Composing features by managing inconsistent requirements

Conference or Workshop Item

Composing Features by Managing Inconsistent Requirements

Robin Laney, Thein T. Tun, Michael Jackson, and Bashar Nuseibeh

Centre for Research in Computing
The Open University
Walton Hall, Milton Keynes MK7 6AA, UK
{r.c.laney, t.t.tun, m.jackson, b.nuseibeh}@open.ac.uk

Abstract. One approach to system development is to decompose the requirements into features and specify the individual features before composing them. A major limitation of deferring feature composition is that inconsistency between the solutions to individual features may not be uncovered early in the development, leading to unwanted feature interactions. Syntactic inconsistencies arising from the way software artefacts are described can be addressed by the use of explicit, shared, domain knowledge. However, behavioural inconsistencies are more challenging: they may occur within the requirements associated with two or more features as well as at the level of individual features. Whilst approaches exist that address behavioural inconsistencies at design time, these are over-restrictive in ruling out all possible conflicts and may weaken the requirements further than is desirable. In this paper, we present a lightweight approach to dealing with behavioural inconsistencies at run-time. Requirement Composition operators are introduced that specify a run-time prioritisation to be used on occurrence of a feature interaction. This prioritisation can be static or dynamic. Dynamic prioritisation favours some requirement according to some run-time criterion, for example, the extent to which it is already generating behaviour.

Key words: Feature Interaction, Pervasive Software, Event Calculus, Problem Frames

1 Introduction

Given a good description of requirements for a feature-rich system, there are advantages, including scalability and traceability [3,14,27,28,19], in solving the feature sub-problems in isolation before composing the partial solutions to give a complete system. Deferring the composition problem supports a better separation of concerns between requirements analysis and the design phase, and is in line with an iterative approach to development [22,12].

The composition problem also raises a number of questions: Are the requirements to be composed consistent with each other? Do the specifications to be composed share assumptions about their environment? Do they embody consistent models? How do we deal with interference between the effects of features

on the system's environment? We focus on the first and last of these questions, but in doing so address the others to varying degrees.

The contribution of this paper is an approach to resolve, at runtime, undesirable feature interactions arising from inconsistent requirements. Runtime resolution techniques have many advantages over compile time techniques, including minimal weakening of the requirements, and allowing features developed by disparate developers to plug and play [17,4].

Our approach synthesises two complementary techniques: (i) a form of temporal logic called the Event Calculus [18,26], and (ii) a way to compose problems and solutions called Composition Frames [19]. We use a version of the Event Calculus [18,26] to express requirements and domain properties, and systematically derive feature specifications in a way that makes inconsistencies more explicit. We add a *Prohibit(...)* predicate to the Event Calculus, and use it in feature specifications to prohibit events over specific periods of time, facilitating non-intrusive composition of features. Composition Frames, introduced in [19], are used to mediate between the features at runtime, and provide an argument showing that they satisfy a family of weakened conjunction requirements.

The paper is organised as follows. In Section 2, we present a motivating example whilst giving a brief introduction to the Problem Frames approach and also the Event Calculus. In Section 3, we begin by showing how to express requirements and domain properties in the Event Calculus before deriving machine specifications. We then consider the semantics of requirements composition and discuss Composition Frames as a way of reasoning about the relationship between composed requirements and composed specifications in Section 4. In Section 5, we compare our work with other approaches. In Section 6, we discuss some lessons about the composition of requirements, of solutions, and their relationship. We conclude in Section 7 and present future work.

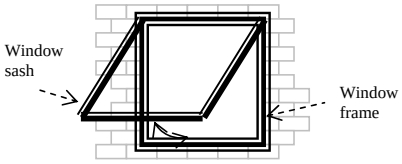
2 Background

In this section we introduce the problem frames notation and philosophy, and present an example system that will be used in Sections 3 and 4 to illustrate our technique. We then give an introduction to the Event Calculus and motivate its choice as a tool for addressing some composition concerns.

2.1 Introductory Example

Throughout this paper we will use an example that involves developing the specification for a simple "smart home" application [17]. In order to facilitate convenient living, household appliances, such as air conditioners, security alarms and windows are increasingly connected to home digital networks. The functioning of these appliances is controlled by complex software systems known as smart home applications. For example, a security feature may switch on and off lights of the home when the homeowners are away, to give an impression that the house is occupied. The specific example discussed in this paper has two features, and

is mainly concerned with the control of a motorized awning window, illustrated below.



Requirements for features. The requirement for one feature is concerned with the house security (SR), whilst the requirement for the other feature is concerned with the climate control and energy efficiency of the house (TR). Informal descriptions of these requirements are given below.

> **SR**: "Keep the awning window shut at night."
> **TR**: "If it is hot indoors (i.e. hotter than the required temperature) and cold outside (i.e. colder than the temperature indoors), open the awning window."

