SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **39**:973–1002
Published online 8 May 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.923

Specifying features of an evolving software system

Thein Than Tun[1,*,†], Tim Trew[2], Michael Jackson[1], Robin Laney[1] and Bashar Nuseibeh[1]

[1]*Department of Computing, The Open University, Milton Keynes MK7 6AA, U.K.*
[2]*NXP Semiconductors, High Tech Campus 37, 5656 AE Eindhoven, The Netherlands*

SUMMARY

Software development is increasingly concerned with maintaining and extending existing software systems to meet the evolving user requirements. Many of these systems are feature-rich and are developed incrementally. As structures of existing software systems—in addition to the user requirements—influence the specifications, specifying these systems poses unique challenges. This paper reports on our experience of applying an engineering approach to specifying an evolving feature-rich television software system. In this approach, features are specified modularly by first fitting their problems to known problem patterns, and then analyzing typical concerns—meaning the potential causes of errors—associated with those patterns. In cases where the existing design poses difficulties when fitting problems to patterns, we transform its structure using known design mechanisms so that the problems fit the patterns. After deriving specifications of individual features, possible interactions between features are detected, before declaratively specifying resolutions to undesired interactions. As the concerns of features and their composition are addressed separately, the specifications derived are modular, thus, providing rich treaceability to their requirements. As well as discussing how features may be specified using natural language, we also show how their descriptions may be formalized using a form of temporal logic called the Event Calculus, and prove their correctness using an off-the-shelf tool. Copyright © 2009 John Wiley & Sons, Ltd.

Received 19 March 2008; Revised 13 February 2009; Accepted 19 February 2009

KEY WORDS: software features; evolution; specifications; feature composition; problem frames

1. INTRODUCTION

It is widely recognized that incremental or evolutionary development of software is now the rule, not the exception: even where evolution was not initially envisaged by the developer,

*Correspondence to: Thein Than Tun, Department of Computing, The Open University, Milton Keynes MK7 6AA, U.K.
†E-mail: t.t.tun@open.ac.uk

Contract/grant sponsor: EPSRC; contract/grant number: EP/C007719/1

it is necessitated by feedback from customers and users, by evolving requirements, and by competitive market pressures [1,2]. The evolutionary dimension highlights the need for clarity in the relationship between changing customer requirements and a stable yet maintainable system structure. Our goal is to understand how requirements for *features*, or 'coherent and identifiable bundle[s] of system functionality' [3], can be systematically mapped to software modules, and vice versa, and to make that mapping a practical intellectual tool in software development and maintenance.

A simple example from feature-rich consumer television (TV) software illustrates the need for such an intellectual tool. TVs in the past had few features and they were relatively simple and easy to implement. For example, the problem of turning the audio on and off in response to the user command could be solved by a simple program involving as little as ten lines of code. As the software system evolves, new features are continually added. These include features to suppress noises when TV signals are unstable, to mute the sound when the child lock is on, and so on. As a result, this seemingly easy operation of muting and unmuting the TV sound becomes very complex. Giving assurance that the system will continue to function correctly after new features have been introduced becomes a challenge. There are several ways to approach this problem. For instance, Philips has used a domain-specific TV architecture [4] and the Koala component model [5], based on build-time binding of reusable components. Anton and Potts [6] have studied the functional evolution of a long-lived software system in order to understand the nature and extent of this evolution.

In this paper we apply an engineering approach that addresses this challenge by focusing on two specific issues. *Developers are better at solving specific software problems with which they are familiar*. One key part of our solution, therefore, is a way to make unfamiliar and complex problems familiar and simpler by decomposing, transforming and fitting subproblems into known patterns of problem structures, called *problem frames* [7]. *Formal analysis can be helpful when reasoning about critical parts of a system*. Therefore, we apply a lightweight formal description language called the Event Calculus (EC), and in doing so we extend this language to make our descriptions succinct, and perform rigorous analysis when necessary. The main contribution of the paper is an engineering approach to specifying evolving feature-rich software, by drawing on past development experience and formal analysis.

