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 specifications 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 home-owners 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 20XX ACM X-XXXX-XX/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 (\( S \)) 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-
requirements 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 LightSwitchsTurnedOff 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 (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 develop 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].

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

<table>
<thead>
<tr>
<th>Table 1: Elementary Predicates of the Event Calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formula</td>
</tr>
<tr>
<td>Meaning</td>
</tr>
<tr>
<td>$\text{Initiates}(\alpha, \beta, \tau)$</td>
</tr>
<tr>
<td>$\text{HoldsAt}(\beta, \tau_1)$</td>
</tr>
<tr>
<td>$\text{Terminates}(\alpha, \beta, \tau)$</td>
</tr>
<tr>
<td>$\text{Happens}(\alpha, \tau)$</td>
</tr>
<tr>
<td>$\text{Trajectory}(\beta_1, \tau, \beta_2, \delta)$</td>
</tr>
<tr>
<td>$\text{Clipped}(\tau_1, \beta, \tau_2)$</td>
</tr>
</tbody>
</table>

$$\text{HoldsAt}(\beta, \tau_1) \leftarrow \text{Initially}(\beta) \land \neg \text{Clipped}(0, \beta, \tau_1) \quad (\text{EC1})$$

$$\text{HoldsAt}(\beta, \tau_2) \leftarrow \text{Happens}(\alpha, \tau_1) \land \text{Initiates}(\alpha, \beta, \tau_1) \land \tau_1 < \tau_2 \land \neg \text{Clipped}(\tau_1, \beta, \tau_2) \quad (\text{EC2})$$

$$\text{HoldsAt}(\beta, \tau_3) \leftarrow \text{Happens}(\alpha, \tau_1) \land \text{Initiates}(\alpha, \beta, \tau_1) \land \text{Trajectory}(\beta_1, \tau_1, \beta, \delta) \land \tau_2 = \tau_1 + \delta \land \tau_1 < \tau_2 \leq \tau_3 \land \neg \text{Clipped}(\tau_1, \beta_1, \tau_2) \land \neg \text{Clipped}(\tau_2, \beta, \tau_3) \quad (\text{EC3})$$

$$\text{Clipped}(\tau_1, \beta, \tau_2) \leftarrow \exists \alpha, \tau [\text{Happens}(\alpha, \tau) \land \tau_1 < \tau < \tau_2 \land \text{Terminates}(\alpha, \beta, \tau)] \quad (\text{DEF1})$$

**Figure 3: Event Calculus Meta-rules**
HoldsAt(HouseEmpty, t5) ∧ ¬Clipped(t5, HouseEmpty, t6) ∧ 
\( t5 + 4 < t < t6 \) → HoldsAt(LightsOff, t)

\((\text{HoldsAt(HouseEmpty, t5)}) \land
\neg(\text{Clipped(t5, HouseEmpty, t6)}) \land\)
\( t5 + 119 < t4 < t6 \land 0 \leq t4 \mod 60 \leq 29 \) →
HoldsAt(LightsOff, t4) \)

\( (\text{HoldsAt(HouseEmpty, t5)}) \land\)
\( \neg(\text{Clipped(t5, HouseEmpty, t6)}) \land\)
\( t5 + 119 < t3 < t6 \land 30 \leq t3 \mod 60 \leq 59 \) →
HoldsAt(LightsOff, t3) \)

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 \(\beta\) 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 \(\tau1\) and \(\tau2\) if and only if there is an
event \(\alpha\) that happens between \(\tau1\) and \(\tau2\) 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 con-
clude 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 hy-
pothesize 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 com-
pleteness 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
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 for-
malize 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 for-
malized 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 (AH)
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 do-
 mains 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 descrip-
tions 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 do-
 mains. The definitions (L1-L4) say that the lights can be

Figure 5: Abstracting the Lights domain

\begin{itemize}
  \item \text{Initiates(SwitchLightsOn, LightsOn, } \tau) \quad \text{(L1)}
  \item \text{Terminates(SwitchLightsOn, LightsOff, } \tau) \quad \text{(L2)}
  \item \text{Initiates(SwitchLightsOff, LightsOff, } \tau) \quad \text{(L3)}
  \item \text{Terminates(SwitchLightsOff, LightsOn, } \tau) \quad \text{(L4)}
  \item \text{Initiates(DetectEmpty, EmptyDetected, } \tau) \rightarrow\text{HoldsAt(HouseEmpty, } \tau) \quad \text{(OSHE1)}
  \item \text{Terminates(DetectEmpty, OccupiedDetected, } \tau) \quad \text{(OSHE2)}
  \item \text{Initiates(DetectOccupied, OccupiedDetected, } \tau) \rightarrow\text{HoldsAt(HouseOccupied, } \tau) \quad \text{(OSHE3)}
  \item \text{Terminates(DetectOccupied, EmptyDetected, } \tau) \quad \text{(OSHE4)}
\end{itemize}
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.

\[
\text{HoldsAt}(\text{HouseEmpty}, t_5) \land \\
\neg \text{Clipped}(t_5, \text{HouseEmpty}, t_6) \land \\
t_5 + 4 < t < t_6 \rightarrow \\
\text{HoldsAt}(\text{LightsOff}, t)
\]

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

Initially(LightsOff) \land \\
\neg \text{Clipped}(0, \text{LightsOff}, t)

We now have an Initially clause and a \(\neg\text{Clipped}\) clause. Initially clauses are treated as a sort of precondition to the event sequences, and therefore Initially(LightsOff) is no refined further.

