Early Identification of Problem Interactions: A Tool-Supported Approach

Thein Than Tun, Yijun Yu, Robin Laney, and Bashar Nuseibeh

Department of Computing
The Open University
Walton Hall, Milton Keynes
{t.t.tun, y.yu, r.c.laney, b.nuseibeh}@open.ac.uk

Abstract. [**Context and motivation**] The principle of "divide and conquer" suggests that complex software problems should be decomposed into simpler problems, and those problems should be solved before considering how they can be composed. The eventual composition may fail if solutions to simpler problems interact in unexpected ways. [**Question/problem**]Given descriptions of individual problems, early identification of situations where composition might fail remains an outstanding issue. [**Principal ideas/results**] In this paper, we present a tool-supported approach for early identification of all possible interactions between problems, where the composition cannot be achieved fully. Our tool, called the OpenPF, (i) provides a simple diagramming editor for drawing problem diagrams and describing them using the Event Calculus, (ii) structures the Event Calculus formulae of individual problem diagrams for the abduction procedure, and (iii) communicates with an off-the-shelf abductive reasoner in the background and relates the results of the abduction procedure to the problem diagrams. The theory and the tool framework proposed are illustrated with an interaction problem from a smart home application. [**Contribution**] This tool highlights, at an early stage, the parts in problem diagrams that will interact when composed together.

Keywords. Problem Composition, Problem Interactions, Problem Frames, Event Calculus

1 Introduction

One general approach to problem solving in requirements engineering is to decompose complex software problems into simpler familiar problems [1]. A software problem refers to the challenge of specifying a software system that satisfies an expressed user requirement [2]. In this approach to problem solving, no provision is initially made about the questions of if and how the subproblems obtained can be composed to solve the larger complex problem. Only when subproblems have been solved, are the concerns for composition considered and addressed as separate problems in their own right. This deferral of the concerns for composition is seen as an effective way of managing complexity in software development. However, when solutions to subproblems are found, several questions arise [3]:

Are the problems free from interactions? If they interact, how do they interact? If there are undesired interactions, what can be done to remove them? In this paper, we are primarily concerned with the second question.

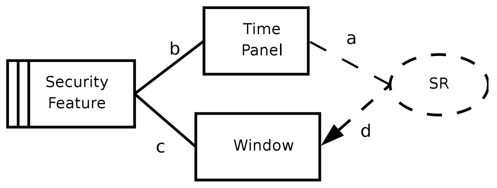
We will consider this question within the framework of the Problem Frames (PF) approach [2], which has been recognised as providing a core ontology for requirements engineering [4]. Following this approach, complex problems are decomposed by fitting their subproblems into known problem patterns. Subproblems, when composed together, may interact in unexpected ways. By problem interactions, we refer to situations where the composition of solutions to subproblems does not constitute a solution to the complex problem (from which subproblems were obtained). These interactions may be related to the issues of *consistency*, *precedence*, *interference* and *synchronisation* [2].

Checking whether subproblems in an event-based reactive system can be composed can only give event sequences where the composed requirement is satisfied (if it can be satisfied at all): it cannot tell us event sequences where the composed requirement is not satisfied. One way to solve this problem is to monitor the runtime behaviour of the system, and diagnosis failures whenever they are detected [5]. However, event sequences are identified only after the occurrence and detection of failures.

The main contribution of the paper is a tool-supported approach that uses abductive reasoning [6] to identify all possible event sequences that may cause the composition to fail. Given a description of system behaviour, an abductive procedure [7] obtains all event sequences that can satisfy a requirement, if the requirement is satisfiable. Otherwise, the procedure will report no event sequence.

In order to identify event sequences leading to a failure in composition, we negate the conjunction of all requirements to be composed as the requirement to satisfy, and attempt to abduce all possible event sequences for the negated requirement. In other words, we assume that the problems cannot be composed and ask an abductive reasoner to find out why. If the procedure returns no event sequence, there is no interactions between problems. On the other hand, any event sequence returned by the abduction is a possible interaction in the composition, and may require further analysis. Our use of logical abduction is reminiscent of [8]. In this paper, we will focus on the identification of possible interactions, whilst the issue of how undesired interactions can be removed is discussed in [9].

We have implemented a tool chain called the OpenPF to demonstrate how event sequences leading to failures in composition can be detected using the technique suggested. The front-end of the OpenPF helps requirements engineers create problem diagrams and describe the elements in the diagram using the Event Calculus, a form of temporal logic [10, 11]. The back-end of our tool encodes the input into an executable specification for an off-the-shelf Event Calculus abductive reasoner [12, 13] to check whether the new diagram can be composed with the existing ones. If event sequences for possible failures are found, the tool takes the abduction results and relates them back to relevant problem diagrams.



Interface Phenomena Set	
TiP!a	{Night, Day}
TiP!b	{NightStarted, DayStarted}
SF!c	{TiltIn, TiltOut}
Win!d	{Open, Shut}

Fig. 1. Problem Diagram: Security Feature

The rest of the paper is organised as follows. In Section 2, we present an brief overview of the Problem Frames approach and the Event Calculus, and explain how they are used in this paper. Our approach to identifying interacting problems is explained in Section 3, and application of the OpenPF tool is discussed in Section 4. Related work can be found in Section 5, whilst Section 6 provides some concluding remarks.

