An Abductive Approach for Handling Inconsistencies in SCR Specifications

Alessandra Russo[*] **Rob Miller**[#] **Bashar Nuseibeh**[*] **Jeff Kramer**[*]

[*] Department of Computing
Imperial College
180, Queens' Gate, London
SW7 2BZ, United Kingdom
{ar3, ban, jk}@doc.ic.ac.uk

[#] S.L.A.I.S.
University College London
Gower Street
London WC1E 6BT
rsm@ucl.ac.uk

ABSTRACT

We present a formal approach for handling inconsistencies in Software Cost Reduction (SCR) specifications. The approach uses an event-based logic, called the *Event Calculus*, to represent SCR mode transition tables. Building on this formalism, the approach provides an abductive reasoning mechanism that enables the analysis of inconsistencies between SCR mode transition tables and global requirements (invariants), and the identification of alternative changes that would resolve such inconsistencies. Changes include addition of new invariants, refinement of existing invariants, and changes on conditions of mode transitions. The methodology is widely applicable, in particular to systems embedded in complex environments whose initial conditions cannot be completely predicted. A case study of an automobile cruise control system is used to illustrate our approach. The technique described is implemented using existing tools for abductive logic programming.

1 INTRODUCTION

Handling inconsistencies in requirements specifications is a critical activity in the software development process. Inconsistent specifications can lead to system failures, and defects detected late in development can be more expensive to correct than specification inconsistencies discovered early. Therefore, techniques for the detection and resolution of inconsistencies in requirements specifications can be crucial for successful development of software systems.

A variety of techniques have been developed for checking specifications for inconsistencies. These range from informal but structured inspections [11], to more formal techniques such as those based on model checking or theorem proving [6]. While many of these approaches provide rigorous, and often automated, analysis of software specifications to reveal inconsistencies, they often also do not support the analyst in handling these inconsistencies after they have been discovered.

This paper presents an approach to support inconsistency handling of requirements specifications, focusing in particular on requirements expressed as Software Cost Reduction (SCR) tabular specifications [13]. The choice of SCR is a pragmatic one – it has been proven useful for expressing the requirements of a wide range of large-scale real-world systems, and for checking the consistency and validity of such requirements [1; 9; 13; 27]. The approach is supported by a suite of automated tools for consistency checking and simulation, and is complemented by model checking tools and techniques for checking specification invariants [3; 4; 15].

However, while SCR provides a host of tools for analysing requirements specifications, once a violation of an invariant has been detected, the identification of (possible) changes to perform on the specification is still primarily a human task. Inconsistencies are reported to the requirement engineer, who must then investigate ways of changing the specifications to fix the inconsistencies.

To address this issue, we have developed an approach based on *abduction* [19], to suggest ways of changing an SCR specification, given the satisfaction of an invariant as a goal. In Artificial Intelligence (AI), abduction is used as one of the three fundamental modes of reasoning, the others being deduction and induction. Abductive techniques are able to generate "explanations" for a given property ("goal") to be satisfied in a specification. These techniques have been shown to be particularly suitable for addressing problems such as diagnosis [7], planning [10], theory update [8; 18; 20], and knowledge-based software development [25]. Of particular interest to us in this paper are abductive techniques that allow reasoning about specifications expressed in event-based formalisms that can be mapped to and from SCR specifications. One such formalism is the *Event Calculus* [23] based on classical logic. Abductive Event Calculus techniques provide "explanations" in terms of events and domain properties, and can be used to (automatically) identify instances of system behavior that are inconsistent with given system invariants.

The paper describes and demonstrates our approach of using an abductive Event Calculus technique for reasoning about discrepancies between required systems properties (invariants) and SCR tabular specifications. In this paper, we only consider SCR specifications composed of mode transition tables and system invariants[1]. An overview of our approach is shown schematically in Figure-1.

SCR mode transition tables and invariants are both

[1] The application of the approach to full SCR specifications is discussed in section 5.

expressed in the Event Calculus language. A table denotes a "domain-description", while invariants denote the goals that such a domain-description should satisfy. Given a goal and an SCR table, our abductive technique identifies whether the goal is satisfied by the specification. This consists of checking if there are possible instances of system behavior (i.e. possible system configurations and input events) that would imply the negation of the goal. These instances of system behavior (also called "explanations") would thus be inconsistent with the system invariant. Due to the completeness of our approach, SCR specifications would instead be consistent with system invariants if no such explanations can be found. Explanations identified by our abductive technique can be used to determine possible changes. Changes include the addition of new invariants, the refinement of existing invariants, and changes to mode transitions. Heuristics can then be used to prune the set of possible changes to a smaller set of proposed changes. Of course, performing a change on a specification often initiates a sequence of additional related changes, and so the approach must then be deployed iteratively, considering either a new table or a new invariant, or both.