Analyzing a requirement, such as SR or TR, using the Problem Frames approach involves identifying the problem context and matching it to one of several well-known diagram forms. Starting with the SR requirement, Fig. 1 shows the problem diagram for the security feature. A problem diagram such as this shows the relationship between descriptions of (i) a *machine domain* denoted by a rectangle with two vertical stripes, (ii) problem world domains, denoted by plain rectangles and (iii) a requirement denoted by a dotted oval. The machine domain implements a solution in order to satisfy the SR requirement. In our discussions, we may refer to a machine as a feature specification or just specification. The *problem domains* are entities in the world that the machine must interact with, such as Time Panel and Window in Fig. 1, in satisfying the *requirement*, in this case, SR. The thick lines are called phenomena (a and b) representing shared states and events between the domains involved. Dotted lines are requirement phenomena (a and c). Broadly speaking, SR in Fig. 1 says that if the time panel indicates night time, we expect the window to be shut.

The problem diagram for the climate control and energy efficiency feature in Fig. 2 is similar. Again, broadly speaking, the requirement is that if the

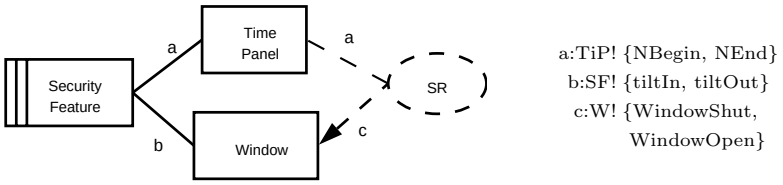


Fig. 1. Problem Diagram for the security feature

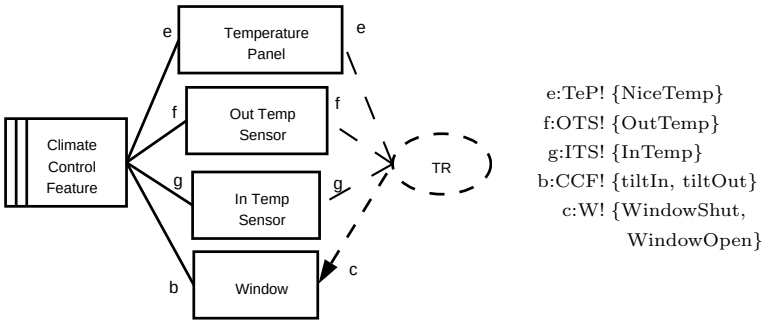


Fig. 2. Problem Diagram for the climate control and energy efficiency feature

desired temperature and the indoors and outdoors temperatures are in a certain relationship, we expect the window to be opened.

Having informally described the requirements, we now examine the properties of the problem and machine domains.

Problem Domains. In Fig. 1, when the time falls between NBegin and NEnd of the Time Panel (TiP) domain, it is night. The prefix TiP! specifies that values of NBegin and NEnd are controlled by Time Panel. The awning window (W), in both Fig. 1 and Fig. 2, has the following properties. When the window sash has a zero degree angle on the window frame, the window is fully shut (WindowShut is true). When the window sash has a twenty degree angle on the window frame, the window is fully open (WindowOpen is true). When the event tiltOut is fired, the window sash starts to tilt out until either the window is fully open, or tiltIn is fired. Similarly, when the event tiltIn is fired, the window sash starts to tilt in until either the window is fully shut, or tiltOut is fired. OutTemp is the temperature outdoors and InTemp is the temperature indoors. NiceTemp of the Temperature Panel (TeP) domain indicates the temperature level desired by the house owner.

Machine Domains. When describing the machines individually, it is necessary to ensure that the specification for each feature's machine, along with the descriptions of the appropriate domains, is sufficient to establish that each requirement is satisfied. The obligation to demonstrate this is known as the *frame concern*, and the case that it holds must be made either formally or informally depending on context. In Section 3.2, we discuss a way to do this based on deriving the feature specifications from formal descriptions of the requirements and the window domain.

Each of these individual features in isolation can satisfy its own requirement. However, they will conflict whenever the TR machine needs to open the window at night time to adjust the indoors temperature by admitting cooler air, and the SR machine needs to keep the window closed. This conflict is dynamic, in the sense that it will only occur in certain circumstances. Our refinement of require-

Table 1. Some Event Calculus Predicates

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

ments into specifications in Section 3.2 highlights this conflict by identifying the events, occurrence of which at certain times may lead to a failure to satisfy some requirement. Therefore, a significant strength of this approach is that it identifies the ways in which a feature could interact with other feature(s) in terms of event occurrences without necessarily knowing what those other features are. Having derived the specification for each feature we must compose the specifications in a way that resolves this conflict at run time. We propose such a technique in Section 4.