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

Initially(LightsOff) \land \\
\neg \exists a_1, t_1 \cdot \text{Happens}(a_1, t_1) \land \\
\text{Terminates}(a_1, \text{LightsOff}, t_1) \land \\
0 < t_1 < t

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.

Initially(LightsOff) \land \\
\neg \exists a_1, t_1 \cdot \text{Happens}(SwitchLightsOn, t_1) \land \\
\text{Terminates}(SwitchLightsOn, \text{LightsOff}, t_1) \land \\
0 < t_1 < t

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

Initially(LightsOff) \land \\
\neg \exists a_1, t_1 \cdot \text{Happens}(SwitchLightsOn, t_1) \land \\
0 < t_1 < t

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

\[
\text{Prohibit}(\alpha, \tau_1, \tau_2) \equiv \neg \exists a_0, \tau \cdot \text{Happens}(\alpha, \tau) \land \tau_1 < \tau < \tau_2
\]

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.

\[
\text{Initially}((\text{LightsOff}) \land \\
\text{Prohibit}(\text{SwitchLightsOn}, 0, t))
\]

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.

\[
\text{HoldsAt}(\text{HouseEmpty}, t_5) \land \\
\neg \text{Clipped}(t_5, \text{HouseEmpty}, t_6) \land \\
t_5 + 4 < t < t_6 \rightarrow \\
\text{Initially}((\text{LightsOff}) \land \\
\text{Prohibit}(\text{SwitchLightsOn}, 0, t))
\]

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 \(t_1\) before \(t\) and the event SwitchLightsOn is prohibited between \(t_1\) and \(t\), then the lights will be off at time \(t\).

\[
\text{HoldsAt}(\text{HouseEmpty}, t_5) \land \\
\neg \text{Clipped}(t_5, \text{HouseEmpty}, t_6) \land \\
t_5 + 4 < t < t_6 \rightarrow \\
\text{Happens}(\text{SwitchLightsOff}, t_1) \land t_1 < t \land \\
\text{Prohibit}(\text{SwitchLightsOn}, t_1, t)
\]

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.

\[
\text{HoldsAt}(\text{HouseEmpty}, t_5) \land \\
\neg \text{Clipped}(t_5, \text{HouseEmpty}, t_6) \land \\
t_5 + 4 < t < t_6 \rightarrow \\
((\text{Initially}((\text{LightsOff}) \land \\
\text{Prohibit}(\text{SwitchLightsOn}, 0, t))) \land \\
\text{Happens}(\text{SwitchLightsOff}, t_1) \land t_1 < t \land \\
\text{Prohibit}(\text{SwitchLightsOn}, t_1, t))
\]

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.

\[
\text{holds}_{\text{at}}(F_1, T_3) := \text{happens}(A, T_1, T_2), T_2 < T_3, \text{initiates}(A, F_1, T_1), \not\text{not clipped}(T_1, F_2).
\]

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

\[
026 \hspace{1em} \text{abdemo}(\text{holds}_{\text{at}}(F_1, T_3) | Gs_1), R_1, R_5, N_1, N_4) := \\
027a \hspace{1em} F_1 \not\text{\neg neg}(F_2), \\
027b \hspace{1em} \text{abresolve}\text{-}\text{initiates}(A, F_1, T_1), R_1, Gs_2, R_1, \\
028 \hspace{1em} \text{abresolve}\text{-}\text{happens}(A, T_1, T_2), R_1, [], R_2, \\
029 \hspace{1em} \text{abresolve}(\text{before}(T_2, T_3), R_2, [], R_3), \\
030 \hspace{1em} \text{append}(Gs_2, Gs_1, Gs_3), \\
031 \hspace{1em} \text{add}\_\text{-}\text{neg}([\text{\neg clipped}(T_1, F_1, T_3)], N_1, N_2), \\
032 \hspace{1em} \text{abdemo}\_\text{-}\text{nafs}(N_2, R_3, R_4, N_2, N_3), \\
033 \hspace{1em} \text{abdemo}(Gs_3, R_4, R_5, N_3, N_4).
\]

Execution of the program mimics the (EC2) rule quite closely. In Line 026, \text{holds}_{\text{at}}(F_1, T_3) is part of the possible composite goal we want to prove. \text{R}_1 and \text{R}_5 are the input and output residues of \text{Happens} literals the tool is maintaining. \text{N}_1 and \text{N}_4 are the negated sub-goals that need to be proved as a result of adding \text{Happens} literal(s) into the residue (we shall revisit this point shortly). In Line 27a, \text{F}_1 \not\text{\neg neg}(\text{F}_2) ensures that the goal is not a negated goal in the form of \text{\neg (F}_2\text{)}), which is dealt with separately.