1.1. Overview of the methodology

Figure 1 shows an overview of the methodology used in this paper. In this methodology, user requirements for features in TV software, as in many real software development projects, are assumed to be expressed in an informal Natural Language (NL). The requirements statements describe software problems in the physical world context which need to be solved.

Problem analysis. The Problem Frames (PF) approach is used to analyze those problems by fitting each of them into one of the known basic frames—such as the required behavior, commanded behavior, information display, simple workpieces and transformation frames—or a composite frame [7]. A problem fitted to a frame is typically described using a Problem Diagram (PD), with accompanying NL descriptions of the requirement and the problem world context in the diagram. Unlike the requirements statements given by the user, these descriptions are separated and structured according to the chosen frame. Feature specifications are then

DOI: 10.1002/spe

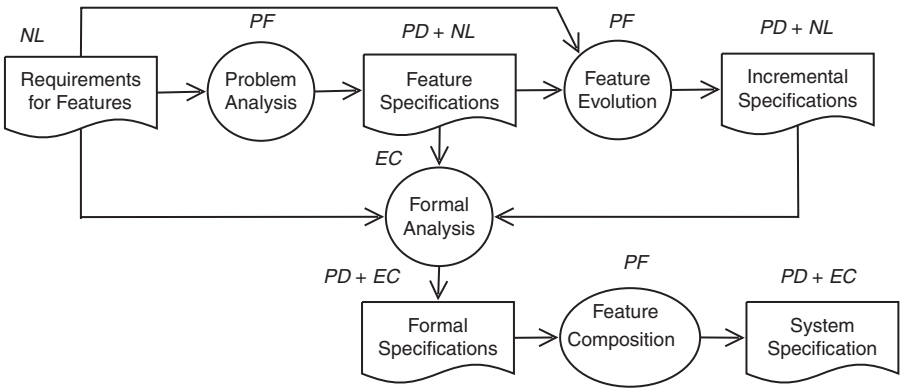


Figure 1. Overview of our methodology.

derived using cause-to-effect and effect-to-cause reasoning on thosedescriptions. Each PF has encoded some knowledge about the typical concerns of a type of software problem. By fitting problems of TV features into basic and composite frames, specifications of these features and their composition benefit from the developer's past experience of dealing with similar problems.

The description of a solution to the problem(s) posed by a user requirement for a feature is called a feature specification. These specifications are modular in the sense that they can be mapped onto the requirements easily. When specifying incremental features (due to new requirements), the existence of partial solutions in the existing design often contributes to the complexity of problem structures.

Feature evolution. Two rules are introduced in order to add *wrappers* which transform and simplify complex problem structures and fit them into known PF. Therefore, the concerns of features can still be analyzed by applying the frames. By treating existing components as black boxes, modular specifications for those features are derived using the same approach used in the problem analysis. Again, we use PDs and their NL descriptions to describe the incremental specifications.

The specifications derived so far are expressed using NL, reflecting the widespread practice in real development. However, for specifications of certain critical features, deviation from the expected behavior is not acceptable, and therefore their correctness needs to be checked rigorously.

Formal analysis. The EC is used to formalize relevant NL descriptions of PDs, and prove the soundness and completeness of the specifications with respect to their requirements through logical deduction. This reasoning can be done both manually and with the help of off-the-shelf reasoning tools. In this paper, the Discrete EC Reasoner [8,9] is used to prove the correctness of feature specifications.

Although specifications for individual features have been derived, the overall system behavior has not yet been specified. One major difficulty of composing individual features is the inconsistencies between their requirements. Having clear and explicit problem structures and descriptions helps to identify the potential interactions between features.

DOI: 10.1002/spe

Feature Composition. Overlapping elements between problem structures of different features are identified as potential sites for feature interactions and their defined properties are examined in the context of the composition. When necessary, compositional wrappers are introduced and specified to detect and resolve runtime conflicts.