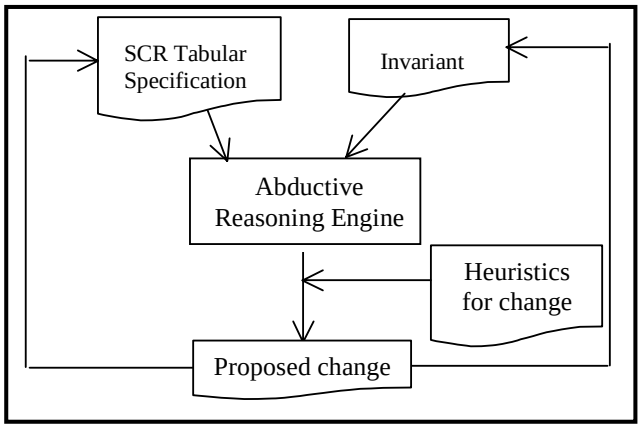

Figure 1: The Abductive Event Calculus Approach

Section 2 of the paper reviews SCR, focusing on mode transition tables and invariants. Section 3 reviews the Event Calculus and illustrates the kinds of reasoning that it can provide. A review of abductive reasoning in the event calculus is also given. Section 4 provides a case study based description of our approach. The domain of the case study is taken from [4]. It shows how SCR tables and invariants can be mapped into an event-based specification, the kind of explanations that our abductive technique is able to provide, and the heuristics that can be used to identify possible changes. The tool we used to implement our approach is also described. The paper concludes with a discussion of lessons learned from our case study, and a summary of related and future work.

2 SCR SPECIFICATIONS.

SCR is a formal requirements engineering method that facilitates the tabular definition of requirements specifications and their analysis [13; 14; 15]. The method is based on Parnas's "Four Variable Model" [28], which describes a required system's behavior as a set of mathematical relations between four types of variables – monitored and controlled variables, and input and output data items. *Monitored variables* are environmental entities that influence the system behavior, and *controlled variables* are environmental entities that the system controls. Systems are assumed to include an input device to measure monitored variables and map them into software *input data items*, and an output device to use the software *output data items* for setting the controlled variables. Four main relations characterize this model – REQ, NAT, IN, and OUT. The relations IN and OUT specify the accuracy with which the input device measures the monitored variables, and the output device sets the controlled variables. NAT describes natural constraints on the system behavior such as constraints imposed by physical laws, while REQ defines the system requirements in terms of relations between monitored and controlled variables. SCR specifications describe the NAT and REQ relations.

The components of an SCR specification are monitored and controlled variables, mode classes and terms. A *mode class* partitions the monitored environment's state space into modes, whereas *terms* are "internal" variables calculated and used within the system. Each mode is a collection of system states sharing common properties on monitored variables. Variables are of different types – Boolean, integers, real number, and enumerated domains. Non-boolean variables can always be reduced to Boolean variables, i.e. predicates defined over their values. For instance, a predicate TempTooHot can be defined to indicate that a monitored variable RoomTemp, over the real numbers, has a value RoomTemp>(SetTemp+ 3°C). These predicates are called *conditions* and are defined over single system states. An *event* occurs when a system component (i.e. a monitored or controlled variable, mode class or term) changes value. Special events are *monitored events*, when monitored variables change value, and *conditioned events*, when an event occurs while a specified condition is true.

SCR specifications use three special tables, mode transition, event and condition tables. A condition table describes a controlled variable or a term as a function of a mode and a condition; an event table describes a controlled variable or a term as a function of a mode and an event.

Mode transition tables describe a mode as a function of another mode and an event. In addition to these tables, SCR specifications also include *assertions*, properties of the environment, and *invariants (goals)*, properties that are required to hold in the system. As mentioned in section 1, for the purpose of this paper, we will regard SCR specifications as consisting simply of a mode transition table and a list of system invariants. These two components are therefore described in more detail in the rest of this section. For a more detailed description of the SCR

approach the reader is referred to [5; 13; 14].

