Specifying Software Features for Composition: A Tool-Supported Approach

Thein Than Tun[a,*], Robin Laney[a], Yijun Yu[a], Bashar Nuseibeh[a,b]

[a] *The Open University, Milton Keynes, UK*
[b] *Lero, Limerick, Ireland*

Abstract

Development of several computing and communication technologies is enabling the widespread availability of pervasive systems. In smart home applications, household appliances—such as security alarms, heating systems, doors and windows—are connected to home digital networks. These applications offer features that are typically developed by disparate vendors, and when composed together, these features are expected to work together harmoniously. Engineering these systems poses two main challenges. The first challenge is: how can developers of individual features specify the features in order to make them composable with other hitherto unknown features? The second challenge is: when composition of features does not produce the desired behaviour, what can be done to resolve this non-intrusively? This article argues that the two issues are intrinsically related, and proposes an approach that addresses the first challenge in a way that makes the second challenge manageable. In particular, we describe a way of writing feature specifications in which assumptions about the problem world are made explicit. These feature assumptions can then be evaluated at runtime in order to preserve the desired system behaviour to the extent possible. Our approach is illustrated with examples from smart home applications.

Keywords:
Smart Home Applications, Feature Interactions, Feature Composition

1. Introduction

Convergence of several computing and communication technologies is enabling the development of pervasive systems. In smart home applications, household appliances—such as air conditioners, security alarms, doors and windows—are controlled by software systems through both wired and wireless home digital

*Corresponding author
Email address: thein.tun@open.ac.uk (Thein Than Tun)

networks (Grimm et al., 2004; Park et al., 2003; Kolberg et al., 2003). The systems have mobile and autonomous devices that communicate with each other by sending and receiving discrete signals. These are distributed event-based systems, components of which are developed independently. Smart home applications are used for several purposes including, but not limited to, health, entertainment, security, and education. Several consumer electronics manufacturers such as Siemens and Philips are developing such applications, while light-weight operating systems for such applications are also becoming available from software vendors (Microsoft, 2012; LinuxMCE, 2012).

Smart home applications typically offer *features* which represent units of user accessible functionality of the system. For example, a security *feature* of a smart home application may switch on and off lights when home occupants are away to give an impression that the house is occupied. Developing such event-based software systems posses two main challenges. First, the requirements of individual features are rooted in their environments in the sense that they are expressed in terms of the property of the system rather than the software. For example, the requirement for a security feature might be to "keep the window shut at night", rather than to implement the instruction "IF time=20:00:00 THEN Call tiltIn", although that may turn out to be part of the specification. A challenge here is to obtain, if possible, a correct specification of the software from the description of a desired property of the system.

Second, different features of the application are developed by separate vendors, but when put together in a particular environment they are expected to work together harmoniously (Kolberg et al., 2003). For example, the entertainment feature, developed by one vendor, may allow the user to record TV programmes according to a predetermined schedule, and the security feature, developed by another vendor, may allow the user to automatically capture the video images when an intrusion is detected. Such subtle behavioural inconsistencies are difficulty to identify and resolve at development time. A challenge here is to provide a mechanism that can detect behavioural inconsistencies and resolve them at runtime to the extent possible.

The main focus of this article is to address the first challenge of specifying individual features. The proposed approach, however, aims to tackle the first challenge in a way that helps address the second challenge, which has been discussed in greater detail by Laney et al. (2007).

The conceptual framework of Problem Frames (Jackson, 1995, 2001) characterizes the relationship between the specification (S), the problem world domains (W) and the requirement (R) as $W, S \vdash R$. The entailment operator ($\vdash$) emphasizes the fact that specifications rely on explicit domain properties in satisfying the requirements. Typically in software development, W and R are given, and the challenge is to find a correct and constructive S if possible. In event-based temporal systems, such as smart home applications, a specification describes the behaviour of a software component that observes and manipulates properties of the problem world through events. In dealing with the first challenge, we suggest that the correct specification for features can be obtained through logical abduction and model abstraction. Given appropriate descrip-

tions of W and R of a particular feature, logical abduction generates all possible scenarios of a feature specification satisfying the requirement. We use a form of temporal logic, called the Event Calculus, to describe W and R and use Decreasoner (Mueller, 2006) to perform abductive reasoning. Based on the scenarios generated by the tool, the developer provides rules for the specifications, which should be an abstraction of the validated scenarios. Correctness of the specification with respect to the requirement can then be checked automatically.

In the process of obtaining specifications, the problem world assumptions of the specifications are made explicit in a way that they can be evaluated at runtime. Making problem world assumptions explicit in specifications is difficult for the following reason: as suggested by the conceptual framework of Problem Frames, specifications must be described only in terms of events that can be controlled and observed by the software. We use a predicate called *prohibit* (Laney et al., 2007) to express problem world assumptions in terms of specification events. These specifications with problem world assumptions are monitored by a runtime composition controller that detects potential conflicts and enforces user preferences to resolve them.

This article builds on several previous works, some by the authors. Zave and Jackson (1997) make the distinction between requirements and specifications. The notion of Composition Frames as a way to compose inconsistent requirements is introduced by Laney et al. (2004), and the Event Calculus is used to manually refine requirements into specifications in (Laney et al., 2007). Using the Event Calculus to reason about problem diagrams is discussed in (Classen et al., 2008) whilst a way of detecting interactions between subproblems and tool-support for it are presented by Tun et al. (2009b).

