An Abductive Approach for Analysing Event-Based Requirements Specifications

Alessandra Russo* Rob Miller# Bashar Nuseibeh♮ Jeff Kramer*

* Department of Computing
Imperial College
180, Queens' Gate, London
SW7 2BZ, U.K.
{ar3, jk}@doc.ic.ac.uk

S.L.A.I.S.
University College London
Gower Street, London
WC1E 6BT, U.K.
rsm@ucl.ac.uk

♮Computing Department
The Open University
Walton Hall, Milton Keynes
MK7 6AA, U.K.
B.A.Nuseibeh@open.ac.uk

Technical Research Report DoC 2001/7

ABSTRACT

We present a logic-based approach for analysing event-based requirements specifications given in terms of a system's reaction to events and safety properties. The approach uses an event-based logic, called the *Event Calculus*, to represent such specifications declaratively. Building on this formalism, the approach uses an abductive reasoning mechanism for analysing safety properties. Given a system description and a safety property, the abductive mechanism is able to identify a complete set of counterexamples (if any exist) of the property in terms of symbolic "current" states and associated event-based transitions. If it fails to find such an answer, this establishes the validity of the safety property with respect to the system description. The approach is supported by a decision procedure that (i) always terminates and (ii) facilitates analysis of this type of properties even in the presence of incomplete domain knowledge, where initial conditions are not completely specified. A case study of an automobile cruise control system specified in SCR is used to illustrate our approach. The technique described is implemented using existing tools for abductive logic programming.

Keywords

Requirements, event-driven specifications, logic-based analysis, abduction, Event Calculus, logic programming.

1. Introduction

Analysis of requirements specifications is a critical activity in the software development process. Specification errors, which if undetected often lead to system failures, are in general less expensive to correct than defects detected later in the development process. Techniques for the detection and analysis of errors in requirements specifications are therefore crucial for successful and efficient development of software systems.

This paper describes a formal approach to this task, and an associated tool, that have the following two desirable characteristics. *First*, the tool is able to verify some properties and detect some errors even when requirements specifications are only partial, and even when only partial knowledge about the domain is available. In particular, our approach does not rely on a complete description of the initial state(s) of the system, making it applicable to systems embedded in complex environments whose initial conditions cannot be specified completely. *Second*, the tool provides diagnostic information about detected errors (e.g. violated safety properties) as a debugging aid for the engineer. In practical terms, it is the integration of *both* these characteristics that distinguishes our approach from other formal techniques, such as those based on model checking or theorem proving [9].

Our focus is on *event-based* requirements specifications. For the purposes of this paper, we will regard such specifications as composed of *system descriptions*, i.e. system requirements expressed in terms of required reactions to *events* (inputs, changes in environmental conditions, etc.), and safety properties.

The approach uses an event-based logic, called the *Event Calculus* [32], to declaratively model event-based requirements specifications. This choice of representation is motivated by both practical and formal needs, and has several advantages. First, and in contrast to pure state-transition based representations, the Event Calculus ontology includes an explicit structure of time that is independent of any (sequence of) events under consideration. This characteristic makes it straightforward to model event-based systems where a number of input events may occur simultaneously, and where the system behavior may in some circumstances be non-deterministic (see [37]). Second, the Event Calculus ontology is close enough to existing types of event-based requirements specifications to allow them to be mapped automatically into the logical representation. This allows our approach and tool to be used as a back-end to existing requirements engineering representations, with the additional advantage that both the semantics of the front-end requirements specification language and individual specifications themselves have a common declarative formal representation. Third, we can prove a general property of the particular class of Event Calculus representations employed here that allows us to reason with a reduced two-state representation (see Section 2.3), thus substantially improving the efficiency of our tool. Fourth, we can build on a substantial body of existing work in applying *abductive reasoning* techniques to Event Calculus representations [30; 37].

This brings us to the second corner stone of our approach, which is the use of *abduction*. In Artificial Intelligence (A.I.) abduction is one of three common modes of automated reasoning (the other two being *deduction* and *induction*). In pure logical terms, assuming T to be a theory, Δ a set of assumptions, and α a formula, then deduction is an analytic reasoning process that uses the assumptions in Δ to infer from T the consequence α. Abduction is, on the other hand, a constructive reasoning process that identifies the set of

assumptions Δ, which when added to the given theory allows the inference of the consequence α. Abduction is a technique for generating "explanations" or "plans" (e.g. Δ) for given "observations" or "goals"(e.g, α)[1]. In software engineering terms, the theory T can be seen as a system description, the set of assumptions Δ as a set of specific instances of system behavior, and the consequence α as system property that has to hold after the execution of system behavior(s). Abduction has been shown to be suitable for automating tasks such as diagnosis [11], planning [15], theory and database updates [10; 24; 26], and knowledge-based software development [35; 43; 44]. Our approach employs abduction in a *refutation mode* to verify safety properties with respect to event-based system descriptions[2]. Given a system description and a safety property, the abductive mechanism is able to identify a (possibly *complete*) set of counterexamples (if any exist) of the system property, where each such counterexample is in terms of a "current" system state and an associated event-based transition. Failure to find such a counterexample establishes the validity of the safety property with respect to the system description. (Thus, in A.I. terminology, each counterexample is an "explanation" for the "observation" which is the negation of the property at an arbitrary symbolic time-point.) The particular form of these counterexamples makes them ideal as diagnoses that can be used to modify the specification appropriately, by altering either the event-based system description, or the safety properties, or both.

The abductive decision procedure employed by our approach has several desirable features. First, in contrast to most conventional theorem proving techniques, it always terminates. Second, in contrast to model-checking approaches, it does not rely on a complete description of some initial system state. Third, like the Event Calculus representation, it supports reasoning about specifications of systems whose state-spaces may be infinite. This last feature is mainly because the procedure is *goal-* or *property-driven*.

The paper is organised as follows. Section 2 describes our general approach. It first reviews the general technique of abduction, and how it may be employed in refutation mode to prove or disprove safety properties. It then describes the variant of Event Calculus we have developed for our approach, how we can efficiently apply abduction in this context, and our prototype tool. Section 3 describes a case study in analysing a Software Cost Reduction (SCR) tabular specification. It reviews SCR focusing on mode transition tables and mode invariants. Then it shows how SCR tables and mode invariants can be straightforwardly mapped into an Event Calculus specification, the kind of diagnostic information that our abductive technique is able to provide, and some heuristics for identifying possible changes. Section 4 describes our logic programming tool and results about its soundness and completeness with respect to the logical formalization of our approach given in Section 2. Section 5 discusses comparison with related work, and Section 6 concludes the paper with a discussion about possible future extensions to our approach.

[1] Induction is a synthetic reasoning process that generates universally quantified formulae from collections of specific instances.

[2] This specific use of abduction in refutation mode is an innovative way of deploying such reasoning mechanism with respect to the various existing applications of abduction presented so far in the Artificial Intelligence literature.

2. Our Approach

As stated above, we will regard requirements specifications as composed of system descriptions and safety properties. The analysis task that we are concerned with is to discover whether a given system description satisfies all safety properties, and if not why not. We express a collection of safety properties as logical sentences $I_1,\ldots,I_n$ and an event-based system description as a set of rules S. Thus for each safety property I_i, we need to evaluate whether $S \models I_i$, and, if not, to generate appropriate diagnostic information. The Event Calculus representation we have employed allows us to use an abductive reasoning mechanism to combine these two tasks into a single automated decision procedure. The next two sections describe in detail the two features of our approach: (i) the Event Calculus formalism used for representing event-based specifications, and (ii) the analysis process for safety properties, based on abduction.

2.1 The Event Calculus

The Event Calculus [32] is a logic-based formalism for representing and reasoning about dynamic systems. Its ontology includes an explicit structure of time that is independent of any (sequence of) events or actions under consideration. As we shall see, this characteristic makes it straightforward to model a wide class of event-driven systems including those that are non-deterministic, those in which several events may occur simultaneously, and those for which the state space is infinite. Our approach has, so far, been tested only on specifications for deterministic systems, such as the case study described in Section 3 (which is an SCR style specification). However, we are currently investigating its applicability to LTS style specifications [33], which may be for concurrent and non-deterministic systems.

Different forms of Event Calculus have been presented in the literature [32; 45]. Our approach adapts a simple classical logic form [37], whose ontology consists of (i) a set of *time-points* isomorphic to the non-negative integers, (ii) a set of time-varying properties called *fluents*, and (iii) a set of *event types* (or *actions*). The logic is correspondingly sorted[3], and includes the predicates *Happens*, *Initiates*, *Terminates* and *HoldsAt*, as well as some auxiliary predicates defined in terms of these. *Happens(a,t)* indicates that event (or action) *a* actually occurs at time-point *t*. *Initiates(a,f,t)* (resp. *Terminates(a,f,t)*) means that if event *a* were to occur at *t* it would cause fluent *f* to be true (resp. false) immediately afterwards. *HoldsAt(f,t)* indicates that fluent *f* is true at *t*. So, for example, to indicate that events *A1* and *A2* occur simultaneously at time-point *T4* it is sufficient to assert:

$$[Happens(A1,T4) \wedge Happens(A2,T4)].$$

System descriptions as axiomatisations.

Every Event Calculus description includes a core collection of domain-independent axioms (sentences) that describe general principles for deciding when fluents hold or do not hold at particular time-points. In addition, each specification includes a collection of domain- or scenario-dependent sentences, describing the particular effects of events or actions (using the predicates *Initiates* and *Terminates*), and may also include sentences stating the

[3] The language may contain function as well as constant symbols of sorts fluent and event, in order for example to allow for parameterised fluents and infinite state spaces.

particular time-points at which instances of these events occur (using the predicate *Happens*).

To write the domain-independent axioms succinctly, it is convenient to introduce two auxiliary predicates, *Clipped* and *Declipped*. *Clipped(t1,f,t2)* means that some event occurs between the times *t1* and *t2* which terminates the fluent *f*. In logic, this is:

$$Clipped(t1,f,t2) \equiv \exists a,t[Happens(a,t) \wedge t1 \leq t < t2 \wedge Terminates(a,f,t)]$$

(In this and all other axioms all variables are assumed to be universally quantified with maximum scope unless otherwise stated.) Similarly, *Declipped(t1,f,t2)* means that some event occurs between the times *t1* and *t2* that initiates the fluent *f*:

$$Declipped(t1,f,t2) \equiv \exists a,t[Happens(a,t) \wedge t1 \leq t < t2 \wedge Initiates(a,f,t)]$$

Armed with this notational shorthand, we can state the three general (commonsense) principles that constitute the domain-independent component of the Event Calculus: (i) fluents that have been initiated by event occurrences continue to hold until events occur that terminate them:

$$HoldsAt(f,t2) \leftarrow \exists a,t1[Happens(a,t1) \wedge Initiates(a,f,t1) \wedge t1 < t2 \wedge \neg Clipped(t1,f,t2)]$$

(ii) fluents that have been terminated by event occurrences continue not to hold until events occur that initiate them:

$$\neg HoldsAt(f,t2) \leftarrow \exists a,t1[Happens(a,t1) \wedge Terminates(a,f,t1) \wedge t1 < t2 \wedge \neg Declipped(t1,f,t2)]$$

and (iii) fluents only change status via occurrences of initiating and terminating events:

$$HoldsAt(f,t2) \leftarrow [HoldsAt(f,t1) \wedge t1 < t2 \wedge \neg Clipped(t1,f,t2)]$$

$$\neg HoldsAt(f,t2) \leftarrow [\neg HoldsAt(f,t1) \wedge t1 < t2 \wedge \neg Declipped(t1,f,t2)]$$

The above axiomatisation also allows the representation of event-based systems where a number of input events may occur simultaneously and where the system behavior may in some circumstances be non-deterministic. The non-determinism can, in general, be caused by either the occurrence of an (input) event, which can have complementary effects on a given fluent (e.g., the event “toss a coin” can both initiate and terminate a fluent “head”), or by the occurrence of simultaneous events, which have complementary effects on the same fluent. In either these two cases, the above axiomatisation would not allow the inference of information about the effects of non-deterministic event(s), as it would accept different models for different non-deterministic behaviors.

From the viewpoint of formal representation and reasoning, the above axiomatisation is expressive enough to model both deterministic and non-deterministic event-based system specifications, and also specifications with sequential and simultaneous events. However, for the purpose of abductive analysis of safety properties, the event-based system specifications have to be deterministic. In the case of non-deterministic systems, failure to identify examples of system behavior, which together with the system description entail the negation of a safety property, can be due to the property being actually satisfied as well as to the non-

determinism. In the latter case, our abductive approach would provide unsound results[4]. This is, however, not a real limitation of the approach, since any non-deterministic system description can be transformed into an equivalent deterministic system specification behavior. To model deterministic event-driven requirements specifications it is sufficient to strengthen the above Event Calculus axiomatisation by adding the following deterministic axiom. This axiom simply excludes the two possible causes for non-determinism described above:

$$\neg\exists\, t, a1, a2, f\ [Initiates(a1,f,t) \wedge Terminates(a2,f,t) \wedge Happens(a1,t) \wedge Happens(a2,t)]$$

To model event-driven requirements specifications where events only happen one at a time, as in the case study described in Section 3, it is sufficient to strengthen the above Event Calculus axiomatisation by adding the following axiom:

$$\neg\exists\, t, a1, a2, f\ [Happens(a1,t) \wedge Happens(a2,t) \wedge a1 \neq a2]$$

We can now define the domain-independent Event Calculus axiomatisation used in our approach and denoted by EC_{Ax}.