2 Preliminaries

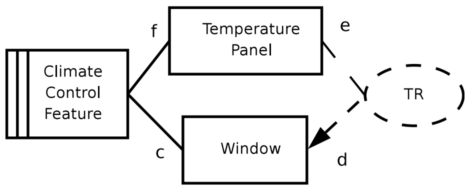
In this section, we introduce two decomposed problems related to the security feature and climate control feature of a smart home application [14]. The purpose of the simple example is to help us illustrate the approach and tool-support. An overview of Problem Frames, the Event Calculus and how we use minimal Event Calculus predicates to describe problem diagrams are also explained.

2.1 Problem Diagrams and Their Descriptions

An important feature of the Problem Frames approach (PF) is that it makes a clear distinction between three descriptions: the *requirements* (R), the *problem world domains* (W) and the *specification* (S). Broadly speaking, the requirements describe the desired property of the system, the problem world domains describe the given structure and behaviour of the problem context, and the specifications describe the behaviour of the software at the machine-world interface [2].

Security Feature in Smart Home The problem diagram of the security feature (SF), shown in Figure 1, emphasises the high-level relationship between the requirement, written inside a dotted oval, the problem world domains, denoted by plain rectangles, representing entities in the problem world that the machine must interact with, and a machine specification, denoted by a box with a double stripe, implementing a solution to satisfy the requirement.

The problem world domains in the context of the security feature are Window (Win) and Time Panel (TiP). When describing their behaviour, we will use labels such as (Win1) and (TiP1), and refer to them later in the discussion. The window



Interface Phenomena Set
TeP!e {Hot, Cold}
TeP!f {HotStarted, ColdStarted}
CCF!c {TiltIn, TiltOut}
Win!d {Open, Shut}

Fig. 2. Problem Diagram: Climate Control Feature

has two *fluents*, or time-varying properties, Open and Shut, each a negation of the other. At the interface SF!c, the machine SF may generate instances of events (or simply *events* henceforth) `TiltIn` and `TiltOut`, which are only observed by the window (Win). The behaviour of the window is such that once it observes the event `TiltIn`, it will soon become shut (Win1); once it observes the event `TiltOut`, the window will become open (Win2), when it starts to tilt in, it is no longer open (Win3), and when it starts to tilt out, it is no longer shut (Win4), and the window cannot be both open and shut at the same time (Win5).

Similarly, the Time Panel domain has two opposing fluents, Day and Night. When time switches from day to night, the panel generates the event `NightStarted` once at the interface TiP!b, observed by the machine SF (TiP1). Likewise, when the daytime begins, `DayStarted` is generated once (TiP2). Solid lines between these domains represent *shared phenomena*: for example, c is a set of the phenomena `TiltIn` and `TiltOut`, and SF!c indicates that these phenomena are controlled by the security feature and are observed by Window. In such event-based systems, states of the problem world domains are represented by *fluents*, whilst these domains communicate by sending/receiving *events*. This is a neat mapping to the Event Calculus ontology, as we shall see later.

The requirement for the security problem (SR) can be expressed informally as follows: "Keep the window shut during the night." Notice that the requirement statement *references* the fluent Night of the TiP at the interface TiP!a, and *constrains* the fluent Shut of the window at the interface Win!d.

A possible specification for the security feature (SF) may be: "Fire `TiltIn` whenever `NightStarted` is observed (SF1). Once `TiltIn` is fired, do not fire `TiltOut` as long as `DayStarted` is not observed (SF2)." It should be noted that the specification is written only in terms of events at the interface between the machine and the world, while the requirement is written only in terms of the fluent properties of the problem world domains.

Climate Control Feature in Smart Home The problem of the climate control feature (CCF) is shown in Figure 2. The requirement of this problem is: "Keep the window open when it is hot". Temperature Panel (TeP) fires an event `HotStarted` or `ColdStarted` to indicate the relationship between the preferred and the actual temperatures.

Correctness of Specifications In addition to the three descriptions, the Problem Frames approach also provides a way of relating these descriptions through the entailment relationship $W, S \models R$, showing how the specification, within a particularly context of the problem world, is sufficient to satisfy the requirement. This provides a template for structuring correctness proofs, and/or arguments for sufficiency/adequacy, of specifications [2].

An informal argument for adequacy of the specification may be provided as a positive event sequence: When the night starts, the time panel will generate the event `NightStarted` observed by the machine (TiP1). The security feature will fire `TiltIn` as soon as it observes `NightStarted` (SF1). `TiltIn` makes the window shut (Win1). Since the specification does not allow the window to tilt out until `DaytStarted` is observed (SF2), the window will remain shut during the night, thus satisfying the requirement (SR).

Although we have so far described the problem diagrams using an informal language, the Problem Frames approach is agnostic about the particular choice of the description language. We now give an overview of the description languge used in remainder of the paper.

