ISSN 1744-1986

Technical Report N° 2008/ 13

# Tool support to derive specifications for conflict-free composition

Thein Than Tun
Robin Laney
Michael Jackson
Bashar Nuseibeh

31st July, 2008

Department of Computing
Faculty of Mathematics, Computing and Technology
The Open University

Walton Hall, Milton Keynes, MK7 6AA
United Kingdom

http://computing.open.ac.uk

# Tool support to derive specifications for conflict-free composition

Thein Than Tun, Robin Laney, Michael Jackson and Bashar Nuseibeh
Department of Computing, Open University
Milton Keynes, MK7 6AA, UK
{t.t.tun, r.c.laney, m.jackson, b.nuseibeh}@open.ac.uk

## ABSTRACT

Finding specification of pervasive systems is difficult because it requires making certain environmental assumptions explicit at design-time, and describing the software in a way that facilitates runtime composition. This paper describes how a systematic refinement of specifications from descriptions of the system's environment and requirements can be automated. Our notion of requirements allows individual features in the system to be inconsistent with each other. Resolution of conflicts at design-time is often over-restrictive because it uses the strongest possible conditions for conjunctions and rules out many possible interactions between features. In order to support runtime resolution, our tool examines specifications for potential conflicts and augments them with information to enable detection at runtime. We use a form of temporal logic, the Event Calculus, as our formalism, and characterize the refinement of requirements as a kind of abductive planning. This allows us to use an existing Event Calculus planning tool, implemented in Prolog, as a basis to develop a reasoning tool for obtaining specifications from potentially inconsistent requirements. We validate our tool by applying it to find specifications of smart home software.

## 1. INTRODUCTION

In order to facilitate convenient living, household appliances, such as air conditioners, security alarms, doors and windows are increasingly connected to home digital networks, and the functioning of these appliances is controlled by complex pervasive "smart home" software applications [12, 25, 17, 16]. For example, a security feature of a smart home application may switch on and off lights when homeowners are away to give an impression that the house is occupied. Although there are several tools to *analyze* specifications for certain properties, there are relatively few tools that help *find* specifications. In this paper, we discuss an approach to specifying these systems, and suggest tool support to help develop their specifications.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

Pervasive systems have particular characteristics which call for specific tool support in obtaining their specifications. First, the requirements of such systems are rooted in their environment. As characterized by Jackson [14], the relation between software and its environment can be described as $W, S \vdash R$, where $W$ represents problem world domains, $S$, specifications of the software, and $R$, requirements for the system. The entailment operator ($\vdash$) emphasizes the fact that specifications rely on explicit domain properties in satisfying the requirements. Consider a security requirement ($R$) to record, on VCR, pictures from the security camera when movement in the house is detected while the owners are away ($W$). The security system ($S$) is expected to generate appropriate events to start recording pictures from the security camera when an intrusion is detected. However, this reasoning makes several assumptions about the problem world properties. It assumes, amongst other things, that the VCR is functioning, and is loaded with an appropriate recordable medium, and that it has enough capacity to record for the length of time necessary. In requirements engineering approaches, such as the Problem Frames approach [15], these assumptions are made explicit and reflected appropriately in the specifications.

Second, various features of the smart home software are often implemented by disparate third-party developers [16], and when put together, these features are expected to work collaboratively. This means that inconsistency between requirements is often difficult, or impossible, to resolve at compile-time. For example, if the entertainment feature is preset to record a TV programme and an intrusion is detected while the programme is being recorded, conflicts arise. Since such conflicts may manifest only at runtime, the specifications need to be appropriately augmented to enable runtime conflict detection and resolution.

In this paper, we build on our earlier work that formally refined requirements into specifications in the presence of inconsistencies in the requirements [20] and composed specifications to resolve conflicts at run-time [19]. We now make the refinement rules used in [20] explicit and introduce tool support to find specifications of pervasive systems. The reasoning tool we present here is based on an abductive Event Calculus planner, implemented as a Prolog meta-interpreter, by Shanahan [29]. We extend Shanahan's partial-order planning tool in order to, perhaps iteratively and incrementally, find specifications of smart home software.

The remainder of the paper is organized as follows. We first explain and justify our choices of the requirements engineering approach and logical formalism to describe the re-

quirements and software artifacts in Section 2. We explain our refinement technique in Section 3. We review an existing Event Calculus tool that supports part of our refinement process, and identify its limitations in Section 4. Section 5 describes how these limitations are overcome in our extension of the tool, and the extended tool is applied and evaluated in Section 6. We then discuss the limitations of our extended tool in Section 7. Section 8 presents an overview of related work, and Section 9 concludes the paper.

## 2. BACKGROUND

In this section, we present an overview of the Problem Frames approach, and illustrate it with a motivating example from smart home software. We then discuss the need for tool support in deriving specifications, and explain what the tool is expected to do. We also briefly discuss the Event Calculus used in the paper.

### 2.1 The Problem Frames Approach

Introduced by Jackson in [15], the Problem Frames approach has some key principles, two of which are relevant to our discussions in this paper.

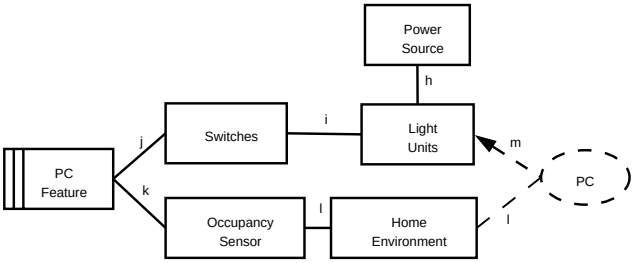
One principle is concerned with the properties of software artifacts in requirements engineering. Intuitively, it suggests that requirements are expressed in terms of properties of its environment (or problem world domains), and specifications, *within the context of problem world domains*, are expected to satisfy the requirements. When applied to a particular type of software problem, this entailment is called a "frame concern" [15].

The problem diagram for the smart home "power control" feature [16] in Fig. 1 can be used to illustrate this characterization of artifacts. The diagram shows a high-level relationship between (i) the *requirement*, written inside a dotted oval, that needs to be satisfied, (ii) *problem domains*, denoted by plain rectangles, representing entities in the problem world that the machine must interact with, and (iii) a *machine domain*, denoted by a box with a double stripe, implementing a solution to satisfy the requirement. In other words, problem domains represent the properties of the problem world that are necessarily true, and requirements represent the properties of the problem world that users wish to hold true, whilst the machine domain represents the properties of a computer that will enact the required properties in that problem world context.