Definition 1. [EC_{Ax}] The domain-independent Event Calculus axiomatisation EC_{Ax} consists of the following seven axioms:

$$Clipped(t1,f,t2) \equiv \exists a,t[Happens(a,t) \wedge t1 \leq t < t2 \wedge Terminates(a,f,t)] \quad \text{(EC1)}$$

$$Declipped(t1,f,t2) \equiv \exists a,t[Happens(a,t) \wedge t1 \leq t < t2 \wedge Initiates(a,f,t)] \quad \text{(EC2)}$$

$$HoldsAt(f,t2) \leftarrow \exists a,t1[Happens(a,t1) \wedge Initiates(a,f,t1) \wedge t1 < t2 \wedge \neg Clipped(t1,f,t2)] \quad \text{(EC3)}$$

$$\neg HoldsAt(f,t2) \leftarrow \exists a,t1[Happens(a,t1) \wedge Terminates(a,f,t1) \wedge t1 < t2 \wedge \neg Declipped(t1,f,t2)] \quad \text{(EC4)}$$

$$HoldsAt(f,t2) \leftarrow [HoldsAt(f,t1) \wedge t1 < t2 \wedge \neg Clipped(t1,f,t2)] \quad \text{(EC5)}$$

$$\neg HoldsAt(f,t2) \leftarrow [\neg HoldsAt(f,t1) \wedge t1 < t2 \wedge \neg Declipped(t1,f,t2)] \quad \text{(EC6)}$$

$$\neg\exists\, t, a1, a2, f\ [Initiates(a1,f,t) \wedge Terminates(a2,f,t) \wedge Happens(a1,t) \wedge Happens(a2,t)] \quad \text{(ECD)}$$

□

To illustrate how the effects of particular events may be described in the domain-dependent part of a specification using the predicates *Initiates* and *Terminates*, we will describe an electric circuit consisting of a single light bulb and two switches A and B all connected in series. We need three fluents, SwitchAOn, SwitchBOn and LightOn, and two actions FlickA and FlickB. We can describe facts such as (i) that flicking switch A turns the light on, provided that switch A is not already on and that switch B is already on (i.e. connected) and is not simultaneously flicked:

$$Initiates(FlickA,LightOn,t) \leftarrow [\neg HoldsAt(SwitchAOn,t)$$

[4] A technical justification for restricting our approach to just event-based specifications of deterministic systems can be found in the proof of Theorem 1, given in Section 2.2.

$$\wedge\ HoldsAt(SwitchBOn,t)$$
$$\wedge\ \neg Happens(FlickB,t)]$$

(ii) that if neither switch is on, flicking them both simultaneously causes the light to come on:

$$Initiates(FlickA,LightOn,t) \leftarrow$$
$$[\neg HoldsAt(SwitchAOn,t)$$
$$\wedge\ \neg HoldsAt(SwitchBOn,t)$$
$$\wedge\ Happens(FlickB,t)]$$

and (iii) that if either switch is on, flicking it causes the light to go off (irrespective of the state of the other switch):

$$Terminates(FlickA,LightOn,t) \leftarrow [HoldsAt(SwitchAOn,t)]$$
$$Terminates(FlickB,LightOn,t) \leftarrow [HoldsAt(SwitchBOn,t)]$$

In fact, in this example we need a total of five such sentences to describe the effects of particular events or combinations of events on the light, and a further four sentences to describe the effects on the switches themselves. Although for readability these sentences are written separately here, it is the *completions* (i.e. the if-and-only-if transformations) of the sets of sentences describing *Initiates* and *Terminates* that are actually included in the specification. The completion of the two *Terminates* clauses above, for example, is:

$$Terminates(a,f,t) \equiv [[a=FlickA \wedge f=LightOn \wedge HoldsAt(SwitchAOn,t)] \vee$$
$$[a=FlickB \wedge f=LightOn \wedge HoldsAt(SwitchBOn,t)]\ ]$$

The use of such completions avoids the frame problem, i.e. it allows us to assume that the only effects of events are those explicitly described[5].

For many applications, it is appropriate to include similar (completions of) sets of sentences describing which events occur (when using the predicate *Happens*). However, in this paper we wish to prove properties of systems under all possible scenarios, i.e. irrespective of which events actually occur. Hence our descriptions leave *Happens* undefined, i.e. they allow models with arbitrary interpretations for *Happens*. In this way, we effectively simulate a branching time structure that covers every possible series of events. In other words, by leaving *Happens* undefined we effectively consider, in one model or another, every possible path through a state-transition graph. The formal definition of a domain-dependent Event Calculus description, denoted with EC_{Spec} is as follows[6].

---

[5] The "frame problem" is the problem of stating concisely that in general almost all fluents that hold true at a given instant of time continue to hold after an event has been performed [43].This corresponds to the fact that requirements engineers do not want to have to explicitly list "non-effects" of (input) events. For example, SCR mode transition tables do not explicitly describe sets of conditions and input types for which no mode transition occurs.

[6] In the case of finite domains the set of uniqueness-of-name axioms can be expressed in the form of $\forall f, \forall e.[(f = F_1 \vee f = F_2 \vee \ldots \vee f = F_n) \wedge (e = E_1 \vee e = E_2 \vee \ldots \vee e = E_n)]$, where F_i and E_i are, respectively, ground fluents and ground events in the Event Calculus language. The inclusion of such axioms and of the existence-of-name axioms guarantees the models of a given (Event Calculus) specification to be based only on the domain defined in the specification.

Definition 2. [Specification EC_{Spec}] A set EC_{Spec} is an Event Calculus *specification* if it contains exactly the following:

a. The completion of a set of *Initiates* clauses each of the form
   $Initiates(A,F,t) \leftarrow \Pi$
   where Π is a conjunction of (positive and negative) *HoldsAt* and *Happens* literals whose event and fluent terms are ground and whose only timepoint term is *t*.

b. The completion of a set of *Terminates* clauses each of the form
   $Terminates(A,F,t) \leftarrow \Pi$
   where Π is a conjunction of (positive and negative) *HoldsAt* and *Happens* literals whose event and fluent terms are ground and whose only timepoint term is *t*.

c. A set of uniqueness-of-name axioms for all event and fluent terms.

d. A set of existence-of-name axioms or an existence-of-names schema for each sort of the language. □

Safety properties are assertions about attributes of system states that must hold for every possible system execution. They can therefore be expressed in Event Calculus as universally quantified formulae of the form $\forall t.I(t)$, where $I(t)$ is a logical expression without quantifiers, using only the predicate *HoldsAt* and ground fluent terms corresponding to the system attributes relevant to the properties.

Definition 3. [*Safety property*] A safety property is an expression of the form $\forall t.I(t)$, where $I(t)$ is a formula without quantifiers, whose only predicate is *HoldsAt* and whose fluent terms are ground. □

Other types of properties can also be formalised in Event Calculus. In particular, temporal logic formulae (e.g. [5; 42]) can be represented as equivalent *reified*[7] Event Calculus first-order formulae. For the purpose of this paper, we will consider only the class of safety properties defined below. Examples of such properties are given in Section 3.

2.2 Abduction for Verification

2.2.1 Defining an abductive framework

Abduction is commonly defined as the problem of finding a set of hypotheses (an "explanation" or "plan") of a specified form that, when added to a given specification, allows an "observation" or "goal" sentence to be inferred, without causing contradictions [30]. In logical terms[8], given a domain description D and a sentence (goal) G, abduction attempts to identify a set Δ of assertions such that:

$$D \cup \Delta \models G, \qquad (1)$$
$$D \cup \Delta \text{ is consistent.} \qquad (2)$$

The set Δ is often required to satisfy two main properties: (1) it must consist only of *abducible* sentences, where the definition of what is abducible is generally domain-specific (e.g., assertions about performances of particular actions in the case of planning), and (2) it

[7] Reified in the sense that properties, which might otherwise be represented as predicates or propositions, are instead represented as fluents inside the HoldsAt predicate, and thus becoming terms of the language.

[8] Throughout the paper, the mathematical symbol $\models$ denotes logical entailment.

is minimal. The set of abducible sentences is defined a priori to reflect some domain-specific notion of causality with respect to the given properties. For instance, in the electric circuit example given before, the occurrences of events *FlickA* and *FlickB* cause the light bulbs to be on and/or off, depending on the current states of the system. This state recursively depends on previous events occurrences of *FlickA* and *FlickB*. So observations about properties of system states can be explained in terms of sequences of ground *Happens* literals and ground literals on what is true or false at the initial state. These types of information would constitute our set of abducible sentences. This is also the case for arbitrary event-driven specifications – system behaviors are caused by occurrences of event transitions in specific system states. Observations such as violation of a safety property can be explained in terms of sequences of environmental events and conditions over the initial state of the system. Therefore, in our approach, abducible sentences are ground *Happens* literals and specific types of ground *HoldsAt* literals, as stated in the following two definitions.

Definition 4. [*Abductive predicates set* Ab] The set Ab of *abductive predicates* is given by the predicates *HoldsAt* and *Happens*. □

Definition 5. [*Abducible sentence*] Given an abductive predicates set Ab, an *abducible sentence* is a ground literal over Ab given by a ground *HoldsAt* literal whose timepoint term is *0* or a ground *Happens* literal. □

The minimality property means that the set Δ of abduced explanations should not be subsumed by other explanations. For instance, in the above electric circuit example, given the observation *¬HoldsAt(LightOn,t+1)*, the explanation {*HoldsAt(SwitchAOn,t)*, *HoldsAt(SwitchBOn,t)*, *Happens(FlickA,t)*, *Happens(FlickB,t)*} is not minimal since there are two other possible explanations, namely, {*HoldsAt(SwitchAOn,t)*, *Happens(FlickA,t)*} and {*HoldsAt(SwitchBOn,t)*, *Happens(FlickB,t)*}, which subsume the first one.

An abductive reasoning process also takes into account domain specific *constraints* [30]. These are in general used to define the classes of "correct" models for a given system specification. Examples of constraints are, for instance, natural physics laws of the environment in which a system is supposed to run, or, more specifically, environmental assertions in SCR style requirements specifications. In the presence of such constraints, the abductive reasoning process is able to generate explanations that satisfy such constraints. In logical terms, given a domain description D, a set of integrity constraints C, and a sentence (goal) G, abductive reasoning in the presence of constraints attempts to identify a set Δ of assertions such that:

$$D \cup \Delta \models G \qquad (1')$$

$$D \cup \Delta \text{ is consistent and } D \cup \Delta \models C. \qquad (2')$$

Condition (2') is stronger than condition (2) given before, since it has the effect of further reducing the collection of explanations generated by the abductive reasoning process to just those that satisfy the given set of constraints.

In our approach, constraints take the form of formulae without quantifiers, whose only predicate is *HoldsAt*, whose fluent terms are ground and whose only timepoint is a timepoint term τ (see Definition A1 in Appendix). To summarise, the *abductive framework* used in our approach is the triple <D, Ab, C> where D is a domain description given by the domain

independent Event Calculus axiomatisation EC_{Ax} and (the Event Calculus representation of) an event-based specification EC_{Spec}, the component Ab is the set of abductive predicates and C is a set of constraints (see Definition A2 in Appendix).

2.2.2 Using abduction to analyse safety properties

To analyse safety properties in event-based specifications means showing, in general, that given a safety property I_i and a (formal) specification D, it is the case that $D \models I_i$. In our approach this analysis task is translated into an equivalent problems; that is, to show that it is not possible to consistently extend the specification with assertions that particular events have actually occurred (i.e. with a Δ) in such a way that the extended description entails $\neg I_i$. In other words, there is no set Δ such that $D \cup \Delta \models \neg I_i$. The equivalence of these two problems, illustrated in Figure 1 and proved in Theorem 1 below, depends on the particular Event Calculus representation used in our approach.

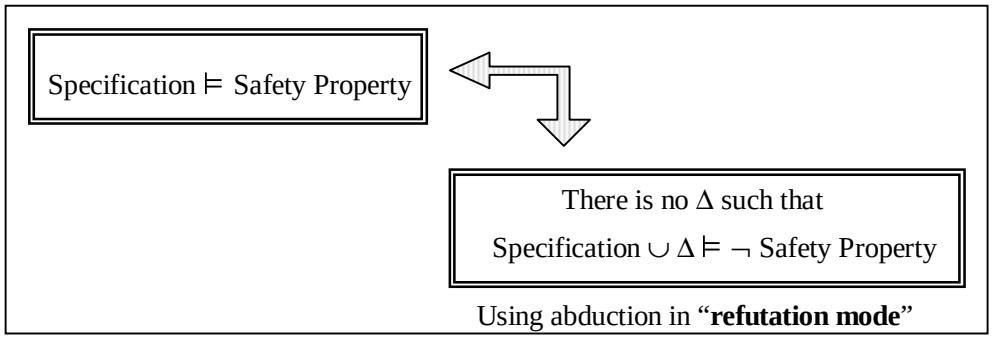

Figure 1. Using abduction to analyse safety properties

The problem of showing that there is no set of abducibles Δ, such that $D \cup \Delta \models \neg I_i$ for a given domain description and safety property, is solved by attempting to generate such a Δ using a complete abductive decision procedure. Given an abductive framework defined above and a safety property I_i, the analysis of this property consists in trying to identify a set Δ of assertions such that $D \cup \Delta \models \neg I_i$, and, in the presence of domain dependent constraints C, such that $D \cup \Delta \models C$ as well. We refer to this analysis process as using abduction in a *refutation mode*. If the abductive procedure finds such a Δ, then the set Δ acts as a counterexample (see Definition 6 below). The completeness of the abductive proof procedure (see Theorem A1 in the Appendix) guarantees that failing to find such a set Δ establishes the validity of the property in the given event-based specification.

Definition 6. [*Counterexample of a safety property*] Let <D, Ab, C> be an abductive framework where D is a domain description given by the set EC_{Ax} and a specification EC_{Spec}; Ab is a set of abducible predicates and C is a (possibly empty) set of constraints. Let $\forall t.I(t)$ be a safety property. A set Δ of abducibles over Ab is a counterexample of the safety property if and only if the following two conditions hold:

(1) $D \cup \Delta \models \neg \forall t.I(t)$
(2) $D \cup \Delta$ is consistent, and $D \cup \Delta \models C$ (when $C \neq \varnothing$) □

As we shall see, the form of such counterexamples makes them ideal as diagnostic information that can be utilised to change the description and/or safety properties. The

counterexamples generated by our approach describe particular events occurring in particular "contexts" (i.e. classes of current states). These contexts must themselves satisfy the property. This is ensured by considering the property instantiated on a symbolic current state also as part of the domain description, so as to prune the set of possible counterexamples. A detailed description of the particular abductive proof procedure used in our approach is given in Section 4.2.

Theorem 1. [*Abduction theorem*]. Let <D, Ab, C> be an abductive framework, and let $\forall t.\mathrm{I}(t)$ be a safety property. Then, $\mathrm{D} \models \forall t.\mathrm{I}(t)$ if and only if there exists no counterexample of the safety property $\forall t.\mathrm{I}(t)$.

