An Abductive Approach for Analysing Event-Based Requirements Specifications

Alessandra Russo[1], Rob Miller[2], Bashar Nuseibeh[3], and Jeff Kramer[1]

[1] Imperial College of Science, Technology and Medicine, London SW7 2BT, U.K.
{ar3,jk}@ic.ac.uk
[2] University College London, London WC1E 6BT, U.K.
rsm@ucl.ac.uk
[3] The Open University, Walton Hall, Milton Keynes, MK7 6AA, U.K.
B.A.Nuseibeh@open.ac.uk

Abstract. We present a logic and logic programming based approach for analysing event-based requirements specifications given in terms of a system's reaction to events and safety properties. The approach uses a variant of Kowalski and Sergot's Event Calculus to represent such specifications declaratively and 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. A case study of an automobile cruise control system specified in the SCR framework is used to illustrate our approach. The technique described is implemented using existing tools for abductive logic programming.

1 Introduction

Requirements specification analysis is a critical activity in software development. 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. This paper describes a formal approach to the detection and analysis of errors, and an associated logic programming 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 partially completed, 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 completely predicted. Second, the tool provides "pinpoint" 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 characteristics that distinguishes our approach from other formal techniques, such as model checking or theorem proving [6].

Our focus is on event-based requirements specifications. In this paper, we will regard such specifications as composed of system descriptions, i.e. expressed in terms of required reactions to events (inputs, changes in environmental conditions, etc.), and global system invariants. For simplicity we restrict our attention to "single-state" invariants (e.g. safety properties) although we speculate that our approach could be adapted for other types of property.

The approach uses the Event Calculus (EC) [17] to declaratively model event-based requirements specifications. The choice of EC is motivated by both practical and formal needs, and gives several advantages. First, in contrast to pure state-transition representations, the EC ontology includes an explicit time structure 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 [22]). Second, the EC 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 representational methods. Both the semantics of the front-end specification language and individual specifications themselves can be represented in EC. Third, we can prove a general property of the particular class of EC representations employed here which 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 EC representations [16, 22].

This brings us to the second corner stone of our approach – the use of abduction. Abduction has already proved suitable for automating knowledge-based software development [20, 27]. Our approach employs abduction in a refutation mode to verify global system invariants with respect to event-based system descriptions. Given a system description and an invariant, the abduction is able to identify a complete set of counterexamples (if any exist) to the system invariant, where each counterexample is in terms of a "current" system state and an associated event-based transition. Failure to find a counterexample establishes the validity of the invariant 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 invariant at an arbitrary symbolic time-point.) The particular form of these counterexamples makes them ideal as diagnoses which can be used to modify the specification appropriately, by altering either the event-based system description, or the set of global system invariants, or both.

The abductive decision procedure employed by our approach has several desirable features. It always terminates, in contrast to most conventional theorem proving techniques. It does not rely on a complete description of an initial state (in contrast to model-checking approaches). Like the EC representation, it supports reasoning about specifications whose state-spaces may be infinite. This last feature is mainly because the procedure is goal- or property-driven.

The next section describes our general approach. It is followed by an illustrative case study involving analysis of an SCR tabular specification. We conclude with some remarks about related and future work.

2 Our Approach

As stated above, we will regard requirements specifications as composed of system descriptions and global system invariants. The analysis task that we are concerned with is to discover whether a given system description satisfies all system invariants, and if not why not. We express a collection of system invariants as logical sentences $I_1, \ldots, I_n$ and an event-based system description as a set of rules S. Thus for each system invariant I_i, we need to evaluate whether $S \models I_i$, and to generate appropriate diagnostic information if not. 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.

2.1 Abduction for Verification

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 formal specification, allows an "observation" or "goal" sentence to be inferred, without causing contradictions [16]. In logical terms, given a domain description D and a sentence (goal) G, abduction attempts to identify a set Δ of assertions such that $(D \cup \Delta) \models G$ and $(D \cup \Delta)$ is consistent. The set Δ must consist only of *abducible* sentences, where the definition of what is abducible is generally domain-specific. Δ is often required to be minimal.