2.2 The Event Calculus

The Event Calculus (EC) is a system of logical formalism, which draws from first-order predicate calculus. It can be used to represent actions, their deterministic and non-deterministic effects, concurrent actions and continuous change [10]. Therefore, it is suitable for describing and reasoning about event-based temporal systems such as the smart home application. Several variations of EC have been proposed, and the version we adopted here is based on the discussions in [11]. Some elementary predicates of the calculus and their respective meanings are given in Table 1.

Table 1. Elementary Predicates of the Event Calculus

Predicate	**Meaning**
Happens(a, t)	Action a occurs at time t
Initiates(a, f, t)	Fluent f starts to hold after action a at time t
Terminates(a, f, t)	Fluent f ceases to hold after action a at time t
HoldsAt(f, t)	Fluent f holds at time t
$t1 < t2$	Time point $t1$ is before time point $t2$

The Event Calculus also provides a set of domain-independent rules to reason about the system behaviour. These rules define how fluent values may change as a result of the events.

$$\begin{aligned} Clipped(t1, f, t2) \overset{\text{def}}{\equiv} \exists a, t[Happens(a, t) \wedge \\ t1 \leq t < t2 \wedge Terminates(a, f, t)] \end{aligned} \tag{EC1}$$

$$\begin{aligned} HoldsAt(f, t2) \leftarrow [Happens(a, t1) \wedge \\ Initiates(a, f, t1) \wedge t1 < t2 \wedge \neg Clipped(t1, f, t2)] \end{aligned} \tag{EC2}$$

For instance, the rule (EC1) states that Clipped(t1,f,t2) is a notational shorthand to say that the fluent f is terminated between times t1 and t2, whilst the rule (EC2) says that fluents that have been initiated by occurrence of an event continue to hold until occurrence of a terminating event. There are other such rules in the Event Calculus but we will omit them for space reasons. All variables in our formulae are universally quantified except where otherwise shown. We also assume linear time with non-negative integer values. We follow the rules of circumscription in formalizing commonsense knowledge [12], by assuming that all possible causes for for a fluent are given and our reasoning tool cannot find anything except those causes.

In the Event Calculus, given a requirement expressed using a `HoldsAt` formula, the abduction procedure will find all `Happens` literals via a system description and the Event Calculus meta-rules. For example, given the requirement `HoldsAt(Open, 4)`, and the domain rule `Initiates(TiltOut, Open, t)`, the meta-rule (EC2) allows us to abduce that `Happens(TiltOut, 3)` is a possible event squence to satisfy the requirement.

2.3 Relating Event Calculus to Problem Frames

As discussed in [9, 15], requirements are largely about some desired properties in the problem world, and can be expressed using the `HoldsAt` predicate. Problem world domains are about causality between events and fluents: event-to-fluent causality can be expressed using the `Initiates` and `Terminates` predicates, and the fluent-to-event causality is expressed using the `HoldsAt` and `Happens` predicates. Specifications are about events generated by machines, and can be expressed using the `Happens` predicate. Event and fluent names will be parameterised with the names of domains that control them. For instance, we will write `TiltIn(SF)` to refer to the event `TiltIn` controlled by the domain Security Feature. The same event or fluent name with different parameters denote distinct events or fluents respectively.

Once described in this way, the $W, S \models R$ entailment of problem diagrams can be instantiated in the Event Calculus, allowing us to prove the correctness of a specification, with respect to a problem world context and a requirement [15].

3 Identifying Problem Interactions

This section describes the formal basis of our approach, while working through the running examples introduced in Section 2.

3.1 Abducing Failure Event Sequences in Problem Composition

Let n be the total number of problem diagrams that have been initially created in the workspace, where $1 \leq n$. Each problem diagram has the Event Calculus descriptions of the requirement, relevant problem world domains, and the specification. Furthermore, each specification is assumed to be correct with respect to its problem world context and the requirement: for every diagram i in the workspace, where $1 \leq i \leq n$, the entailment $W_i, S_i \models R_i$ holds. Since problem diagrams are created incrementally, the n^{th} diagram is typically the newly added diagram.

Let W be $W_1 \wedge \cdots \wedge W_n$, denoting the finite conjunction of all `Initiates`, `Terminates`, and `Happens` formulae in the current workspace. It is a conjunction because each `Initiates` and `Terminates` formula, for instance, describes a rule by which a fluent changes its value, and if the rule can be applied in a subproblem, it should be possible to apply the same rule in the composed system. We assume that W is consistent, meaning for instance that there are no two rules in W that allow a fluent to be true and false at the same time.

Let S be $S_1 \wedge \cdots \wedge S_n$, denoting the finite conjunction of the `Happens` formulae in the specifications of problem diagrams in the workspace. Finally, let R be $R_1 \wedge \cdots \wedge R_n$, denoting the finite conjunction of `HoldsAt` formulae of the requirements in the problem diagrams. We consider conjunction, rather than disjunction, as the composition operator for any composed requirement, because any disjuncted requirement will be trivially satisfied if one of the individual requirements has been satisfied (which is the case).