Proof. The "only if" part of the theorem can be easily proved reasoning by contradiction and using the monotonicity property of classical logic. The proof of the "if part" of the theorem is more elaborate. Let us assume, for simplicity, that the set of constraints C are always satisfied in any possible extension $\mathrm{D} \cup \Delta$ of the domain description D, for any set Δ. Therefore, we know that there is no Δ such that $\mathrm{D} \cup \Delta \models \neg \forall t.\mathrm{I}(t)$, and $\mathrm{D} \cup \Delta$ is consistent. Let us suppose now that $\mathrm{D} \not\models \forall t.\mathrm{I}(t)$. This means that there is a classical model $\mathcal{M}$ that satisfies D and $\neg \forall t.\mathrm{I}(t)$ (i.e. $\mathcal{M} \Vdash \mathrm{EC}_{\mathrm{Ax}} \wedge \mathrm{EC}_{\mathrm{Spec}} \wedge \neg \forall t.\mathrm{I}(t)$, which means that the axioms in $\mathrm{EC}_{\mathrm{Ax}}$ and in $\mathrm{EC}_{\mathrm{Spec}}$ and the formula $\neg \forall t.\mathrm{I}(t)$ are true in the model $\mathcal{M}$. Given the syntactic form of the safety property, $\mathcal{M} \Vdash \neg \forall t.\mathrm{I}(t)$ implies that there is a time point n such that $\mathcal{M} \Vdash \neg \mathrm{I}(n)$. By the *deterministic* property of $\mathrm{EC}_{\mathrm{Spec}}$ (see Lemma 1 in Appendix), we also know that there is no other model $\mathcal{M}'$ that agrees with $\mathcal{M}$ up to the time point n, and such that $\mathcal{M}' \Vdash \forall t.\mathrm{I}(t)$. We can construct now from $\mathcal{M}$ the set of abducibles Δ as follows:

$$\Delta = \{\neg \mathit{HoldsAt(F',0)} \mid \mathcal{M} \Vdash \neg \mathit{HoldsAt(F',0)}\} \cup \{\mathit{HoldsAt(F',0)} \mid \mathcal{M} \Vdash \mathit{HoldsAt(F',0)}\} \cup$$
$$\{\mathit{Happens(E',0)} \mid \mathcal{M} \Vdash \mathit{Happens(E',0)}\} \cup \{\neg \mathit{Happens(E',0)} \mid \mathcal{M} \Vdash \neg \mathit{Happens(E',0)}\}$$

By construction, Δ is consistent with D, and by the *deterministic* property of $\mathrm{EC}_{\mathrm{Spec}}$ $\mathrm{D} \cup \Delta \models \neg \forall t.\mathrm{I}(t)$, which contradicts the initial hypothesis. □

It is easy to see that the above theorem would not hold if the Event Calculus axiomatisation did not include the deterministic axiom (ECD). This is because the proof of existence of one counter model for a given safety property would not be sufficient to guarantee the existence of a (minimal) extension $\mathrm{D} \cup \Delta$ of the given specification that would violate the property in all the possible (non-deterministic) models of system behaviors.

2.3 Efficient Abduction with Event Calculus

In the previous section we defined the analysis of a safety property as the process of detecting, by means of abduction, counterexample(s) of the property. However, given that such properties are sentences universally quantified over time, it is (potentially) computationally expensive to demonstrate their truth by standard (deductive or abductive) theorem-proving techniques. To overcome this problem we make use of a "reduction step" that simplifies the given inference task to a simpler one based on an appropriately instantiated domain description and safety property. The abductive proof procedure is then applied to this reduced (ground) description so making the reasoning process more efficient. The proof of soundness and completeness of this reduction step is given in Theorem 2.

This simplification step can be seen as a form of abstraction, which reduces a complex semi-decidable and computationally expensive inference process to two propositional inference

tasks, which are thus fully decidable and computationally tractable. This abstraction process is based on proof by induction, and it can therefore be used only to analyse the class of safety properties considered in this paper (i.e. universally quantified formulae with no nesting temporal quantifiers). We speculate, however, that abductive reasoning in general, as described in section 2.2.1, can also be used to analyse other classes of system properties, such as liveness properties, but this is a topic for future investigation.

The reduced domain description of an event-based specification is given by considering a simple time structure consisting of two (arbitrary) timepoints *Sc* and *Sn*, respectively, such that *Sc* < *Sn*. The Event Calculus axiomatisation and event-based specification are both grounded over this simple time structure as shown in the following two definitions.

Definition 7. [EC_{Ax}(**S**)**]** Let **S** be a time structure consisting of two timepoint *Sc* and *Sn*, with *Sc* < *Sn*. The set EC_{Ax}(**S**) consists of the following axioms:

$$Clipped(Sc,f,Sn) \equiv \exists a[Happens(a,Sc) \wedge Terminates(a,f,Sc)] \quad \text{(ECS1)}$$

$$Declipped(Sc,f,Sn) \equiv \exists a[Happens(a,Sc) \wedge Initiates(a,f,Sc)] \quad \text{(ECS2)}$$

$$HoldsAt(f,Sn) \leftarrow \exists a[Happens(a,Sc) \wedge Initiates(a,f,Sc) \wedge \neg Clipped(Sc,f,Sn)] \quad \text{(ECS3)}$$

$$\neg HoldsAt(f,Sn) \leftarrow \exists a[Happens(a,Sc) \wedge Terminates(a,f,Sc) \wedge \neg Declipped(Sc,f,Sn)] \quad \text{(ECS4)}$$

$$HoldsAt(f,Sn) \leftarrow [HoldsAt(f,Sc) \wedge \neg Clipped(Sc,f,Sn)] \quad \text{(ECS5)}$$

$$\neg HoldsAt(f,Sn) \leftarrow [\neg HoldsAt(f,Sc) \wedge \neg Declipped(Sc,f,Sn)] \quad \text{(ECS6)}$$

$$\neg \exists\, a1, a2, f\, [Initiates(a1,f,Sc) \wedge Terminates(a2,f,Sc) \wedge Happens(a1,Sc) \wedge Happens(a2,Sc)] \quad \text{(ECSD1)}$$

$$\neg \exists\, a1, a2, f\, [Initiates(a1,f,Sn) \wedge Terminates(a2,f,Sn) \wedge Happens(a1,Sn) \wedge Happens(a2,Sn)] \quad \text{(ECSD2)}$$

□

The set of axioms EC_{Ax}(**S**) is then an instantiation of the set of axioms EC_{Ax} over the two timepoint structure **S**. Similar instantiation is performed on the domain description EC_{Spec} as shown below.

Definition 8. [EC_{Spec}(**S**)] Let EC_{Spec} be a specification and let **S** be a time structure consisting of two timepoints *Sc* and *Sn*, with *Sc* < *Sn*. The set EC_{Spec}(**S**) is the corresponding ground specification containing exactly the following ground rules:

a. The completion of a set of *Initiates* clauses each of the form

   $Initiates(A,F,Sc) \leftarrow \Pi$

   where Π is a conjunction of (positive and negative) *HoldsAt* and *Happens* literals whose event and fluent terms are ground and whose only timepoint term is *Sc*.

b. The completion of a set of *Terminates* clauses each of the form

   $Terminates(A,F,Sc) \leftarrow \Pi$

   where Π is a conjunction of (positive and negative) *HoldsAt* and *Happens* literals whose event and fluent terms are ground and whose only timepoint term is *Sc*.

c. A set of uniqueness-of-name axioms for all event and fluent terms.

d. A set of existence-of-name axioms or an existence-of-names schema for each sort of the language. □

Theorem 2. [*Reduction theorem*]. Let D be a domain description given by the set EC_{Ax} and a specification EC_{Spec}. Let $\forall t.I(t)$ be a safety property. Let **S** be a time structure consisting of two timepoints *Sc* and *Sn*, with *Sc* < *Sn*. Let D(**S**) be the ground domain description given by the set EC_{Ax}(**S**) and the ground specification EC_{Spec}(**S**). Then $D \models \forall t.I(t)$ if and only if $D \models I(0)$ and $D(\mathbf{S}) \cup \{I(Sc)\} \models I(Sn)$.

Proof. By Lemma A1 in the Appendix, $D \models \forall t.I(t)$ if and only if $D \models I(0)$ and $D \cup \{I(Sc)\} \models I(Sn)$. By Lemma A2 in the Appendix, $D \cup \{I(Sc)\} \models I(Sn)$ if and only if $D(\mathbf{S}) \cup \{I(Sc)\} \models I(Sn)$. Hence, $D \models \forall t.I(t)$ if and only if $D \models I(0)$ and $D(\mathbf{S}) \cup \{I(Sc)\} \models I(Sn)$.

□

As a corollary of both Theorems 1 and 2 given above, we can now state the soundness and completeness of our abductive reasoning technique for the analysis of safety properties.

Corollary 1. Let D be a domain description given by the set EC_{Ax} and a specification EC_{Spec}. Let $\forall t.I(t)$ be a safety property. Let **S** be a time structure consisting of two timepoints *Sc* and *Sn*, with *Sc* < *Sn*. Let D(**S**) be the ground domain description given by the set EC_{Ax}(**S**) and the ground specification EC_{Spec}(**S**) and let C be a set of domain specific constraints. Then $D \models \forall t.I(t)$ if and only if $D \models I(0)$, and in the abductive framework <D(**S**), Ab, C> there exists no counterexample Δ of the safety property $\forall t.I(t)$. □

Hence, to show for some safety property $\forall t.I(t)$ that $EC_{Ax} \models \forall t.I(t)$ it is sufficient to show (i) that the property is initially true (i.e. $I(0)$ is true), and (ii) that $D(\mathbf{S}) \cup \{I(Sc)\} \models I(Sn)$, where **S** is a simple time structure consisting of just two points *Sc* and *Sn* such that *Sc* < *Sn* ("*c*" for "current" and "*n*" for "next"), and D(**S**) is the ground domain description $EC_{Ax}(\mathbf{S}) \cup EC_{Spec}(\mathbf{S})$. Therefore, given that a safety property is satisfied at the initial timepoint *0*, to verify that it is always true it is sufficient to consider only a symbolic time-point *Sc* and its immediate successor *Sn*, assume the property to be true at *Sc*, and demonstrate that its truth follows at *Sn*. The results given in Theorems 1 and 2 and Corollary 1 are also applicable even when complete information about the initial state of the system is not available. Their utilisation reduces computational costs considerably because, in the context of EC_{Ax}(**S**), it allows us to re-write all our Event Calculus axioms with ground time-point terms (as shown in Definition 7).

Once the Event Calculus representation of an event-based requirements specification is provided (possibly by automatic translation), the reasoning task of showing that a safety property is true at the initial state is a simple theorem proving task where the formula to prove (i.e. $I(0)$) is a ground sentence. The second reasoning task, i.e. $D(\mathbf{S}) \cup \{I(Sc)\} \models I(Sn)$, is, on the other hand, where our abductive analysis technique, defined and proved in Theorem 1, is applied. Using the reduced time structure described above, our approach proves assertions of the form $D(\mathbf{S}) \cup \{I(Sc)\} \models I(Sn)$ by showing that a *complete abductive proof procedure* fails to produce a set Δ of *HoldsAt* and *Happens* facts, grounded at *Sc*, such that $D(\mathbf{S}) \cup \{I(Sc)\} \cup \Delta \models \neg I(Sn)$. If, on the other hand, the abductive procedure produces such a set *Δ*, then this *Δ* is an explicit indicator of where in the specification there is a

problem. The case study gives such an example of generation of diagnostic information from the violation of a safety property.

3. A Case Study

In this section we describe, via an example, an application of our approach to analysing Software Cost Reduction (SCR) specifications. We show how our tool analyses particular SCR-style safety properties, called *mode invariants*, with respect to event-based system descriptions expressed as *SCR mode transition tables*. The SCR approach has been shown to be useful for expressing the requirements of a wide range of large-scale real-world applications [1; 14; 20; 38] and is a popular requirements engineering method for specifying and analysing event-based systems. Other requirement engineering representations for event-based specifications include [17].

3.1 SCR Specifications

The SCR method is based on Parnas's "Four Variable Model", which describes a required system's behavior as a set of mathematical relations between *monitored* and *controlled* variables, and input and output data items [41]. Monitored variables are environmental entities that influence the system behavior, and controlled variables are environmental entities that the system controls. For simplicity, our case study uses only Boolean variables. (This is not a major restriction, since non-Boolean variables can always be reduced to Boolean variables, i.e. predicates defined over their values.) SCR facilitates the description of natural constraints on the system behavior, such as those imposed by physical laws, and defines system requirements in terms of relations between monitored and controlled variables, expressed in tabular notation. Predicates representing monitored and controlled variables are called *conditions* and are defined over single system states. An *event* occurs when a system component (e.g., a monitored or controlled variable) changes value.

Full SCR specifications can include *mode transition*, *event* and *condition* tables to describe a required system behavior, *assertions* to define properties of the environment, and *invariants* to specify properties that are required to always hold in the system (see [6; 19; 20]). However, this case study concerns a simple SCR specification consisting of just a single mode transition table and a list of system properties.

3.1.1 Mode transition tables

Mode classes are abstractions of the system state space with respect to monitored variables. Each mode class can be seen as a state machine, defined on the monitored variables, whose states are modes and whose transitions, called *mode transitions*, are triggered by changes on the monitored variables. Mode transition tables represent mode classes and their respective transitions in a tabular format. The mode transition table for our case study, taken from [4], is given in Table-1. It is for an automobile cruise control system. Note that the table already reflects basic properties of monitored variables. For example, the two transitions from "Inactive" to "Cruise" take into account the environmental property that in any state a cruise control lever is in exactly one of the three positions "Activate", "Deactivate" or "Resume". So, for example, whenever "Activate" changes to true, either "Deactivate" or "Resume" changes to false. For a more detailed description of this case study, the reader is referred to [4].