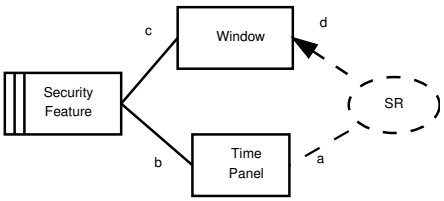
The remainder of the article is organized as follows. Section 2 provides the background discussion before the proposed approach is described in Section 3. Tool-assisted derivation of feature specifications and the composition controller is detailed in Sections 4 and 5. Finally, related work and conclusions can be found in Sections 6 and 7 respectively.

2. Preliminaries

This section begins by characterizing and introducing a simple example from the smart home application. We then discuss the key concepts used in the proposed approach, namely, the Problem Frames approach to requirements analysis, and the Event Calculus formalism to describe problem diagrams.

2.1. Smart Home Application

The smart home applications considered in this article are event-based temporal systems involving autonomous agents, many of which are controlled by software systems. There are hardware/software devices, as well as human agents, communicating with each other largely through instances of event types (or simply 'events'). Each event type is controlled either by the software components or the problem world domains, but not both. Although domains communication with each other through events, in some cases, problem world domains



Interface Name	*Phenomena Set*
a:	TiP!{Night, Day}
b:	TiP!{NightStarted(), DayStarted()}
c:	SF!{TiltIn(), TiltOut()}
d:	W!{Open, Shut}

Figure 1: Problem Diagram: Security Feature

may also expose their states directly to other agents. We assume partial ordering of events and more than one event may occur at a time within the system. A linear time where all time points can be mapped to non-negative integer values is also assumed.

Smart home applications provide features that may have been developed independently by different vendors. A feature in this case is a software component that brings about certain behaviour of the smart home application in order to satisfy a user requirement. Requirements are usually expressed in terms of properties of the problem world, in this case, the home environment. Specifications of the software components are expressed in terms of the events the software can fire, and the events of the environment the software can observe.

Users of smart home applications may buy features separately and put them together in different ways. Therefore it is important to ensure that the specification of the feature is not only correct, but also composable at runtime.

2.2. Problem Frames

According to the conceptual framework of Problem Frames (Jackson, 2001), requirements for software systems can be analysed by applying some principles, two of which are relevant to our discussions here.

One principle of Problem Frames is concerned with the properties of software artefacts in requirements engineering. Intuitively, it suggests that requirements (R) are expressed in terms of properties of their environment (or problem world domains) (W), and specifications (S), and within the context of problem world domains, are expected to satisfy the requirements. This relationship can be described as the logical entailment $W, S \vdash R$ (Jackson, 2001).

The problem diagram for the smart home security feature in Fig. 1 can be used to illustrate this characterization of artifacts. The diagram shows a high-level relationship between the requirement R, denoted by a dotted oval, the problem world domains W, denoted by plain rectangles and solid lines, and the specification S denoted by a box with a double stripe. W represents the properties of the problem world that are necessarily true, and R represent the properties of the problem world that users wish to hold true, whilst S represents the properties of a computer that will enact the required properties in that problem world context. S is also called the specification of the *machine*.

The requirement and problem world domains in Fig. 1 can be described informally as follows. The *requirement* for the security feature (SR) is to have

Table 1: Elementary Predicates of the Event Calculus

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

the window shut during the night. The requirement is expressed in terms of problem world properties: it *references* the property of the domain Time Panel that it is night (the undirected dotted line a) and *constrains* the property of the domain Window that it is shut (the directed dotted arrow d). The requirement, therefore, is a desired relationship between the property of the time panel and the window.

The *problem world domain* Window in Fig. 1 has the following given properties. The event TiltIn() makes the window shut. The event TiltOut makes the window open. The window cannot be both open and shut at the same time.

Domain interfaces such as b and c, as in all problem diagrams, are represented by undirected solid lines because they do not indicate any data flow, but instead, indicate the sharing of states and events. Those states and events are controlled by one domain (indicated by the symbol ! suffixing the domain initials), and observed by the other implicit domain. For instance, at the interface b, the time panel controls the events NightStarted() and DayStarted(), denoted by TiP!, and since the interface b connects the time panel with the machine Security Feature, the machine observes those events when they are generated. Similarly, at the interface c, Security Feature can fire the events TiltIn() and TiltOut(), denoted by SF!, which are observed by the window.

We assume that problem world domains communicate with each other by sending and receiving events, while the state properties are internal to the problem world domains.

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 problems. Therefore individual features may be inconsistent. This principle nicely fits the development practice of disparately constructing features before considering how they might be composed. Therefore, Problem Frames allow individual feature specifications to be initially inconsistent with each other.

We will use Composition Frames (Laney et al., 2004) to compose specifications by detecting and resolving runtime inconsistencies. Composition operators of Composition Frames can intercept events from the individual features and block certain events in order to resolve interactions between features.

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

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

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

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

Figure 2: Event Calculus Domain Independent rules

2.3. The Event Calculus

The Event Calculus (EC) is a system of logical formalism which draws from first-order predicate calculus. We chose EC as our formalism, because it is suitable for describing and reasoning about event-based temporal systems. Several variations of EC have been proposed, dealing with system properties such as nondeterminism, concurrency, and continuous change, as well as different modes of reasoning, such as abduction and default reasoning. Many of these capabilities of the Event Calculus have been surveyed by Mueller (2006). In general, the Event Calculus is appropriate for analysing the dynamic behaviour of the smart home application, while the analysis of static structures such as numbers and arrays may be best served by other relevant formalisms.