Our focus, therefore, is on requirements: in particular, requirements that interaction. When the composed requirements is not always satisfiable, we would like to identify the event sequences where the composition may fail. Again, the possibility of composition failures does not necessarilly mean the system is not useful: these failures may be tolerated, even desired in some cases, or prevented from arising by modifying the problem context. The composed system should have the property $W, S \models R$. The question raised then is: What are the possible failure event sequences in this system? In other words: What are the event sequences where this entailment may not hold?

Let Δ be the set of all possible event sequences (ordered Happens literals) permitted by the system, i.e. W and S. One way to check whether R can be failed is as follows. For every $\sigma \in \Delta$, verify whether the entailment (1) holds.

$$W, S, \sigma \models R \tag{1}$$

Let us denote the set of event sequences that satisfy the entailment (1) as Δ_1 and the set of event sequences that do not satisfy the entailment (1) as Δ_2. Clearly, $\Delta = \Delta_1 \cup \Delta_2$. There are two major limitations to finding Δ_2 through deduction. In a system with a reasonable number of events and fluents, verifying the relationship for a good length of time will require a large Δ, which is usually difficult to obtain. Secondly, this approach can be highly inefficient because it requires checking exhaustively [8].

In such circumstances, logical abduction is regarded as more efficient [8]. Logical abduction is a procedure that, given a description of a system, finds all event sequences that satisfy a requirement. Since an abduction procedure will return the event sequences Δ_1 that satisfy the goal, it is not possible to obtain Δ_2 using the abduction procedure on the entailment relation (1).

In order to identify failure event sequences, we take the negation of the composed requirement $\neg R$ as the requirement to be satisfied, whilst W and S serve as the same description of the composed system. Given the uniqueness of fluent and event names, completion of `Initiates` and `Terminates` predicates, and the event calculus meta-rules, the procedure will find a complete set of event sequences Δ_2, such that the entailment (2) holds for every member ϵ of Δ_2 [12].

$$W, S, \epsilon \models \neg R \tag{2}$$

Since Δ_2 also is a set of event sequence permitted by the composed system, any ϵ is a member of Δ. Each ϵ is a failure event sequence, and is a refutation of (1). If Δ_2 is empty, Δ equals Δ_1, and all valid sequences of events permitted by the composed system will lead to the satisfaction of the requirement in (1).

Since our Event Calculus formulae are annotated with the names of problem world domains in problem diagrams, when Δ_2 is not empty, each ϵ will contain references to elements in the problem diagrams. This information allows us to relate the results of abduction procedure back to the corresponding problem diagrams in the workspace.

3.2 Smart Home Example

In order to illustrate a simple application of the approach, we will first formalise the requirements, specifications and the descriptions of the problem world domains discussed in Section 2. Natural language descriptions of all formulae given below are provided in Section 2.

Security Feature The requirement for the security feature (SR), described in Section 2, can be formalised as follows.

$$HoldsAt(Night(TiP), t) \rightarrow HoldsAt(Shut(Win), t + 1) \tag{SR}$$

This formula is, in fact, stronger than the natural language statement. The formula says that at every moment that is night, the window should be shut at the next moment, requiring the window to be shut until one time unit after the night has passed. This formulation is chosen for its simplicity. The behaviour of the window domain in the security problem is given below.

$$Initiates(TiltIn(SF), Shut(Win), time) \tag{Win1}$$

$$Initiates(TiltOut(SF), Open(Win), time) \tag{Win2}$$

$$Terminates(TiltIn(SF), Open(Win), time) \tag{Win3}$$

$$Terminates(TiltOut(SF), Shut(Win), time) \tag{Win4}$$

$$HoldsAt(Open(Win), time) \leftrightarrow \neg HoldsAt(Shut(Win), time) \tag{Win5}$$

Parameterisation of the event and fluent names is important because (Win1), for instance, allows only the TiltOut event generated by the security feature to affect the fluent Shut. The behaviour of the time panel domain is described below.

$$[HoldsAt(Day(TiP), time-1) \wedge HoldsAt(Night(TiP), time)] \leftrightarrow Happens(NightStarted(TiP), time) \tag{TiP1}$$

$$[HoldsAt(Night(TiP), time-1) \wedge HoldsAt(Day(TiP), time)] \leftrightarrow Happens(DayStarted(TiP), time) \tag{TiP2}$$

Finally, the specification of the security feature can be formalised as follows.

$$Happens(NightStarted(TiP), time) \rightarrow Happens(TiltIn(SF), time) \tag{SF1}$$

$$\begin{aligned}&[Happens(NightStarted(TiP), time) \wedge \\ &\neg Happens(DayStarted(TiP), time1) \wedge time \leq time1] \\ &\rightarrow \neg Happens(TiltOut(SF), time1)\end{aligned} \tag{SF2}$$

Climate Control Feature Formalisation of the requirements, problem world domains and the specification of the climate control feature is given below. Since the behaviour of the window is the same in both problems, we will omit their formulae in this feature, but note that the TiltIn and TiltOut events will be parameterised with CCF, instead of SF.