1.2. Organization

The remainder of the paper is organized as follows. Section 2 introduces the key concepts in the PF approach and EC, before the PF approach is applied to derive informal specifications of two features in Section 3. Descriptions of a simple feature are formalized and analyzed using the EC in Section 4. Section 5 describes application of simple rules and general patterns for evolving existing designs when implementing new features, and discusses how an incremental feature specification may be formalized and analyzed. Possible interactions between three features are detected and resolved in Section 6. Concluding remarks are given in Section 7.

2. PRELIMINARIES

We begin with a short discussion of the PF approach by summarizing its basic vocabulary and how we will apply this approach in specifying the features in TV software. The discussion is illustrated by a simple feature of user-commanded muting and unmuting of the TV sound. This section also gives an overview of our choice of formalism, EC.

2.1. Descriptions in the PF approach

Specifying a feature using the PF approach begins with an analysis of the problem and its context. The PF approach emphasizes that descriptions of each problem should cover three things:

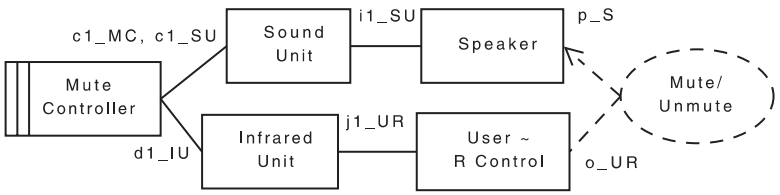
- the *problem world domains* that are relevant to the problem,
- the *requirement* that needs to be satisfied, and
- the *machine*, a programmed computer, that the developer needs to build.

In addition to these descriptions, there should be an *adequacy argument* that justifies how the description of the machine, or the specification, together with the descriptions of the problem world domains, are sufficient in satisfying the requirement [7]. In our specification of features in TV software, we will follow this separation of descriptions.

2.2. The mute/unmute feature

With this relatively simple feature of most TVs, the problem is that of muting/unmuting the TV sound in response to user commands. An analysis of the problem using the PF approach involves fitting the problems into some known frames, and considering their concerns. For example, this muting/unmuting problem can be fitted to a frame called the *Commanded Behavior* frame [7]. In doing so, we typically draw a *problem diagram*, such as the one in Figure 2, to describe the structure of the problem, i.e. the requirement, the problem world domains, the machine, and how they are connected to each other.

DOI: 10.1002/spe



c1_MC:MC!{setMuteOn, setMuteOff}
d1_IU:IU!{toggleMute}
j1_UR:UR!{signalMuteButton}
o_UR:UR!{pressMuteButton}

c1_SU:SU!{MuteOn}
i1_SU:SU!{startAudioStream, stopAudioStream}
p_S:S!{NoSound}

Figure 2. Problem diagram for muting/unmuting.

2.2.1. *Requirement*

The NL description of the user-commanded muting/unmuting problem in TV software is given as follows:

> *When the user presses the mute button on the TV remote control, muting and unmuting of the TV sound should be toggled.*

In Figure 2, this requirement is indicated by the dotted oval Mute/Unmute. Dotted and solid lines are called *interfaces*. These interfaces represent a set of *phenomena*, classes of events and properties controlled and shared by the domains involved. The dotted lines *o_UR* and *p_S*, representing the *requirement phenomena*, reference some phenomena of the User ~ R Control domain ('When the user presses the mute button on the TV remote control'), and constrains some phenomena of the Speaker domain ('muting and unmuting of the TV sound should be toggled'). These phenomena are often referred to as *reference* and *constraint* phenomena, respectively.

2.2.2. *Problem world domains*

Problem world domains, represented by plain rectangles in Figure 2, are some abstraction or projection of objects in the physical world with which the machine is concerned. Table I describes the relevant properties of the problem world domains in Figure 2. These properties of the domains are assumed to be pre-determined, and therefore cannot be modified by software engineers. For example, the fact that the remote control produces infrared signals—rather than some other signals—when its buttons are pressed cannot be changed by software engineers.