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. An example of a mode transition table, taken from [4], is given in Table-1 for an automobile cruise control system. Note that the table already reflects basic properties on monitored variables. For example, the two transitions from "Inactive" to "Cruise" take into account the environmental property that a cruise control lever is in exactly one of the three positions "Activate", "Deactivate" and "Resume" at any state. So whenever Activate is becoming true, it can only be the case that Deactivate is becoming false or Resume is becoming 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 (condition on) 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 a temperature control system, the event @T(TempTooHot) would denote that the temperature in a room has changed from not being too hot (i.e. RoomTemp ≤ (SetTemp + 3°C)) to be too hot (i.e. RoomTemp > (SetTemp + 3°C)). 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) while Running is false. Different semantics have been used for conditioned events [13]. These are all equally expressible in our approach. In this paper, we will adopt the following interpretation. An event @T(C) conditioned to (a condition) 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. Similarly for an event @F(C) conditioned to 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 condition 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. Two main features need to be considered when reading an SCR table. The first one is that 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. The second feature to

consider and which makes tables concise, is the non-specification of transitions between identical modes. Mode transition tables are basically functions that define for each current mode and each combination of conditions' values, a new mode of the system. This new mode may or may not be equal to the current mode. These functions uniquely "model" the system requirements. However in specifying real system behavior only the transitions between current and new modes that are different are explicitly represented in SCR tables; the other (transitions between identical current and new modes) are implicitly assumed (i.e. the system stays in the same mode) and "hidden" away from the table. Models of SCR tables (i.e. full extended SCR mode transition tables) thus include, for each possible combination of pairs of modes, a number of transitions (rows) given by all possible combinations of the four condition' entries for all the conditions used in the table. Therefore, both hidden rows and dashes need to be taken into account when analysing invariants with respect to the concise version of an SCR mode transition table. As discussed in section 4, both are indeed often causes for mismatch between SCR tables and invariants, as they may hide away system behaviors that violate system invariants.

Goals as Invariants. Invariants are properties (specification assertions) of the system behavior regarding mode classes, which ought to be satisfied by the system specification. Considering an automobile cruise control system, an example of 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. The example property given here is called a *mode invariant*. 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. Different approaches have been developed for checking invariants of SCR specifications. These are mainly based on model checking techniques, such as state-based model checking [4; 16] using Spin [17], and symbolic model checking using SMV [24]. This paper provides an alternative logic-based approach for detecting violation of invariants and for generating explanations. The latter are pointers to the rows in the (extended) table that violate the invariant. A case study investigation of our approach for the cruise control system given in Table-1 has highlighted the fact that such rows can be hidden mode transitions. This shows that it is often the implicit assumptions used by the requirements engineer that cause inconsistencies. This is discussed in more detail in Section 4. Extensions of the approach to full SCR specifications (i.e. event and condition tables, and global requirements properties that are not necessarily mode invariants) are discussed in Section 5, together with a comparative analysis of our approach with respect to model checking techniques A brief overview of the event-based formalism used in our approach, and of the basic notion of abductive reasoning, is given in the next section.

3 THE EVENT CALCULUS

The Event Calculus [23; 26] is a logic-based formalism for representing and reasoning about dynamic systems. In contrast to pure state-transition based formalisms, its ontology includes an explicit structure of time, which is independent of any (sequence of) events or actions under consideration. As we shall see, this characteristic makes it straightforward to model concurrent, possibly non-deterministic, event-driven systems. In this paper, we consider only deterministic systems, since SCR specifications assume deterministic system behaviors. However, the approach is equally applicable to non-deterministic and concurrent event-driven system specifications.

For our purposes, a simple classical logic form [26] of the Event Calculus is sufficient, 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, 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* occurs at time-point *t*, *Initiates(a,f,t)* (resp. *Terminates(a,f,t)*) means that event *a* causes fluent *f* to be true (resp. false) immediately after *t*, and *HoldsAt(f,t)* indicates that fluent *f* is true at *t*. So, for example, to indicate that events *A1* and *A2* occur concurrently at time-point *T4* it is sufficient to assert $[Happens(A1,T4) \wedge Happens(A2,T4)]$.

Specifications as Axiomatisations. Every Event Calculus specification (i.e. 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 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:

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

(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*:

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

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:

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

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

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

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

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

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

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:

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

(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 (see [26] for details). 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[2], i.e. it allows us to assume that the only effects of events are those explicitly described.

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 the present 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.

Efficient Abductive Reasoning in the Event Calculus. In the context of this paper, we wish to take an Event Calculus specification such as described above and use it to test system invariants. In the language of the Event Calculus these are expressions involving *HoldsAt* and universally quantified over time, such as:

$$\forall t.[HoldsAt(SwitchAOn,t) \vee \neg HoldsAt(LightOn,t)]$$

It is (potentially) computationally expensive to demonstrate the truth of such sentences by standard (deductive or abductive) theorem-proving techniques. However, fortunately we can reduce this inference task to a simpler one as stated by the following theorem.