$$HoldsAt(Hot(TeP), t) \rightarrow HoldsAt(Open(Win), t+1) \tag{TR}$$

$$[HoldsAt(Cold(TeP), time-1) \wedge HoldsAt(Hot(TeP), time)] \leftrightarrow Happens(HotStarted(TeP), time) \tag{TeP1}$$

$$[HoldsAt(Hot(TeP), time-1) \wedge HoldsAt(Cold(TeP), time)] \leftrightarrow Happens(ColdStarted(TeP), time) \tag{TeP2}$$

$$Happens(HotStarted(TeP), time) \rightarrow Happens(TiltOut(CCF), time) \tag{TF1}$$

$$\begin{aligned}&Happens(HotStarted(TeP), time) \wedge \\ &[\neg Happens(ColdStarted(TeP), time1) \wedge time \leq time1] \\ &\rightarrow \neg Happens(TiltIn(CCF), time1)\end{aligned} \tag{TF2}$$

Detecting Interactions R in this case is $TR \wedge SR$; W is the conjunction of (Win1–Win5), similar formulae for the window in the climate control problem, (TiP1), (TiP2), (TeP1) and (TeP2); and S is the conjunction of (SF1), (SF2), (TF1) and (TF2). In order to detect posssible failure event sequences in the composition of the two problems, we will first take the negation of the composed requirement, which can be stated as:

$$\begin{aligned}&(HoldsAt(Night(TeP), t) \wedge HoldsAt(Open(Win), t+1)) \vee \\ &(HoldsAt(Hot(TeP), t) \wedge HoldsAt(Shut(Win), t+1))\end{aligned} \tag{$\neg R$}$$

The abduction procedure, in this case, will work as follows. It may be either day or night, and hot or cold, and the window may be open or shut at the beginning, and the procedure will condsider every valid combination. Suppose the window is initially open during a hot day at a symbolic time t1. In order to abduce event sequence for $HoldsAt(Hot(TeP), t1) \wedge HoldsAt(Shut(Win), t1 + 1)$, for instance, the procedure will look for events that can make the fluents hold.

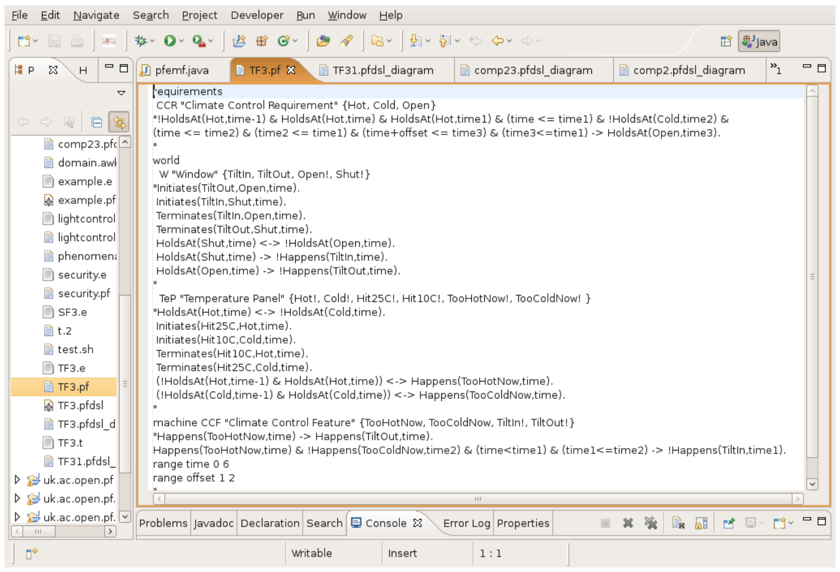
Since it is already hot, $HoldsAt(Hot(TeP), t1)$ is true. In order to satisfy $HoldsAt(Shut(Win), t1+1)$, the procedure will look for event that can turn the current state $HoldsAt(Open(Win), t1)$ into $HoldsAt(Shut(Win), t1 + 1)$.

According to the domain rule (Win1), its counterpart for the climate control problem, and the Event Calculus meta-rule (EC2), if the event TiltIn(SF) or TiltIn(CCF) happens, the window will be shut at the next time point, provided the event TiltOut(SF) or TiltOut(CCF) does not happen at the same time. The event TiltIn(SF) will be fired when NightStarted(TeP) is fired, according to (SF1), and NightStarted(TeP) is triggered when the day turns into night, according to (TeP1). The event TiltOut(SF) will not happen at the same time because of (SF2). TiltOut(CCF) in (TF1) will not happen at the same time because it has been hot for a while.

In other words, in one possible failure event sequence, the window is open during a hot day, and the night soon begins. In that case, the window will be shut according to the specification of the security feature, because the climate control feature cannot prevent the window from being shut, thus resulting in a failure situation where the smart home is hot but the window is shut. This, of course, is a single event sequence and there may be other such event sequences, and the abduction procedure will find all of them in one pass. From the event sequence obtained, we know how the security problem and the climate control problem can interact.

The failure event sequence in this composition is due to the fact that no precendence between the security and the climate control requirements has been defined. It is, of course, possible that the smart home is situated in a world where it is never too hot at night. The interaction only arises under certain conditions, and the abduction procedure can find those conditions. (Strictly speaking, the abduction procedure cannot reason about the changes from day to night unless the events for these changes are given. That is due to the frame axiom in the Event Calculus. The examples we implemented in the next section include these events, but for space reasons we have ommitted them.)