The calculus relates events and event sequences to 'fluents', time-varying properties, which denote states of a system. Table 1, based on Miller and Shanahan (1999), gives the meanings of the elementary predicates of the calculus we use in this article. The domain-independent rules in Fig. 2, taken from Miller and Shanahan (1999), state that: $Clipped(t1, f, t2)$ is a notational shorthand to say that the fluent f is terminated between times $t1$ and $t2$ (EC1), $Declipped(t1, f, t2)$ is another notational shorthand to say that the fluent f is initiated between times $t1$ and $t2$ (EC2), fluents that have been initiated by occurrence of an event continue to hold until occurrence of a terminating event (EC3), and fluents that have been terminated by occurrence of an event continue not to hold until occurrence of an initiating event (EC4). Following Shanahan, we assume that all variables are universally quantified except where otherwise shown. We also assume linear time with non-negative integer values. In EC, we follow the rules of circumscription in formalizing commonsense knowledge (McCarthy, 1986), by assuming that all possible causes for a fluent are given in the database and our reasoning tool cannot find anything except those causes.

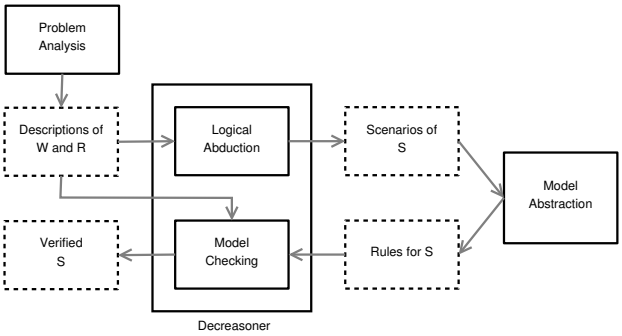


Figure 3: Overview of the approach

3. The Proposed Approach

A schematic overview of our approach is summarised in Figure 3. This approach uses the Problem Frame approach to decompose complex problems into subproblems and recompose them (Problem Analysis). Requirements (R) and domain assumptions (W) identified are described using the Event Calculus. In order to find the instances of the specification, descriptions of W and R are encoded and presented to Decreasoner (Mueller, 2006) to find event-based narratives of actions, "plans", or scenarios (Logical Abduction). These scenarios show how the requirement can be satisfied given the domain descriptions.

Unrealistic scenarios, impossible sequences of actions, are then eliminated and the remainder abstracted into rules for the specification (S) (Model Abstraction). In our derivation of specifications we not only find the events leading to the satisfaction of the requirement, we also identify the events that may happen, and events that must not happen in the mean time. This information is used in order to weaken feature specifications and detect potential inconsistencies. The specification is then checked (Model Checking) to produce the verified specifications (with respect to W and R).

3.1. Describing Problem Diagrams using the Event Calculus

In our approach to specifying event-based systems, requirements are described as combinations of fluents capturing the required states of the problem world, problem world domains are described as event-to-fluent and fluent-to-event causality, and specifications are described as conditions constraining the occurrences and non-occurrences of events.

Definition 3.1. Observations *consist of a finite conjunction of* $(\neg)HoldsAt$ *formulae.* Reference phenomena *(Γ) are observations describing the given state of the system, while* controlled phenomena *(Γ') are observations describing the desired state of the system. A* requirement *in the Problem Problems approach is expressed either as (i) ground observations Γ', without any reference to the given state of the system, or (ii) as a relationship between the reference and*

the controlled phenomena, such as a state constraint *of the form* $\Gamma \rightarrow \Gamma'$ *, or an* action precondition axiom *of the form* $(\neg)Happens(f1, t) \rightarrow \Gamma'$ *where the antecedent is an occurrence of an action in the system (for example, to say that when an event a1 happens at time t, the fluent f1 must be true at t1).*

Since the requirements tend to be about (desired) properties of the system over time, they will be formulated in terms of fluents holding, rather than in terms of (instantaneous) event occurrences.

Definition 3.2. *A* domain description *in our approach is expressed as event-to-condition and condition-to-event causality. The first causality deals with what happens to the fluents when events occur, and the second causality deals with the domain properties that lead to the occurrence of certain events. In the Event Calculus, the event-to-condition causality is described as a finite conjunction of* positive *and* negative effect axioms *(*Σ*) of the form* $Initiates(a, f, t) \leftarrow \Pi$ *or* $Terminates(a, f, t) \leftarrow \Pi$ *where* Π *has the form* $(\neg)HoldsAt(f_1, t) \wedge \cdots \wedge (\neg)HoldsAt(f_n, t)$ *and t, and* f_1 *to* f_n *are terms for the time and fluents respectively. The condition-to-event causality is described as a finite conjunction of* trigger axioms *(*Δ_2*) of the form* $Happens(a, t) \leftarrow \Pi$*.*

3.2. Specifying Features

Generally, specifications tend to describe the events that must be generated at certain times in order to satisfy requirements. In our specifications, we also describe the events that must not happen within certain time ranges in order to satisfy requirements. The *prohibit* predicate is used to specify those events. There may also be other events which are left undescribed because their occurrence or non-occurrence is assumed not to affect the requirement satisfaction. When there are such undescribed events, the specification is partially open, meaning that there can be more than one program that satisfies the specification.

If the specification asserts that certain event instances are prohibited, this assertion is universal in the sense that all specifications, including the one making the assertion, must not generate the events. It may be non-trivial to implement this assertion, especially when event are controlled by more than one device, as is the case with smart home applications. Therefore, a composition controller must be able to observe and when necessary block events that may cause interactions.

Definition 3.3. *A* specification *is expressed as a finite conjunction of* event occurrence constraints *(*Ψ*) of the form* $(\neg)Happens(a_1, t) \wedge (\neg)HoldsAt(f, t) \rightarrow (\neg)Happens(a_2, t)$ *where* a_1*,* a_2*, t, and f are terms for the action, time point, and fluent respectively.*

3.3. Important Properties

The simplest specification in the problem frames approach is a pro-active machine that addresses a subtype of problem known as *Required Behavior*. In

this type of problem, a specification is required to bring about certain states in the system, without relying on the feedback from the problem world. In such cases, the basic property of the descriptions we want is:

$$\Sigma \wedge \Psi \models \Gamma'$$

That is, given a theory of problem world domains (Σ), a specification (Ψ), and an appropriate deductive system ($\models$), we want to show that the requirement (property of the controlled phenomena) are satisfied non-trivially, meaning that the system has some liveness properties (Gunter et al., 2000).