From a computational view, abductive procedures (i.e. procedures to find Δ's) are usually 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. Furthermore, the abducibles together with the system description often have to satisfy a given set of *integrity constraints.* In general this (re)checking for consistency and satisfaction of constraints can be computationally expensive, but the particular form of our EC specifications together with a theoretical result regarding plan consistency in [21] allows us to avoid such pitfalls.

In our abductive approach the problem of proving that, for some invariant I_i, $D \models I_i$ is translated into an equivalent problem of showing that it is not possible to consistently extend D 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 problems is dependent on the particular Event Calculus representation used (see [26]). We solve the latter problem by attempting to generate such a Δ using a complete abductive decision procedure, and refer to this process as using abduction in a *refutation mode.* If the procedure finds a Δ then the assertions in Δ act as a counterexample. 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 invariants. The counterexamples that our approach generates describe particular events occurring in particular "contexts" (i.e. classes of "current states"). To be relevant, these contexts must themselves satisfy the invariants. This is ensured by considering the invariants as integrity constraints on a symbolic current state, which prunes the set of possible counterexamples. A detailed description of the particular abductive proof procedure used in our approach can be found in [14].

2.2 The Event Calculus

The Event Calculus is a logic-based formalism for representing and reasoning about dynamic systems. Its ontology includes an explicit structure of time 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. However, we are currently investigating its applicability to LTS style specifications [18], which may be for concurrent and non-deterministic systems.

Our approach adapts a simple classical logic form of the EC [22], 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. $\mathit{Happens}(a, t)$ indicates that event (or action) a actually occurs at time-point t. $\mathit{Initiates}(a, f, t)$ (resp. $\mathit{Terminates}(a, f, t)$) means that if event a were to occur at t it would cause fluent f to be true (resp. false) immediately afterwards. $\mathit{HoldsAt}(f, t)$ indicates that fluent f is true at t. So, for example $[\mathit{Happens}(A_1, T_4) \wedge \mathit{Happens}(A_2, T_4)]$ indicates that events A_1 and A_2 occur simultaneously at time-point T_4.

System Descriptions as Axiomatisations Every EC description includes a core collection of domain-independent axioms 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-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*).

It is convenient to introduce two auxiliary predicates, *Clipped* and *Declipped*. $\mathit{Clipped}(T_1, F, T_2)$ means that some event occurs between the times T_1 and T_2 which terminates the fluent F:

$$\mathit{Clipped}(t_1, f, t_2) \overset{\text{def}}{\equiv} \exists a, t[\mathit{Happens}(a, t) \wedge t_1 \leq t < t_2 \wedge \mathit{Terminates}(a, f, t)] \tag{EC1}$$

(In all axioms all variables are assumed to be universally quantified with maximum scope unless otherwise stated.) Similarly, $\mathit{Declipped}(T_1, F, T_2)$ means that some event occurs between the times T_1 and T_2 which initiates the fluent F:

$$\mathit{Declipped}(t_1, f, t_2) \overset{\text{def}}{\equiv} \exists a, t[\mathit{Happens}(a, t) \wedge t_1 \leq t < t_2 \\ \wedge \mathit{Initiates}(a, f, t)] \tag{EC2}$$

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

$$\mathit{HoldsAt}(f, t_2) \leftarrow [\mathit{Happens}(a, t_1) \wedge \mathit{Initiates}(a, f, t_1) \\ \wedge t_1 < t_2 \wedge \neg \mathit{Clipped}(t_1, f, t_2)] \tag{EC3}$$

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

$$\neg \mathit{HoldsAt}(f, t_2) \leftarrow [\mathit{Happens}(a, t_1) \wedge \mathit{Terminates}(a, f, t_1) \\ \wedge t_1 < t_2 \wedge \neg \mathit{Declipped}(t_1, f, t_2)] \tag{EC4}$$