A similar failure may arise if the specifications of individual problems are too strong. For example, another way to satisfy the security requirement is to have a specification that fires the TiltIn event at every time point, thus ensuring that the window is shut at all times. Similarly, the climate control requirement can be satisfied by another specification that fires the TiltOut event at every time point. They both satisfy the individual requirements, but any chance of composition is prevented because the specifications are too strong. Again, the same procedure



```
requirements
 CCR "Climate Control Requirement" {Hot, Cold, Open}
"!HoldsAt(Hot,time-1) & HoldsAt(Hot,time) & HoldsAt(Hot,time1) & (time <= time1) & !HoldsAt(Cold,time2) &
(time <= time2) & (time2 <= time1) & (time+offset <= time3) & (time3<=time1) -> HoldsAt(Open,time3).
"
world
  W "Window" {TiltIn, TiltOut, Open!, Shut!}
"Initiates(TiltOut,Open,time).
 Initiates(TiltIn,Shut,time).
 Terminates(TiltIn,Open,time).
 Terminates(TiltOut,Shut,time).
 HoldsAt(Shut,time) <-> !HoldsAt(Open,time).
 HoldsAt(Shut,time) -> !Happens(TiltIn,time).
 HoldsAt(Open,time) -> !Happens(TiltOut,time).
"
  TeP "Temperature Panel" {Hot!, Cold!, Hit25C!, Hit10C!, TooHotNow!, TooColdNow! }
"HoldsAt(Hot,time) <-> !HoldsAt(Cold,time).
 Initiates(Hit25C,Hot,time).
 Initiates(Hit10C,Cold,time).
 Terminates(Hit10C,Hot,time).
 Terminates(Hit25C,Cold,time).
 (!HoldsAt(Hot,time-1) & HoldsAt(Hot,time)) <-> Happens(TooHotNow,time).
 (!HoldsAt(Cold,time-1) & HoldsAt(Cold,time)) <-> Happens(TooColdNow,time).
"
machine CCF "Climate Control Feature" {TooHotNow, TooColdNow, TiltIn!, TiltOut!}
"Happens(TooHotNow,time) -> Happens(TiltOut,time).
Happens(TooHotNow,time) & !Happens(TooColdNow,time2) & (time<time1) & (time1<=time2) -> !Happens(TiltIn,time1).
range time 0 6
range offset 1 2
"
```

Fig. 3. An input file to create a problem diagram

can be used to identify those conditions in descriptions of problem diagrams. A detailed discussion, however, is beyond the scope of the paper.

Performing this abduction procedure manually is both labourious and error-prone. Fortunately, there are several implementation of this procedure for the Event Calculus. In the next section, we describe an end-to-end tool for detecting interacting problems that automates much of what has been discussed.

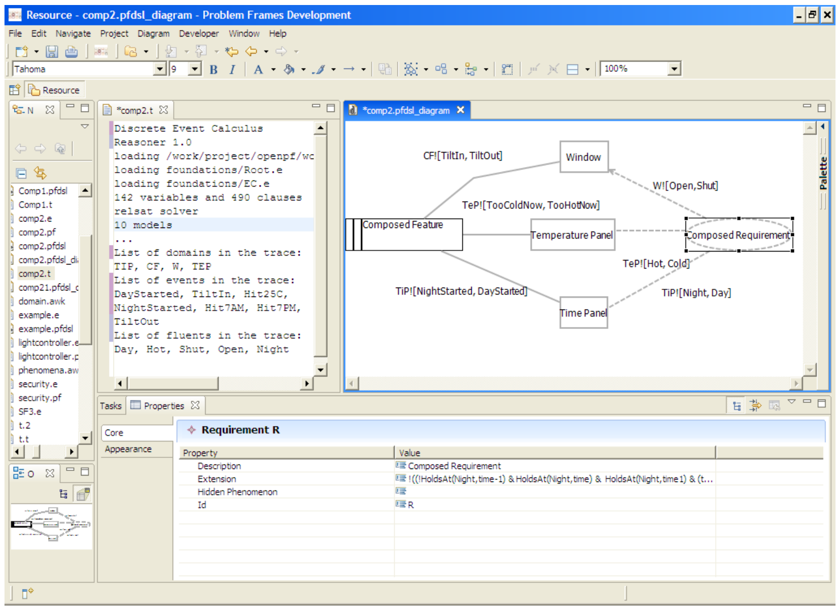
4 Detecting Interacting Problems Using the OpenPF

This section gives a brief overview of the OpenPF tool, with a particular emphasis on how interacting problems can be discovered early in the development.

4.1 Creating Problem Diagrams

An easy way to create problem diagrams using the OpenPF is through an input file that defines the names of the requirement, problem world domains and the machine, together with the Event Calculus formulae as their descriptions. Figure 3 shows an extract from the input file to create the climate control problem diagram and its descriptions.