In more common cases, the system has to rely on the feedback from the environment (Δ_2) and observations about the environment (Γ) .

$$\Sigma \wedge \Gamma \wedge \Delta_2 \wedge \Psi \models \Gamma'$$

The challenge in each case is to find (Ψ): what are the conditions under which certain events should be generated by the machine, and the conditions under which certain events must not be generated by the machine, in order to satisfy the requirement?

3.4. Deriving Feature Specifications

Our approach to specifying the features involves four steps. Before that we will make certain assumptions clear. First, these have to rely on the consistency of the domain description Σ (see Def. 3.2) and observations Γ and Γ' (see Def. 3.1). Second, we assume uniqueness of fluent and event names, meaning that no two names denote the same thing. This uniqueness axiom is represented by Ω. Finally, we assume the completion of predicates in Σ Mueller (2006).

Our four steps are:

1. We first pose a *logical abduction* problem in order to find all constructive hypotheses (Δ_1) explaining how, given the description of the problem world domains ($\Sigma \wedge \Gamma \wedge \Delta_2$), the requirement ($\Gamma'$) can be satisfied, i.e.

$$CIRC[\Sigma; Initiates, Terminates] \wedge$$
$$CIRC[\Delta_1 \wedge \Delta_2; Happens] \wedge \Gamma \wedge \Omega \models \Gamma'$$

   where Δ_1 is consistent with the domain description. Δ_1 is a partially ordered sequences of event occurrences that, given the problem world domains, leads to the requirement being satisfied. The circumscription operator assumes that no events other than those by Δ_1 and Δ_2 may occur (otherwise the requirement is not satisfied). Therefore, Δ_1 tells us events that must happen and that may happen. Event occurrences that do not appear in Δ_1 must not happen.
   However, some of the hypotheses in Δ_1 may not be "realistic": for example, a scenario may assume competence and co-operation of users to a level that cannot be guaranteed. Furthermore, Δ_1 may also contain stuttering events that can be eliminated without affecting requirements satisfactions (Lamport, 1983).

2. The developer then identifies the 'unrealistic' hypotheses in Δ_1 and eliminates them by providing further information about the problem world domains. Similarly, event stuttering is removed by adding further constraints (which are then used to weaken the specifications). Although each hypothesis is a model of a running program that implements the specification, we want specifications to be axiomatic because they are easier to reason about and are highly composable.
3. The models are then abstracted and the event occurrence constraints are formulated by creating the rules Ψ for S. This and the previous step are done interactively. User input is important because it is difficult to identify unrealistic hypotheses automatically. There are many approaches to merging scenarios (for example Hélouët et al. (2006); Uchitel et al. (2003)). There are also logic-based approaches to learning specification rules from positive and negative scenarios such that the rules will permit all positive scenarios and eliminate negative scenarios (Alrajeh et al., 2009).
4. Finally, specifications are verified via checking the completeness of the constraints by showing that

$$CIRC[\Sigma; Initiates, Terminates] \wedge$$
$$CIRC[\Delta_2; Happens] \wedge \Psi \wedge \Gamma \wedge \Omega \models \Gamma'$$

   Notice that Ψ is not circumscribed: it will have to make explicit all events that must not happen. Therefore, Ψ describe all events that must happen, and all events that must not happen, the remainder being events that may happen.

In practice, the process is highly iterative and will rely on the developer's expertise. The state-space of the model tends to grow exponentially, and the number of scenarios also increase accordingly. When the number of the scenarios becomes very large, there are tactics the developer can use in order to limit the number of scenarios. For instance, Step 2 suggests removing stuttering events. This may reduce the number scenarios significantly. The developer can also write assertions removing scenarios that cannot happen in practice (such as the clock going backward, or a burglar not wanting to steal after gaining entry to the house). Finally, the developer can limit the length of time so that periodic episodes in the system behaviour are not repeated too many times. All of these tactics require a lot of care because they can be error-prone. In our experience, after removing these three kinds of scenarios, the remainder is relatively small and mostly realistic. Precise numbers vary from case to case. For the examples discussed in the paper, tens of scenarios were reduced to a handful.

3.5. Tool Support

Logical reasoning in our approach is done using the Discrete Event Calculus Reasoner, Decreasoner, tool (Mueller, 2006). The tool solves the Event Calculus problems by translating them into satisfiability problems for SAT solvers. In principle, the abductive procedure may not terminate if the goal is not satisfiable; however, since the reasoning in Decreasoner uses a bounded and discrete time range, the tool forces a termination when the time limit is reached.

4. Specifying Individual Features

This section provides examples of how the proposed approach can be applied in order to specify and check individual features and their composition.

4.1. Tool-assisted Deriving of the Security Feature Specification

The requirement for the Security Feature (SF), initially discussed in Section 2.2, is to ensure that the window is shut during the night. In this case, the machine by itself does not know whether it is night or day at any given time, and thus relies on the domain Time Panel for this information.

The requirement for the security feature is to keep the window shut during the night. It can be formalised in the Event Calculus as follows:

$$\begin{aligned} \neg HoldsAt(Night, time-1) \wedge HoldsAt(Night, time) \wedge \\ HoldsAt(Night, time1) \wedge time \leq time1 \wedge \neg HoldsAt(Day, time2) \wedge \\ time \leq time2 \leq time1 \wedge time + delay \leq time3 \leq time1 \rightarrow \\ HoldsAt(Shut, time3) \end{aligned} \tag{SR}$$

The requirement says that soon after the time point of nightfall, and until the daybreak, the window should be shut.