2.2 The Event Calculus

The Event Calculus [26], first introduced in [18], is a logic system grounded in the predicate calculus. The calculus relates events and event sequences to 'fluents', which denote states of a system. It has been used as a way of permitting inconsistency in reasoning about requirements [25]. In our approach to this example 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 will work with a version of the calculus based on Shanahan [26] that is intended to be simple whilst fully supporting the contribution of Section 3. Since the machines for individual features are executed sequentially, the Event Calculus does not have to deal with concurrent events. Concurrency that arises due to composition of multiple features are handled by the composition controller introduced in Section 4. Table 1, also based on Shanahan [26], gives the meanings of the elementary predicates of the calculus.

The EC rules in Fig. 3, taken from Shanahan [26], are a way of stating that the fluent β holds if: it held initially and nothing has happened since to stop it holding (EC1); the event α has happened to make the fluent hold and nothing has happened since to stop it holding (EC2); or, the event α happened that caused some fluent $\beta 1$ to hold, that in turn, after a period of time δ caused this fluent β to hold, and again nothing has happened since to stop the second fluent

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

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

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

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

Fig. 3. Event Calculus Meta-rules

holding (EC3). Finally, the rule DEF1 says that the fluent β is clipped between $\tau 1$ and $\tau 2$ if and only if there is an event α that happens between $\tau 1$ and $\tau 2$ and the event terminates the fluent β. Following Shanahan, we assume that all variables are universally quantified except where otherwise shown.

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 Formalising Feature Specifications

We now address the derivation of feature specifications to meet the requirements in Fig. 1 and Fig. 2. In Section 3.1, we formalize our requirements and the description of the window domain by translating them into the language of the Event Calculus described in the previous section. We then derive feature specifications in Section 3.2 by refining our requirements using the window domain semantics. In this way, we are establishing the argument for the frame concern.

3.1 Formalizing Requirements and Domains

The natural language specifications of SR and TR, described in Section 2.1, can be formalized as follow:

$$HoldsAt(IsIn(t, NBegin, NEnd), t) \rightarrow HoldsAt(WindowShut, t) \quad \text{(SR)}$$

$$HoldsAt(InTemp > NiceTemp + 1, t) \wedge HoldsAt(InTemp > OutTemp + 1, t) \rightarrow HoldsAt(WindowOpen, t) \quad \text{(TR)}$$

The definition of SR says that if the current time is in the range of NBegin and NEnd, the machine should make sure that the window is shut. The definition of TR says that if the required temperature is lower than the temperature indoors

$$Initiates(tiltOut, TiltingOut, \tau) \quad \text{(D1)}$$
$$Trajectory(TiltingOut, \tau, WindowOpen, suffopentime) \quad \text{(D2)}$$
$$Initiates(tiltIn, TiltingIn, \tau) \quad \text{(D3)}$$
$$Trajectory(TiltingIn, \tau, WindowShut, suffshuttime) \quad \text{(D4)}$$
$$Terminates(tiltOut, TiltingIn, \tau) \quad \text{(D5)}$$
$$Terminates(tiltOut, WindowShut, \tau) \quad \text{(D6)}$$
$$Terminates(tiltIn, TiltingOut, \tau) \quad \text{(D7)}$$
$$Terminates(tiltIn, WindowOpen, \tau) \quad \text{(D8)}$$

Fig. 4. Domain Descriptions in EC

by more than one unit, and the outside temperature is lower than the temperature indoors by more than one unit, the machine should make the window fully open.

The natural language specification of the window, described in Section 2.1, can be formalized as shown in Fig. 4. In other words, if the window is tilted out, it starts tilting out (D1) until the window is fully open (D2) or the window is tilted in (D7). Similarly, if the window is tilted in, it starts tilting in (D3) until the window is fully shut (D4) or the window is tilted out (D5). When the window is tilted out, it is no longer shut (D6) and when it is tilted in, it is no longer open (D8).

3.2 Deriving Feature Specifications

The Event Calculus provides three options for dealing with a fluent expressed using HoldsAt – namely, EC1, EC2 and EC3. Since no window events shuts or opens the window instantaneously, the feature specification based on EC2 does not apply. We, therefore, focus on EC1 and EC3 only.

We begin with a refinement based on EC1 which deals with the case where the window was initially shut and nothing has changed. In our refinement, 'initially' or time point 0 means the time at which the system containing all composed features is turned on.