The solid lines such as $j$ in Fig. 1 represent shared events and states between the domains involved. The description of $j$, for example, indicates that the events SwitchLightsOn and SwitchLightsOff are controlled by the machine domain (denoted by the prefix PCF!), whilst the states LightSwitchsTurnedOn and LightSwitchesTurnedOff are controlled by the Switches domain (denoted by the prefix S!). (Events have verb sounding names and states have noun sounding names). It means that, at the interface $j$, the machine domain may fire these two events, but can only observe the two states; the Switches domain, on the other hand, may manipulate the values of the two states, but can only observe the events being fired.

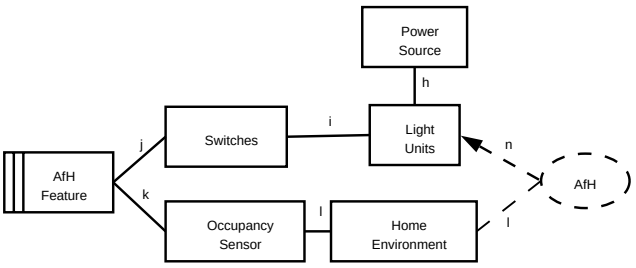
The **requirement** for the power control feature (PC) can be expressed informally as follows: "*When the house is empty, switch off the lights.*"

The **problem world domains** in Fig. 1 have the following properties. When the system switches the lights on




h:PS! {PowerSourceOK}
i:S! {LightSwitchesTurnedOn, LightSwitchesTurnedOff}
j:PCF! {SwitchLightsOn, SwitchLightsOff}
j:S! {LightSwitchesTurnedOn, LightSwitchesTurnedOff}
k:OS! {OccupiedDetected, EmptyDetected}
l:HE! {HouseOccupied, HouseEmpty}
m:LU! {LightsOn, LightsOff}

**Figure 1: Problem diagram for the power control feature**




h:PS! {PowerSourceOK}
i:S! {LightSwitchesTurnedOn,LightSwitchesTurnedOff}
j:AfH! {SwitchLightsOn,SwitchLightsOff}
j:S! {LightSwitchsTurnedOn,LightSwitchesTurnedOff}
k:OS! {OccupiedDetected, EmptyDetected}
l:HE! {HouseOccupied, HouseEmpty}
n:LU! {LightsOn, LightsOff}

**Figure 2: Problem diagram for the away-from-home feature**

(SwitchLightsOn), the switches are turned on (LightSwitchesTurnedOn is true and LightSwitchesTurnedOff is false), and when the system switches the lights off (SwitchLightsOff), the switches are turned off (LightSwitchesTurnedOff is true and LightSwitchesTurnedOn is false). When the switches are turned on (LightSwitchesTurnedOn is true), the lights in Light Units are on, provided that the light units are working well and the power source is OK. The system detects occupancy of Home Environment through Occupancy Sensor; for instance, if the house is occupied (HouseOccupied is true), the sensor will detect it (OccupiedDetected is true).

The **specification** for the machine PC Feature should describe what the machine must do to enact the required properties of the problem world. Typically in requirements engineering, descriptions of requirements and problem world properties are provided by various stake-holders of the system, and specifications are obtained from these two descriptions [15]. There are several systematic approaches to find-

ing specifications [33, 20, 26, 28, 31, 32]. We adopted the Event Calculus-based refinement approach suggested in [20] because its formalism appears to be well suited to describing event-based temporal systems.

The other principle of the Problem Frames approach is related to separation of concerns. When dealing with complex problems, the Problem Frames approach suggests, that individual subproblems should be solved before considering how they may be recomposed to satisfy the requirements of (larger) composed problem.

Consider another feature of the smart home software called "away-from-home". The diagram in Fig. 2 describes the problem and its context, which is similar to the context of the power control feature. The requirement (AfH) in this case, however, is concerned with house security and is informally expressed as follows: "*When the house owners are away, give an impression that the house is occupied by switching the lights on and off periodically*".

Clearly there is an inconsistency between the two features: a conflict will arise when the house is empty, and the power control feature wants to switch off the lights to save energy usage, and the away-from-home feature wants to switch on the lights to give an impression of occupancy.

Given these two features, the Problem Frames approach suggests that these two individual problems should be solved before we consider their composition.

The justification for applying the Problem Frames approach to the smart home problem is two fold. Firstly, the principle of distinguishing *requirements*, *problem world domains* and the *machine domains* elegantly characterizes the nature of smart home systems. Secondly, the separation of concerns principle captures the fact that, since disparate vendors develops features of smart home systems independently, certain conflicts between features may be resolved only at runtime.

## 2.2 Tool Support

The need for tool support reflects the two principles discussed. In essence, we want to automate the following two respective tasks.

> **Task #1**: From appropriate descriptions of a requirement and relevant domain(s), obtain a correct specification.
>
> **Task #2**: In order to help resolve run-time conflicts, augment the specifications with necessary information to enable conflict detection at runtime. For example, if an occurrence of a particular event at a particular time can cause a conflict, this event should be described appropriately in the specifications. This information is used by composition operators to resolve conflicts at runtime [20].

Before we discuss how to obtain specifications and augment them with information to detect conflicts, we give an overview of our chosen formalism, the Event Calculus.

## 2.3 The Event Calculus

The Event Calculus (EC), first introduced by Kowalski and Sergot [18], is a system of logical formalism, which draws from first-order predicate calculus, and can be used to represent actions, their deterministic and non-deterministic effects, concurrent actions and continuous change [30]. Since it is suitable for describing and reasoning about event-based temporal systems, and our smart home system is one such system, we chose EC as our formalism. Since its introduction in [18], several variations of EC have been proposed [24], and we adopt the version suggested by Shanahan [30].

**Table 1: Elementary Predicates of the Event Calculus**

| Formula | Meaning |
|---|---|
| Initiates($\alpha$, $\beta$, $\tau$) | Fluent $\beta$ starts to hold after action $\alpha$ at time $\tau$ |
| Terminates($\alpha$, $\beta$, $\tau$) | Fluent $\beta$ ceases to hold after action $\alpha$ at time $\tau$ |
| Initially($\beta$) | Fluent $\beta$ holds from time 0 |
| $\tau 1 < \tau 2$ | Time point $\tau 1$ is before time point $\tau 2$ |
| Happens($\alpha$, $\tau$) | Action $\alpha$ occurs at time $\tau$ |
| HoldsAt($\beta$, $\tau$) | Fluent $\beta$ holds at time $\tau$ |
| Trajectory($\beta 1$, $\tau$, $\beta 2$, $\delta$) | If Fluent $\beta 1$ is initiated at time $\tau$ then fluent $\beta 2$ becomes true at time $\tau + \delta$ |
| Clipped($\tau 1$, $\beta$, $\tau 2$) | Fluent $\beta$ is terminated between times $\tau 1$ and $\tau 2$ |