Interfaces between problem world domains, such as *j1_UR* and *i1_SU*, represent a set of *shared phenomena* between the domains involved. A special case of shared phenomena, at the interface between problem world domains and the machine such as *d1_IU* and *c1_MC*, is called the *specification phenomena*. When an event class or a property is shared between two domains, one domain typically *controls* it, and the other domain *observes* it. In the description of the interface *j1_UR*, *UR*!*signalMuteButton* means that generation of an instance of the event (henceforth an event) *signalMuteButton* is controlled by the domain User ~ R Control, while the Infrared Unit only observes

DOI: 10.1002/spe

Table I. Descriptions of problem world domains.

Domains	Descriptions
User ∼ R Control	When the user presses the mute button on the remote control (*o_UR*), the property of the remote control *MuteIsPressed* becomes true for an instant if the control is functioning properly. When *MuteIsPressed* is true, the control sends out an instance of the event class for muting/unmuting (henceforth an event) *signalMuteButton* (*j1_UR*)
Infrared Unit	On receipt of the event *signalMuteButton* from a source near the TV (*j1_UR*), Infrared Unit sets *MuteRequired* to true for an instant. When *MuteRequired* is true the event *toggleMute* is fired (*d1_IU*)
Sound Unit	The event *setMuteOn* makes *MuteOn* true and the event *setMuteOff* makes *MuteOn* false (*c1_MC*). Sound Unit stops the audio stream to the speaker, denoted by the event *stopAudioStream*, when *MuteOn* becomes true, and starts the audio stream to the speaker, denoted by the event *startAudioStream*, when *MuteOn* becomes false (*j1_UR*)
Speaker	On receipt of the event *startAudioStream* the speaker produces the audible sound, and similarly on receipt of the event *stopAudioStream* the speaker produces no audible sound (*p_S*)

it. In our descriptions of phenomena, by convention, property names begin with capital letters and event names begin with small letters.

2.2.3. Specification

In software development, the requirement and properties of the domains relevant to the requirement are given, and the task of specifying the software is to find specifications of machines, represented by a rectangle with two vertical stripes in Figure 2. The important obligation here is that the specification, within the context of the relevant problem world domains, must satisfy the requirement [7].

2.3. The EC

First introduced by Kowalski and Sergot [10], the EC is a system of logical formalism, which draws from first-order predicate calculus. It can be used to represent actions, their deterministic and non-deterministic effects, concurrent actions and continuous change [11]. 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, and the version we adopted here is based on the discussions in [12].

2.3.1. Key predicates

This predicate calculus has three main sorts: (i) sort $\mathscr{A}$ for actions, or events, with variables a, $a1$, $a2$, ..., (ii) sort $\mathscr{F}$ for fluents, or boolean states, with variables f, $f1$, $f2$, $f3$, ..., and (iii) sort $\mathscr{T}$ for time points with non-negative integer variables t, $t1$, $t2$, $t3$, ... Predicates are relations between these sorts and the standard predicates in EC are summarized in Table II. The table says, for example, that the $Happens$ predicate is a subset of the relations between events and time

DOI: 10.1002/spe

Table II. Basic and auxiliary predicates of the Event Calculus.

Predicate	Syntax	Meaning
Happens$\subseteq \mathscr{A} \times \mathscr{T}$	Happens(A, T)	Action A occurs at time T
HoldsAt$\subseteq \mathscr{F} \times \mathscr{T}$	HoldsAt(F, T)	Fluent F holds at time T
Initiates$\subseteq \mathscr{A} \times \mathscr{F} \times \mathscr{T}$	Initiates(A, F, T)	If action A happens at time T, fluent F becomes true at time $T+1$
Terminates$\subseteq \mathscr{A} \times \mathscr{F} \times \mathscr{T}$	Terminates(A, F, T)	If action A happens at time T, fluent F becomes false at $T+1$
$< \subseteq \mathscr{T} \times \mathscr{T}$	$T1 < T2$	Time point $T1$ is before time point $T2$
Clipped$\subseteq \mathscr{T} \times \mathscr{F} \times \mathscr{T}$	Clipped($T1$, F, $T2$)	Fluent F is terminated between times $T1$ and $T2$