(State the requirement)

$$HoldsAt(IsIn(t, NBegin, NEnd), t) \rightarrow HoldsAt(WindowShut, t)$$

(Refine the conclusion by applying EC1)

$$Initially(WindowShut) \wedge \neg Clipped(0, WindowShut, t)$$

(Apply DEF1 to the second sub-clause)

$$Initially(WindowShut) \wedge \neg\exists a1, t1 \cdot Happens(a1, t1) \wedge$$
$$Terminates(a1, WindowShut, t1) \wedge 0 < t1 < t$$

(Unify the Terminate sub-clause with D6)

$$Initially(WindowShut) \wedge \neg\exists t1 \cdot Happens(tiltOut, t1) \wedge \\ Terminates(tiltOut, WindowShut, t1) \wedge 0 < t1 < t$$

(Remove the Terminate sub-clause because it is an axiom)

$$Initially(WindowShut) \wedge \neg\exists t1 \cdot Happens(tiltOut, t1) \wedge 0 < t1 < t$$

At this stage, we have a sub-clause whose role is to prevent a certain event happening over a given time period. In order to simplify our feature specifications, we introduce into our Event Calculus the new predicate, $Prohibit(\alpha, \tau1, \tau2)$, with the meaning that the event α should not occur between times $\tau1$ and $\tau2$. More formally,

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

The refinement can then be completed to give the following partial specification for SR.

$$HoldsAt(IsIn(t, NBegin, NEnd), t) \rightarrow \\ Initially(WindowShut) \wedge Prohibit(tiltOut, 0, t) \qquad \text{(SFa)}$$

This partial specification (SFa) says that if the window is shut initially (time 0), the system should prohibit the tiltOut event from time 0 until time t in order to keep the window shut at time t.

The second refinement based on EC3 deals with the significant case where the machine needs to tilt in the window sufficiently before the night falls (SFb). For space reasons, we only show the refinement results.

$$HoldsAt(IsIn(t, NBegin, NEnd), t) \rightarrow \\ Happens(tiltIn, t1) \wedge t2 = t1 + suffshuttime \wedge \\ t1 < t2 \leq t \wedge Prohibit(tiltOut, t1, t) \qquad \text{(SFb)}$$

The specification ensures that the window is shut when the night falls and remains shut during the night. Since the window is robust in its response to, for instance, the tiltIn event when it is already shut (it remains shut), or when it is already tilting in (it keeps tilting in), these cases are covered by SFb. Therefore, we obtain the full specification for the security feature from a disjunction of the conclusions in SFa and SFb as shown below:

$$HoldsAt(IsIn(t, NBegin, NEnd), t) \rightarrow \\ ((Initially(WindowShut) \wedge Prohibit(tiltOut, 0, t)) \\ \vee (Happens(tiltIn, t1) \wedge t2 = t1 + suffshuttime \wedge \\ t1 < t2 \leq t \wedge Prohibit(tiltOut, t1, t))) \qquad \text{(SF)}$$

Applying the same refinement technique, two partial specifications for TR are derived. The first partial specification deals with the case where the window was initially open and nothing has changed, whilst the second partial specification

deals with the significant case where the machine needs to tilt out the window sufficiently before the temperature difference becomes large.

Again from these two partial specifications, we obtain the following full specification for the climate control and energy efficiency feature.

$$\begin{array}{l} HoldsAt(InTemp > NiceTemp + 1, t) \wedge \\ HoldsAt(InTemp > OutTemp + 1, t) \rightarrow \\ ((Initially(WindowOpen) \wedge Prohibit(tiltIn, 0, t)) \vee \\ (Happens(tiltOut, t1) \wedge t2 = t1 + suf fopentime \wedge \\ t1 < t2 \leq t \wedge Prohibit(tiltIn, t1, t))) \end{array} \quad \text{(CCF)}$$

4 Composing Features

Having derived the specifications for individual features, we now turn to the question of how to compose requirements and feature specifications, using Composition Frames. Since, as Section 2.1 argued, the requirements of the features are not fully consistent, it is not possible to meet the conjunction of SR and TR requirements completely. We will see that the use of Event Calculus in deriving feature specifications in Section 3 and the introduction of the $Prohibit(\alpha, \tau 1, \tau 2)$ predicate in particular, now give us a more succinct approach to reasoning about the composition controller semantics that we require. Using a family of weakened conjunction operators adapted from [19], we formulate the following ways of combining two general requirements R1 and R2, expressed in terms of control on domains. For the window example, R1 and R2 can be regarded as SR and TR respectively.