Theorem 1. Let $EC(\mathcal{N})$ an Event Calculus description with the sort of time-points interpreted as the natural numbers $\mathcal{N}$, and $\forall t.I(t)$ be the invariant we wish to demonstrate. Let is a simple time structure consisting of just two points *Sc* and *Sn* such that *Sc* < *Sn*. Then $EC(\mathcal{N})$ $\forall t.I(t)$ if and only if $EC(\mathcal{N})$ $I(0)$ and $EC(\ ) \cup \{I(Sc)\}$ $I(Sn)$.

[2] 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 [30].

Suppose that $EC(\mathcal{N})$ is an Event Calculus description with the sort of time-points interpreted as the natural numbers $\mathcal{N}$, and $\forall t.I(t)$ is the invariant we wish to demonstrate. Then in logical terms we wish to show that $EC(\mathcal{N}) \quad \forall t.I(t)$. The theorem states that, provided the invariant is initially true (i.e. $I(0)$ is true), it is sufficient to show that $EC(\ ) \ U \{I(Sc)\} \quad I(Sn)$, where is a simple time structure consisting of just two points *Sc* and *Sn* such that $Sc < Sn$ ("*c*" for "current" and "*n*" for "next"). In other words, it is sufficient to consider only a symbolic time-point *Sc* and its immediate successor *Sn*, assume the invariant to be true at *Sc*, and demonstrate that its truth then follows at *Sn*. (Proof of the theorem is straightforward by induction over $\mathcal{N}$.) This theorem is applicable even when complete information about the initial state of the system is not available. Its utilisation reduces computational costs considerably because, in the context of $EC(\ )$, it allows us to re-write all our Event Calculus axioms with ground time-point terms. For example, (EC5) becomes:

$$HoldsAt(f,Sn) \leftarrow [HoldsAt(f,Sc) \wedge \neg Clipped(Sc,f,Sn)]$$

Our final logical tool for efficient reasoning about Event Calculus specifications is *abduction* [19]. Abduction is the process of finding a consistent extension (of a specified form) to a logical specification such that it then entails a given goal. Given an Event Calculus description *EC* and a goal *G* expressed as a collection of *HoldsAt* facts, abductive tools exist that will attempt to identify (the completion of) a collection of *Happens* facts Δ such that $EC \cup \Delta \quad G$ and $EC \cup \Delta$ is consistent. (In the context of *planning*, each fact in Δ is then interpreted as an action for the agent to perform to achieve the goal *G*.) Moreover, some of these abductive tools (e.g. [22]) are *complete*, in the sense that they will always identify such a Δ if one exists.

In this paper, we will use abduction "in reverse". Using the reduced time structure described above, we will prove assertions of the form $EC(\ ) \cup \{I(Sc)\} \quad I(Sn)$ by showing that a complete abductive procedure fails to produce a set Δ of *HoldsAt* and *Happens* facts (grounded at *Sc*) such that $EC(\ ) \cup \{I(Sc)\} \cup \Delta \quad \neg I(Sn)$. This procedure is valid given the reasonable assumptions that only a finite number of events can occur in a given instant and that the total number of system properties (fluents) is finite. The theorem described above then allows us to confirm that, provided $I(0)$ is true, $\forall t.I(t)$ is also true. As we shall see, the abductive procedure has the added advantage that if instead it does produce such a set Δ, then this Δ is an explicit indicator of where in the specification (i.e. in the SCR table) there is a problem.

4 ABDUCTIVE INCONSISTENCY HANDLING

We are now in the position to describe our abductive Event Calculus approach to analysing the consistency between SCR mode transition tables and system invariants. We refer to Table-1 as an example case study to illustrate our approach. Briefly, SCR tables are translated into Event Calculus specifications of the type described above, system invariants are expressed (in the same language) as *HoldsAt* formulae universally quantified over time, and then abduction is used as an inference method to either confirm that the (translation of the) table satisfies the invariants or to identify the parts where it does not. To guarantee the computational efficiency and scalability of this process, the Event Calculus translations are reduced to ground, two time-point versions, of the type described above, prior to the application of the abductive reasoning process. The soundness of this reduction is guaranteed by Theorem 1. In practice, the refinement of an SCR table using this method will be an iterative process, with the abductive tool being reapplied after each change of the table.