$$HoldsAt(\beta, \tau 1) \leftarrow Initially(\beta) \wedge \neg Clipped(0, \beta, \tau 1) \quad \text{(EC1)}$$

$$HoldsAt(\beta, \tau 2) \leftarrow Happens(\alpha, \tau 1) \wedge Initiates(\alpha, \beta, \tau 1) \wedge \tau 1 < \tau 2 \wedge \neg Clipped(\tau 1, \beta, \tau 2) \quad \text{(EC2)}$$

$$HoldsAt(\beta, \tau 3) \leftarrow Happens(\alpha, \tau 1) \wedge Initiates(\alpha, \beta 1, \tau 1) \wedge Trajectory(\beta 1, \tau 1, \beta, \delta) \wedge \tau 2 = \tau 1 + \delta \wedge \tau 1 < \tau 2 \leq \tau 3 \wedge \neg Clipped(\tau 1, \beta 1, \tau 2) \wedge \neg Clipped(\tau 2, \beta, \tau 3) \quad \text{(EC3)}$$

$$Clipped(\tau 1, \beta, \tau 2) \leftrightarrow \exists \alpha, \tau [Happens(\alpha, \tau) \wedge \tau 1 < \tau < \tau 2 \wedge Terminates(\alpha, \beta, \tau)] \quad \text{(DEF1)}$$

**Figure 3: Event Calculus Meta-rules**

The calculus relates events and event sequences to 'fluents', which denote states of a system. In our approach to this smart home problem we use event sequences to describe feature machine behaviours; fluents to describe problem domain states; and we use the rules by which events cause state changes to describe the given properties of the problem domains. Requirements are described as combinations of fluents capturing the required states of the problem world. We also assume linear time with non-negative integer values.

Table 1, based on Shanahan [30], gives the meanings of the elementary predicates of the calculus we use in this paper. The EC rules in Fig. 3, taken from Shanahan [30], are a way of stating that the fluent $\beta$ holds if: it held initially and

$$HoldsAt(HouseEmpty, t5) \wedge \neg Clipped(t5, HouseEmpty, t6) \wedge t5 + 4 < t < t6 \rightarrow HoldsAt(LightsOff, t) \quad \text{(PC)}$$

$$(HoldsAt(HouseEmpty, t5) \wedge \neg Clipped(t5, HouseEmpty, t6) \wedge t5 + 119 < t4 < t6 \wedge 0 \leq t4 \ mod \ 60 \leq 29 \rightarrow HoldsAt(LightsOn, t4)) \wedge (HoldsAt(HouseEmpty, t5) \wedge \neg Clipped(t5, HouseEmpty, t6) \wedge t5 + 119 < t3 < t6 \wedge 30 \leq t3 \ mod \ 60 \leq 59 \rightarrow HoldsAt(LightsOff, t3)) \quad \text{(AfH)}$$

**Figure 4: Formalized requirements**

nothing has happened since to stop it holding (EC1); the event $\alpha$ has happened to make the fluent hold and nothing has happened since to stop it holding (EC2); or, the event $\alpha$ happened that caused some fluent $\beta 1$ to hold, that in turn, after a period of time $\delta$ caused this fluent $\beta$ to hold, and again nothing has happened since to stop the second fluent holding (EC3). Finally, the rule (DEF1) says that the fluent $\beta$ is clipped between $\tau 1$ and $\tau 2$ if and only if there is an event $\alpha$ that happens between $\tau 1$ and $\tau 2$ and the event terminates the fluent $\beta$. Following Shanahan, we assume that all variables are universally quantified except where otherwise shown.

The logical machinery of the Event Calculus works with three components, informally described as (i) "what actions do", (ii) "what happens when", and (iii) "what is true when" [30]. For example, (i) could be "switching on the lights makes them on", (ii), "switching on happened at 12:00 today", and (iii), "the lights were on at 12:01 today". The logical machinery of EC supports three types of reasoning.

If (i) and (ii) are given, through *deduction*, we can conclude that "the lights were on at 12:01 today". Essentially, we are reasoning here from cause to effect. The assumption that nothing else happened between 12:00 and 12:01 is implicit.

If (i) and (iii) are given, through *abduction*, we can hypothesize that "switching on happened at 12:00 today". This effect-to-cause inference may be unsound if we cannot rule out that there are no other possible causes for this effect. In EC, we follow the rules of circumscription [22], in assuming that all possible causes for a fluent are given in the database and our reasoning tool cannot find anything except those causes. This is in line with the Closed World Assumption of Prolog [8], and therefore, ensures the soundness and completeness of the inferences.

If facts such as (ii) and (iii) are given, through *induction*, we can generalize that "switching on the lights turn them on" [30]. As we shall see, our refinement of requirements into specifications is mostly concerned with the abductive reasoning.

We again follow Shanahan in adopting the common sense law of inertia, meaning that fluents do not change value

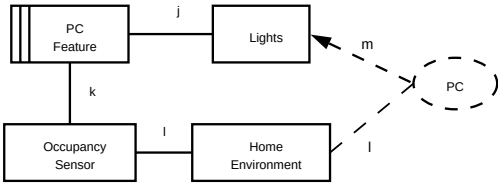



**Figure 5: Abstracting the Lights domain**

$$Initiates(SwitchLightsOn, LightsOn, \tau) \quad \text{(L1)}$$
$$Terminates(SwitchLightsOn, LightsOff, \tau) \quad \text{(L2)}$$
$$Initiates(SwitchLightsOff, LightsOff, \tau) \quad \text{(L3)}$$
$$Terminates(SwitchLightsOff, LightsOn, \tau) \quad \text{(L4)}$$
$$Initiates(DetectEmpty, EmptyDetected, \tau) \leftarrow HoldsAt(HouseEmpty, \tau) \quad \text{(OSHE1)}$$
$$Terminates(DetectEmpty, OccupiedDetected, \tau) \quad \text{(OSHE2)}$$
$$Initiates(DetectOccupied, OccupiedDetected, \tau) \leftarrow HoldsAt(HouseOccupied, t) \quad \text{(OSHE3)}$$
$$Terminates(DetectOccupied, EmptyDetected, \tau) \quad \text{(OSHE4)}$$

**Figure 6: Minimal Domain Descriptions**

unless something happens to cause this. That is, fluents change only in accordance with the meta-rules (EC1), (EC2) and (EC3).

# 3. FINDING SPECIFICATIONS

In order to formally derive the specifications, we first formalize the natural language descriptions of the requirements and the problem world domain discussed in Section 2.1.

## 3.1 Formalizing the requirements

The natural language descriptions of the requirements for the power control and away-from-home features can be formalized using EC as shown in Fig 4.