We then describe the state machine of the window domain as follows. When the event TiltOut() happens, the window opens (W1). When the event TiltIn() happens, the window closes (W2). When the event TiltIn() happens, the window is no longer open (W3). When the event TiltOut() happens, the window is no longer closed (W4). The window cannot be both open and closed at the same time (W5). This prevents inconsistent initial conditions of the window.

$$Initiates(TiltOut(), Open, time) \tag{W1}$$

$$Initiates(TiltIn(), Shut, time) \tag{W2}$$

$$Terminates(TiltIn(), Open, time) \tag{W3}$$

$$Terminates(TiltOut(), Shut, time) \tag{W4}$$

$$HoldsAt(Open, time) \leftrightarrow \neg HoldsAt(Shut, time) \tag{W5}$$

The behavior of the time panel can be described as follows, while omitting the formalisation for space reasons. When the clock hits 7pm, it is night. When the clock hits 7pm it is no longer day. When the clock hits 7am it is day. When the clock hits 7am it is no longer night. It cannot be both night and day at the same time. When it is night and was not night at the previous time point, the night has started. When it is day and was not day at the previous time point, the day has started.

In the first step, when this model is given to the solver to abduce the specification, it finds several models with much similarity. In the second step, we add further domains rules to remove stuttering events: (W6) and (W7) says that

when shut, the window cannot be shut again; when open, the window cannot be opened again. We also define the initial condition of the window: The window is initially open (W8). Finally, the behaviour of the clock is further constrained. The clock hits 7am on the fifth of every ten time units (TP8), and hits 7pm on the tenth of every ten time units (TP9). The clock will not hit either 7am or 7pm at other times (TP10) and (TP11). Initially, it is day when the clock strikes 7pm TP12 and TP13).

$$HoldsAt(Shut, time) \rightarrow \neg Happens(TiltIn(), time) \quad \text{(W6)}$$

$$HoldsAt(Open, time) \rightarrow \neg Happens(TiltOut(), time) \quad \text{(W7)}$$

$$(time = 0) \rightarrow HoldsAt(Open, time) \quad \text{(W8)}$$

$$(time\ mod\ 5 = 0) \wedge (time\ mod\ 10 \neq 0) \wedge (time > 0) \rightarrow Happens(Hit7am(), time) \quad \text{(TP8)}$$

$$(time\ mod\ 10 = 0) \wedge (time > 0) \rightarrow Happens(Hit7pm(), time) \quad \text{(TP9)}$$

$$(time\ mod\ 10 \neq 0) \wedge (time\ mod\ 5 \neq 0) \rightarrow \neg Happens(Hit7pm(), time) \quad \text{(TP10)}$$

$$(time\ mod\ 10 \neq 0) \wedge (time\ mod\ 5 \neq 0) \rightarrow \neg Happens(Hit7am(), time) \quad \text{(TP11)}$$

$$(time = 0) \rightarrow HoldsAt(Day, time) \quad \text{(TP12)}$$

$$(time = 0) \rightarrow Happens(Hit7pm(), time) \quad \text{(TP13)}$$

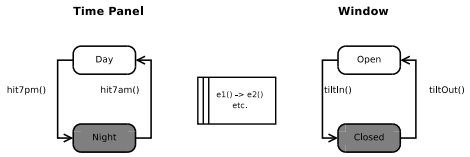


Figure 4: Model Finding in Problem Frames

An intuitive description of the required specification is described in Figure 4. There are two state machines: one for the time panel and another for the window. We need to design a machine such that when some (shaded) properties hold in one domain or state machine, certain (shaded) properties must also hold in another state machine within a certain time range. The machine needs to observe certain alphabets of both machines and make sure that the desired properties hold by firing off appropriate events (or not firing certain events) in time.

We can select any time range including an entire interval of night time, and possibly some day time too. For instance, if we restrict the time range for the abduction to 6 time units, the tool finds four scenarios as shown in Figure 5, which can be read as follows:

```
model 1:
0
Day().
Open().
Happens(Hit7pm(), 0).
1
-Day().
+Night().
Happens(NightStarted(), 1).
Happens(TiltIn(), 1).
2
-Open().
+Shut().
3
4
5
Happens(Hit7am(), 5).
Happens(TiltOut(), 5).
6
-Shut().
-Night().
+Day().
+Open().
---
model 2:
0
Day().
Open().
Happens(Hit7pm(), 0).
1
-Day().
+Night().
Happens(NightStarted(), 1).
Happens(TiltIn(), 1).
2
-Open().
+Shut().
3
4
5
Happens(Hit7am(), 5).
6
-Night().
+Day().
```

```
model 3:
0
Day().
Open().
Happens(Hit7pm(), 0).
Happens(TiltIn(), 0).
1
-Day().
-Open().
+Shut().
+Night().
Happens(NightStarted(), 1).
2
3
4
5
Happens(Hit7am(), 5).
6
-Night().
+Day().
---
model 4:
0
Day().
Open().
Happens(Hit7pm(), 0).
Happens(TiltIn(), 0).
1
-Day().
-Open().
+Shut().
+Night().
Happens(NightStarted(), 1).
2
3
4
5
Happens(Hit7am(), 5).
Happens(TiltOut(), 5).
6
-Shut().
-Night().
+Day().
+Open().
```

Figure 5: Scenarios of how the security requirement can be satisfied

1 It is day, the window is open, and the clock hits 7pm. As a result, Night becomes true. The events NightStarted() and TiltIn() are fired and the window becomes shut. It remains shut until the clock hits 7am, then the event TiltOut() is fired. Notice that the machine Security Feature can observes the universal time and the events NightStarted() and DayStarted(). Occurrences of TiltOut() and TiltIn() may be caused by the time, occurrences of NightStarted() and DayStarted(), or both (in cases of delayed event occurrences). Since Security Feature is a reactive machine, and does not (appear to) generate delayed events, we will look for event to event causality.