Current Mode	Ignited	Running	Toofast	Brake	Activate	Deactivate	Resume	New Mode
Off	@T	-	-	-	-	-	-	Inactive
Inactive	@F	f	-	-	-	-	-	Off
	@F	@F	-	-	-	-	-	
	t	t	-	f	@T	@F	f	Cruise
	t	t	-	f	@T	f	@F	
Cruise	@F	@F	-	-	-	-	-	Off
	t	@F	-	-	-	-	-	Inactive
	t	-	@T	-	-	-	-	
	t	t	f	@T	-	-	-	Override
	t	t	f	-	@F	@T	f	
	t	t	f	-	f	@T	@F	
Override	@F	@F	-	-	-	-	-	Off
	t	@F	-	-	-	-	-	Inactive
	t	t	-	f	@T	@F	f	Cruise
	t	t	-	f	@T	f	@F	
	t	t	-	f	f	@F	@T	
	t	t	-	f	@F	f	@T	

Table 1: Mode Transition Table for an automobile cruise control system

Mode transition events occur when one or more monitored variables change their values. Events are of two types: "**@T**(*C*)" when a condition C changes from false to true, and "**@F**(*C*)", when a condition C changes from true to false. C is called a *triggered condition*. For example, in the automobile cruise control system the event "@T(Ignited)" denotes that the engine of the automobile has changed from not being ignited to being ignited. Event occurrences can also depend on the truth/falsity of other conditions. In this case, the events are called *conditioned events*. For example, in Table-1 the mode transition defined in the second row is caused by the occurrence of conditioned event "@F(Ignited)" whose condition is that "Running" is false. Different semantics have been used for conditioned events [21], all of which are expressible in our Event Calculus approach. In this specific case study, we have adopted the same interpretation as used in [4]. An event "@T(C)" conditional on "D" means that "C" is false in the current mode and is changed to true in the new mode, while "D" is true in the current mode and stays true in the new mode. The interpretation is similar for an event "@F(C)" conditional on "D", but with "C" changing truth value from true to false. In a mode transition table, each row is a transition from a current mode, indicated in the left most column of the table, to a new mode, specified in the right most column. The central part of the table defines the events that cause the transition. A triggered event "C" can have entries equal to "@T" or "@F". Monitored variables that are conditions for the occurrence of an event can have entry equal to "t" or "f". Monitored variables that are irrelevant for the transition have a "–" entry.

SCR mode transition tables can be seen as shorthand for much larger tables in two respects. First, a "–" entry for a condition in the table is shorthand for any of the four possible condition's entries "@T", "@F", "t" and "f". This means that any transition between a

current and a new mode specified in a table using n dashes is in effect shorthand for 4^n different transitions, between the same current and new modes, given by the different combinations of entries for each of the dashed monitored variables. For instance, the first transition in Table-1 from mode Inactive to mode "Cruise" is shorthand for four different transitions between "Inactive" and "Cruise" given, respectively, by each of the four entries "t", "f", "@T" and "@F" for the condition "Toofast". Second, tables are made very much more concise by the non-specification of transitions between identical modes. A mode transition table basically describes a function that defines, for each current mode and each combination of condition values, a next mode of the system. This next mode may or may not be equal to the current mode. The function thus uniquely captures the system requirements. However, in specifying real system behavior only the transitions between current and next modes that are different are explicitly represented in SCR tables. The other "transitions" (where current and next modes are identical) are implicit and thus omitted or "hidden" from the table. Hence for the purposes of discussion we may regard the meaning of real SCR mode transition tables as being given by "full extended" (and very long!) mode transition tables that do not utilise "–" dashes and include a row (which might otherwise be "hidden" in the sense described above) for each possible combination of current mode and "t", "f", "@T" and "@F" condition entries. Therefore, both the implicit "hidden rows" and the dashes need to be taken into account when analysing invariants with respect to the real (concise) version of an SCR mode transition table. Our case study shows that both can indeed be causes for mismatch between SCR tables and system invariants, as they may obscure system behaviors that violate these invariants.

3.1.2 Mode invariants

Mode invariants are unchanging properties (specification assertions) of the system regarding mode classes, which should be satisfied by the system specification. They are specific types of safety properties. In our case study of an automobile cruise control system, an example of an invariant is:

$$Cruise \rightarrow (Ignited \wedge Running \wedge \neg Brake)$$

This means that whenever the system is in mode "Cruise", the conditions "Ignited" and "Running" must be true and "Brake" must be false. In SCR notation mode invariants are formulae of the form:

$$m \rightarrow P \qquad \text{(INV)}$$

where m is a mode value of a certain mode class and P is a logical proposition over the conditions used in the associated mode transition table. A mode transition table of a given mode class has to satisfy the mode invariants related to that mode class.

3.2 Abductive Analysis of Mode Invariants

We are now in the position to illustrate the use of our abductive Event Calculus approach to analysing mode invariants in SCR mode transition tables.

3.2.1 The translation

In our translation, both conditions and modes are represented as fluents, which we will refer to as *condition fluents* and *mode fluents* respectively. Although in reality many different types of external, real-word events may affect a given condition, SCR tables abstract these

differences away and essentially identify only two types of events for each condition – a "change-to-true" (@T) and a "change-to-false" (@F) event. Hence in our Event Calculus translation there are no independent event constants, but instead two functions *@T* and *@F* from fluents to events, and for each condition fluent *C*, the two axioms:

$$\forall t.Initiates(@T(C),C,t) \quad \text{(S1)}$$

$$\forall t.Terminates(@F(C),C,t) \quad \text{(S2)}$$

The translation of tables into Event Calculus axioms (rules) is modular, in the sense that a single *Initiates* and a single *Terminates* rule is generated for each row of the table. For a given row, the procedure for generating the *Initiates* rule is as follows. The *Initiates* literal in the left-hand side of the rule has the new mode (on the far right of the row) as its fluent argument, and the first @T or @F event (reading from the left) as its event argument. The right-hand side of the rule includes a *HoldsAt* literal for the current mode, and a pair of *HoldsAt* and *Happens* literals for each "non-dash" condition entry in the row. Specifically, if the entry for condition C is a "t" this pair is $HoldsAt(C,t) \wedge \neg Happens(@F(C),t)$, for "f" it is $\neg HoldsAt(C,t) \wedge \neg Happens(@T(C),t)$, for "@T" the pair $\neg HoldsAt(C,t) \wedge Happens(@T(C),t)$, and for "@F" the pair $HoldsAt(C,t) \wedge Happens(@F(C),t)$. The *Terminates* rule is generated in exactly the same way, but with the current mode as the fluent argument in the *Terminates* literal. For example, the seventh row in Table-1 is translated as follows[9]:

$$Initiates(@F(Running),Inactive,t) \leftarrow$$
$$[\, HoldsAt(Cruise,t) \wedge$$
$$HoldsAt(Ignited,t) \wedge \neg Happens(@F(Ignited),t) \wedge$$
$$HoldsAt(Running,t) \wedge Happens(@F(Running),t) \,]$$

$$Terminates(@F(Running),Cruise,t) \leftarrow$$
$$[\, HoldsAt(Cruise,t) \wedge$$
$$HoldsAt(Ignited,t) \wedge \neg Happens(@F(Ignited),t) \wedge$$
$$HoldsAt(Running,t) \wedge Happens(@F(Running),t) \,]$$

Clearly, this axiom pair captures the intended meaning of individual rows as described in Section 3.1.1.

The semantics of the whole table is given by the two completions of the collections of *Initiates* and *Terminates* rules. These completions (standard in the Event Calculus) reflect the implicit information in a given SCR table that combinations of condition values not explicitly identified are not mode transitions. Indeed, as discussed in Section 3.1.1 we may regard SCR tables as also containing "hidden" or "default" rows (which the engineer does not need to list) in which the current and the new mode are identical. Violations of system invariants are just as likely to be caused by these "hidden" rows as by the real rows of the table. Because our translation utilises completions, the abductive tool is able to identify problems in "hidden" as well as real rows.

[9] Please note that the automatic translation from SCR specifications into Event Calculus descriptions may also consider as a target formalism the simplified ground version of an Event Calculus description given in Definition 7.

Our Event Calculus translation supplies a semantics to mode transition tables that is independent from other parts of the SCR specification. In particular, the translation does not include information about the initial state, and the abductive tool does not rely on such information to check system invariants. The technique described here is therefore also applicable to systems where complete information about the initial configuration of the environment is not available. The abductive tool does not need to use defaults to "fill in" missing initial values for conditions. (Information about the initial state may of course also be represented in the Event Calculus; e.g., *HoldsAt(Off,0)*, so that system invariants may be checked with respect to the initial state separately

3.2.2 Translation of an alternative SCR semantics.

As mentioned in Section 3.1, different semantics for conditioned events have been used in the literature [21], all of which are expressible in our Event Calculus approach. In recent years, one of these semantics has become widely used. According to this semantics, two basic assumptions are made when representing systems specifications as SCR mode transition tables. The first is that events are deterministic, and the second, called the *one input assumption*, is that only one event happens at a time. In this context, the interpretation of an event "@T(C)" conditional on "D" means that "C" is false in the current mode and is changed to true in the new mode, while "D" is assumed to be true in the current mode, without forcing any condition on the particular value of the variable D in the new mode. Similarly for an event "@F(C)" conditional on "D", but with "C" changing truth value from true to false [18]. It is easy to see that this different way of interpreting conditioned events together with the one input assumption gives, in the case of independent monitored variables, the same interpretation of conditioned events as given in Section 3.1. Our approach is also able to express this particular semantics with the following minor modifications.

The assumption of deterministic events is already captured by the (ECD) axioms included in our Event Calculus axiomatisation EC_{Ax}. To express the one input assumption, our axiomatisation EC_{Ax} also needs to include the axiom:

$$\neg\exists\, t, a1, a2[Happens(a1,t) \wedge Happens(a2,t) \wedge a1 \neq a2] \qquad \text{(ECL)}$$

This eliminates the possibility of simultaneous (different) events happening in the environment. Two ground instantiations of this axiom need to be added to the reduced version of our Event Calculus axiomatisation $EC_{Ax}(\mathbf{S})$, given in Definition 7. These are:

$$\neg\exists\, a1, a2[Happens(a1,Sc) \wedge Happens(a2,Sc) \wedge a1 \neq a2] \qquad \text{(ECSL1)}$$

$$\neg\exists\ a1, a2[Happens(a1,Sn) \wedge Happens(a2,Sn) \wedge a1 \neq a2] \quad \textit{(ECSL2)}$$

In this semantics, mode transition tables always include just single (conditioned) events in each transition row. Each of these rows will then be translated into a single *Initiates* and a single *Terminates* rule, as described in section 3.2.1, but with the following minor change. If the entry for a condition C is a "t" then the right-hand-sides of the corresponding *Initiates* and *Terminates* rules will only include the single literal *HoldsAt(C,t)* rather than the pair $HoldsAt(C,t) \wedge \neg Happens(@F(C),t)$. If the entry is a "f" then the right-hand-sides of the *Initiates* and *Terminates* rules will only include the single literal $\neg HoldsAt(C,t)$ rather than the pair $\neg HoldsAt(C,t) \wedge Happens(@T(C),t)$. Propagations of effects of events on other

(internal) variables of the system are expressed in SCR terms by considering such variables to be dependent on other monitored variables. Such dependencies are expressed by means of condition tables. This guarantees consistency with respect to the one input assumption. In a similar way, propagations of effects of events can be formalised in the Event Calculus by several techniques (see for example [37]).

3.2.3 The abductive proof procedure

For the purposes of discussion, let us suppose Table-1 has been translated into an Event Calculus specification EC_{Spec}. The system mode invariants in this particular case are translated into four universally quantified sentences $\forall t.I_1(t),\ldots,\forall t.I_4(t)$ (where each I_i is expressed with standard logical connectives and the *HoldsAt* predicate). In general, there will be *n* such properties, but we always add an additional property $\forall t.I_0(t)$ that simply states (via an exclusive or) that the system is in exactly one mode at any one time. We use the term $\forall t.I(t)$ to stand for $\forall t.I_0(t) \wedge \ldots \wedge \forall t.I_n(t)$.

Thus, for our case study the invariants are (reading "|" as exclusive or):

I_0: *[HoldsAt(Off,t) | HoldsAt(Inactive,t) | HoldsAt(Cruise,t) | HoldsAt(Override,t)]*

I_1: $HoldsAt(Off,t) \equiv \neg HoldsAt(Ignited,t)$

I_2: $HoldsAt(Inactive,t) \rightarrow [HoldsAt(Ignited,t) \wedge [\neg HoldsAt(Running,t) \vee \neg HoldsAt(Activate,t)]]$

I_3: $HoldsAt(Cruise,t) \rightarrow [HoldsAt(Ignited,t) \wedge HoldsAt(Running,t) \wedge \neg HoldsAt(Brake,t)]$

I_4: $HoldsAt(Override,t) \rightarrow [HoldsAt(Ignited,t) \wedge HoldsAt(Running,t)]$

Using the results illustrated in Corollary 1, the abductive analysis is then performed with respect to a reduced version of the Event Calculus specification, which uses a time structure **S** consisting of just two symbolic points *Sc* and *Sn* such that $Sc < Sn$. Different abductive proof procedures and techniques have been developed [30]. Our approach uses a logic programming abductive proof procedure, so deploying a substantial existing body of work in applying abduction to Event Calculus representations [29; 30]. This proof procedure (i.e. procedure to find Δ's) is composed of two phases, an *abductive phase* and a *consistency phase*, that interleave with each other. Each abducible generated during the first phase is temporarily added to a set of abducibles that have already been generated. But this addition is only made permanent if the second phase confirms that the entire new set of abducibles is consistent with the specification and with the given constraints (if any). A detailed description of the abductive proof procedure is given in Section 4.2, together with an illustrative example, whereas background logic programming definitions and related theoretical results are given in the Appendix.

Our abductive procedure attempts to find system behaviors described by the transition table that are counterexamples of the system invariants, by attempting to generate a consistent set

Δ of *HoldsAt* and *Happens* facts (positive or negative literals grounded at *Sc*), such that $D(\mathbf{S}) \cup \{I(Sc)\} \cup \Delta \models \neg I(Sn)$. We can also check the specification against a particular invariant $\forall t.I_i(t)$ by attempting to abduce a Δ such that $D(\mathbf{S}) \cup \{I(Sc\} \cup \Delta \models \neg I_i(Sn)$. Because the abductive procedure is complete (see Theorem 3 in section 4.1), failure to find such a Δ ensures that the table satisfies the invariant(s). If, on the other hand, the tool generates a Δ, this Δ is effectively a pointer to a particular row in the table that is problematic.