- **Option 1: No Control.** Let R1 $\wedge_{\{any\}}$ R2 be the requirement that R1 and R2 should each be met at times when they are not in conflict; but there is no requirement that any conflicts should be resolved and if there are times when conflicts occur, any emergent behaviour is acceptable. For example, the window might sometimes oscillate in a partly open position.
- **Option 2: Exclusion.** Let R1 $\wedge_{\{control\}}$ R2 be the requirement that both R1 and R2 should hold at all times except when the system is actively attempting to satisfy R1, R2 may not be satisfied during that time; and vice versa. The exclusion here is symmetrical. For example, SR might not be satisfied while TR is keeping the window open, and TR might not be satisfied while SR is keeping the window shut.
- **Option 3: Exclusion with Priority.** Let R1 $\wedge_{\{R1\}}$ R2 be the requirement that both R1 and R2 should hold at all times except when the system is attempting to satisfy R1, R2 may not be satisfied during that time. The exclusion here is asymmetrical in favor of R1.
- **Option 4: Exclusion & Fine Grain Priority.** Let R1 $\wedge_{\{important,R1\}}$ R2 be the requirement that R1 $\wedge_{\{R1\}}$ R2 holds, except that any sub-requirement associated with the phenomenon *important* should be given top priority.

Fig. 5 shows how SR and TR may be recomposed with the Composition Frame. This diagram is a product of a simple syntactic transformation involving

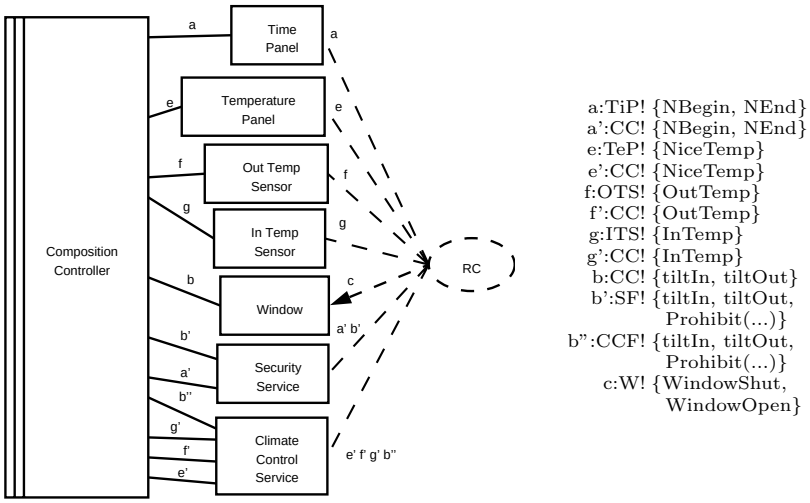


Fig. 5. SR and TR fitted to the Composition Frame

two steps. First, we introduced a new machine, the Composition Controller, between the machine domain Security Feature (SF) and the world domains (Time Panel and Window) in Fig. 1. The original machine domain (SF) became a world domain in the new diagram, and the phenomena a and b were split by insertion of the new machine. Now, Time Panel, for example, reports to the new machine (phenomena a prefixed by the Time Panel domain TiP) and the new machine may pass it on to the SF domain (phenomena a' prefixed by the composition controller CC). The same transformation was also applied to the problem diagram in Fig. 2. Second, the resulting two diagrams were merged to give the diagram in Fig. 5.

We also added the $Prohibit(\alpha, \tau1, \tau2)$ events to the phenomena b' and b". These prohibit events will be generated on the basis of the $Prohibit(\alpha, \tau1, \tau2)$ predicates in our feature specification. The composition controller will interpret them, possibly acting on them and possibly ignoring them, in order to resolve conflicts.

We will now specify four versions of the composition controller in Fig. 5 that meet the composition requirement RC as described by each of the conjunction operators (Options 1-4). To choose a resolution of the requirement conflict between SR and TR is to choose the appropriate composition controller.

Composition Controller for SR $\wedge_{\{any\}}$ TR. The semantics of the first type of composition operator is straightforward. We use a simple formalism to describe the semantics of the controller in which $\rightarrow$ should be read as stating that the composition controller generates the event on the right when the event on the left happens.

Definitions (1 to 4) in Fig. 6 say that the events from Time Panel, Temperature Panel, Out Tempt Sensor and In Tempt Sensor are passed to the SF and

a:e → a':e	(1)	g:e → g':e	(4)
e:e → e':e	(2)	b':e → b:e	(5)
f:e → f':e	(3)	b":e → b:e	(6)

Fig. 6. The semantics of SR $\wedge_{\{any\}}$ TR

CCF domains respectively without prohibition. Similarly in (5 and 6) the events from SF and CCF are propagated to the window without prohibition. That is, all of the prohibit events transmitted in the interfaces b' and b" to the composition controller are ignored. Since the controller applies no prohibition on events generated by the domains, in particular by SF and CCF domains, any emergent behaviour of the window is possible. For example, if SF has generated tiltIn to shut the window, and as a result the window is closing, and in the mean time the CCF domain generates the tiltOut event to open the window, the composition controller will allow CCF to open the window.