Assuming each time unit represents a minute, the first definition (PC) says that if the house has been empty for at least 5 minutes, the lights should be turned off, and remain so until it is no longer empty. The second definition (AfH) says that if the house has been empty for at least two hours, the lights should be on for the first 30 minutes and off for next 30 minutes of every hour, as long as the house is empty.

## 3.2 Formalizing domain descriptions

In formalizing the descriptions of the problem world domains in Fig. 1, we first obtain minimal description of these domains by *projecting* [15] the Switches, Light Units and Power Source, as a single domain called Lights, as shown in Fig. 5. It means that we will initially assume, for instance, that the lights can be switched on and off instantaneously, and that they are always reliable. This abstraction is useful for two reasons: first, it reflects the fact the formal descriptions are often developed iteratively and incrementally (and our extended tool supports such development), and second, it allows us to focus on the essential concern. We shall, of course, enhance the descriptions in Section 5.3.

Fig. 6 shows minimal descriptions of the problem domains. The definitions (L1-L4) say that the lights can be

switched on and off instantaneous, and that the on and off states oscillate. If the house is empty, the event DetectEmpty, generated by the sensor, makes the fluent EmptyDetected true (OSHE1), and OccupiedDetected false (OSHE2). When occupancy is detected, the event DetectOccupied, also generated by the sensor, makes the fluent OccupiedDetected true (OSHE3), and EmptyDetected false (OSHE4).

## 3.3 Deriving Specifications

We now examine the refinement of the requirement for the power control feature using the technique suggested in [20], and annotate the refinement with the refinement rule used in each step. The aim of this refinement is to find what the feature machine, PCF, needs to do (in terms of Happen, Initially and Prohibit predicates) to achieve a requirement (HoldsAt clause).

We begin by stating the PC requirement as follows.

$$\begin{aligned} HoldsAt(HouseEmpty, t5) \wedge \\ \neg Clipped(t5, HouseEmpty, t6) \wedge \\ t5 + 4 < t < t6 \rightarrow \\ HoldsAt(LightsOff, t) \end{aligned}$$

We refine the conclusion of the statement by applying EC1. This is an abduction. Disjoined cases for EC2 and EC3 are considered separately.

$$\begin{aligned} &Initially(LightsOff) \wedge \\ &\neg Clipped(0, LightsOff, t) \end{aligned}$$

We now have an Initially clause and a ¬Clipped clause. Initially clauses are treated as a sort of precondition to the event sequences, and therefore Initially(LightsOff) is not refined further.

We apply (DEF1) to the sub-clause *Clipped (0, LightsOff, t)*. This is an equivalence rewrite.

$$\begin{aligned} &Initially(LightsOff) \wedge \\ &\neg\exists a1, t1 \cdot Happens(a1, t1) \wedge \\ &Terminates(a1, LightsOff, t1) \wedge \\ &0 < t1 < t \end{aligned}$$

We unify the Terminate sub-clause with the domain property (L2) in Fig. 6. This is a simple database lookup for *a1*. Apart from SwitchLightsOn, there are no other cases of *a1* to consider in this case.

$$\begin{aligned} &Initially(LightsOff) \wedge \\ &\neg\exists t1 \cdot Happens(SwitchLightsOn, t1) \wedge \\ &Terminates(SwitchLightsOn, LightsOff, t1) \wedge \\ &0 < t1 < t \end{aligned}$$

As a result of this unification, we remove the Terminate sub-clause because it is a domain axiom, (L2) in Fig. 6.

$$\begin{aligned} &Initially(LightsOff) \wedge \\ &\neg\exists t1 \cdot Happens(SwitchLightsOn, t1) \wedge \\ &0 < t1 < t \end{aligned}$$

At this point in refinement, in order to simplify the expression, in [20] we introduce a new predicate, Prohibit, which has the following meaning:

$$\begin{aligned} \text{Prohibit}(\alpha, \tau1, \tau2) \equiv \neg\exists \alpha, \tau\bullet \text{ Happens } (\alpha, \tau) \\ \wedge \tau1 < \tau < \tau2 \end{aligned}$$

This predicate indicatively describes events whose occurrences terminate the fluent that needs to hold. Whether or not these events do occur needs to be evaluated at runtime. As we shall see in Section 6.1, such information is needed to help identify and resolve run-time conflicts [20].

Continuing with the refinement, the next step is to rewrite the last predicate using Prohibit. This is also an equivalence rewrite.

$$\begin{aligned} &Initially(LightsOff) \wedge \\ &Prohibit(SwitchLightsOn, 0, t) \end{aligned}$$

We have now completed derivation of the following partial specification (PCFa), and the specification says that if the lights are initially off (at time 0), the system should prohibit the SwitchLightsOn event from time 0 until time $t$ in order to satisfy PC the requirement.

$$\begin{aligned} HoldsAt(HouseEmpty, t5) \wedge \\ \neg Clipped(t5, HouseEmpty, t6) \wedge \\ t5 + 4 < t < t6 \rightarrow \\ Initially(LightsOff) \wedge \\ Prohibit(SwitchLightsOn, 0, t) \end{aligned} \quad \text{(PCFa)}$$

This refinement is repeated using the (EC2) rule to give the following second partial specification (PCFb) for the PC requirement. The specification says that if the event SwitchLightsOff happens at a time $t1$ before $t$ and the event SwitchLightsOn is prohibited between $t1$ and $t$, then the lights will be off at time $t$.

$$\begin{aligned} HoldsAt(HouseEmpty, t5) \wedge \\ \neg Clipped(t5, HouseEmpty, t6) \wedge \\ t5 + 4 < t < t6 \rightarrow \\ Happens(SwitchLightsOff, t1) \wedge t1 < t \wedge \\ Prohibit(SwitchLightsOn, t1, t) \end{aligned} \quad \text{(PCFb)}$$

The (EC3) rule does not apply in this domains description because the lights are assumed to come on and off instantaneously at all times. Therefore, from these two partial specifications, we obtain the full specification (PCF) for the PC requirement (using minimal domain descriptions) as follows.

$$\begin{aligned} HoldsAt(HouseEmpty, t5) \wedge \\ \neg Clipped(t5, HouseEmpty, t6) \wedge \\ t5 + 4 < t < t6 \rightarrow \\ ((Initially(LightsOff) \wedge \\ Prohibit(SwitchLightsOn, 0, t)) \vee \\ (Happens(SwitchLightsOff, t1) \wedge t1 < t \wedge \\ Prohibit(SwitchLightsOn, t1, t))) \end{aligned} \quad \text{(PCF)}$$

Manual derivation of such specifications is time-consuming, and error-prone. An automated tool can improve the quality of refinement process and specifications.