First, the program tries to prove the sub-goal \text{initiates}(A, F_1, T_1) by looking up the object-level clauses. \text{initiates}(A, F_1, T_1) may have its own sub-goals, and they are retrieved by \text{Gs}_2 in Line 27b and added to the list of goals to prove in Line 030. The program attempts to resolve another sub-goal \text{happens}(A, T_1, T_2) in Line 028 by adding happen literal to the residue \text{R}_2, and temporally order the \text{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 \text{initiates} literals before \text{happens} prevents looking 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.

\[
? - \text{abdemo}([\text{holds}_{\text{at}}(\text{lights}_{\text{off}}(\text{lights}), t)], N).
\]

Shanahan’s planner returns the following plan.

\[
H = [[\text{happens}(\text{switch}_{\text{lights}_{\text{off}}}(\text{lights}), t_1, t_1)], \\
\text{[before}(t_1, t_1)]].
\]

No

The planner finds only one model in the plan, and the model says that if the event \text{switch}_{\text{lights}_{\text{off}}} happens at a time \(t_1\) 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 we add a predicate to our domain description to say that lights are off initially (\text{axiom}(\text{initially}(\text{lights}_{\text{off}}(L)), [])), and query for the same goal, the planner returns the following plan.

\[
H = [[], []]; \\
H = [[\text{happens}(\text{switch}_{\text{lights}_{\text{off}}}(\text{lights}), t_1, t_1)], \\
\text{[before}(t_1, t_1)]].
\]

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 temporarily-ordered \text{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 \text{Happens} literals, our specifications also include \text{Initially} and \text{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 the \text{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 describes 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 (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 abprohibit(0,F1,T,P1,P2),
A26 abdemo(Gs3, I2, I3, R2, P2, P3, N3, N4).

If we now issue a query for the power control requirement, SpecPlanner generates the possible initial state of the system and describes 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 tool 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 t 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, 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([abprohibit(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 = [abprohibit([switch_lights_on(lights)], 0, t)];
I = [],
H = [[happens(switch_lights_off(lights), t1, t1)],
  [before(t1, t)]],
P = [abprohibit([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.

\[
\begin{align*}
\text{Initiates} & (\text{SwitchLightsOff}, \text{LightSwitchesTurnedOff}, \tau) \quad \text{— (L3a)} \\
\text{HoldsAt} & (\text{SwitchesWorkingOK}, \tau) \\
\text{Trajectory} & (\text{LightSwitchesTurnedOff}, \tau, \text{LightsOff}, 2) \quad \text{— (L3b)} \\
\text{HoldsAt} & (\text{LightUnitsWorkingOK}, \tau)
\end{align*}
\]

The statement (L3a) says that the event SwitchLightsOff caused 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 are resolved (Line 049 - before Line 051).

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).

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 \( t \).

\[
\begin{align*}
I & = [\text{initially(lights_off(lights))}], \\
H & = [\text{[]} , \text{[]} ] , \\
P & = [\text{prohibit([switch_lights_on(lights)], 0, t)]) ; \\
I & = [\text{initially(switches_working_ok(lights))}], \\
H & = [\text{[happens(switch_lights_off(lights), t1, t1)]}, \\
& \text{[before(t1, zero_plus(t1))],} \\
& \text{[before(t1, t1)]}], \\
P & = [\text{prohibit([], 0, t1)]}, \\
& \text{prohibit([], 0, t1)]}, \\
& \text{prohibit([switch_lights_on(lights)], zero_plus(t1), t)} , \\
& \text{[before(t1, zero_plus(t1))]} ; \\
& \text{[before(t1, t1)]}]
\end{align*}
\]

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 $\text{HoldsAt(LightsOn,t4)}$ and $\text{HoldsAt(LightsOff,t3)}$ 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 $\text{AfH}$ and $\text{PC}$, there are four different ways in their conflict is handled.

1. $\text{AfH} \land \{\text{temp}\} \text{PC}$ – when conflicts happen, the system need not apply any control because any emergent behaviour is acceptable

2. $\text{AfH} \land \{\text{contrad}\} \text{PC} – \text{AfH}$ and $\text{PC}$ may prohibit each other’s events to gain mutually exclusive control of a domain, where exclusion is symmetrical

3. $\text{AfH} \land \{\text{AfH}\} \text{PC}$ – similar to exclusion, except that $\text{AfH}$’s control of the domain has a higher priority over $\text{PC}$

4. $\text{AfH} \land \{\text{important, AfH}\} \text{PC}$ – similar to $\text{AfH} \land \{\text{AfH}\} \text{PC}$, 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 $\text{AfH}$ and $\text{PC}$, 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 $\text{AfH} \land \{\text{AfH}\} \text{PC}$ is chosen, and when $\text{PC}$ wants to keep the lights off whilst $\text{AfH}$ has them on, the mediator will not pass on the $\text{switch\_lights\_off}$ event from the $\text{PC}$ 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>
<thead>
<tr>
<th>Domain Predicates</th>
<th>8</th>
<th>8</th>
<th>12</th>
<th>12</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Time</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

### 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