For example, when checking the table against the invariant I_3 the tool produces the following Δ:

Δ = {*HoldsAt(Ignited,Sc), HoldsAt(Running,Sc),*
HoldsAt(Toofast,Sc), ¬*HoldsAt(Brake,Sc),*
HoldsAt(Cruise,Sc), ¬*Happens(@F(Ignited),Sc),*
¬*Happens(@F(Running),Sc),*
¬*Happens(@F(Toofast),Sc),*
Happens(@T(Brake),Sc) }

Clearly, this Δ identifies one of the hidden rows of the table in which a "@T(Brake)" event merely results in the system staying in mode "Cruise". The requirements engineer now has a choice: (1) alter the new mode in this (hidden) row so that invariant I_3 is satisfied (in this case the obvious choice is to change the new mode from "Cruise" to "Override", and make this previously hidden row explicit in the table), (2) add an extra invariant that forbids the combination of *HoldsAt* literals in Δ (e.g. add the invariant I_5 = *[HoldsAt(Cruise,t)* $\rightarrow$ ¬*HoldsAt(Toofast,t)]*) or (3) weaken or delete the system invariant (in this case I_3) that has been violated.

This example illustrates all the types of choices for change that will be available when violation of an invariant is detected. Choices such as these will be highly domain-specific and therefore appropriate for the requirements engineer, rather than the tool, to select. After the selected change has been implemented, the tool should be run again, and this process repeated until no more violations of properties are identified (i.e. until the tool fails to generate a Δ). As an indication, a single execution of our abductive reasoning over the invariant I_3, for example, takes XXXX, using a simple unoptimised implementation of the abductive proof procedure.

4. Tool Support

The results illustrated in Section 2 describe the logic foundations of our technique for analysing safety properties. They are based on a classical notion of logical abduction, as given in Definition A2 in the Appendix, and on a notion of counterexample of a safety property as given in Definition 6. They are therefore applicable to any abductive proof procedure or abductive algorithm that is sound and complete with respect to these classical notions. As mentioned in the previous section, our approach uses a logic programming abductive proof procedure based on two phases, *abductive phase* and a *consistency phase*, interleaving with each other. In this section, we describe these two phases in detail and the overall implementation of our abductive framework. The tool is implemented in Prolog, and it uses (i) a logic program conversion of the given (classical logic) Event Calculus specification, based the method described in [29], and (ii) the abductive logic program module described in [28]. Building upon the theoretical results illustrated, it has been

sufficient to implement our abductive analysis approach only with respect to the reduced two timepoints structure **S**. An implementation example of the case study given in Section 3, with some of the abductive analysis results, is also given. Background logic programming definitions and related theoretical results are given in the Appendix.

4.1 The Event Calculus Implementation

The Event Calculus description given in Section 2.1 is based on standard classical logic. The logic program implementation of the Event Calculus used in our tool is instead based on "negation-as-failure" [8], whose underlying semantics differs from the semantics of classical negation [25]. In our logic program tool, the negation (i.e. negation-as-failure) of a predicate, such as *\+(HoldsAt(X,T))*[10] for some fluent X and time point T, is assumed to be true whenever its positive instance *HoldsAt(X,T)* cannot be proven from the specification. This semantic definition of negation-as-failure builds upon a notion of "Closed Word Assumption" (CWA) [31], which implicitly assumes that all models of a given Event Calculus logic program are uniquely defined by only those assertions that are in the program or that can be proved from it. Our classical Event Calculus representation of a given specification (see Definitions 1 and 2), however, is not based on any general CWA[11]. It is possible for a given Event Calculus domain description EC_{Spec} to be incomplete, in the sense that for some fluent constant *F* and time point *T*, neither *HoldsAt(F,T)* nor *¬HoldsAt(F,T)* can be derived from the description. We have overcome this mismatch by adopting the following implementation method [29]. In our tool, negative fluent literals are represented *inside* the *HoldsAt* predicate. In the program translation of our domain description, assertions of the form *¬HoldsAt(F,T)* are represented by the positive literal *HoldsAt(neg(F),T)*, and assertions of the form *HoldsAt(F,T)* are represented by the positive literal *HoldsAt(pos(F),T)*. In this way, negative literals *\+(HoldsAt(F,T))* of our logic programs will correctly denote that the predicate *HoldsAt(F,T)* is not provable from our domain description. A mapping function ϑ, called *implementation mapping*, from the classical logic Event Calculus description into a Prolog program is given in Definition A9 in the Appendix. The logic program implementation of our Event Calculus description is given by a Prolog implementation of the Event Calculus axiomatisation, i.e. $LPEC_{Ax}(\mathbf{S})$, and a Prolog implementation of an Event Calculus system specification $LPEC_{Spec}(\mathbf{S})$. These two implementations are given in Definitions 9 and 10 respectively.

Definition 9. [$LPEC_{Ax}(\mathbf{S})$**]** Let **S** be a time structure consisting of two timepoints *Sc* and *Sn*, with *Sc* < *Sn*, and let ϑ be an implementation mapping. Let $EC_{Ax}(\mathbf{S})$ be a ground Event Calculus axiomatisation. The set $LPEC_{Ax}(\mathbf{S})$ is the Prolog implementation of $EC_{Ax}(\mathbf{S})$ generated by the mapping ϑ, consisting of the following Horn clauses:

clipped(pos(F),sc) ← *happens(A,sc), possiblyTerminates(A,F,sc).* (LPECS1)

clipped(neg(F),sc) ← *happens(A,sc), possiblyInitiates(A,F,sc).* (LPECS2)

holdsAt(pos(F),sn) ← (LPECS3)

[10] Note that in logic programs symbols starting with capital letters denote variables whereas symbols starting with small cases denote constants. The operator "\+" denotes negation-as-failure [21].

[11] We remind the reader that in our Event Calculus descriptions only *Initiates* and *Terminates* clauses are completed but not the *HoldsAt* clauses.

happens(A,sc), initiates(A,F,sc), \+ clipped(pos(F),sc).

holdsAt(neg(F),sn) ← (LPECS4)
happens(A,sc), terminates(A,F,sc), \+ clipped(neg(F),sc).

holdsAt(pos(F),sn) ← initiallyTrue(F) , \+ clipped(pos(F),sc). (LPECS5)

holdsAt(neg(F),sn) ← initiallyFalse(F) , \+ clipped(neg(F),sc). (LPECS6)

□

The implementation of an Event Calculus system specification EC_{Spec}(**S**) is also based on the mapping ϑ, but it is slightly more elaborate than the implementation of EC_{Ax}(**S**). For each *initiates* and *terminates* clause of a given EC_{Spec}(**S**), the implementation includes also a *possiblyInitiates* and *possiblyTerminates* clauses. This is formally described in the following definition.

Definition 10. [*Prolog Specification* $LPEC_{Spec}$(**S**)] Let EC_{Spec}(**S**) be a ground specification and ϑ be an implementation mapping. A Prolog implementation of EC_{Spec}(**S**) based on ϑ, is the set $LPEC_{Spec}$(**S**) consisting of the following Horn clauses:

a. For each *Initiates* clause in EC_{Spec}(**S**), an *initiates* clause of the form
initiates(A,F,Sc)$^{\vartheta}$ ← *Π*$^{\vartheta}$

b. For each *Terminates* clause in EC_{Spec}(**S**) a *terminates* clause of the form
terminates (A,F,Sc)$^{\vartheta}$ ← *Π*$^{\vartheta}$

c. For each *Initiates* clause in EC_{Spec}(**S**), a *possiblyInitiates* clause of the form
possiblyInitiates(a,f,sc)← *Π'*
where *Π'* includes (i) literals *\+ holdsAt(neg(f),t)*, for each positive literal *HoldsAt* in *Π*, (ii) literals *\+ holdsAt(pos(f),t)*, for each negative literal *HoldsAt* in *Π*, (iii) literals *Happens*$^{\vartheta}$, for each (positive and/or negative) literal *Happens* in *Π* .

d. For each *Terminates* clause in EC_{Spec}(**S**), a *possiblyTerminates* clause of the form
possiblyTerminates(a,f,sc)← *Π'*
where *Π'* includes (i) literals *\+ holdsAt(neg(f),t)*, for each positive literal *HoldsAt* in *Π*, (ii) literals *\+ holdsAt(pos(f),t)*, for each negative literal *HoldsAt* in *Π*, (iii) literals *Happens*$^{\vartheta}$, for each (positive and/or negative) literal *Happens* in *Π* . □

An example is given below that corresponds to the logic program implementation of the first row of the SCR specification given in Table 1:

initiates(t(ignited), inactive, sc) ←
[holdsAt(pos(off),sc) ∧
holdsAt(neg(ignited),sc) ∧ happensAt(t(ignited),sc)].
terminates(t(ignited),off,sc) ←
[holdsAt(pos(off),sc) ∧
holdsAt(neg(ignited),sc) ∧ happensAt(t(ignited),sc)].
possiblyInitiates(t(ignited), inactive, sc) ←
[\+ holdsAt(neg(off),sc) ∧
\+ holdsAt(pos(ignited),sc) ∧ happensAt(t(ignited),sc)].

possiblyTerminates(t(ignited),off,sc) ←
[\+ holdsAt(neg(off),sc) ∧
\+ holdsAt(pos(ignited)),sc) ∧ *happensAt(t(ignited),sc)].*

The implementation of safety properties and constraints also uses the same implementation mapping ϑ defined in the Appendix. The logical assumptions that safety properties hold at the current state *Sc* are implemented as integrity constraints. These are added to any existing domain dependent constraint.

Definition 11. [*Prolog safety property* $I(sn)^{\vartheta}$] Let I(*sn*) be a ground safety property with timepoint *sn* and let ϑ be an implementation mapping. Let $I_1(sn)\wedge\ldots\wedge I_n(sn)$ be the conjunction of normal clauses generated from I(sn). The Prolog implementation of the safety property I(sn) over the mapping ϑ is the set $I(sn)^{\vartheta}$ of Prolog clauses $\{I_i(sn)^{\vartheta} \mid i \leq n\}$. Similarly for $\neg I(sn)$. □

Definition 12. [*Integrity Constraints set* $I(sc)^{\vartheta}$ *for safety property*] Let I(*sc*) be an ground safety property with timepoint *sc* and let ϑ be an implementation mapping. Let $I_1(sc)\wedge\ldots\wedge I_n(sc)$ be the conjunction of normal clauses generated from I(*sc*). The set of integrity constraints associated with I(*sc*) is the set $\{IC_1,\ldots,IC_n\}$ where for each $1\leq i\leq n$, IC_i is the denial form of $I_i(sc)$. □

4.2 The Abductive Module

The abductive module is a reasoning engine that takes as input the implementation of an Event Calculus specification with constraints (if any), the implementation of a goal, and generates a set of assertions of a predefined specific form that are consistent with the specification and a given set of constraints, and that allow the given goal to be inferred from the specification. The implementation is based on a standard abductive proof procedure for logic programming [28; 30; 46]. As mentioned in section 3.2.2, this consists of two types of derivations (or phases), the abductive derivation and the consistency derivation, which interleave with each other. A formal definition of these derivations is given below. To clarify some of the standard logic programming notations used in the following definitions, note that the symbol "←" denotes the logic program representation of the classical "if" logical operator, a "normal goal" in logic programming is a clause of the form $\leftarrow L_1, \ldots, L_n$, which means "prove L_1, and … and prove L_n"; an empty goal "□" is a goal with no more literals L_i left to prove. Formal definitions of these concepts and notations are given in the Appendix.

Definition 13. [*Abductive derivation*] Let $<P, Ab_P, IC_P>$ be a logic program abductive framework and let R be a safe selection rule (see Definitions A12 and A13 in the Appendix). *An abductive derivation* from (G_1, Δ_1) to (G_n, Δ_n), with respect to R, is a sequence:

$$(G_1, \Delta_1), \ldots, (G_n, \Delta_n)$$

such that for each $1\leq i\leq n-1$, G_i is a normal goal of the form $\leftarrow L,Q$, where L is the literal selected by R and Q is a possibly empty conjunction of literals. For each $1\leq i\leq n$, Δ_i is a set of Prolog abducibles and G_{i+1} is obtained by applying one of the following rules:

1. If L is not abducible, then
   $G_{i+1} = C$
   $\Delta_{i+1} = \Delta_i$,
   where C is the resolvent of G_i with some clause in P that defines L.

2. If L is abducible and $L \in \Delta_i$, then
   $G_{i+1} = \leftarrow Q$
   $\Delta_{i+1} = \Delta_i$.
3. If L is abducible, $L \notin \Delta_i$ and $\text{Ł} \notin \Delta_i$ and there exists a successful consistency derivation $(L, \Delta_i \cup \{L\}),\ldots,(\emptyset, \Delta')$, then
   $G_{i+1} = \leftarrow Q$
   $\Delta_{i+1} = \Delta'$.

A **successful abductive derivation** is an abductive derivation of the kind $(G_1, \Delta_1),\ldots,(\Box, \Delta_n)$ with $n \geq 1$. □

Definition 14. [*Consistency derivation***]** Let <P, Ab_P, IC_P> be a logic program abductive framework and let R be a safe selection rule. A consistency derivation from $(L, \Delta_1),\ldots, (S_n, \Delta_n)$ is a sequence $(L, \Delta_1), (S_1, \Delta_1),\ldots, (S_n, \Delta_n)$ such that L is an abducible, S_1 is the set of all normal goals of the form $\leftarrow \phi$, obtained by resolving the abducible L with the integrity constraints in $IC_P \cup \{\bot \leftarrow Q, \backslash+ Q \mid Q$ atomic literal in P$\}$, with $\Box \notin S_1$; for each $1 \leq i \leq n\text{-}1$, S_i can be rewritten in the form $\{\leftarrow L,Q\} \cup S_i'$; for each $1 \leq i \leq n\text{-}1$, Δ_i is a set of abducibles; and (S_{i+1},Δ_{i+1}) is obtained according to one of the following rules:

1. If L is not abducible, then
   $S_{i+1}=C' \cup S_i'$
   $\Delta_{i+1}=\Delta_i$
   where C' is the set of all resolvents of rules in P with $\leftarrow L,Q$ on L and $\Box \notin C'$;
2. If L is an abducible and $L \in \Delta_i$ and $Q \neq \Box$, then
   $S_{i+1}=\{\leftarrow Q\} \cup S_i'$
   $\Delta_{i+1}=\Delta_i$
3. If L is an abducible and $\text{Ł} \in \Delta_i$, then
   $S_{i+1}=S_i'$
   $\Delta_{i+1}=\Delta_i$;
4. If L is an abducible, $L \notin \Delta_i$, and $\text{Ł} \notin \Delta_i$, then
   (i) if there exists a successful abductive derivation $(\leftarrow \text{Ł}, \Delta_i),\ldots,(\Box,\Delta')$ then
   $S_{i+1}=S_i'$
   $\Delta_{i+1}=\Delta'$;
   (ii)otherwise, if $Q \neq \Box$, then
   $S_{i+1}=\{\leftarrow Q\} \cup S_i'$
   $\Delta_{i+1}=\Delta_i$.