Implementing the refinement in Prolog is relatively easier because Prolog supports some of the refinement rules well. Implementation of the EC abductive reasoning, however, requires two things. First, since occurrence of several events over a period of time may contribute to achieving a certain goal, performing effect to cause reasoning requires finding a correct temporal ordering of these events. Second, when the tool attempts to prove a goal, we not only want to know whether the goal is provable or not, but how it can be proved. This record of how a goal can be proved, or a residue, is also needed.

# 4. USING AN ABDUCTIVE PLANNER

In this section we briefly review Shanahan's planner [29, 4]. Implemented in Prolog, this partial order planning tool was designed as an abductive theorem prover, based on resolution. The key idea was to deploy a vanilla meta-interpreter so that the EC axioms could be expressed as object-level clauses.

```
holds_at(F1,T3) :-
  happens(A,T1,T2), T2 < T3, initiates(A,F,T1),
  not clipped(T1,F,T2).
```

For example the above (EC2) rule was compiled into the following Prolog code in [29].

```
026  abdemo([holds_at(F1,T3)|Gs1],R1,R5,N1,N4) :-
027a  F1 \= neg(F2),
027b  abresolve(initiates(A,F1,T1),R1,Gs2,R1),
028   abresolve(happens(A,T1,T2),R1,[],R2),
029   abresolve(before(T2,T3),R2,[],R3),
030   append(Gs2,Gs1,Gs3),
031   add_neg([clipped(T1,F1,T3)],N1,N2),
032   abdemo_nafs(N2,R3,R4,N2,N3),
033   abdemo(Gs3,R4,R5,N3,N4).
```

Execution of the program mimics the (EC2) rule quite closely. In `Line 026`, holds_at (F1,T3) is part of the possible composite goal we want to prove. R1 and R5 are the input and output residues of Happens literals the tool is maintaining. N1 and N4 are the negated sub-goals that need to be proved as a result of adding Happens literal(s) into the residue (we shall revisit this point shortly). In `Line 27a`, F1 \= neg(F2) ensures that the goal is not a negated goal in the form of $\neg(F2)$, which is dealt with separately.

First, the program tries to prove the sub-goal initiates (A,F1,T1) by looking up the object-level clauses. initiates(A, F1,T1) may have its own sub-goals, and they are retrieved by Gs2 in `Line 27b` and added to the list of goals to prove in `Line 030`. The program attempts to resolve another sub-goal happens(A,T1,T2) in `Line 028` by adding happen literal to the residue *R2?*, and temporally order the happens literal in `Line 029`. In `Line 031` and `Line 032` the tool attempts to prove that the event that has just been added to the happen literal does not clip a fluent that has been proved to hold. The tool then continues to prove other sub-goals in a similar fashion in `Line 033`.

Although the program generally follows the EC rules closely, the order in which the certain sub-goals are resolved in the program is different. For example, resolving the initiates literals before happens prevents looping and minimizes the search space [29, 9].

Two simple queries shall illustrate the tool's functionality. For example, using the domain description in Fig. 6, if we query what needs to happen to satisfy the power control requirement at time $t$ by issuing the following query.

```
?- abdemo([holds_at(lights_off(lights),t)],H).
```

Shanahan's planner returns the following plan.

```
H = [[happens(switch_lights_off(lights), t1, t1)],
     [before(t1, t)]] ;
No
```

The planner finds only one model in the plan, and the model says that if the event switch_lights_off happens at a time $t1$ before $t$, then the lights will be on at time $t$. This is correct according to our refinement using the (EC2) rule in Section 3.3, (PCFb).

If now we add a predicate to our domain description to say that lights are off initially (axiom(initially(lights_off(L)),[])), and query for the same goal, the planner returns the following plan.

```
H = [[], []] ;
H = [[happens(switch_lights_off(lights), t1, t1)],
     [before(t1, t)]] ;
No
```

This time, the planner correctly finds two (disjoined) models. The first model says that if the lights are initially off (at time 0), (and nothing has happened since then), then they remain off (at time $t$). This model is coded in H = [[], []], and notice that it is not clear from the plan what that initial state might be. The second model involving the happens clause is same as the one in the previous query.

In essence, the planner of Shanahan abductively finds the temporally-ordered Happens literals for a given goal.

## 4.1 Refinement of requirements as planning

Conceptually, the relationship between a planner and the artifacts in our refinement is as follows:

- the *domain descriptions and EC meta-rules* in the planner are part of the *problem world domains* in our refinement,
- a *goal* the planner tries to prove is a *requirement*, and
- a plan the planner finds for a goal is part of our *specification*.

There is a close parallel between planning and our refinement. In our refinement, largely through effect to cause reasoning, we want to develop proofs that requirements can be satisfied, which is essentially what classic planners do. Therefore our refinement can be cast as a kind of abductive planning, so we use Shanahan's tool [29] as the basis to implement our extension of the tool.

However, there are some important differences between the two approaches. We note four points in particular:

1. In addition to Happens literals, our specifications also include Initially and Prohibit literals where appropriate.
2. In the planner, the initial state of the lights needs to be stated explicitly. However, in our refinement, the initial state is generated as a kind of precondition to the plan.
3. The fact that "nothing else happened since" some fluent started to hold is implicit in the tool, as it is in the EC reasoning. In our refinement, we derive in the Prohibit clause all the events that could potentially terminate a fluent that needs to hold.
4. The tool does not support the (EC3) rule involving the trajectory clause. This rule is required in our refinement (as we shall discuss in Section 5.3).

The EC meta-rules Shanahan adopted in [29] are a more general formulation of the rules suggested in [30], on which our rules are based. The rules (EC1) and (EC2) we used

here are equivalent to the rules (EC1) and (EC2) in [29], but other rules in [29] are not needed for our derivations. However, our rule (EC3) is not used in [29], and therefore part of the refinement involving the trajectory literal is not supported by Shanahan's tool.

# 5. EXTENDING THE TOOL

When implementing the tasks specific to our refinement, we first split the residue of literals into three parts: a residue of Initially literals (I), a residue of Happens literals (H), and a residue of Prohibit literals (P). The residue I describes the preconditions of the event sequence, the residue H describes the event sequence and the residue P describe guards for the fluents. In extending the tool, we will eventually include two further variables in the query command, representing the I and P residues.

```
?- abdemo([holds_at(lights_off(lights),t)],I,H,P).
```

This separation of residues allows us to preserve the integrity of original implementation and carefully control the changes we make. We will now discuss our implementation step by step, and in order to avoid confusion with Shanahan's original tool, our extended tool will be called `SpecPlanner` [1].

## 5.1 Implementing the Initially residue

Since the tool of Shanahan [29] requires the domain description to be explicit about the initial state of the system, and since our refinement *generates* the assumption about the initial state of the system, the tool needs to maintain a residue for Initially literals. To do that, we declare that the system may already be in whatever state it should be in by including the statement `axiom(initially(X),[])` in our domain descriptions. We then add a new variable for the initially literals, I, in our query.