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" it is $\neg HoldsAt(C,t) \wedge Happens(@T(C),t)$,and for "@F" it is $HoldsAt(C,t) \wedge Happens(@F(C),t)$. 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:

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

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

$$HoldsAt(Running,t) \wedge Happens(@F(Running),t)]$$

Clearly, this axiom pair captures the intended meaning of individual rows as described in Section 2.

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 2 we may regard SCR tables as also containing "hidden" or "default" rows (which the engineer does not bother to list) in which the current and the new mode are identical. Mismatches between the system invariants and the table are just as likely to be caused by these hidden rows as by the explicit rows of the table. Because our translation utilises completions, the abductive tool is able to identify problems in hidden as well as explicit rows.

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).

The Abductive Procedure. For the purposes of discussion, let us suppose that the mode transition table in question has been translated into an Event Calculus specification $EC(\mathcal{N})$ (where the $\mathcal{N}$ signifies that our structure of time-points is isomorphic to the natural numbers) and that the system invariants have been expressed as n universally quantified sentences $\forall t.I_1(t),\ldots,\forall t.I_n(t)$ (where each I_n is expressed with standard logical connectives and the *HoldsAt* predicate). We add an additional constraint $\forall t.I_0(t)$ to the specification which 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 the conjunction $\forall t.I_0(t) \wedge \ldots \wedge \forall t.I_n(t)$. In the case of the cruise control specification, the invariants are (reading "|" as exclusive or):

I_0: $[HoldsAt(Off,t) \mid HoldsAt(Inactive,t) \mid HoldsAt(Cruise,t) \mid HoldsAt(Override,t)]$

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

As stated in Section 3, a general theoretical result about the Event Calculus allows us to use an abductive tool with a reduced version of the Event Calculus specification. The specification is reduced in the sense that it uses a time structure consisting of just two symbolic points *Sc* and *Sn* such that $Sc < Sn$. Our abductive procedure attempts to find system behaviors described by the transition table that are inconsistent with the system invariants (i.e. potential inconsistencies between $EC(\mathcal{N})$ and $\forall t.I(t)$) by attempting to generate a consistent set Δ of *HoldsAt* and *Happens* facts (positive or negative literals grounded at *Sc*), such that $EC( ) \cup \{I(Sc)\} \cup \Delta \quad \neg I(Sn)$. We can also check the specification against a particular invariant $\forall t.I_i(t)$ by attempting to abduce a Δ such that $EC( ) \cup \{I(Sc)\} \cup \Delta \quad \neg I_i(Sn)$. Because the abductive procedure is complete, 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, in the case of the cruise control specification, 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)*, $\neg$*HoldsAt(Brake,Sc)*, *HoldsAt(Cruise,Sc)*, $\neg$*Happens(@F(Ignited),Sc)*, $\neg$*Happens(@F(Running),Sc)*, $\neg$*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) weaken or delete the system invariant (in this case I_3) that has been violated, or (3) add an extra invariant that forbids the combination of *HoldsAt* literals in Δ (e.g. add $I_5 = [HoldsAt(Cruise,t) \rightarrow \neg HoldsAt(Toofast,t)]$). Choices such as this 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 inconsistencies are identified (i.e. until the tool fails to abductively generate a Δ).

This example illustrates in general all the types of choices for change that will be available when an inconsistency is detected. In particular, as described in Section 2 any mode transition table employing "-"s as values for monitored variables is equivalent to a (greatly) expanded table in which the "-"s have been eliminated, and in which all "hidden" rows (i.e. rows where the current and new modes are the same) have been added. Thus any change in the

concise version of the table (e.g. changing a "t" into a "@F" or a "-") is equivalent to changing the new mode (i.e. the value in the right-most column) in some collection of rows in the expanded table. This in turn means that performing a change in the concise version of the table is equivalent to replacing zero or more of its rows by a collection of rows taken from the newly modified, expanded version of the table. (Of course, as this collection is added, it should be appropriately collapsed to a manageable size by re-introduction of "-"s.) In other words, the only real underlying choice that the engineer has when defining or altering a mode transition table is which new mode any given set of conditions and events will result in.

Tool Support. It is beyond the scope of this paper to describe in detail the implementation of our abductive tool. However, it is worth briefly mentioning how the tool avoids the pitfalls sometimes associated with theorem proving (and in particular abductive theorem proving) techniques, these being computational inefficient and non-scalable. We are able to avoid both these problems because we are able to reduce the representation to a ground expression with just two symbolic time-points *Sc* and *Sn*. Furthermore, the particular structure of this expression allows us to largely avoid the consistency checking that often imposes a high computational cost on automated abductive procedures as they construct an extension Δ. This is because the particular form of the Event Calculus we use already ensures that any internally consistent, finite collection of *Happens* literals is consistent with any related specification. Therefore, it is only necessary to check the consistency of candidate *HoldsAt* literals against the system invariants, and this can be done efficiently because both these types of expression are grounded at *Sc*.