A **successful consistency derivation** is a consistency derivation of the kind $(L, \Delta_1),\ldots,(\emptyset, \Delta_n)$. □

In simple words, the abductive derivation chooses a single (sub)goal to prove and checks if it is an abducible predicate. If it is not, it keeps performing resolution steps until it finds a sub-goal that is an abducible predicate. When the current sub-goal is an abducible, if it has already been identified then the abductive derivation continues to prove the remaining list of (sub)goals. If it is a new abducible predicate then it is temporarily added to a set of abducible assertions, which have already been generated. The new abducible is then checked for consistency with the specification, the temporary set of generated abducibles, and the

integrity constraints (step 3 of the abductive derivation). If the consistency checking succeeds, the temporary assertion is permanently accepted in the set of abducibles, otherwise discharged. The consistency phase may itself invoke the abductive phase (step 4 in the consistency derivation), in order to verify the (abductive) inference of some intermediate sub-goals. In general this (re)checking for consistency and satisfaction of constraints can be computationally expensive, but because of the particular logic program implementation of our Event Calculus system descriptions (see Definition 8), the computational costs can be further reduced by largely avoiding an exhaustive consistency checking. Any internally consistent, finite collection of *Happens* literals is consistent with our logic program implementation of any Event Calculus system descriptions. Therefore, it is necessary only to check the consistency of candidate abducibles *HoldsAt* literals against the system properties, and this can be done efficiently because both these types of expression are grounded at *Sc*. An abductive solution Δ provided by this procedure can therefore be defined as follows.

Definition 15. [*Prolog abductive solution*] Let <P, Ab_P, IC_P> be a logic program abductive framework and let R be a safe selection rule. Let G be a normal goal. The set Δ of Prolog abducibles is a *Prolog abductive solution* for G if and only if there exists a successful abductive derivation from (G,∅) to (□,Δ). □

To give an example of how the abductive proof procedure works, in the case study described in Section 3 the abductive analysis of the property I_3 (given in section 3.2.2), includes the proof of the sub-goal *holdsAt(pos(brake),sn)*. Part of the abductive derivation for this sub-goal is diagrammatically represented in Figure 2. At the first interaction, the goal *holdsAt(pos(brake),sn)* is not an abducible. So a resolution step is applied with the clause (LPECS3), given in Definition 9, which generates the next (intermediate) list of sub-goals ←*happens(A,sc), initiates(A,pos(brake),sc),* \+ *clipped(pos(brake),sc)*. In order to succeed on the first goal, the abductive derivation has to succeed on each of these three sub-goals. The next immediate step is therefore to prove *happensAt(t(brake),sc)*. Since this is an abducible predicate and it is not in the given specification, it is temporarily added to the set Δ of abducibles already identified (for simplicity we have assumed that at this point of the proof Δ is empty). The consistency derivation (double lined box in Figure 2) is then invoked to show that this addition is consistent with the specification and Δ[12]. For *happensAt(t(brake),sc)*} to be consistent, the abductive derivation should fail to prove \+*happensAt(t(brake),sc)*. Failing to prove \+*happensAt(t(brake),sc)* means, by negation-as-failure, to show that it is possible to prove *happensAt(t(brake),sc)* from the specification together with Δ, which is obviously the case. Hence the consistency checking succeeds and the new Δ = {*happensAt(@t(brake),sc)*} is passed to the external cycle of the abductive derivation for the proof of the other two intermediate sub-goals.

The consistency derivation is slightly more complex than the example given above. This is because our abductive analysis of safety properties has to take into account any domain dependent *integrity constraints* included in the specification. For instance, as shown in Theorem 3, a first class of constraints included in our logic program abductive framework is the set of denials corresponding to the assumption I(*sc)* of the property being true at the current state of the system. Another type of constraints in the case of SCR specifications is

[12] Our case study example did not include any domain dependent constraint.

the assumption that "for any @T(C) event occurring at a time point t, the variable C has to be false at t", which captures the SCR semantics of events. Constraints of this form are for instance implemented by means of denials of the form:

$$\bot \leftarrow happensAt(t(X),T) \wedge holdsAt(neg(X),T).$$

where "*T*" is an arbitrary time point within an hypothetical running behavior of the specified system. Such integrity constraint would then provide a way for constructing only counterexamples that are correct with respect to the semantics of the event-driven specification under consideration. Therefore, if in the example given in Figure 2 the set Δ was not empty but equal to the set {*initiallyTrue(brake)*}, the consistency derivation on the abducible *happensAt(t(brake),sc)* would have failed. Instantiating the variable *T* with *sc*, the tool would have attempted to fail to prove either one of the two conditions *happensAt(t(X),sc)*, *holdsAt(neg(X),sc)*, of the integrity constraint, in order to show that the new abducible is consistent with it. It is easy to see that in each case it would have not succeeded.

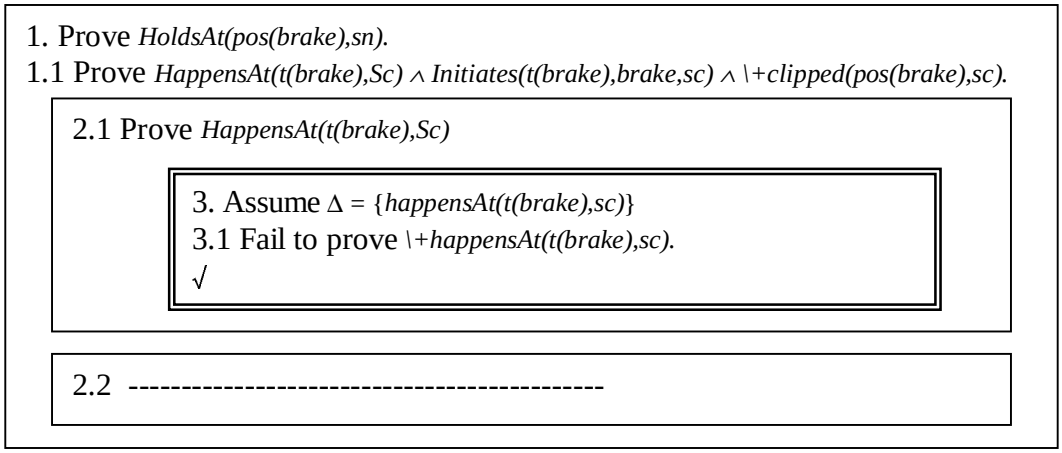

Figure 2: An example of a partial abductive derivation.

The implementation of our abductive framework is based on the property that our logic program Event Calculus is sound and complete with respect to the classical Event Calculus formalism. This result is fundamental to our approach and it is proved in the following two theorems.

Theorem 3. [*Soundness of the Event Calculus abductive implementation*] Let **S** be a time structure consisting of two timepoints *Sc* and *Sn*, with *Sc* < *Sn* and let ϑ be an implementation mapping. Let <D(**S**), Ab, IC> be a classical abductive framework and let I(*Sn*) be a safety property in disjunctive normal form[13]. Let <LPD(**S**), Ab_P, IC_P> be the logic program abductive framework where IC_P includes the Prolog implementation of the constraints IC and the integrity constraint set associated with I(*Sc*). If Δ_P is a Prolog abductive solution for $\neg I(Sn)^{\vartheta}$ with respect to LPD(**S**) then Δ is an abductive solution for $\neg I(Sn)$ with respect to D(**S**)$\cup$I(*Sc*).

[13] A safety property I is in disjunctive normal form if it is of the form $L_1 \vee \ldots \vee L_n$ where L_i is a literal for each $1 \leq i \leq n$. The negation $\neg I$ is therefore a formula of the form $\neg L_1 \wedge \ldots \wedge \neg L_n$.

Proof. Because of the sets of uniqueness-of-name axioms and existence-of-name axioms included in $EC_{Spec}(\mathbf{S})$, the class of models of D(**S**) is isomorphic to a class of Herbrand models. Assume that Δ_P is a Prolog abductive solution for $\neg I(Sn)^{\vartheta}$ with respect to LPD(**S**). By the soundness result stated in Theorem A1 in the Appendix, there exists a stable model $\mathcal{M}(\Delta_P)$ such that $\mathcal{M}(\Delta_P) \Vdash \neg I(Sn)^{\vartheta}$ and $\mathcal{M}(\Delta_P) \Vdash \varphi$, for each $\varphi \in IC_P$. We then reason by contradiction and assume that Δ is not an abductive solution. This means that either D(**S**)$\cup$I(*Sc*)$\cup\Delta$ violate the integrity constraints (or is inconsistent when IC=$\varnothing$) or that D(**S**)$\cup$I(*Sc*)$\cup\Delta \not\models \neg I(Sn)$. In the first case we reach immediately a contradiction since the stable model $\mathcal{M}(\Delta_P)$ is an Herbrand model of LPD(**S**) which satisfies IC_P by definition, and since the program is acyclic[14] there is only one such model. Therefore, there exists only one isomorphic model $\mathcal{M}$ of D(**S**)$\cup$I(*Sc*)$\cup\Delta$, that satisfies the integrity constraints IC. We consider now the case when D(**S**)$\cup$I(*Sc*)$\cup\Delta \not\models \neg I(Sn)$. This second case assumes that all models of D(**S**)$\cup$I(*Sc*)$\cup\Delta$ satisfy IC and that there exists a model $\mathcal{M}$ of D(**S**)$\cup$I(*Sc*)$\cup\Delta$ such that $\mathcal{M} \nVdash \neg I(Sn)$. Let $\mathcal{M}$ be such a model. It is possible to construct a corresponding Herbrand model $\mathcal{M}(\Delta_P)$, which satisfies LPD(**S**)$\cup\Delta_P$ and the integrity constraints IC_P, and such that $\mathcal{M}(\Delta_P) \nVdash \neg I(Sn)^{\vartheta}$. It is possible to show, by construction, that the model $\mathcal{M}(\Delta_P)$ is a minimal Herbrand model for LPD(**S**)$\cup\Delta_P$. Since the logic program LPD(**S**) is acyclic, $\mathcal{M}(\Delta_P)$ is the only such stable model. Hence, $\mathcal{M}(\Delta_P) \Vdash \neg I(Sn)^{\vartheta}$ by hypothesis and $M(\Delta_P) \nVdash \neg I(Sn)^{\vartheta}$ by construction, which is a contradiction. □

Theorem 4.[*Completeness of the Event Calculus abductive implementation*] Let **S** be a time structure consisting of two timepoints *Sc* and *Sn*, with *Sc* < *Sn* and let ϑ be an implementation mapping. Let <D(**S**), Ab, IC> be a classical abductive framework and let I(*Sn*) be a safety property in disjunctive normal form[15]. Let <LPD(**S**), Ab_P, IC_P> be the logic program abductive framework where IC_P includes the Prolog implementation of the constraints IC and the integrity constraint set associated with I(*Sc*). If Δ is an abductive solution for $\neg I(Sn)$ with respect to D(**S**)$\cup$I(*Sc*) then there exists a subset $\Delta'_P \subseteq \Delta_P$ which is a Prolog abductive solution for $\neg I(Sn)^{\vartheta}$ with respect to LPD(**S**).

Proof. Assume that Δ is an abductive solution for $\neg I(Sn)$ with respect to D(**S**)$\cup$I(*Sc*). Therefore D(**S**)$\cup$I(*Sc*)$\cup\Delta \models \neg I(Sn)$ and D(**S**)$\cup$I(*Sc*)$\cup\Delta$ satisfies IC (or is consistent when IC=$\varnothing$). This means that there exists at least one model of D(**S**)$\cup$I(*Sc*)$\cup\Delta$ and that for each model $\mathcal{M}$ of D(**S**)$\cup$I(*Sc*)$\cup\Delta$, $\mathcal{M} \Vdash \neg I(Sn)$ and $\mathbf{M} \Vdash$ IC. We then reason by contradiction and assume that each subset $\Delta'_P \subseteq \Delta_P$ is not a Prolog abductive solution for $\neg I(Sn)^{\vartheta}$ with respect to LPD(**S**). By the completeness result stated in Theorem A1 in Appendix, for all stable models $\mathcal{M}(\Delta_P)$ of the program LPD(**S**)$\cup\Delta_P$ such that $\mathcal{M}(\Delta_P) \Vdash \varphi$, for each $\varphi \in IC_P$, it is the case that $\mathcal{M}(\Delta_P) \nVdash \neg I(Sn)^{\vartheta}$. Following an argument similar to that used in Theorem 3, it is possible to construct from $\mathcal{M}(\Delta_P)$ a corresponding model $\mathcal{M}$ of D(**S**)$\cup$I(*Sc*)$\cup\Delta$ such that, by construction, $\mathcal{M} \nVdash \neg I(Sn)$ and $\mathcal{M} \Vdash$ IC, so getting a contradiction with the initial hypothesis. □

[14] Informally, a logic program is acyclic if it does not include loops. For a formal definition the reader is referred to [3].

[15] A safety property I is in disjunctive normal form if it is of the form $L_1 \vee \ldots \vee L_n$ where L_i is a literal for each $1 \leq i \leq n$. The negation $\neg I$ is therefore a formula of the form $\neg L_1 \wedge \ldots \wedge \neg L_n$.

5. Related Work

A variety of techniques have been developed for analysing requirements specifications. These range from informal but structured inspections [16], to more formal techniques such as those based on model checking or theorem proving [9] or logic-based approaches (e.g. [35; 44; 48]). In general terms, techniques based on model checking facilitate automated analysis of requirements specifications and generation of counterexamples when errors are detected [2; 6; 21]. However, in contrast to our approach they presuppose complete description of the initial state(s) of the system, in order to compute successor states; they require the application of abstraction techniques to reduce the size of the state space; and infinite state systems are abstracted to (possibly equivalent) finite state systems.

For example, in the context of SCR, Heitmeyer *et al.* [21] illustrate how both explicit state model checkers, such as Spin [23], and symbolic model checkers, like SMV [34], can be used to detect safety violations in SCR specifications. The first type of model checking verifies system properties by means of state exploration, whereas the second technique uses reasoning by refutation. Problems related to state explosion are dealt with by the use of sound and complete abstraction techniques that reduce the number of variables to just those that are relevant to the property to be tested [21]. The goal-driven nature of our abductive Event Calculus has the same effect, in that abduction focuses reasoning on goals relevant to the property, and the Event Calculus ensures that this reasoning is at the level of relevant variables (fluents) rather than via the manipulation of entire states.