Our OpenPF tool will check the syntax of the above input and the conformance of the Event Calculus formulae to the problem diagram. For instance, the tool will not allow two problem world domains to have the same names as otherwise there may be ambiguity in the produced EC formalae. Furthermore, it will check whether the requirement description refers to event names: since



```
Discrete Event Calculus
Reasoner 1.0
loading /work/project/openpf/wc
loading foundations/Root.e
loading foundations/EC.e
142 variables and 490 clauses
relsat solver
10 models
...
List of domains in the trace:
TIP, CF, W, TEP
List of events in the trace:
DayStarted, TiltIn, Hit25C,
NightStarted, Hit7AM, Hit7PM,
TiltOut
List of fluents in the trace:
Day, Hot, Shut, Open, Night
```

Requirement R

Property	Value
Description	Composed Requirement
Extension	!((!HoldsAt(Night,time-1) & HoldsAt(Night,time) & HoldsAt(Night,time1) & (t...
Hidden Phenomenon	
Id	R

Fig. 4. Generated problem diagram with the Event Calculus descriptions

the requirements should be written in terms of fluents, such descriptions are not allowed. The tool will also annotate the event calculus formulae with the appropriate domain names: for instance, the fluent term Open will be written as Open(W) because the window domain assigns values to the fluent. It will also generate a problem diagram from the input.

4.2 Detecting Interactions in the Running Example

Once a problem diagram is created, the OpenPF tool can generate a composition diagram such as the one shown in Figure 4 for the running example. When the diagram is created, the tool automatically generates the Event Calculus script to abduce failure event sequences for the composition. The script will include the conjunction of all formulae for problem world domains in all individual problem diagrams, the conjunction of all formulae for the specifications, and the negation of the conjunction of the requirements formulae (as shown in the property window in Figure 4).

The Event Calculus script is then fed to the off-the-shelf abductive reasoner, `Decreasoner` [12, 13], in order to abduce the event sequences satisfying the negated requirement. `Decreasoner` will translate the abduction problem in the Event Calculus into a SAT problem and solve it using the solver `Relsat` [16, 17]. SAT results are then translated back into the Event Calculus literals by `Decreasoner`. From the output of `Decreasoner`, the OpenPF tool will capture the abduction output, and relate it to the elements in the problem diagrams. In one view, the tool can show the event sequences of the interactions, and in another view, it will pinpoint the list of problem world domains, events and fluents involved in a the interaction (as shown in the panel to the left of the diagram in Figure 4). Once an input file as shown in Figure 3 is provided, the rest of the tasks of finding interacting problem diagrams, or even individual elements with a diagram, is done automatically.

Our initial evaluation criterion is to implement the idea that identifying possible failure event sequences in problem composition can be done efficiently through logical abduction. Our implementation of the OpenPF using Problem Frames, Event Calculus, decreasoner, and Model-driven Eclipse plugins, and the smart home examples have demonstrated the viability of our idea. An abduction example involving 140 variables and 440 Event Calculus clauses has been computed in less than one second on a standard personal laptop. Since the abduction procedure and modern SAT solvers such as Relsat are efficient, it gives us confidence that a framework such as the OpenPF will scale when applied to larger examples.

5 Related Work

Russo *et al.* [8, 18] provide theoretical insights on the use of abductive reasoning in the analysis of requirements. Given a system description and an invariant, Russo *et al.* propose using of an abduction procedure to generate a complete set of counterexamples, if there is any, to the invariant. Rather than analyse explicit safety properties, we use logical abduction to identify possible interactions between problems, with the assumption that conjunction will be the eventual composition operator. Failure event sequences suggested by our approach may not be sound (if the interactions can be tolerated), but they provide an early indication of possible failures in composition.

Wang *et al.* [5] propose a SAT-based framework for monitoring run-time satisfaction of requirements and diagnosing the problems when errors happens. In order to diagnose the components where errors originate, the framework logs the system execution and when goals are not satisfied, the traces are preprocessed and transformed into a SAT problem using propositional logic. Although their aim and ours are similar, we work with early requirements models where there is no running system to monitor. Moreover, our approach generates possible failure event sequences by abduction procedure.

van Lamsweerde et al [19] propose a framework for detecting requirements inconsistencies at the goal level and resolving them systematically. Unlike the KAOS approach [19], we reason about the system behaviour in a bottom-up fashion. Once the behaviour of individual solutions are specified, failure event sequences can be generated automatically. A way of resolving the composition problem such as precedence are discussed in [9].

Nentwich *et al* [20] describe a tool-supported approach for describing consistency constraints and checking them across various resources. Similarly, Egyed [21] presented a tool for instantly checking consistency of UML diagrams. Although inconsistency checking also plays an important role in our approach, we are detecting run-time interactions rather than static inconsistency in the representation of artefacts.

Seater and Jackson [22] propose a systematic way to derive specifications using the Problem Frames approach, and they use the Alloy language and Alloy Analyzer to check the validity of the derivation. Our work is complementary in the sense that they are concerned with decomposing and specifying individual problems, but we are concerned with the composition of the individual problems.