It is possible to fire TiltIn() when NightStarted() is observed (time 1). As the event Hit7am() is not observable by the machine, this causality is not feasible (time 5).

2 This is similar to the previous scenario, except for the fact that nothing happens when the clock hits 7am. This is true because the requirement does not say what happens when it is day. This is a realistic scenario.

3 In this scenario, the event TiltIn() is fired before NightStarted() is observed. This is not realistic, so it is ignored.

4 This scenario also is unrealistic for reasons given above.

From these scenarios, we can say in the positive mode that the event TiltIn() should be fired whenever NightStarted() is observed.

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

In addition, we examine the list of $\neg Happens()$ produced by the tool. Removing those covered by (TP10 and TP11), and abstracting the remainder yields the following rule.

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

(SF1) and (SF2) are the specification for Security Feauture. In order to simplify, our feature specifications, we introduce into our Event Calculus the predicate, $Prohibit(a, t1, t2)$, with the meaning that the event a should not occur between times $t1$ and $t2$. More formally,

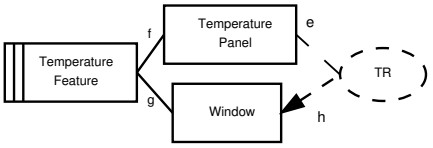
$$Prohibit(a, t1, t2) \stackrel{\text{def}}{\equiv} \neg\exists t \cdot Happens(a, t) \wedge t1 \leq t \leq t2 \tag{EC7}$$

Using this definition, we can rewrite (SF2) as follows:

$$\begin{aligned} Happens(NightStarted(), time) \wedge \neg Happens(DayStarted(), time1) \wedge \\ time \leq time1 \rightarrow Prohibit(TiltOut(), time, time1) \end{aligned} \tag{SF2b}$$

Notice that the predicate *Prohibit* explicitly defines the domain assumptions that must hold in order that the feature satisfy its requirement. The variable $time1$ in SF2b indicates the time point until which the door should remain shut and we will refer to it as ShutUntil. Having derived the specification, we can now prove the specification in the fourth and final step. Let SF2b be the specification (Ψ) for the Security Feature. Let W1 to W8, and TP8 to TP13 be the domain description (Σ), and let SR be the requirement Γ'. Let EC be a conjunction of all domain independent EC rules, namely $(EC1) \wedge \ldots (EC7)$. Then the following property can be proved using the tool.

$$CIRC[\Sigma; Initiates, Terminates] \wedge \Psi \wedge EC \wedge \Omega \models \Gamma'$$



Interface Phenomena Name Set	
e,f:	TeP!{NiceTemp, OutTemp,InTemp}
g:	TF!{tiltIn(), tiltOut()}
h:	W!{Open,Shut}

Figure 6: Problem diagram for the temperature feature

4.2. Temperature Feature

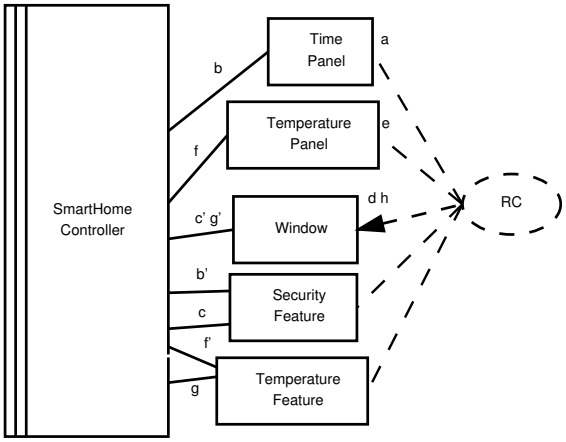
In order to illustrate our approch to composition of features, we will first briefly introduce a new smart home feature. The problem diagram for the temperature feature, shown in Fig. 6 is similar to the diagram in Fig. 1. The requirement (TR) here is that if the desired temperature (NiceTemp) and the indoors and outdoors temperatures (OutTemp and InTemp) are in a certain relationship, the window should be kept open. The temperature readings are controlled by the temperature panel (TeP!), and the temperature feature can observe them at the interface f. One description of the specification Temperature Feature is to fire the tiltOut() event at the interface g whenever the conditions $NiceTemp < InTemp$ and $OutTemp < InTemp$ hold and to ensure that the tiltIn() is not fired as long as that relation remains true.

Notice that the two requirements above do not say anything about what to do during the daytime, and when it is not hot indoors. Composing these two features can lead to a divergent behaviour under certain conditions. During a hot night, according to the temperature feature, the window should be open, but according to the security feature, the window should be shut. It is important to note that although the temperature feature will not close the window by firing the tiltIn event, it cannot stop the security feature from firing the same event during the hot night. Likewise, although the security feature will not open the window by firing the tiltOut event, it cannot stop the temperature feature from firing the same event during the hot night.

5. Runtime Composition of Features

As shown in Fig. 7, the two features can be composed by introducing the new software component SmartHome Controller, which is obtained by merging two wrappers that sit at the interfaces a and b of the security feature in Fig. 1 and the interfaces f and g of the temperature feature in Fig. 6. In effect, SmartHome Controller intercepts the information and events going in and coming out of the two features. Tun et al. (2009a) discuss rules for obtaining and merging wrappers.