Our prototype tool is implemented in Prolog, using a simplified version of the abductive logic program module described in [21]. The logic program conversion of the given (classical logic) Event Calculus specification is achieved using the method described in [22], which overcomes the potential mismatch between the negation-as-failure used in the implementation and the classical negation used in the specification.

5 DISCUSSION AND CONCLUSIONS

Observations. The efficiency and, we believe, intuitiveness of our approach derives partly from the ontology of the Event Calculus. In particular, the Event Calculus' explicit representation of event occurrences means that there is a one-to-one correspondence between a given row in the mode transition table and a particular pair of sentences in the translated specification. This correspondence facilitates the implementation of automatic translators from SCR specifications to Event Calculus specifications and vice-versa. This would allow the abductive approach to be used as "back-end" of existing requirements engineering tools without requiring engineers to write specifications directly in the logical form. Furthermore, Theorem 1, by justifying the reduction of Event Calculus specifications to two-time-point counterparts, reflects directly the primary intuition behind mode transition tables. This is that the extended dynamic behavior of the system can be reduced to a description involving just two symbolic states labeled "current mode" and "new mode". Finally, the use of predicate completion for *Initiates* and *Terminates* exactly mirrors the default assumption implicit in mode transition tables. This is that combinations of events and conditions, not explicitly represented in any row, are not transitions between different modes.

A key characteristic of our approach is that, because the computation does not utilise information about the "initial state", it is applicable to systems whose initial environmental conditions are not fully known. This is likely to be the case in many large or complex event-driven systems. The downside to this characteristic is that the tool will in certain cases be over-zealous in its reporting of potential problems, in that it will also report inconsistencies 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 specification. However, in such cases the resolution of these problems would result only in overly robust, rather then incorrect, specifications.

Related Work. A number of logic-based approaches for handling inconsistency have been proposed in the literature. Specifically, Zowghi and Offen [32] suggest belief revision for default theories as a formal approach for resolving inconsistencies arising during the evolution of requirements specifications. Similarly, Ryan [29] defines (epistemic entrenchment) ordering relations on default information, and changes on these relations rather then the specifications facilitate conflict resolution. A logic-based method more closely related to ours has been proposed by van Lamsweerde *et al.* [31]. 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 similar to our abductive technique in that the notion of goals is comparable to our notion of invariants, and obstacles are comparable to the abduced facts detected by our abductive technique. However, whereas our approach identifies only explanations that are consistent with the specification, van Lamsweerde's approach does not consider consistency checking as part of the process of generating obstacles. Furthermore, the generation of obstacles as well as of refined goals is still purely a human task [31]. We believe that our abductive proof procedure and tool can also be used in the context of van Lamsweerde's goal-driven approach to provide a proof procedure and a tool support for the analysis of goals and obstacles. Recent work by Menzies has also demonstrated the applicability of abductive reasoning to knowledge-based software engineering, using an inference procedure for "knowledge-level modeling" that can support prediction, explanation, and planning [25].

Especially relevant to this paper is existing work on inconsistency analysis of SCR requirements specifications using model checking. Heitmeyer *et al.* [16] illustrate how both explicit state model checkers, such as Spin, and symbolic model checkers, like SMV, can be used to detect safety violations in SCR specifications. The first type of model checking verifies systems' invariants by means of state exploration. Problems related to state explosion are dealt with by the use of sound and complete abstraction techniques, which basically reduce the number of variables to just those that are relevant to the invariant to be tested [16]. In our case, the combination of abduction and Event Calculus has the same effect. Abduction focuses reasoning on goals relevant to the invariant, and the Event Calculus ensures that this reasoning is at the level of relevant variables (fluents) rather than via the manipulation of entire states. The essential differences between our approach and this type of model checking 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 transitions (which correspond directly to rows in the tables) rather than in terms of particular paths through a state space.