Both model checking techniques use the standard notion of validity, whereby showing the existence of a model of a specification S, which satisfies the negation of a property P is sufficient to prove that the specification violates (or does not validate) the property. Our abductive Event Calculus, on the other hand, proves the violation of a property by identifying a minimal extension of a given specification S, sufficient to identify a class of semantic models that violate a given property. The focus is therefore on assumptions to add to a given specifications rather then on the construction of a full counter model. As a consequence, the abductive answer can be mapped more easily back to the given requirements specifications than the answer provided by a model checking technique. Other essential differences between our approach and the explicit state model checking technique are that our system (i) can deal with specifications in which information about the initial state is incomplete, and (ii) reports problems in terms of individual mode transitions (which correspond directly to rows in the tables) rather than in terms of particular paths through a state space. The answers of our approach can therefore more easily be interpreted within the representation of the requirements specification. On the other hand, the approach will in certain cases be over-zealous in its reporting of potential errors, in that it will also report problems associated with system states that are in reality unreachable from the initial state (or set of possible initial states) if such information is given elsewhere in the specifications. However, this feature of the system can only result in overly robust, rather then incorrect, specifications, and it does not in practice constitute a limitation of our approach. In fact, if so desired, we can always reapply the abductive procedure, with the current state as a goal, a full rather than a reduced time structure, and information about the initial state, in order to test for reachability.

Formal techniques based on theorem proving [40] can also provide an alternative way of performing analysis on requirements specifications, even for infinite state systems. However, in contrast to our approach they do not provide useful diagnostic information when a verification fails, and may not always terminate.

Recent work by Bharadwaj and Sims [7] uses a hybrid approached based on a combination of specialised decision procedures and model checking for overcoming some of the limitations described above. This approach makes use of induction to prove the safety-critical properties in SCR specifications, and so again states identified as counterexamples may not be reachable.

Of logic-based approaches, the work of van Lamsweerde *et al*. [47; 48; 49] is particularly relevant. This describes a goal-driven approach to requirement engineering in which "obstacles" are parts of a specification that lead to a negated goal. This approach is comparable to ours in that its notion of goals is similar to our notion of safety properties, and its notion of obstacles is analogous to our notion of abducibles. However, the underlying *goal-regression* technique is not completely analogous to our abductive decision procedure. It does use backward reasoning and classical unification in the same way as our abductive derivation uses (logic program) resolution, but no checking for consistency or satisfaction of domain-dependent constraints is performed once an obstacle is generated. In simple terms, from a technical viewpoint, the goal regression technique illustrated in [47; 48; 49] is comparable to step 1 of our abductive derivation phase (see Definition 13), but our approach extends van Lamsweerde's work on obstacle generation in that it provides a *minimal* counterexample to a given property, which is consistent with the specification and with the domain dependent constraints. Moreover, our generation of counterexamples is fully automated where as no implementation of identification of obstacles has yet been reported. In this respect, we believe that our procedure might also be used effectively to support automated identification of obstacles in van Lamsweerde's framework. The only current limitation of our approach with respect to van Lamsweerde's work regards the class of properties that we can analyse using abductive reasoning. Currently, our framework can only analyse safety properties that are universally quantified formulae with no nested temporal quantifiers, whereas van Lamsweerde's approach covers a much wider class of temporal properties (i.e. goals).

The notion of abductive explanations, as described in Section 2.1 is also related to the notion of Dijkstra's weakest-precondition for programming languages [12; 13]. These are pre-conditions for a list of program statements, for which the execution of these statements terminates leaving the system in a state satisfying some desirable post-conditions. However, the analogy is not complete since weakest pre-conditions are used mainly to provide an axiomatic definition of programming language semantics, and a calculus for a formal derivation of programs. This approach is not supported by any automated procedure for the generation of weakest-preconditions from given programs and post-conditions. Our abductive approach could be considered to be a declarative counterpart of weakest-precondition calculus, but also provides an automated decision procedure for the generation of conditions.

Recent work has also demonstrated the applicability of abductive reasoning to software engineering in general. Menzies has proposed the use of abductive techniques for

knowledge-based software engineering, providing an inference procedure for "knowledge-level modeling" that can support prediction, explanation, and planning [35]. Menzies *et al.* shows how abductive techniques can also be used to reason about inconsistent (multi-perspective) requirements specifications to identify their consistent subsets [36]. Satoh has also proposed the use of abduction for handling the evolution of (requirements) specification, by showing that minimal revised specifications can efficiently be computed using logic programming abductive decision procedures [44]. Finally, we have also explored the use of abduction for handling inconsistencies in requirements specifications using QC logic, a paraconsistent logic that allows non-trivial reasoning in the presence of inconsistency [39].

6. Conclusions & Future Work

Our case study illustrates the two characteristics of our approach mentioned in the introduction. It was able to detect violations of safety properties even though the SCR specification used did not include information about an initial state. The counterexamples generated acted as pointers to rows in the mode transition tables and to individual properties that were problematic. It avoids high computational overheads because of the choice of logical representation, and because of our theoretical results, which allow us to reduce the reasoning task before applying the tool. Although the case study used an SCR tabular representation, early indications are that this approach could be much more widely applicable. In particular, we are currently investigating its use in analysing Labeled Transition Systems (LTS) specifications [33].

The abductive approach described in this paper is a (theoretical and practical) foundation for the development of an efficient logic-based method for automated analysis of event-driven requirements specifications. However, a number of issues are still open to further investigation.

First, in our case study, our approach has been applied to an SCR specification composed only of a single mode transition table and some system invariants. But it seems likely that it could also facilitate reasoning about full SCR specifications. A second extension of our approach is to consider requirements specifications of systems with infinite states. As mentioned in Section 2.2, the Event Calculus allows the representation of such types of specifications, but further experimentation is needed. A third direction is to extend our approach to facilitate the analysis of other system properties. This might require the development of different abductive proof procedures appropriately tailored to the specific type of property under consideration. A fourth extension is to allow for non-determinism and concurrency in specifications, since again the Event Calculus makes it straightforward to represent both. As regards this last point, a natural first step is to tailor our approach for analysing LTS specifications, which are prime examples of event-based specifications for non-deterministic and concurrent systems.

Finally, practical tasks for the future include developing automated tools for translating event-based requirements specifications into Event Calculus specifications, and a user-friendly interface for our abductive tool. In the case of SCR specifications, both these tasks are feasible because of the systematic way of generating Event Calculus rules from SCR tables.

Acknowledgements

Many thanks to Connie Heitmeyer, Ramesh Bharadwaj, Axel van Lamsweerde, Didar Zowghi, Gianpaolo Cugola, Jonathan Moffett, and our colleagues in the DSE group at Imperial College for useful feedback and suggestions about earlier drafts of this paper. Thanks also to Peter Grimm and Bruce Labaw of NRL for assisting us in installing SCR* at Imperial College, to Jo Atlee for providing and clarifying the case study, and to Tony Kakas for providing many useful insights about abduction and abductive tools. This work was partially funded by the UK EPSRC projects MISE (GR/L 55964) and VOICI (GR/M 38582).

REFERENCES

s1] Alspaugh, T., et al. (1988). Software Requirements for the A-7E Aircraft, *Technical Report*, Naval Research Laboratory, Washington.

[2] Anderson, R., et al. (1996). Model Checking Large Software Specifications. *ACM SIGSOFT Proceedings of 4th International Symposium on Foundation of Software Engineering*, San Francisco, USA, October 16-18, pp. 156-166

[3] Apt, K. R., and Bol, R. (1994). Logic programming and negation: a survey. *Journal of Logic Programming*, 19: 9-71.

[4] Atlee, J. M., and Gannon, J. (1993). State-Based Model Checking of Event-Driven System Requirements. *IEEE Transactions on Software Engineering*, 19(1): 24-40.

[5] Ben-Ari, M., Manna, Z. and Pneuli, A. (1983). The temporal logic of branching time. *Acta Informatica*, 20: 207-226.

[6] Bharadwaj, R., and Heitmeyer, C. (1997). Model Checking Complete Requirements Specifications Using Abstraction, *Technical Report* No. NRL-7999, Naval Research Laboratory, Washington.

[7] Bharadwaj, R. l., and Sims, S. (2000). Salsa: Combining Solvers with BDD for Automated Invariant Checking. *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, Springer Verlag.

[8] Clark, K. (1978). Negation as Failure. In H. Gallaire, and Minker, J. (Eds.), *Logic and Data Bases* (pp. 293-322). New York: Plenum.

[9] Clarke, M., and Wing, M. (1996). Formal Methods, State of the Art and Future Directions. *ACM Computing Surveys*, 28(4): 626-643.

[10] Console, L., Sapino, M.L., and Theseider Dupre, D. (1994). The Role of Abduction in Database view Updates. *Journal of Intelligent Systems*.

[11] Console, L., Portinale, L., and Theseider Dupre, D. (1996). Using Compiled Knowledge to Guide and Focus Abductive Diagnosis. *IEEE Transaction on Knowledge and Data Engineering*, 8(5): 690-706.

[12] Dijkstra, E. W. (1971). Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1: 115-138.

[13] Dijkstra, E. W. (1975). Guarded commands, non-determinacy and a calculus for the derivation of programs. *Proceedings of International Conference on Reliable Software*, Los Angeles, CA., 2, pp. 2-13

[14] Easterbrook, S., and Callahan, J. (1998). Formal Methods for Verification and Validation of Partial Specifications. *Journal of Systems and Software*, 40(3).

[15] Esghsi, K. (1988). Abductive Planning with the Event Calculus. *Proceedings of International Conference on Artificial Intelligence*, 1, pp. 3-8

[16] Gilb, T., and Graham, D. (1993). *Software Inspection*. Addison-Wesley.

[17] Heimdahl, M. P. E., and Leveson, N.G. (1996). Completeness and Consistency in Hierarchical State-based Requirements. *IEEE Transaction on Software Engineering*, 22(6): 363-377.

[18] Heitmeyer, C. L., Labaw, B., and Kiskis, D. (1995). Consistency Checking of SCR-style Requirements Specifications. *Second International Symposium on Requirements Engineering*, York, pp. 27-29, IEEE Publisher.

[19] Heitmeyer, C. L., Jeffords, R.D., and Labaw, B.G. (1996). Automated Consistency Checking of Requirements Specifications. *ACM Transaction of Software Engineering and Methodology*, 5(3): 231-261.

[20] Heitmeyer, C. L., et al. (1998). Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications. *IEEE Transaction on Software Engineering*, 24(11): 927-947.

[21] Hogger, C. (1990). *Essentials of Logic Programming*. Clarendon Press, Oxford.

[22] Holzmann, G. J. (1997). The Model Checker SPIN. *IEEE Transaction on Software Engineering*, 23(5): 279-295.

[23] Inoue, K., and Sakama, C. (1995). Abductive Framework for Non-monotonic Theory Change. *International Joint Conference on Artificial Intelligence*, 1, pp. 204-210

[24] Kakas, A., and Mancarella, P. (1990). Generalised Stable Models: a Semantics for Abduction. *Proceedings of the 9th European Conference on Artificial Intelligence (ECAI'90)*, Stockolm, pp. 385-391.

[25] Kakas, A. C., and Mancarella, P. (1990). Database Updates Through Abduction. *Proceedings of 16th International Conference on Very Large Database*, Brisbane, Australia

[26] Kakas, A. C., and Mancarella, P. (1990). Generalised Stable Models: a semantics for abduction. *Proceedings of the European Conference on Artificial Intelligence (ECAI90)*, Stockolm, pp. 385-391, Pitman.

[27] Kakas, A. C., and Michael, A. (1995). Integrating Abductive and Constraint Logic Programming. *Proceedings of 12th International Conference on Logic Programming*, Tokyo

[28] Kakas, A. C., and Miller R. (1997). A Simple Declarative Language for Describing Narratives with Actions. *Journal of Logic Programming, Special issue on Reasoning about Actions and Events*, 31(1-3): 157-200.

[29] Kakas, A. C., Kowalski, R.A., and Toni, F. (1998). The Role of Abduction in Logic Programming. In D. M. Gabbay, Hogger, C.J., and Robinson, J.A. (Eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming* (pp. 235-324). Oxford University Press.

[30] Kowalski, R. (1979). *Logic for Problem Solving*. The Computer Science Library.

[31] Kowalski, R. A., and Sergot, M.J. (1986). A Logic-Based Calculus of Events. *New Generation Computing*, 4: 67-95.

[32] Magee, J., and Kramer, J. (1999). *Concurrency: State Models & Java Programs*. John Wiley & Sons Ltd.

[33] McMillian, K. L. (1993). *Symbolic Model Checking*. Kluwer Academic Publishers.

[34] Menzies, T. (1996). Applications of Abduction: Knowledge Level Modeling. *International Journal of Human Computer Studies*, 45: 305-355.

[35] Miller, R., and Shanahan, M. (To appear). The Event Calculus in Classical Logic. In *Journal of Electronic Transactions on Artificial Intelligence - Alternative Axiomatisations* Linköping University Electronic Press, http://www.ida.liu.se/ext/etai/.

[36] Miller, S. (1998). Specifying the mode logic of a Flight Guidance System in CoRE and SCR. *Proceedings of 2nd Workshop of Formal Methods in Software Practice*

[37] Nuseibeh, B., and Russo, A. (1999). Using Abduction to Evolve Inconsistent Requirements Specifications. *Australian Journal of Information Systems, Special Issue on Requirements Engineering. ISSN:1039-7841*: 118-130.

[38] Owre, S., Rushby, J. Shankar, N., and von Henke, F. (1995). Formal verification for fault-tolerant architecture: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2): 107-125.

[39] Parnas, D. L., and Madey, J. (1995). Functional Documentation for Computer Systems, *Technical Report* No. CRL 309, McMaster University.

[40] Pneuli, A. A Temporal Logic of Concurrent Programs. *Theoretical Computer Science*, 13: 45-60.

[41] Russo, A., and Nuseibeh, B. (2000). On the Use of Logical Abduction in Software Engineering. In S. K. Chang (Eds.), *(To appear in) Handbook in Software Engineering and Knowledge Engineering* World Scientific Publishing Corporation.