The variable ShutUntil is used by the security feature to indicate the time point until which it does not want other features to open the window. A similar variable OpenUntil is used by the temperature feature to indicate the time point



Interface Phenomena Name Set
a: TiP!{Night, Day}
b: TiP!{NightStarted(), DayStarted()}
b': SC!{NightStarted(), DayStarted()}
c: SF!{TiltIn(), TiltOut()}
c': SC!{TiltIn(), TiltOut()}
d: W!{Open, Shut}
e,f: TeP!{NiceTemp, OutTemp,InTemp}
g: TF!{tiltIn, tiltOut}
g': SC!{tiltIn, tiltOut}
h: W!{Open,Shut}

Figure 7: Composition of the security and temperature features

until which it does not want other features to shut the window. Again, notice that each of the features does not prevent another feature from opening or shutting the window. Each feature only declares what it wants other features not to do within a certain duration. ShutUntil and OpenUntil will then be used by the SmartHome Controller to mediate when conflicts arise.

The requirement for composition (RC) can be defined in several ways in resolving the conflicts: we will call them the semantics of the composition operator. They include:

- **No Control:** In this composition, the requirements for the security and temperature features should each be met at times when they are not in conflict; but when conflicts occur, any emergent behaviour is acceptable. It allows, for example, the window to oscillate in a partly open position.

- **Exclusion:** In this composition, the requirements for the security and temperature features should each be met at times when they are not in conflict; but when conflicts occur, the requirement of the feature that started first should have priority. For example, if the security feature shuts the window before the temperature feature needs to open it, the temperature feature will not be able to shut the window until the security requirement has been satisfied. This exclusion is symmetrical.

- **Exclusion with Priority.** In this composition, the exclusion is asymmetrical, for instance, in favour of the security requirement. The security feature can shut the window during the time in which the temperature feature wants the window open. The temperature feature, however, cannot open the window if the security feature wants it shut.

A precise specification of these semantics of the composition operator have been discussed by Laney et al. (2007).

6. Related Work

There is a long history of research into feature interaction problems, surveyed comprehensively, for instance, by Calder et al. (2003). Therefore, the discussion here will relate mostly to research into feature interactions in smart home applications, into tool-assisted specification and composition of features.

6.1. Feature Interactions in Smart Home Applications

Nakamura et al. (2005) propose an object-oriented approach to detecting feature interactions in smart home applications. They model each device, as well as the home environment, as an object, whose properties can be accessed through the object methods. Pre- and post-conditions of the methods are used to detect possible feature interactions. They distinguish between two kinds of feature interactions: *appliance interactions* can be detected within the properties of a single device, while *environment interactions* can only be detected when properties of several devices (or the environment as a whole) are considered.

Shehata et al. (2007) examine the runtime policy interactions in smart home applications. In particular, they consider the issue of divergent policies invoked by different users of the smart home application. They propose different ways to detect possible interactions and propose a KNX-based policy interaction management module to resolve undesired interactions at runtime.

Rashidi and Cook (2009) argue that many smart home technologies often do not fit user expectations. Part of the difficulties is that these technologies do not adapt to changing environments and user needs. They propose to use machine learning techniques to discover patterns of user behaviour and modify system behaviour accordingly. Although feature interaction is not the central to this work, it may be possible to learn how users wish to resolve new interactions.

6.2. Deriving Specifications

Although there are several tools to *analyze* specifications for certain properties, there are relatively few tools that help *find* specifications. In this article, we discuss an approach to specifying these systems, and suggest tool support to help develop their specifications. There are several systematic approaches to finding specifications (Yu and Mylopoulos, 1994; Laney et al., 2007; Rapanotti et al., 2006; Seater and Jackson, 2006; van Lamsweerde et al., 1995; van Lamsweerde and Willemet, 1998). There are few tools to support systematic derivation of specifications from requirements using abductive temporal logic.

The Event Calculus has previously been used for reasoning about evolving specifications (d'Avila Garcez et al., 2003; Russo et al., 2002), and distributed systems policy specifications (Bandara et al., 2003). Our work is complementary to such approaches in that it will allow inconsistencies to be resolved at run-time.

Various specification analysis tools exist; Lespérance et al. (1999) and Heitmeyer et al. (1996), for example, propose tool suites to perform specific analyses tasks, such as consistency checks. However, they are less concerned with automated derivation of specifications.

7. Conclusion

This article has examined two challenges in engineering smart home applications, relating to the question of deriving correct feature specifications from requirements and to the question of runtime composition of features and resolution of interactions. The first challenge is addressed by using the Problem Frames approach in order to pose the specification as a logical abduction problem. Results from the abduction procedure are used by the developer to refine and write rule-base specifications which can then be checked by the tool. The second challenge is addressed by making the domain assumptions of the individual features explicit, so that those assumptions, in the form of the *Prohibit* predicate, are used by a composition controller to detect and resolve feature interactions.

Acknowledgements

We thank the anonymous reviewers for providing helpful and detailed comments. This work was funded in part by the SFI grant 10/CE/I1855 and ERC Advanced Grant 291652.

8. References

References

Alrajeh, D., Kramer, J., Russo, A., Uchitel, S., 2009. Learning operational requirements from goal models. In: Proceedings of ICSE. IEEE Computer Society, Washington, DC, USA, pp. 265–275.

Bandara, A. K., Lupu, E., Russo, A., 2003. Using event calculus to formalise policy specification and analysis. In: POLICY. IEEE Computer Society, pp. 26–39.

Calder, M., Kolberg, M., Magill, E. H., Reiff-Marganiec, S., 2003. Feature interaction: a critical review and considered forecast. Computer Networks 41 (1), 115 – 141.