(iii) fluents only change status via occurrence of initiating or terminating events:

$$\mathit{HoldsAt}(f, t_2) \leftarrow [\mathit{HoldsAt}(f, t_1) \wedge t_1 < t_2 \\ \wedge \neg \mathit{Clipped}(t_1, f, t_2)] \tag{EC5}$$

$$\neg \mathit{HoldsAt}(f, t_2) \leftarrow [\neg \mathit{HoldsAt}(f, t_1) \wedge t_1 < t_2 \\ \wedge \neg \mathit{Declipped}(t_1, f, t_2)] \tag{EC6}$$

To illustrate how the effects of particular events may be described in the domain-dependent part of a specification using *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, $\mathit{SwitchAOn}$, $\mathit{SwitchBOn}$ and $\mathit{LightOn}$, 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, (ii) that if neither switch is on, flicking them both simultaneously causes the light to come on, and (iii) that if either switch is on, flicking it causes the light to go off (irrespective of the state of the other switch):

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

$$\mathit{Terminates}(\mathit{FlickB}, \mathit{LightOn}, t) \leftarrow \mathit{HoldsAt}(\mathit{SwitchBOn}, t)$$

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 [22] for details). The use of completions avoids the frame problem, 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 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.

2.3 Efficient Abduction with Event Calculus

In this paper, we wish to take an EC description such as that above and use it to test system invariants. In the language of the EC these are expressions involving *HoldsAt* and universally quantified over time, such as $\forall t.[\mathit{HoldsAt}(\mathit{SwitchAOn}, t) \vee \neg \mathit{HoldsAt}(\mathit{LightOn}, t)]$. It is (potentially) computationally expensive to prove such sentences by standard (deductive or abductive) theorem-proving. To overcome this problem we have reduced this inference task to a simpler one as stated by the following theorem.

Theorem 1. *Let* $EC(\mathbb{N})$ *be an Event Calculus description with time-points interpreted as the natural numbers* $\mathbb{N}$*, and let* $\forall t.I(t)$ *be an invariant. Let* $\mathbb{S}$ *be the time structure consisting of two points* S_c *and* S_n*, with* $S_c < S_n$*. Then* $EC(\mathbb{N}) \models \forall t.I(t)$ *if and only if* $EC(\mathbb{N}) \models I(0)$ *and* $EC(\mathbb{S}) \cup I(S_c) \models I(S_n)$*. (Proof by induction over* $\mathbb{N}$*, see [26].)*

Hence to show for some invariant $\forall t.I(t)$ that $EC(\mathbb{N}) \models \forall t.I(t)$ it is sufficient to consider only a symbolic time-point S_c and its immediate successor S_n ("c" for "current" and "n" for "next"), assume the invariant to be true at S_c, and demonstrate that its truth then follows at S_n. Theorem 1 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 $EC(\mathbb{S})$, it allows us to re-write all our EC axioms with ground time-point terms. For example, (EC5) becomes:

$$\mathit{HoldsAt}(f, S_n) \leftarrow [\mathit{HoldsAt}(f, S_c) \wedge \neg \mathit{Clipped}(S_c, f, S_n)]$$

Once the EC representation of an event-based requirements specification is provided (perhaps by automatic translation), the approach applies existing abductive tools to analyse this specification. Using the reduced time structure described above, our approach proves assertions of the form $EC(\mathbb{S}) \cup I(S_c) \models I(S_n)$ by showing that a *complete abductive procedure* fails to produce a set Δ of *HoldsAt* and *Happens* facts (grounded at S_c) such that $EC(\mathbb{S}) \cup I(S_c) \cup \Delta \models \neg I(S_n)$. This procedure is valid given the particular form of the EC descriptions and under reasonable assumption that only a finite number of events can occur in a given instant. Theorem 1 then allows us to confirm that, provided $I(0)$ is true, $\forall t.I(t)$ is also true. 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 invariants.