6 Conclusions and Future Work

In this paper, we examined the issue of problem interactions: these are situations where the conjunction of solutions to smaller problems introduce new, often unexpected, problems in the composed solution. Although checking whether some given problems can be composed is relatively easy, identifying situations where the composition may fail can be difficult. In this paper, we proposed that identification of problem interactions where composition cannot be achieved fully can be done through logical abduction. We have used the OpenPF tool to demonstrate our idea using examples taken from a smart home application.

The issue of identifying possible failure event sequences is closely related to the question of suggesting viable corrective actions to resolve undesired interactions. We are currently investigating how requirements engineers can use the early feedback obtained through logical deduction in order to come up with proposals for corrective actions.

Although, we focused on problem solving approaches that defer the concerns of composition, other problem solving approaches in requirements engineering may have a similar issue. For instance, when there is a need to modify a goal tree, or to merge smaller goal trees, the question of finding event sequences leading to possible failures may be raised. Therefore, we conjecture that our approach can be applied, with little or no modification, in those cases. We are also investigating in this direction.

Acknowledgements We would like to thank our colleagues at the Open University, in particular, Michael Jackson, and the anonymous reviewers for their helpful comments and suggestions. This research is funded by the EPSRC, UK.

References

1. Parnas, D.L., Lawford, M.: The role of inspection in software quality assurance. IEEE Trans. Softw. Eng. **29**(8) (2003) 674–676
2. Jackson, M.: Problem Frames: Analyzing and structuring software development problems. ACM Press & Addison Wesley (2001)
3. Robinson, W.N., Pawlowski, S.D., Volkov, V.: Requirements interaction management. ACM Computing Surveys **35**(2) (2003) 132–190

4. Jureta, I., Mylopoulos, J., Faulkner, S.: Revisiting the core ontology and problem in requirements engineering. In: Proceedings of the 2008 16th IEEE International Requirements Engineering Conference, IEEE Computer Society (2008) 71–80
5. Wang, Y., McIlraith, S.A., Yu, Y., Mylopoulos, J.: An automated approach to monitoring and diagnosing requirements. In: Proceedings of the International Conference on Automated Software Engineering, ACM (2007) 293–302
6. Shanahan, M.: Prediction is deduction but explanation is abduction. In: Proceedings of the International Joint Conference on Artificial Intelligence, Morgan Kaufmann (1989) 1055–1060
7. Denecker, M., Schreye, D.D.: Sldnfa: an abductive procedure for normal abductive programs. In: Proc. of the International Joint Conference and Symposium on Logic Programming, MIT Press (1992) 686–700
8. Russo, A., Miller, R., Nuseibeh, B., Kramer, J.: An abductive approach for analysing event-based requirements specifications. In Stuckey, P.J., ed.: ICLP. Volume 2401 of Lecture Notes in Computer Science., Springer (2002) 22–37
9. Laney, R., Tun, T.T., Jackson, M., Nuseibeh, B.: Composing features by managing inconsistent requirements. In: Proceedings of 9th International Conference on Feature Interactions in Software and Communication Systems (ICFI 2007). (2007) 141–156
10. Shanahan, M.P.: The event calculus explained. In Woolridge, M.J., Veloso, M., eds.: Artificial Intelligence Today, Lecture Notes in AI no. 1600. Springer (1999) 409–430
11. Miller, R., Shanahan, M.: The event calculus in classical logic - alternative axiomatisations. Journal of Electronic Transactions on Artificial Intelligence (1999)
12. Mueller, E.T.: Commonsense Reasoning. Morgan Kaufmann (2006)
13. Decreasoner: `http://decreasoner.sourceforge.net/`.
14. Kolberg, M., Magill, E., Marples, D., Tsang, S.: Feature interactions in services for internet personal appliances. In: In Proceedings of IEEE International Conference on Communications (ICC-2002). Volume 4., New York (2001) 2613–2618
15. Classen, A., Laney, R., Tun, T.T., Heymans, P., Hubaux, A.: Using the event calculus to reason about problem diagrams. In: Proceedings of International Workshop on Applications and Advances of Problem Frames, NY, USA, ACM (2008) 74–77
16. Jr., R.J.B., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: AAAI/IAAI. (1997) 203–208
17. Relsat: `http://code.google.com/p/relsat/`.
18. Russo, A., Nuseibeh, B.: On the use of logical abduction in software engineering. In Chang, S.K., ed.: Software Engineering and Knowledge Engineering. World Scientific Publishing Corporation (2000)
19. Lamsweerde, A.v., Letier, E., Darimont, R.: Managing conflicts in goal-driven requirements engineering. IEEE Trans. Softw. Eng. **24**(11) (1998) 908–926 http://dx.doi.org/10.1109/32.730542.
20. Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: xlinkit: a consistency checking and smart link generation service. ACM Trans. Interet Technol. **2**(2) (2002) 151–185
21. Egyed, A.: Instant consistency checking for the uml. In: Proceedings of the International Conference on Software Engineering, NY, USA, ACM (2006) 381–390
22. Seater, R., Jackson, D.: Requirement progression in problem frames applied to a proton therapy system. In: Proceedings of RE'06, Washington, DC, USA, IEEE Computer Society (2006) 166–175