In order to address the other composition operators, it is necessary for the composition controller to remember and act on some of the prohibit events it has received. For this purpose, an additional, but quite minimal, machinery is required. Let P be a set that hold tuples of form (e,t1,t2,m) which will represent an assertion that event e is prohibited by the specification of machine m between times t1 and t2. We now allow the → to be guarded by an optional predicate (enclosed in square brackets following the first operand). In the following specifications for composition controller, we assume that no machine can prohibit another machine issuing a prohibit event.

Composition Controller for SR $\wedge_{\{control\}}$ TR. The controller semantics for dealing with events generated by world domains (1 to 4) applies to this controller. Definitions (5.a to 5.d) and (6.a to 6.d) replace (5) and (6) respectively. Note that t in the expression t1 ≤ t ≤ t2 in Fig. 7 denotes current time.

Definitions 1 to 4 and the following:

b':prohibit (e, t1, t2) → insert((e, t1, t2, 'SF'), P)	(5.a)
b':e [∀ t1, t2, m · t1 ≤ t ≤ t2 ∧ m ≠ 'SF' ∧ (e, t1, t2, m) ∉ P] → b:e	(5.b)
b':e [∃ t1, t2, m · t1 ≤ t ≤ t2 ∧ m ≠ 'SF' ∧ (e, t1, t2, m) ∈ P] → ignore	(5.c)
b':e [∀ t1, t2, m · t1 ≤ t ≤ t2 ∧ m = 'SF' ∧ (e, t1, t2, m) ∈ P] → error	(5.d)
b":prohibit(e, t1, t2) → insert((e, t1, t2, 'CCF'), P)	(6.a)
b":e [∀ t1, t2, m · t1 ≤ t ≤ t2 ∧ m ≠ 'CCF' ∧ (e, t1, t2, m) ∉ P] → b:e	(6.b)
b":e [∃ t1, t2, m · t1 ≤ t ≤ t2 ∧ m ≠ 'CCF' ∧ (e, t1, t2, m) ∈ P] → ignore	(6.c)
b":e [∀ t1, t2, m · t1 ≤ t ≤ t2 ∧ m = 'CCF' ∧ (e, t1, t2, m) ∈ P] → error	(6.d)

Fig. 7. The semantics of SR $\wedge_{\{control\}}$ TR

Controller semantics (5.a) says that when the domain SF issues a prohibition on the event e between t1 and t2, the composition controller records the assertion by adding a tuple into P. When SF issues any other event, the controller passes on the event to the window domain, only if the event has not been prohibited by another machine for that time (5.b); otherwise the event is ignored (5.c). If self-prohibitions happen, an error is generated, (5.d). (6.a to 6.d) describes the controller dealing with the events from CCF in a similar fashion. In effect, this controller gives to the SF and CCF domains mutually exclusive control of the window domain over a period of time.

Composition Controller for SR $\wedge_{\{SR\}}$ TR. The semantics of this controller differs from the previous one in one respect: since events from the prioritized machine SF should not be prohibited, (5.b to 5.d) are not necessary. (5.a) is needed in order that SF can prohibit events and (5) is added in order that SF events are passed on to the window domain unprohibited, thus giving SF events precedence over events from CCF. CCF events are handled in the same way as before (6.a to 6.d).

Composition Controller for SR $\wedge_{\{emgOpenWindow,SR\}}$ TR. Assume that SF and CCF can open the window in emergency situations (for example, if a fire is detected in the house) by firing the emgOpenWindow event. Again, the semantics of this controller differs from the previous in one respect: since the prioritized event, emgOpenWindow, from the CCF machine should not be prohibited, (6.e) is added. (5) already allows the emgOpenWindow event from the SF machine to pass unprohibited.

$$\text{b''}:\text{emgOpenWindow} \rightarrow \text{b}:\text{e} \qquad (6.e)$$

It is easy to see that there is nothing in the above composition controller semantics that refers directly to the machine specifications or requirements of the sub-problems. If we treat Fig. 5 as a composition pattern, then the controller we have specified is actually generic, and can be applied to any requirements R1 and R2 that can be specified using the Event Calculus of Section 2.2.

5 Related Work

Our work is related, first and foremost, to the feature interaction problem, common in the field of telecommunications [16,27], as well as other domains such as email [13]. In particular it is found in application domains where feature interactions are manifest in the environment rather than inside the software [17]. While less ambitious about the extent to which requirements can be composed, our work is also less domain-specific. In [28], work is presented on the conjunction of specifications as composition in a way that addresses multiple specification languages, but the emphasis is less on the relationship between requirements and specifications. Nakamura et al [21] propose an object-oriented approach to

detecting feature interactions in services of home appliances. However, their approach uses a design-time, rather than run-time, technique.

The whole area of inconsistency management offers a variety of contributions to dealing with inconsistencies in specifications [9,10,11]. Robinson [24], in particular, reviews a variety of techniques for requirements interaction management and Nuseibeh et al [23] discuss a range of ways of acting in the presence of inconsistency. None of these approaches address the decomposition and recomposition of requirements to facilitate problem solving.