The particular form of our EC system descriptions allows us to further reduce computational costs by largely avoiding the consistency checking normally associated with abduction. This is because it ensures that any internally consistent, finite collection of *Happens* literals is consistent with any related description. Therefore, it is necessary only 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 S_c.

Logic Programming Implementation Page limitations prevent us from describing in detail the implementation of our abductive tool. However, it is implemented in Prolog, using a simplified version of the abductive logic program module described in [14]. The logic program conversion of the given (classical logic) Event Calculus specification is achieved using the method described in [15], which overcomes the potential mismatch between the negation-as-failure used in the implementation and the classical negation used in the specification.

We have been able to formally prove the correctness of our Prolog tool with respect to the theoretical framework described in this paper, and this is fully documented in [26]. Because we are using abduction in what we have described as "refutation mode", the proof relies on demonstrating both the soundness and completeness of the Prolog abductive computation w.r.t. the classical logic description of abduction (at least in the context of the EC axiomatisation) described here. The proof of completeness builds on the work in [13] on a generalised stable model semantics for abduction, and is valid for a well-defined class of deterministic EC domain descriptions.

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 system invariants, called *mode invariants*, with respect to event-based system descriptions expressed as SCR *mode transition* *tables.* The SCR approach has been proven useful for expressing the requirements of a wide range of large-scale real-world applications [1, 7, 10, 23] and is an established method for specifying and analysing event-based systems.

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 [25]. 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. (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 [4, 9, 10]). However, this case study concerns a simple SCR specification consisting of just a single mode transition table and a list of system invariants.

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 [3], 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 see [3].

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 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, they 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 [11], all of which are expressible in our Event Calculus approach. In this case study, we have adopted the following interpretation. 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.

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

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 entries “@T”, “@F”, “t” and “f”. This means that any transition between a current and new mode specified in a table using n dashes is in effect shorthand for up to 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 “Inactive” to “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 much more concise by the non-specification of transitions between identical modes. A 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 we may regard the meaning of real SCR mode transition tables as being given by "full extended" (and very long!) mode transition tables which 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. 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.

Mode Invariants Mode invariants are unchanging properties (specification assertions) of the system regarding mode classes, which should be satisfied by the system specification. In our case study of an automobile cruise control system, an example of an invariant is $[\mathit{Cruise} \rightarrow (\mathit{Ignited} \wedge \mathit{Running} \wedge \neg \mathit{Brake})]$. 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 $m \rightarrow P$, 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 Invariants

The Translation We can now illustrate the use of our abductive EC approach to analysing mode invariants in SCR mode transition tables. 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 EC 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:

$$\mathit{Initiates}(@T(C), C, t) \qquad \text{(S1)}$$
$$\mathit{Terminates}(@F(C), C, t) \qquad \text{(S2)}$$

The translation of tables into EC axioms (rules) is modular, in that a single *Initiates* and a single *Terminates* rule is generated for each row. 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 $\mathit{HoldsAt}(C,t) \wedge \neg \mathit{Happens}(@F(C),t)$, for "f" it is $\neg \mathit{HoldsAt}(C,t) \wedge \neg \mathit{Happens}(@T(C),t)$, for "@T" it is $\neg \mathit{HoldsAt}(C,t) \wedge \mathit{Happens}(@T(C),t)$, and for "@F" it is $\mathit{HoldsAt}(C,t) \wedge \mathit{Happens}(@F(C),t)$. 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:

$$\begin{aligned}\mathit{Initiates}(@F(\mathit{Running}),\mathit{Inactive},t) \leftarrow\ & [\mathit{HoldsAt}(\mathit{Cruise},t) \\ & \wedge \mathit{HoldsAt}(\mathit{Ignited},t) \wedge \neg \mathit{Happens}(@F(\mathit{Ignited}),t) \\ & \wedge \mathit{HoldsAt}(\mathit{Running},t) \wedge \mathit{Happens}(@F(\mathit{Running}),t)]\end{aligned}$$

$$\begin{aligned}\mathit{Terminates}(@F(\mathit{Running}),\mathit{Cruise},t) \leftarrow\ & [\mathit{HoldsAt}(\mathit{Cruise},t) \\ & \wedge \mathit{HoldsAt}(\mathit{Ignited},t) \wedge \neg \mathit{Happens}(@F(\mathit{Ignited}),t) \\ & \wedge \mathit{HoldsAt}(\mathit{Running},t) \wedge \mathit{Happens}(@F(\mathit{Running}),t)]\end{aligned}$$