Symbolic model checking techniques, such as SMV, use special-purpose languages to represent system specifications, and the (branching time temporal logic) CTL language to express system invariants. Whereas explicit state model checkers detect violation of invariants by enumerating the set of reachable states, symbolic model checkers consider the set of reachable states as logical formulae. The complexity of SMV analysis is therefore given by the number of "possible" (consistent) states, in contrast with the first type of model checking, where the complexity depends on the number of reachable states [3; 5]. Applications of these two techniques to various case studies suggest that explicit state methods are less expensive than symbolic model checking for error detection [5]. Our approach is similar to symbolic model checking in that consistency checking is also needed. However, as explained in Section 4, this checking can be done efficiently, as it only applies to abduced *HoldsAt* literals and system invariants, and both these two types of expressions are grounded with respect to the current state. SMV model checking has also been used to analyse requirements specifications expressed in the Requirements State Machine Language (RSML) [2; 12], in which state explosion problems are addressed by performing the consistency analysis directly on the model of the specifications. Decomposition and function composition rules are adopted to guarantee the scalability of the approach to large system specifications [12].

Future work. The abductive approach described in this paper lays the (theoretical and practical) foundations for the development of a logic-based method with efficient tool support for inconsistencies handling in SCR requirements specifications and more generally in event-driven systems specifications. However, a number of specific technical and general issues are still open to further investigation.

In this paper, our approach has been applied to single mode transition tables and system invariants. However, full SCR specifications also include event tables and condition tables. Our approach could also be extended to facilitate reasoning about such tables. Event tables could be translated into *Initiates* and *Terminates* rules, where modes are expressed by *HoldsAt* literals, events by *Happens* literals, and the controlled variables or terms defined by the table are expressed using *Initiates* or *Terminates* literals. Condition tables, on the other hand, could be formalised as additional constraints of the Event Calculus specification by using *HoldsAt* formulae. In a very similar way, environmental constraints could also be included in the Event Calculus specification as *HoldsAt* constraints. For such extensions, abduced *HoldsAt* literals would have also to be checked for consistency with respect to these additional constraints. A second extension of the approach would be to consider SCR specifications with non-Boolean variables. Our approach facilitates the representation of SCR specifications with non-Boolean variables only when such variables can be re-expressed in terms of auxiliary Boolean conditions. The approach needs to be extended in order to handle specifications that do need to refer to the non–Boolean values of their variables explicitly.

A third line of future investigation is to allow for non-determinism. Although the Event Calculus makes it straightforward to model non-deterministic and concurrent systems, in this present paper we have considered only deterministic systems. This is also the case for other existing model checking approaches [4; 5]. In our approach both the abductive reasoning principles and the tool will need to be appropriately adapted in order to reason about non-deterministic and concurrent event transitions.

A more general issue for future work is the development of (i) automated tools for translating SCR tables into Event Calculus specifications, and (ii) user-friendly interfaces for the abductive tool. Both these issues are feasible, in particular because of the systematic way of generating Event Calculus rules from SCR tables.

Acknowledgements. We gratefully acknowledge the feedback and suggestions of Connie Heitmeyer, Ramesh Bharadwaj, Axel van Lamsweerde, Didar Zowghi, Gianpaolo Cugola, Jonathan Moffet and our colleagues in the DSE group at Imperial College. Thanks also to Peter Grimm and Bruce Labaw of NRL for assisting us in installing SCR* at Imperial College. Tony Kakas also provided many insightful comments about abductive tools. This work was partially funded by the UK EPSRC projects MISE (GR/L 55964) and VOICI (GR/M 38582).

REFERENCES

1. Alspaugh, T. *et al.*, "Software Requirements for the A-7E Aircraft", *Technical Report*, Naval Research Laboratory, March 1988.