Classen, A., Laney, R., Tun, T. T., Heymans, P., Hubaux, A., 2008. Using the event calculus to reason about problem diagrams. In: Proceedings IWAAPF'08, Leipzig, Germany, May 2008.

d'Avila Garcez, A. S., Russo, A., Nuseibeh, B., Kramer, J., 2003. Combining abductive reasoning and inductive learning to evolve requirements specifications. IEE Proceedings - Software 150 (1), 25–38.

Grimm, R., Davis, J., Lemar, E., MacBeth, A., Swanson, S., Anderson, T. E., Bershad, B. N., Borriello, G., Gribble, S. D., Wetherall, D., 2004. System support for pervasive applications. ACM Trans. Comput. Syst. 22 (4), 421–486.

Gunter, C. A., Gunter, E. L., Jackson, M., Pamela, Z., 2000. A reference model for requirements and specifications. IEEE Software 17 (3), 37–43.

Heitmeyer, C. L., Jeffords, R. D., Labaw, B. G., 1996. Automated consistency checking of requirements specifications. ACM Trans. Softw. Eng. Methodol. 5 (3), 231–261.

Hélouët, L., Hénin, T., Chevrier, C., 2006. Automating scenario merging. In: Gotzhein, R., Reed, R. (Eds.), SAM. Vol. 4320 of Lecture Notes in Computer Science. Springer, pp. 64–81.

Jackson, M., 1995. Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices. ACM Press.

Jackson, M., 2001. Problem Frames: Analyzing and structuring software development problems. ACM Press & Addison Wesley.

Kolberg, M., Magill, E. H., Wilson, M., 2003. Compatibility issues between services supporting networked appliances. IEEE Communications Magazine 41 (11), 136–147.

Lamport, L., 1983. What good is temporal logic? In: IFIP Congress. pp. 657–668.

Laney, R., Barroca, L., Jackson, M., Nuseibeh, B., 2004. Composing requirements using problem frames. In: Proceedings of RE'04. IEEE Computer Society, pp. 122–131.

Laney, R., Tun, T. T., Jackson, M., Nuseibeh, B., 2007. Composing features by managing inconsistent requirements. In: Proceedings of ICFI. pp. 141–156.

Lespérance, Y., Kelley, T. G., Mylopoulos, J., Yu, E. S. K., 1999. Modeling dynamic domains with congolog. In: Jarke, M., Oberweis, A. (Eds.), CAiSE. Vol. 1626 of LNCS. Springer, pp. 365–380.

LinuxMCE, 2012. Linux media center edition.

McCarthy, J., 1986. Applications of circumscription to formalizing common sense knowledge. Artificial Intelligence 28, 89–116, reprinted in McCarthy (1990).

McCarthy, J., 1990. Formalization of common sense, papers by John McCarthy edited by V. Lifschitz. Ablex.

Microsoft, 2012. Homeos: Enabling smarter homes for everyone.
URL `http://research.microsoft.com/en-us/projects/homeos/`

Miller, R., Shanahan, M., 1999. The event calculus in classical logic - alternative axiomatisations. Journal of Electronic Transactions on Artificial Intelligence.

Mueller, E. T., 2006. Commonsense Reasoning. Morgan Kaufmann.

Nakamura, M., Igaki, H., ichi Matsumoto, K., 2005. Feature interactions in integrated services of networked home appliance. Feature Interactions in Telecommunications And Software Systems VIII.

Park, S. H., Won, S. H., Lee, J. B., Kim, S. W., 2003. Smart home - digitally engineered domestic life. Personal Ubiquitous Comput. 7 (3-4), 189–196.

Rapanotti, L., Hall, J. G., Li, Z., 2006. Deriving specifications from requirements through problem reduction. IEE Proceedings Software 153 (5), 183–198.

Rashidi, P., Cook, D., sept. 2009. Keeping the resident in the loop: Adapting the smart home to the user. Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on 39 (5), 949 –959.

Russo, A., Miller, R., Nuseibeh, B., Kramer, J., 2002. An abductive approach for analysing event-based requirements specifications. In: Stuckey, P. J. (Ed.), ICLP. Vol. 2401 of LNCS. Springer, pp. 22–37.

Seater, R., Jackson, D., 2006. Requirement progression in problem frames applied to a proton therapy system. In: Proceedings of RE'06. IEEE Computer Society, Washington, DC, USA, pp. 166–175.

Shehata, M., Eberlein, A., Fapojuwo, A., 2007. Managing policy interactions in knx-based smart homes. In: Computer Software and Applications Conference. Vol. 2. pp. 367 –378.

Tun, T. T., Trew, T., Jackson, M., Laney, R. C., Nuseibeh, B., 2009a. Specifying features of an evolving software system. Softw., Pract. Exper. 39 (11), 973–1002.

Tun, T. T., Yu, Y., Laney, R., Nuseibeh, B., 2009b. Early identification of problem interactions: A tool-supported approach. In: Glinz, M., Heymans, P. (Eds.), Proceedings of REFSQ. LNCS 5512. Springer, pp. 74–88.

Uchitel, S., Kramer, J., Magee, J., 2003. Synthesis of behavioral models from scenarios. IEEE Trans. Software Eng. 29 (2), 99–115.

van Lamsweerde, A., Darimont, R., Massonet, P., 1995. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In: RE. IEEE Computer Society, pp. 194–203.

van Lamsweerde, A., Willemet, L., 1998. Inferring declarative requirements specifications from operational scenarios. IEEE Trans. Software Eng. 24 (12), 1089–1114.

Yu, E. S. K., Mylopoulos, J., 1994. Understanding "why" in software process modelling, analysis, and design. In: ICSE. pp. 159–168.

Zave, P., Jackson, M., 1997. Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology (TOSEM) 6 (1), 1–30.