Since the domain description says that the system may be initially in the required state, when the goal is proved according to (EC1), the Initially literal can be resolved immediately (`Line A21b`). The tool then simply adds the residue for Initially literals to I in (`Line A22`). Other sub-goals of (EC1) are proved in a way similar to those of (EC2) are proved in Section 4. Since Prolog proves the sub-goals sequentially, this ordering of sub-goals in (`Line A23a-A24`) is necessary.

```
A20 abdemo([holds_at(F1,T)|Gs1],I1,I3,
                          R1,R3,N1,N4) :-
A21a  F1 \= neg(F2),
A21b  abresolve(initially(F1),R1,Gs2,R1),
A22   append([initially(F1)],I1,I2),
A23a  append(Gs2,Gs1,Gs3),
A23b  add_neg([clipped(0,F1,T)],N1,N2),
A24   abdemo_naf([clipped(0,F1,T)],R1,R2,N2,N3),
A25   abdemo(Gs3,I2,I3,R2,R3,N3,N4).
```

If we now issue a query for the power control requirement, `SpecPlanner` generates the possible initial state of the system and describe it as a precondition in the residue I.

```
I = [initially(lights_off(lights))],
H = [[], []],

I = [],
H = [[happens(switch_lights_off(lights), t1, t1)],
     [before(t1, t)]],
No
```

In this specification, from the residue I, it becomes clear what the initial states of the system must be for each model. In the first model, the tools says that, the lights are initially off, nothing happened since then; therefore, the lights will be off at time $t$. In the second model, the initial state does not matter; occurrence of the event switch_lights_off at a time $t1$ before $t$ will keep the lights off at $t$.

Implementation of other EC rules does not need to maintain the Initially residue, and therefore they are not affected by this implementation in a significant way.

## 5.2 Implementing the Prohibit residue

Our notation of prohibited events is a further elaboration of a not clipped clause. Therefore, after having resolved the not clipped sub-goal, we need to identify events that could potentially terminate the fluent. First, we add a further variable P to our query for the Prohibit literals.

When the goal is proved using (EC1) for example, having proved the not clipped sub-goal (`Line B23b` and `Line B24`), the program attempts to maintain a residue of Prohibit literals in `Line B25`.

```
B20 abdemo([holds_at(F1,T)|Gs1],I1,I3,R1,R3,
                        P1,P3,N1,N4) :-
B21a  F1 \= neg(F2),
B21b  abresolve(initially(F1),R1,Gs2,R1),
B22   append([initially(F1)],I1,I2),
B23a  append(Gs2,Gs1,Gs3),
B23b  add_neg([clipped(0,F1,T)],N1,N2),
B24   abdemo_naf([clipped(0,F1,T)],R1,R2,N2,N3),
B25   abprohibit(0,F1,T,P1,P2),
B26   abdemo(Gs3,I2,I3,R2,R3,P2,P3,N3,N4).
```

The residue of Prohibit literals is maintained by querying all events that terminate the fluent that needs to hold. It uses the built-in Prolog function findall.

```
133a  abprohibit(T1,F1,T,P1,P7) :-
133b  findall(A2, axiom(terminates(A2,F1,T),[]),P6),
134    append([prohibit(P6,T1,T)],P1,P7).
```

Continuing with the implementation of (EC1), when we now issue a query for the requirement of the power control feature, the tool finds the following specification.

```
I = [initially(lights_off(lights))],
H = [[], []],
P = [prohibit([switch_lights_on(lights)], 0, t)] ;

I = [],
H = [[happens(switch_lights_off(lights), t1, t1)],
     [before(t1, t)]],
P = [prohibit([switch_lights_on(lights)], t1, t)] ;

No
```

Now `SpecPlanner` returns specifications complete with description of the possible initial states, event sequence for the machine domain, and events that should be prohibited in order to satisfy the requirement fully.

Similar changes are made to the implementation of (EC2) and our (EC3) to maintain the residue of Prohibit literals. Having implemented these changes, we finally need to implement our (EC3) rule involving the Trajectory predicate.

## 5.3 Implementing EC3

In our discussions so far, we have simplified parts of the problem domains, for instance, in stating that the event SwitchLightsOff causes LightsOff to be true instantaneously. As described in Fig. 1, realistic descriptions will have to consider more complex assumptions, such as the chain of causality and time delay from the point the event SwitchLightsOff is fired to the point the lights actually go off. Similarly, we also have to make explicit the assumptions that the light switches and light unit works properly. It involves revising parts of our domain descriptions as shown in Fig. 1 and Fig. 2, and obtaining revised specifications with the tool again.

Therefore, we can revise statements such as (L3) in Fig. 6, by saying that (i) the event SwitchLightsOff only sets the LightSwitchesTurnedOff *provided that the light switches are working properly*, (ii) when the light switches are turned off, the lights go off after a delay, *if the lights are working well*. Formally, we can replace the statement (L3) with (L3a-L3b) as follows.

$$\begin{aligned} &Initiates(SwitchLightsOff,\\ &LightSwitchesTurnedOff, \tau) \leftarrow \\ &HoldsAt(SwitchesWorkingOK, \tau) \end{aligned} \quad \text{(L3a)}$$

$$\begin{aligned} &Trajectory(LightSwitchesTurnedOff,\\ &\tau, LightsOff, 2) \leftarrow \\ &HoldsAt(LightUnitsWorkingOK, \tau) \end{aligned} \quad \text{(L3b)}$$

The statement (L3a) says that the event SwitchLightsOff, causes the fluent LightSwitchesTurnedOff to hold, *if the switches are working properly.* Similarly, the statement (L3b) says that when the fluent the fluent LightSwitchesTurnedOff starts to hold at time $\tau$, the lights will be off at time $\tau + 2$, provided that the light units are working properly. The derivation now needs to apply our (EC3) rule.