A number of formal approaches exist where emergent behaviours due to composition can be identified and controlled [1,7]. Our approach differs from these in that we identify how requirements interact and remove non-deterministic behaviour by imposing priorities over the requirements set.

In [8], a run-time technique for monitoring requirements satisfaction is presented. This approach is taken further in [6], where requirements are monitored for violations and system behaviour dynamically adapted, whilst making acceptable changes to the requirements to meet higher-level goals. This requires that alternative system designs be represented at run-time. One view of our approach is that it involves the monitoring of when a requirement leads to a machine taking control (including event prohibition) and the taking of appropriate action. Our approach differs further, in that it is more lightweight: we do not need to maintain alternative system designs at run-time.

In [15] we sketched some options in composing a sluice gate control machine with a safety machine in order to address safety concerns. That was in the context of a more philosophical discussion of composition and decomposition. The work presented in this paper differs in that we embody the composition as a separate extra machine. This gives us the potential to deal with a wider range of compositions.

The Event Calculus has previously been used in software development for reasoning about evolving specifications [5,25] and distributed systems policy specifications [2]. Our work should be seen as complementary to such approaches in that it will allow inconsistencies to be resolved at run-time.

Finally, our approach is strongly related to the mutual exclusion problem of concurrent resource usage, but with an explicit emphasis on requirements satisfaction.

6 Discussion

In solution space terms composition controllers correspond to the notion of an architectural connector [1]. This allows us to move backwards and forwards between architectural and requirements perspectives using the Composition Frame as a reasoning tool.

We now consider how our work can be generalized, alternative composition semantics and the significance of the work.

It is well understood that in producing a machine to solve a real-world problem there is often a need to implement an analogic model [14] of at least part of

the problem domain. Arriving at a conceptual model that can subsequently be implemented is often difficult in itself. In the case of the SF and CCF machines, the models are very simple. This is partly because of the domain assumption that the window is robust. If the window is less robust, it is necessary to explicitly model the position of the window. Composing machines containing such models can be complex because the model in one machine may become inconsistent with the world, due to the world being changed by another machine.

It is not difficult to see how the Composition Frame can be generalized to any two machines with a common domain under their control. In the specification we used the notion of a particular machine being in control of the window, including passive partial control specified using the $Prohibit(\alpha, \tau 1, \tau 2)$ predicate. The same technique should be usable with any two machines.

Although our Composition Frame in this example deals with two problems fitting a type of problems called the Required Behaviour Frame, it is easy to see that it would generalize to composing two problems fitting other basic Problem Frames [14] in a similar fashion. For example, in [20] we demonstrate how to compose two problems fitting the Required Behaviour and Commanded Behaviour frames.

Whilst much work has been done on protocols for controlling mutual access to resources in program code, less attention seems to have been paid to the problem of systematically gaining control over domains in the real world [14]. Working explicitly with the notion of a machine being in control at certain times, and the use of a temporal semantics, allows us to express the concerns at the requirements stage. In particular, our requirements composition operators make the issue of control explicit.

7 Conclusions and Future Work

We have shown how by expressing requirements and domain properties in a temporal logic we can formally derive feature specifications. In itself this refinement style approach is not new. However, we have placed it in the context of a development process based on Problem Frames. The value of this is that in making the properties of the application domain explicit, we increase our confidence that the specified machine will meet the system requirements. Furthermore, by adding the $Prohibit(\alpha, \tau 1, \tau 2)$ predicate to the Event Calculus and making use of it in machine specifications we have obtained an important new element in our toolbox for composing solutions to feature subproblems. The composition controller needs only to be parameterized and the composition is done non-intrusively in the sense that we have made no changes to the specifications of the machines being composed. We have illustrated this through the application of our approach to an awning window control system in a smart home application.

We have also shown how to combine two inconsistent requirements in terms of the operators given in Section 4. The Composition Frame allowed us to reason about the relationship between sub-solutions and sub-requirements. We were

able to specify composition at a requirements level rather than solely in design or implementation terms.

We believe that our approach is scalable, as composition controllers have a simple semantics. Although the specification is in terms of set operations, it would be simple to bound the size of these sets in practice and to implement them efficiently.

Future work is planned to formalize the relationship between our requirements composition operators, the Problem Frames for sub-problems, and the composition requirements. We also need to address a wider range of compositions, both in terms of the options in Section 4 and across a larger set of basic Problem Frames. In a large Problem Frames development, sub-parts of domains and amalgamations of domains can appear in different frames. Related to this is the need to apply the approach to more significant case studies. It might be possible to develop patterns for particular domain areas. Given the use of formal derivations of machine specifications, we are developing a reasoning tool to automate our approach in order to support its use in larger systems.