Clearly, this axiom pair captures the intended meaning of individual rows as described in Section 3.1.

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 EC) reflect the implicit information in a given SCR table that combinations of condition values not explicitly identified are not mode transitions. As discussed in Section 3.1 we may regard SCR tables as also containing "hidden" rows (which the engineer does not 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.

Our EC 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. Our technique 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 also be represented; e.g, $\mathit{HoldsAt}(\mathit{Off},0)$, so that system invariants may be checked w.r.t. to the initial state separately).

The Abductive Procedure For the purposes of discussion, let us suppose Table 1 has been translated into an EC specification $EC_A(\mathbb{N})$. The system invariants in this particular case are translated into 4 universally quantified sentences $\forall t.I_1(t),\ldots,\forall t.I_4(t)$. In general there will be n such constraints, but we always add an additional constraint $\forall t.I_0(t)$ 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 $\forall t.I_1(t), \ldots, \forall t.I_n(t)$. For our case study the invariants are (reading "|" as exclusive or):

$$
\begin{array}{ll}
I_0: & [\mathit{HoldsAt}(\mathit{Off}, t) \mid \mathit{HoldsAt}(\mathit{Inactive}, t) \mid \\
 & \qquad \mathit{HoldsAt}(\mathit{Cruise}, t) \mid \mathit{HoldsAt}(\mathit{Override}, t)] \\
I_1: & \mathit{HoldsAt}(\mathit{Off}, t) \equiv \neg \mathit{HoldsAt}(\mathit{Ignited}, t) \\
I_2: & \mathit{HoldsAt}(\mathit{Inactive}, t) \rightarrow [\mathit{HoldsAt}(\mathit{Ignited}, t) \wedge \\
 & \qquad [\neg \mathit{HoldsAt}(\mathit{Running}, t) \vee \neg \mathit{HoldsAt}(\mathit{Activate}, t)]] \\
I_3: & \mathit{HoldsAt}(\mathit{Cruise}, t) \rightarrow [\mathit{HoldsAt}(\mathit{Ignited}, t) \wedge \\
 & \qquad \mathit{HoldsAt}(\mathit{Running}, t) \wedge \neg \mathit{HoldsAt}(\mathit{Brake}, t)] \\
I_4: & \mathit{HoldsAt}(\mathit{Override}, t) \rightarrow [\mathit{HoldsAt}(\mathit{Ignited}, t) \wedge \mathit{HoldsAt}(\mathit{Running}, t)]
\end{array}
$$

As stated previously, Theorem 1 allows us to use our tool with a reduced version of the EC specification that uses a time structure $\mathbb{S}$ consisting of just two points S_c and S_n with $S_c < S_n$. To recap, our abductive procedure attempts to find system behaviors that are counterexamples of the system invariants by generating a consistent set Δ of *HoldsAt* and *Happens* facts (positive or negative literals grounded at S_c), such that $EC(\mathbb{S}) \cup I(S_c) \cup \Delta \models \neg I(S_n)$. We can also check the specification against a particular invariant $\forall t.I_i(t)$ by attempting to abduce a Δ such that $EC(\mathbb{S}) \cup I(S_c) \cup \Delta \models \neg I_i(S_n)$. 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, when checking the table against I_3 the tool produces the following:

$$
\begin{array}{ll}
\Delta = & \{\mathit{HoldsAt}(\mathit{Ignited}, S_c), \mathit{HoldsAt}(\mathit{Running}, S_c), \mathit{HoldsAt}(\mathit{Toofast}, S_c), \\
 & \neg \mathit{HoldsAt}(\mathit{Brake}, S_c), \mathit{HoldsAt}(\mathit{Cruise}, S_c), \neg \mathit{Happens}(@F(\mathit{Ignited}), S_c), \\
 & \neg \mathit{Happens}(@F(\mathit{Running}), S_c), \neg \mathit{Happens}(@F(\mathit{Toofast}), S_c), \\
 & \mathit{Happens}(@T(\mathit{Brake}), S_c)\}
\end{array}
$$

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 $I_5 = [\mathit{HoldsAt}(\mathit{Cruise}, t) \rightarrow \neg \mathit{HoldsAt}(\mathit{Toofast}, t)]$). 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 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.

4 Conclusions, Related and Future Work

Our case study illustrates the two characteristics of our approach mentioned in the introduction. It was able to detect violations of invariants 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 invariants that were problematic. It avoids high computational overheads because of the choice of logical representation and theoretical results, which allow us to reduce the reasoning task before applying the tool. We believe our approach could be more widely applicable. In particular, we are investigating its use in analysing LTS [18] specifications.

A variety of techniques have been developed for analysing requirements specifications. These range from structured inspections [8], to more formal techniques such as model checking, theorem proving [6] and other logic-based approaches (e.g. [20, 27, 28]). Most techniques based on model checking facilitate automated analysis of requirements specifications and generation of counterexamples when errors are detected [2, 4, 11]. However, in contrast to our approach they presuppose complete descriptions of the initial state(s) of the system to compute successor states. Moreover, they need to apply abstraction techniques to reduce the size of the state space, and can only handle finite state systems.

For example, in the context of SCR, [11] illustrates how both explicit state model checkers, such as Spin [12], and symbolic model checkers, like SMV [19], can be used to detect safety violations in SCR specifications. The first type of model checking verifies system invariants by means of state exploration. Problems related to state explosion are dealt with by the use of sound and complete abstraction techniques, which reduce the number of variables to just those that are relevant to the invariant to be tested [11]. The goal-driven nature of our abductive EC has the same effect, in that abduction focuses reasoning on goals relevant to the invariant, and the EC 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) deals 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 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 any possible initial state if such information is given elsewhere in the specifications. However, this feature can only result in overly robust, rather then incorrect, specifications. If desired we can reapply the abductive procedure, with information about the initial state and a full time structure, to test for reachability.

Theorem proving [24] provides an alternative way of analysing requirements specifications, even for infinite state systems. However, in contrast to our approach this does not provide useful diagnostic information when a verification fails, and computations may not always terminate. [5] uses a hybrid approach 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 in [28] 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 invariants, 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. Although it uses backward reasoning and classical unification as in the abductive phase of our decision procedure, no checking for consistency or satisfaction of domain-dependent constraints is performed once an obstacle is generated. Moreover, the identification of obstacles is not automated. Our procedure might also be used effectively to support automated identification of obstacles in [28]'s framework.

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 [20]. 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 [27].

Our next aim is to test our approach on larger and more complex specifications, for example of systems with infinite states or including non-determinism. As mentioned in Section 2.2, the EC allows the representation of such types of specifications, but further experimentation is needed.

Acknowledgements Thanks to R. Bharadwaj, G. Cugola, P. Grimm, C. Heitmeyer, A. Kakas, B. Labaw, A. van Lamsweerde, J. Moffet, D. Zowghi and the DSE group at IC. This work was supported by the EPSRC projects MISE (GR/L 55964) and VOICI (GR/M 38582).