2. Anderson, R, Beame, P. Burns S., Chan, W., Modugno F., Notkin D. and Reese, J., "Model Checking Large Software Specifications", *Proc. of 4th ACM Symp. on Foundations of Soft. Eng.*, October 1996.
3. Atlee, J.M. and Buckley, M.A., "A Logic-Model Semantics for SCR Software Requirements", *Proc. of the Int. Symp. on Soft. Testing and Analysis*, 280-292, January 1996.
4. Atlee, J.M. and Gannon, J., "State-Based Model Checking of Event-Driven System Requirements", *IEEE Trans. on Soft. Eng.*, 19(1):24-40, January 1993.
5. Bharadwaj, R. and Heitmeyer, C., "Model Checking Complete Requirements Specifications Using Abstraction", *Technical Report NRL-7999*, Naval Research Laboratory, Washington, 10th Nov. 1997.
6. M.Clarke, E. and M.Wing, J., "Formal Methods, State of the Art and Future Directions", *ACM Computing Surveys 28(4)*:626-643, December 1996.
7. Console, L., Portinale, L. and Theseider Dupre, D., "Using Compiled Knowledge to Guide and Focus Abductive Diagnosis", *IEEE Trans. on Knowledge and Data Eng.*, 8(5):690-706, 1996.
8. Console, L., Sapino, M.L. and Theseider Dupre, D., "The Role of Abduction in Database View Updates", *Journal of Intelligent Systems*, 1994.
9. Easterbrook, S. and Callahan, J. "Formal Methods for Verification and Validation of Partial Specifications: A Case Study", *Journal of System and Software*, 1997.
10. Esghsi, K., "Abductive Planning with the Event Calculus", *Proc. of Int. Joint Conf. on AI*, 1, 3-8, 1988.
11. Gilb, T. and Graham, D., *Software Inspection*, Addison-Wesley 1993.
12. Heimdahl, M.P.E. and Leveson, N.G., "Completeness and Consistency in Hierarchical State-Based Requirements", *IEEE Trans. on Soft. Eng.*, 22(6):363-377, June 1996.
13. Heitmeyer, C.L., Jeffords, R.D. and Labaw, B.G. "Automated Consistency Checking of Requirements Specifications", ACM *Trans. of Soft. Eng. and Methodology*, 5(3):231-261, July 1996.
14. Heitmeyer, C.L., Labaw, B. and Kiskis, D., "Consistency Checking of SCR-Style Requirements Specifications", *IEEE Proc. of 2nd Int. Symp. on Requirements Engineering*, 27-29, York, March 1995.
15. Heitmeyer, C. *et al*., "SCR*: A Toolset for Specifying and Analyzing Software Requirements", *Proc. of Computer-Aided Verification*, Canada, 1998.
16. Heitmeyer, C. *et al.*, "Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications", *IEEE Trans. on Soft. Eng.*, 24(11):927-947, November 1998.
17. Holzmann, G.J., "The Model Checker SPIN", *IEEE Trans. on Soft. Eng.*, 23(5): 279-295, May 1997.
18. Inoue, K. and Sakam, C., "Abductive Framework for Non-monotonic Theory Change", *Proc. of Int. Joint Conf. on AI*, 1, 204-210, 1995.
19. Kakas, A.C., Kowalski, R.A. and Toni, F., "The Role of Abduction in Logic Programming", *Handbook of Logic in Artificial Intelligence and Logic Programming, 5*, (1998), D.M. Gabbay, C.J. Hogger and J.A. Robinson eds., Oxford University Press, 235-324,.
20. Kakas, A.C. and Mancarella, P., "Database Updates Through Abduction", *Proc. of 16th Very Large Database Conference*, Brisbane, Australia, 1990.
21. Kakas, A.C. and Michael, A., "Integrating Abductive and Constraint Logic Programming", *Proc. of the 12th Int. Conf. on Logic Programming*, Tokyo 1995.
22. Kakas, A. C. and Miller, R., "A Simple Declarative Language for Describing Narratives with Actions", *Journal of L.P.*, 31(1-3):157-200 (Special Issue on Reasoning about Action and Change), 1997.
23. Kowalski, R. A. and Sergot, M. J., "A Logic-Based Calculus of Events", *New Generation Computing*, 4:67-95, 1986.
24. McMillan, K.L., *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.
25. Menzies, T., "Applications of Abduction: Knowledge Level Modeling", *Int. Journal of Human Computer Studies*, 1996.
26. Miller, R. and Shanahan, S., "The Event Calculus in Classical Logic – Alternative Axiomatisations", *Linköping Electronic Articles in Computer and Information Science*, 4(16), 1999.
27. Miller, S., "Specifying the Mode Logic of a Flight Guidance System in CoRE and SCR", *Proc. of 2nd Workshop of Formal Methods in Soft. Practice*, 1998.
28. Parnas, D. L. and Madey J., "Functional Documentation for Computer Systems, *Technical Report CRL 309*, McMaster University, Canada, September 1995.
29. Ryan, M., "Default in Specification", *IEEE Proc. of Int. Symp. on Req. Eng.*, 266-272, San Diego, Jan. 1993.
30. Shanahan, M. P., "Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia", MIT Press (1997).
31. van Lamsweerde, A., Darimont, R. and Letier, E., "Managing Conflicts in Goal-Driven Requirements Engineering", *IEEE Trans. on Soft. Eng.*, Nov. 1998.
32. Zowghi, D. and Offen, R. "A Logical Framework for Modeling and Reasoning about the Evolution of Requirements", *IEEE Proc. of 3rd Int. Symp. on Req. Eng.*, Annapolis, USA, Jan. 1997.