8 Acknowledgements

We are grateful for the support of our colleagues at The Open University, in particular, Arosha Bandara, Leonor Barroca, Charles Haley, Jon Hall, Lucia Rapanotti and Michel Wermelinger, and Alexandra Russo of Imperial College. We also acknowledge the financial support of EPSRC for this research.

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.
2. A. K. Bandara, E. Lupu, and A. Russo. Using event calculus to formalise policy specification and analysis. In *POLICY*, pages 26–39. IEEE Computer Society, 2003.
3. D. Bjørner. Towards posit & prove calculi for requirements engineering and software design: In honour of the memory of professor Ole-Johan Dahl. In O. Owe, S. Krogdahl, and T. Lyche, editors, *Essays in Memory of Ole-Johan Dahl*, volume 2635 of *LNCS*, pages 58–82. Springer, 2004.
4. M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.
5. A. S. d'Avila Garcez, A. Russo, B. Nuseibeh, and J. Kramer. Combining abductive reasoning and inductive learning to evolve requirements specifications. *IEE Proceedings - Software*, 150(1):25–38, 2003.
6. M. S. Feather, S. Fickas, A. V. Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *Proceedings of IWSSD'98: 9th International Workshop on Software Specification and Design, Ise-Shima*, Japan, 1998. IEEE Computer Society Press.
7. J. L. Fiadeiro, A. Lopes, and M. Wermelinger. A mathematical semantics for architectural connectors. In R. C. Backhouse and J. Gibbons, editors, *Generic Programming*, volume 2793 of *LNCS*, pages 178–221. Springer, 2003.

8. S. Fickas and M. Feather. Requirements monitoring in dynamic environments. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, pages 140 – 147, 1995.
9. A. Finkelstein and I. Sommerville, editors. *Special Issue of the BCS/IEE Software Engineering Journal on "Multiple Perspectives"*. 1996.
10. C. Ghezzi and B. Nuseibeh, editors. *Special Issues on Inconsistency Management in IEEE Transactions on Software Engineering*. 1998.
11. C. Ghezzi and B. Nuseibeh, editors. *Special Issues on Inconsistency Management in IEEE Transactions on Software Engineering*. 1999.
12. J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. Relating software requirements and architectures using problem frames. In *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 137–144. IEEE Computer Society, 2002.
13. R. J. Hall. Fundamental nonmodularity in electronic mail. *Automated Software Engineering*, 12(1):41–79, 2005.
14. M. Jackson. *Problem Frames*. ACM Press & Addison Wesley, 2001.
15. M. Jackson. Why software writing is difficult and will remain so. *Inf. Process. Lett.*, 88(1-2):13–25, 2003.
16. M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Trans. Softw. Eng.*, 24(10):831–847, 1998. http://dx.doi.org/10.1109/32.729683.
17. M. Kolberg, E. H. Magill, and M. Wilson. Compatibility issues between services supporting networked appliances. *IEEE Communications Magazine*, 41(11):136–147, 2003.
18. R. Kowalski and M. Sergot. A logic-based calculus of events. *New Gen. Comput.*, 4(1):67–95, 1986.
19. R. Laney, L. Barroca, M. Jackson, and B. Nuseibeh. Composing requirements using problem frames. In *Proceedings of 12th IEEE International Conference Requirements Engineering (RE'04)*, pages 122–131. IEEE Computer Society, 2004.
20. R. Laney, M. Jackson, and B. Nuseibeh. Composing problems: Deriving specifications from inconsistent requirements. Technical Report 2005/08, The Open University, 2005.
21. M. Nakamura, H. Igaki, and K.-I. Matsumoto. Feature interactions in integrated services of networked home appliances: An object-oriented approach. In S. Reiff-Marganiec and M. Ryan, editors, *FIW*, pages 236–251. IOS Press, 2005.
22. B. Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–117, 2001.
23. B. Nuseibeh, S. Easterbrook, and A. Russo. Making inconsistency respectable in software development. *Journal of Systems and Software*, 58(2):171–180, 2001.
24. W. N. Robinson, S. D. Pawlowski, and V. Volkov. Requirements interaction management. *ACM Computing Surveys*, 35(2):132–190, 2003.
25. A. Russo, R. Miller, B. Nuseibeh, and J. Kramer. An abductive approach for analysing event-based requirements specifications. In P. J. Stuckey, editor, *ICLP*, volume 2401 of *LNCS*, pages 22–37. Springer, 2002.
26. M. Shanahan. The event calculus explained. *LNCS*, 1600:409–430, 1999.
27. P. Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, 26(8):20–30, 1993.
28. P. Zave and M. Jackson. Conjunction as composition. *ACM Trans. Softw. Eng. Methodol.*, 2(4):379–411, 1993.