When implementing our (EC3) rule, we adopted a strategy similar to Shananan's to minimize the search space: the program first attempts to resolve initiates and trajectory immediately (Line 044), and their sub-goals are not resolved until after happens and before are resolved (Line 049 - Line 051).

```
041 abdemo([holds_at(F1,T3)|Gs1],I9,I11,
              R1,R7,P9,P12,N1,N4) :-
042   F1 \ = neg(F2),
043   append([],I9,I10),
044   abresolve(init_and_traj(A,F3,F1,T1,T9),
                              R1,Gs2,R1),
045   abresolve(happens(A,T1,T2),R1,[],R2),
056   abresolve(before(T2,T3),R2,[],R3),
047   abresolve(before(T9,T3),R3,[],R4),
048   abresolve(before(T2,T9),R4,[],R5),
049   append(Gs2,Gs1,Gs3),
050   add_neg([clipped(T1,F1,T3)],N1,N2),
051   abdemo_nafs(N2,R5,R6,N2,N3),
052   abprohibit(T1,F3,T9,P9,P10),
053   abprohibit(T9,F3,T3,P10,P11),
054   abdemo(Gs3,I10,I11,R6,R7,P11,P12,N3,N4).
```

We then introduced the rule to resolve the initiate and trajectory goal in pairs by looking up an event whose effect, when initiated, will eventually lead to the fluent that needs to hold. For example, given the description Initiates(a1,f1,t1), Trajectory(f1,t1,f2,d), if a1 happens at time $t1$, we know that f2 will hold at time $t1 + d$ (provided nothing else happens in the mean time to clip these fluents). Given a target goal f2, the tool attempts to prove that first, there is an appropriate Trajectory clause which leads to F2 holding (Line 059), and second there is an Initiate clause which causes the initial fluent f1 to hold (Line 060). The time delta in the trajectory clause is represented by D, for example in (Line 058).

```
058  abresolve(init_and_traj(A,F1,F2,T,D),
                                  R,Gs,R) :-
059     axiom(trajectory(F1,T,F2,D1),Gs2),
060     axiom(initiates(A,F1,T),Gs1),
061     append([],Gs2,Gs3),
062     append(Gs3,Gs1,Gs4),
063     append([],D1,D),
064     append([],Gs4,Gs).
```

Since our (EC3) allows refinement involving the trajectory predicate, this extension is important for our application of the tool to the smart home software.

# 6. EVALUATION OF SPECPLANNER

Having developed the tool, we applied it to obtain specifications for the requirements in smart home software. For example, the tool found the following specification for the power control feature. The specification says that the lights will be off at time $t$ if either (i) the lights were off at time 0 and nothing (including wear and tear of the devices) has happened since, or (ii) light switches and light units are all in order and nothing has happened to cause them otherwise, and lights were switched off at a time sufficiently before $t$, and switching on is prohibited between the time lights are switched off and time $t$.

```
I = [initially(lights_off(lights))],
H = [[], []],
P = [prohibit([switch_lights_on(lights)], 0, t)] ;

I = [initially(switches_working_ok(lights)),
     initially(light_units_working_ok(lights))],
H = [[happens(switch_lights_off(lights), t1, t1)],
     [before(t1, zero_plus(t1)),
      before(zero_plus(t1), t),
      before(t1, t)]],
P = [prohibit([], 0, t1),
     prohibit([], 0, t1),
     prohibit([switch_lights_on(lights)],
               zero_plus(t1), t),
     prohibit([switch_lights_on(lights)],
               t1, zero_plus(t1))] ;

No
```

Notice that the first part of the specification (the first set of I, H, P clauses above) are same as the first partial specification (PCFa) we obtained in Section 3.3. In our elaboration of the domain descriptions in Section 5.3, we have introduced the trajectory predicates into the descriptions,

for example, in replacing (L3) with (L3a-L3b). Therefore, the tool also found a second model (the second set of I, H, P clauses above) according to our implementation of our (EC3) rule.

Since the new fluents switches_working_ok and light_units_working_ok are not initiated or terminated by events, our tool `SpecPlanner` makes explicit the assumptions that these fluents hold. Explicating and documenting such domain assumptions are an important part of formally obtaining specifications from requirements [28].

As well as deriving the specifications for requirements expressed in terms of individual fluents, we can also use the tool to find specifications for more complex, conjunctive goals. It is one of the significant strengths of Shanahan's tool that we maintained in `SpecPlanner`. For example, for the requirement for the away-from-home feature, we can find specification for HoldsAt(LightsOn,t4) and HoldsAt(LightsOff,t3) together.

We applied `SpecPlanner` to obtain specifications of smart home application involving requirements for six features and several problem world domains (there are various ways to count them). In all cases, `SpecPlanner` found correct specifications.

## 6.1 Runtime resolution of conflicts

Having obtained specifications for individual requirements using the tool, we now briefly discuss how a runtime mechanism we proposed in [20] makes use of these specifications in resolving conflicts dynamically. It involves the use of a mediator called Composition Controller, which listens to system events and act on them according to a scheme expressed by a weakened conjunction operator. For example, given two conflicting requirements such as AfH and PC, there are four different ways in their conflict is handled.

1. AfH $\wedge_{\{any\}}$ PC – when conflicts happen, the system need not apply any control because any emergent behaviour is acceptable
2. AfH $\wedge_{\{control\}}$ PC – AfH and PC may prohibit each other's events to gain mutually exclusive control of a domain, where exclusion is symmetrical
3. AfH $\wedge_{\{AfH\}}$ PC – similar to exclusion, except that AfH's control of the domain has a higher priority over PC
4. AfH $\wedge_{\{important,AfH\}}$ PC – similar to AfH $\wedge_{\{AfH\}}$ PC, except that the event *important*, regardless of the specification that generates it, has the highest priority

The mediator then listens out for the events generated by the machines of AfH and PC, as well as other problem world domains, to monitor their states. When events are generated, the mediator filters those events to domains according to the conjunction operator chosen. For example, if AfH $\wedge_{\{AfH\}}$ PC is chosen, and when PC wants to keep the lights off whilst AfH has them on, the mediator will not pass on the switch_lights_off event from the PC machine to the switch domain during that time.

## 6.2 Performance Issues

Shanahan wrote the original tool in LPA MacProlog 32. We wrote our extensions using SWI-Prolog version 5.6.27 [6]. Our program runs on a laptop, Sony Vaio VGN-TX1XP (CPU 1.20GHz) running Windows XP. Table 2 shows the average runtime statistics, in CPU seconds, for the tests we carried out on the tool. These statistics are not meant to claim efficiency over other implementations similar tools, but rather to demonstrate the scalability of `SpecPlanner`. The results show the time does not increase exponentially when domain predicates are added and more complex derivations are performed.

**Table 2: Runtime Statistics**

| **Domain Predicates** | 8 | 8 | 8 | 12 | 12 | 12 |
|---|---|---|---|---|---|---|
| **Requirements** | 1 | 2 | 3 | 1 | 2 | 3 |
| **Time** | 5 | 8 | 9 | 9 | 10 | 14 |

# 7. LIMITATIONS AND FURTHER WORK

SpecPlanner has some limitations; in particular:

1. We have implemented only the EC rules necessary to derive specifications for the smart home software. There are several other EC rules [24], implementation of which will make the tool more widely applicable. We believe that such extensions may be implemented in the same way we have implemented our (EC3).
2. Our extension of Shanahan's tool work only with positive goals, again, because our application does not require using negative goals. However, such an extension will be relatively straightforward to implement, as Shanahan's tool already supports it.