References

1. Alspaugh, T. et al. (1988). *Software Requirements for the A-7E Aircraft.* Naval Research Laboratory.
2. Anderson, R, et al. (1996). *Model Checking Large Software Specifications.* ACM Proc. of 4th Int. Symp. on the Foundation of Software Engineering.
3. Atlee, J. M, and Gannon, J. (1993). *State-Based Model Checking of Event-Driven System Requirements.* IEEE Transaction on Software Engineering, 19(1): 24-40.
4. Bharadwaj, R, and Heitmeyer, C. (1997). *Model Checking Complete Requirements Specifications Using Abstraction.* Technical Report No. NRL-7999, NRL.
5. Bharadwaj, R, and Sims, S. (2000). *Salsa: Combining Solvers with BDDs for Automated Invariant Checking.* Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in CS, Springer.

6. Clarke, M, and Wing, M. (1996). *Formal Methods, State of the Art and Future Directions.* ACM Computing Survey, 28(4): 626-643.
7. Easterbrook, S, and Callahan, J. (1997). *Formal Methods for Verification and Validation of Partial Specifications.* Journal of Systems and Software.
8. Gilb, T, and Graham, D. (1993). *Software Inspection.* Addison-Wesley.
9. Heitmeyer, C. L, Labaw, B, and Kiskis, D. (1995). *Consistency Checking of SCR-style Requirements Specifications.* Proc. of 2nd Int. Symp. on Requirements Engineering, York, 27-29.
10. 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.
11. 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.
12. Holzmann, G. J. (1997). *The Model Checker SPIN.* IEEE Transaction on Software Engineering, 23(5): 279-295.
13. Kakas, A. C, and Mancarella, P. (1990). *Generalised Stable Models: A Semantics for Abduction.* ECAI'90, Stockholm, pages 385-391.
14. Kakas, A. C, and Michael, A. (1995). *Integrating Abductive and Constraint Logic Programming.* Proc. of 12th Int. Conf. on Logic Programming, Tokyo.
15. 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.
16. Kakas, A. C, Kowalski, R. A, and Toni, F. (1998). *The Role of Abduction in Logic Programming.* In C.J. Hogger, J. A. Robinson D. M. Gabbay (Eds.), Handbook of Logic in Artificial Intelligence and Logic Programming (235-324). OUP.
17. Kowalski, R. A, and Sergot, M. J. (1986). *A Logic-Based Calculus of Events.* New Generation Computing, 4: 67-95.
18. Magee, J, and Kramer, J. (1999). *Concurrency: State Models and Java Programs.* John Wiley.
19. McMillian, K. L. (1993). *Symbolic Model Checking.* Kluwer Academic.
20. Menzies, T. (1996). *Applications of Abduction: Knowledge Level Modeling.* International Journal of Human Computer Studies.
21. Miller, R. (1997) *Deductive and Abductive Planning in the Event Calculus.* Poc. 2nd AISB Workshop on Practical Reasoning and Rationality, Manchester, U.K.
22. Miller, R, and Shanahan, M. (1999). *The Event Calculus in Classical Logic.* Linkoping Electronic Articles in Computer and Information Science, 4(16).
23. Miller, S. (1998). *Specifying the mode logic of a Flitght Guidance System in CoRE and SCR.* Proceedings of 2nd Workshop of Formal Methods in Software Practice.
24. Owre, S, et al. (1995). *Formal verification for fault-tolerant architecture: Prolegomena to the design of PVS.* IEEE Transactions on S.E, 21(2): 107-125.
25. Parnas, D. L, and Madey, J. (1995). *Functional Documentation for Computer Systems.* Technical Report No. CRL 309, McMaster University.
26. Russo, A, Miller, R, Nuseibeh, B, and Kramer, J. (2001). *An Abductive Approach for Analysing Event-based Specifications.* Technical Report no. 2001/7, Imperial College.
27. Satoh, K. (1998). *Computing Minimal Revised Logical Specification by Abduction.* Proc. of Int. Workshop on the Principles of Software Evolution, 177-182.
28. van Lamsweerde, A, Darimont, R, and Letier, E. (1998). *Managing Conflicts in Goal-Driven Requirement Engineering.* IEEE Transactions on S.E.