[42] Satoh, K. (1998). Computing Minimal Revised Logical Specification by Abduction. *Proceedings of International Workshop on the Principles of Software Evolution*, pp. 177 - 182.

[43] Shanahan, M. (1997). *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press.

[44] Toni, F. (1995). A semantics for the Kakas-Mancarella procedure for abductive logic programming. *Proceedings of the logic programming workshop GULP'95*, pp. 231-242

[45] van Lamsweerde, A., and Letier, E. (1998). Integrating Obstacles in Goals-Driven Requirements Engineering. *Proceedings ICSE'98 - 20th International Conference on Software Engineering.*, Kyoto, IEEE-ACM.

[46] van Lamsweerde, A., Darimont, R., and Letier, E. (1998). Managing Conflicts in Goal-Driven Requirement Engineering. *IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development*, November-December 1999.

[47] van Lamsweerde, A., and Letier, E. (2000). Handling Obstacles in Goal-Oriented Requirements Engineering. *IEEE Transactions on Software Engineering, Special Issue on Exception Handling*, 26.

Appendix.

This appendix includes additional technical proofs and definitions used in the paper. These are structured by the sections of the paper.

Our approach

Definition A1. [*Constraint*] Let τ be a timepoint term. A *constraint* is a formulae without quantifiers, whose only predicate is *HoldsAt*, whose fluent terms are ground, and whose only timepoint terms is τ. □

Definition A2. [*Abductive framework*] The *abductive framework* is the triple:

$$<D, Ab, C>$$

where D is a domain description given by the set EC_{Ax} and a specification EC_{Spec}, Ab is the set of abductive predicates and C is a (possibly empty) finite set of constraints. □

Definition A3. [*Model***]** Let D be a domain description given by the set EC_{Ax} and a specification EC_{Spec}. A model $\mathcal{M}$ is a classical interpretation such that $\mathcal{M} \Vdash D$. □

Definition A4. [*Agreeing models***]** Let D be a domain description given by the set EC_{Ax} and a specification EC_{Spec}. Let $\mathcal{M}$ and $\mathcal{M}'$ be two models of D. Then, $\mathcal{M}$ and $\mathcal{M}'$ *agree up to* a given timepoint $n \geq 0$ if for each timepoint term $T \leq n$, ground event A and fluent F,

$$\mathcal{M} \Vdash HoldsAt(F,T) \text{ iff } \mathcal{M}' \Vdash HoldsAt(F,T)$$
$$\mathcal{M} \Vdash Happens(A,T) \text{ iff } \mathcal{M}' \Vdash Happens(A,T)$$

□

Lemma A1. [*Deterministic lemma*] Let D be a domain description given by the set EC_{Ax} and a specification EC_{Spec}, and let $\mathcal{M}$ and $\mathcal{M}'$ be two models of D, which agree up to some time point n. Then, for all ground fluents F, $\mathcal{M} \Vdash HoldsAt(F,n+1)$ if and only if $\mathcal{M}' \Vdash HoldsAt(F,n+1)$.

Proof. Note that because of the deterministic axiom (ECD), the conditions of axioms (EC3)-(EC6) are mutual exclusive. Therefore, for each event that happens at the time point n, which initiates some fluent F, the two models will both satisfy $HoldsAt(F,n+1)$; for each event that happens at the time point n, which terminates some fluent F, the two models will both satisfy $\neg HoldsAt(F,n+1)$; for each event that happens at the time point n and which does neither initiate nor terminate a fluent, the two models will both satisfy $HoldsAt(F,n+1)$ or $\neg HoldsAt(F,n+1)$ for each unaffected fluent F for which $\mathcal{M} \Vdash HoldsAt(F,n)$ and $\mathcal{M} \Vdash \neg HoldsAt(F,n)$ respectively. □

Corollary A1. Let D be a domain description given by the set EC_{Ax} and a specification EC_{Spec}, and let $\mathcal{M}$ and $\mathcal{M}'$ be two models of D, which agree up to some timepoint n. Then, for each ground fluent F, $\mathcal{M} \models \forall t.HoldsAt(F,t)$ if and only if $\mathcal{M}' \models \forall t.HoldsAt(F,t)$. □

Lemma A2. Let D be a domain description given by the set EC_{Ax} and a specification EC_{Spec}. Let $\forall t.I(t)$ be a safety property. Let **S** be a time structure consisting of two timepoints Sc and Sn, with $Sc < Sn$. Then $D \models \forall t.I(t)$ if and only if $D \models I(0)$ and $D \cup \{I(Sc)\} \models I(Sn)$.

Proof. We consider first the "if" part. We assume the time structure underlying D to be isomorphic to the set of natural numbers $\mathbb{N}$. Each timepoint t is therefore interpreted as a natural number. To show that $D \models \forall t.I(t)$ we use induction over $\mathbb{N}$. The base case, i.e.

D⊨I(*0*), is given by the hypothesis. The inductive hypothesis states that for a given natural number *n*, D ⊨ I(*n*). We want to show that D⊨I(*n+1*). Note that the time structure **S** is isomorphic to the structure **{***n***,** *n+1***}.** We reason by contradiction. We assume that D⊭I(*n+1*). Thus, there exists a model $\mathcal{M}$ of D such that $\mathcal{M}$⊮I(*n+1*). By inductive hypothesis, $\mathcal{M}$⊩I(*n*) and therefore $\mathcal{M}$⊩D∪{I(*n*)}. This is equivalent to say that there exists a model $\mathcal{M}$ such that $\mathcal{M}$⊩D∪{I(*Sc*)} and $\mathcal{M}$⊮I(*Sn*), which is in contradiction with the hypothesis. We consider now the "only-if" part. By hypothesis D⊨∀*t*.I*(t)*. This trivially implies that D⊨I(*0*), and that for all n > 0, D⊨I(*n*). Note that the time structure **S** is isomorphic to a structure **{***n***,** *n+1***},** for any arbitrary natural number n**.** We reason by contradiction. We assume that D∪{I(*Sc*)}⊭I(*Sn*). Therefore there exists a model $\mathcal{M}$ of D∪{I(*Sc*)} such that $\mathcal{M}$⊮I(*Sn*). By inductive hypothesis $\mathcal{M}$ is also a model of D. Hence, $\mathcal{M}$⊩D and $\mathcal{M}$⊮I(*Sn*) which is in contradiction with the hypothesis. □

Lemma A3. Let D be a domain description given by the set EC_{Ax} and a specification EC_{Spec}. Let ∀*t.I(t)* be a safety property. Let **S** be a time structure consisting of two timepoints *Sc* and *Sn*, with *Sc* < *Sn*. Let D(**S**) be the ground domain description given by the set EC_{Ax}(**S**) and the ground specification EC_{Spec}(**S**) corresponding to EC_{Spec}. Hence, D∪{I(*Sc*)}⊨I(*Sn*) if and only if D(**S**)∪{I(*Sc*)}⊨I(*Sn*).

Proof. We prove first the "if" part. Note that it is easy to show that D⊨D(**S**)[16]. This implies that D∪{I(*Sc*)} ⊨ D(**S**)∪{I(*Sc*)}. By hypothesis D(**S**)∪{I(*Sc*)}⊨I(*Sn*). Hence by transitivity, D∪{I(*Sc*)}⊨I(*Sn*). The "only if" part is proved by contradiction. We assume that D(**S**)∪{I(*Sc*)}⊭I(*Sn*). This implies that there exists a model $\mathcal{M}$ of D(**S**)∪{I(*Sc*)} such that $\mathcal{M}$⊮I(*Sn*). The model $\mathcal{M}$ can be extended to a model $\mathcal{M}'$ over the time structure ℕ of natural numbers such that $\mathcal{M}'$⊮I(*Sn*)[17]. Hence, D∪{I(*Sc*)}⊮I(*Sn*), which is in contradiction with the hypothesis. □

Tool support

Definition A5. [*Normal clause*] A *normal clause* is an expression of the form H← L_1,…L_n where H is a positive literal, called *head*, and ← L_1,…L_n is a (possibly empty) conjunction of literals (positive and/or negative) called *body*. □

Definition A6. [*Normal goal*] A *normal goal* is a normal clause of the form ← L_1,…L_n, (with an empty head). A normal goal with empty body, i.e. n = 0, is an empty goal and it is denoted with the symbol □. □

Definition A7. [*Normal logic program*] A normal logic program P is a finite set of normal clauses. □

---

[16] For each model $\mathcal{M}$ of D, consider the assignment that maps t1 to Sc and t2 to Sn. The axioms (EC1) and (EC2) are equilvalent to (ECS1) and (ECS2) with t equal to Sc; the axioms (EC3)-(EC6) are equivalent to (ECS3)-(ECS6) and the axiom (ECD) is equivalent to the two axioms (ECSD1) and (ECSD2). Therefore, $\mathcal{M}$ is also a model of D(**S**).

[17] To construct $\mathcal{M}'$ consider the values 0 and 1 to be isomorphic to Sc and Sn. Then make $\mathcal{M}'$ agrees with $\mathcal{M}$ on the interpretation of *HoldsAt*, *Happens*, *Initiates*, *Terminates*, at the timepoint *0* and Clipped and Declipped between *0* and *1*. Assume that for any timepoint t ≥ *1* nothing happens in the model $\mathcal{M}'$, and define the interpretation of the predicates over the rest of the natural numbers in such a way that the axioms of EC_{Ax} are satisfied. This construction will guarantee $\mathcal{M}'$ to be a model of D(**S**)∪{I(*Sc*)} and to be such that $\mathcal{M}'$⊮I(*Sn*).

Definition A8. [*Prolog abducible predicates set* Ab_P] The set Ab_P of Prolog abducible predicates is the set of predicate symbols *initiallyTrue*, *initiallyFalse*, *Happens*. □

Definition A9. [*Implementation mapping***]** Let D(**S**) be a domain description given by the set EC_{Ax}(**S**) and a ground specification EC_{Spec}(**S**). An implementation mapping is a function ϑ from the set of (ground) literals used in D(**S**) to a set of Prolog literals, defined as follows:

i) $HoldsAt(F,T)^{\vartheta} = holdsAt(pos(f),t)$.

ii) $[\neg HoldsAt(F,T)]^{\vartheta} = holdsAt(neg(f),t)$.

iii) $[(\neg)Happens(A,T)]^{\vartheta} = (\backslash+)\ happens(a,t)$.

iv) $[(\neg)Clipped(T1,F,T2)]^{\vartheta} = (\backslash+)\ clipped(t1,pos(f),t2)$.

v) $[(\neg)Declipped(T1,F,T2)]^{\vartheta} = (\backslash+)\ clipped(t1,neg(f),t2)$.

vi) $[(\neg)Terminates(A,F,T)]^{\vartheta} = (\backslash+)\ terminates(a,f,t)$.

vii) $[(\neg)Initiates(A,F,T)]^{\vartheta} = (\backslash+)\ initiates(a,f,t)$.

viii) $[\Gamma 1 \wedge \ldots \wedge \Gamma n]^{\vartheta} = \Gamma 1^{\vartheta},\ldots,\Gamma n^{\vartheta}$, where Γi is a (ground) literal used in D(**S**).

□

Definition A10. [*Prolog abducibles*] Let Ab_P be a Prolog abducible predicates set. A *Prolog abducible* is a literal (positive or negative) constructed from Ab_P. Let Γ be an abducible and let ϑ be an implementation mapping. The Prolog abducible Γ_P associated with Γ is defined as follows. If Γ is of the form *HoldsAt(F,0)* then Γ_P is the literal *initiallyTrue(f)*; if Γ is of the form $\neg$*HoldsAt(F,0)* then Γ_P is the literal *initiallyFalse(f)*; if Γ is of the form *Happens(A,T)* then $\Gamma_P = \Gamma^{\vartheta}$. Δ_P denotes a (possibly infinite) set of Prolog abducibles associated with a given set Δ of abducibles. □

Definition A11. [*Logic program abductive framework*] A *logic program abductive framework* is a triple <P, Ab_P, IC_P> where P is a normal logic program, Ab_P is a Prolog abducible predicates set, whose predicates are not defined in P, and IC_P is a set of integrity constraints in denial form, i.e. $\bot \leftarrow L_1,\ldots,L_n$ where each L_i is a literal and at least one of these literals is an abducible predicate or the negation of an abducible predicate. □

Notation. Let L be a literal. Then the contrapositive Ł is the opposite in sign of the literal L.

Definition A12. [*Safe selection rule* R] Let $\leftarrow L_1,\ldots L_n$ be a *normal goal* (i.e. $L_1,\ldots L_n$ is a sfinite conjunction of literals). A *safe selection rule* R is a (partial) function, which applied to a normal goal selects a conjunct L_i, $1 \le i \le n$, only if it is ground. □

Definition A13. [*Stable model*] Let P be a normal logic program and let M be an Herbrand interpretation of P [22]. M is a *stable model* of P if and only if M is equal to the minimal Herbrand model of Π^M, where Π^M is the set of ground definite[18] Horn clauses:

$\Pi^M = \{H \leftarrow B_1,..B_k \mid H \leftarrow B_1,..B_k,\neg L_1,..,\neg L_m$ is a clause in ground(P), and

$L_i \notin M$ for each $1 \le i \le m\}$. □

Definition A14. [*Logic program abductive solution***]** Let <P, Ab_P, IC_P> be a logic program abductive framework. Let Q be a goal. Then Δ is a logic program abductive solution for Q if

[18] A definite clause is a clause of the form $H \leftarrow B_1,..B_k$, where H is an atomic literal and for each $1 \le i \le k$ B_i is also a positive literal.

and only if there exists a stable model $M(\Delta)$ of $P \cup \{H \leftarrow | H \in \Delta\}$ such that $M(\Delta) \Vdash Q$ and $M(\Delta) \Vdash \varphi$ for each $\varphi \in IC_P$. □

Theorem A1. [Soundness & Completeness [27]] Let $<P, Ab_P, IC_P>$ be a logic program abductive framework and let R be a safe selection rule. Let G be a normal goal. If there is a Prolog abductive solution Δ for G then there exists a stable model $M(\Delta)$ of $P \cup \{H \leftarrow | H \in \Delta\}$ such that $M(\Delta) \Vdash G$ and $M(\Delta) \Vdash \varphi$ for each $\varphi \in IC_P$ (Soundness). Let Δ be a logic program abductive solution then there exists a subset $\Delta' \subseteq \Delta$ such that Δ' is a Prolog abductive solution for G (Completeness). □