Requirements engineering often work with requirements expressed in an informal language. A more intuitive interface for the tool, supported by a semi-formal language to communicate with the users, may facilitate application of the tool. For example, an editor to create syntactically correct descriptions would be useful.

# 8. RELATED WORK

There are few tools to support systematic derivation of specifications from requirements using abductive temporal logic. Seater and Jackson [28] use first-order relational logic and the Alloy Analyzer to analyze transformation of requirements into specifications. However, our choice of temporal predicate logic in our refinement allows explicit reasoning about time.

The Event Calculus has previously been used in software development for reasoning about evolving specifications [11, 27], and distributed systems policy specifications [7]. Our work should be seen as complementary to such approaches in that it will allow inconsistencies to be resolved at runtime.

Various specification analysis tools exist; Lespérance et al [21] and Heitmeyer et al [13], for example, propose tool suites to perform specific analyses tasks, such as consistency checks. However, they are less concerned with automated derivation of specifications. Tools such as Specware [5] also focus on refinement of programs from specifications, rather than specifications from requirements.

There are several tools to support goal-oriented approaches to requirements engineering [31, 32, 33] which are supported by tools such as [10, 3, 2]. However, these tools perform user-assisted goal decompositions and constraint-satisfaction checks

rather than automatic derivation of specifications from given descriptions of requirements and the system's environment.

## 9. CONCLUSIONS AND CONTRIBUTIONS

We believe that the use of a planner in a systematic discovery of specifications, particularly within the context of pervasive software, is novel. The specifications, in this context, need to be augmented with information necessary for runtime conflict detection. Shanahan laid the groundwork with the planner [29], but we have introduced significant changes to make the tool relevant to our application. Our contributions, in terms of implementation and integration, are (i) the residues of Initially, (ii) Prohibit literals, and (iii) our (EC3) rule.

## 10. ACKNOWLEDGEMENTS

We would like to thank our colleagues Arosha Bandara and Yijun Yu at the Open University for their comments and suggestions. This research is funded by EPSRC.

## 11. REFERENCES

[1] http://mcs.open.ac.uk/ttt23/ase/.
[2] http://www.cs.toronto.edu/km/openome/.
[3] http://www.objectiver.com/.
[4] http://www.signiform.com/csr/ecp.html.
[5] http://www.specware.org/.
[6] http://www.swi-prolog.org/.
[7] A. K. Bandara, E. Lupu, and A. Russo. Using event calculus to formalise policy specification and analysis. In *POLICY*, pages 26–39. IEEE Computer Society, 2003.
[8] I. Bratko. *Prolog (3rd ed.): programming for artificial intelligence*. Addison-Wesley, 2001.
[9] L. Chittaro and A. Montanari. Efficient temporal reasoning in the cached event calculus. *Computational Intelligence*, 12:359–382, 1996.
[10] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. Grail/kaos: An environment for goal-driven requirements engineering. In *ICSE*, pages 612–613, 1997.
[11] A. S. d'Avila Garcez, A. Russo, B. Nuseibeh, and J. Kramer. Combining abductive reasoning and inductive learning to evolve requirements specifications. *IEE Proceedings - Software*, 150(1):25–38, 2003.
[12] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. E. Anderson, B. N. Bershad, G. Borriello, S. D. Gribble, and D. Wetherall. System support for pervasive applications. *ACM Trans. Comput. Syst.*, 22(4):421–486, 2004.
[13] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, 1996.
[14] M. Jackson. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*. ACM Press, 1995.
[15] M. Jackson. *Problem Frames: Analyzing and structuring software development problems*. ACM Press & Addison Wesley, 2001.
[16] M. Kolberg, E. H. Magill, and M. Wilson. Compatibility issues between services supporting networked appliances. *IEEE Communications Magazine*, 41(11):136–147, 2003.
[17] T. Koskela and K. Väänänen-Vainio-Mattila. Evolution towards smart home environments: empirical evaluation of three user interfaces. *Personal Ubiquitous Comput.*, 8(3-4):234–240, 2004.
[18] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Gen. Comput.*, 4(1):67–95, 1986.
[19] R. Laney, L. Barroca, M. Jackson, and B. Nuseibeh. Composing requirements using problem frames. In *Proceedings of RE'04*, pages 122–131. IEEE Computer Society, 2004.
[20] R. Laney, T. T. Tun, M. Jackson, and B. Nuseibeh. Composing features by managing inconsistent requirements. *Manuscript Submitted to International Conference on Feature Interactions*, 2007.
[21] Y. Lespérance, T. G. Kelley, J. Mylopoulos, and E. S. K. Yu. Modeling dynamic domains with congolog. In M. Jarke and A. Oberweis, editors, *CAiSE*, volume 1626 of *LNCS*, pages 365–380. Springer, 1999.
[22] J. McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 28:89–116, 1986. Reprinted in [23].
[23] J. McCarthy. *Formalization of common sense, papers by John McCarthy edited by V. Lifschitz*. Ablex, 1990.
[24] R. Miller and M. Shanahan. The event calculus in classical logic - alternative axiomatisations. *Journal of Electronic Transactions on Artificial Intelligence*, 1999.
[25] S. H. Park, S. H. Won, J. B. Lee, and S. W. Kim. Smart home - digitally engineered domestic life. *Personal Ubiquitous Comput.*, 7(3-4):189–196, 2003.
[26] L. Rapanotti, J. G. Hall, and Z. Li. Deriving specifications from requirements through problem reduction. *IEE Proceedings Software*, 153(5):183–198, 2006.
[27] A. Russo, R. Miller, B. Nuseibeh, and J. Kramer. An abductive approach for analysing event-based requirements specifications. In P. J. Stuckey, editor, *ICLP*, volume 2401 of *LNCS*, pages 22–37. Springer, 2002.
[28] R. Seater and D. Jackson. Requirement progression in problem frames applied to a proton therapy system. In *Proceedings of RE'06*, pages 166–175, Washington, DC, USA, 2006. IEEE Computer Society.
[29] M. Shanahan. An abductive event calculus planner. *J. Log. Program.*, 44(1-3):207–240, 2000.
[30] M. P. Shanahan. The event calculus explained. In M. J. Woolridge and M. Veloso, editors, *Artificial Intelligence Today, Lecture Notes in AI no. 1600*, pages 409–430. Springer, 1999.
[31] A. van Lamsweerde, R. Darimont, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In *RE*, pages 194–203. IEEE Computer Society, 1995.
[32] A. van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Trans. Software Eng.*,

24(12):1089–1114, 1998.

[33] E. S. K. Yu and J. Mylopoulos. Understanding "why" in software process modelling, analysis, and design. In *ICSE*, pages 159–168, 1994.