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

$$HoldsAt(f, t2) \leftarrow [Happens(a, t1) \wedge Initiates(a, f, t1) \wedge t1 < t2 \wedge \neg Clipped(t1, f, t2)] \quad \text{(EC2)}$$

Figure 3. Some domain-independent rules in the Event Calculus.

points, written *Happens* $\subseteq \mathscr{A} \times \mathscr{T}$, where *Happens*$(A, T)$ indicates that the event A occurs at time T. Similarly, the predicate *HoldsAt*(F, T) indicates that the fluent F is true at time T, and so on.

2.3.2. Domain-independent rules

In addition to standard predicates, EC has domain-independent rules which define a theory of how fluents change their values over time. In Figure 3, the rule (EC1) says that *Clipped*$(t1, f, t2)$ is a shorthand for writing that an event with a terminating effect on the fluent f occurs between $t1$ and $t2$. The rule (EC2) says that if an event a has happened and initiated the fluent f at time $t1$, which is before time $t2$, and if the fluent f is not terminated between $t1$ and $t2$, then the fluent f will hold at time $t2$. There are other EC domain-independent rules such as these, but we have included only the rules necessary for our discussions. These rules are assumed to be complete, meaning that fluents cannot change their values without following such explicit rules.

2.4. Using the EC in the PF approach

The way that the EC can be used in the PF approach is discussed in [13]. In addition, if an event or a fluent is a part of an interface—especially, if the event or fluent is involved in a problem structure transformation—we will parameterize its name with the name of the interface. For example, we will write $Happens(e1(p), t1)$ to say that the event $e1$ is generated by a controlling domain at the interface p at the time $t1$. For readability, we parameterize the event and fluent names only when necessary.

Similarly when describing the effect of an event on a fluent that is controlled by a domain, we will parameterize the fluent name with the name of the domain. For example, *Initiates*$(e1(p), f2(DB), t)$

DOI: 10.1002/spe

says that when the event $e1$ occurs at the interface p, the fluent $f2$ controlled by Domain B becomes true. Given an action $Happens(e1(p), t1)$ and the domain rule $Initiates(e1(p), f2(DB), t)$, we may conclude that the fluent $f2$ of Domain B is true at $t1+1$ according to (EC2). Like fluent and event names, parameterized names of fluents and events are also unique. For example, given two interfaces $p1$ and $p2$, and an event name e, if $p1 \neq p2$, then $e(p1) \neq e(p2)$.

When describing the behavior of a wrapper machine, we will also use the predicate $PassedOn(e(p1), t, e(p2), d)$, which is a shorthand to say that if the event e at the interface $p1$ happens at the time t, another event e at the interface $p2$ happens at time $t+d$. Formally,

$$PassedOn(e(p1), t, e(p2), d) \stackrel{\text{def}}{\equiv} [Happens(e(p1), t) \wedge d \geq 0 \rightarrow Happens(e(p2), t+d)] \quad \text{(DR1)}$$

The predicate *PassedOn* guarantees that the event gets passed on eventually. In some cases, it may be necessary for a wrapper machine to ignore an event completely by not passing it on. An easy, implicit way to do this, for example, is not to have an appropriate *PassedOn* predicate in the description of the wrapper machine. When it is necessary to say explicitly that a wrapper does not pass on an event, we will say $Fail(e(p1), t, e(p2))$ meaning that when the event e at interface $p1$ occurs at time t, the corresponding event e at the interface $p2$ will be generated at a time point in infinity, ∞.

$$Fail(e(p1), t, e(p2)) \stackrel{\text{def}}{\equiv} PassedOn(e(p1), t, e(p2), \infty) \quad \text{(DR2)}$$

In the same way that a wrapper may delay events being communicated between some domains, it may also delay reporting of fluents changing their values

$$Report(f1(p1), t, f1(p1'), d) \stackrel{\text{def}}{\equiv} [HoldsAt(f1(p1), t) \leftrightarrow HoldsAt(f1(p1'), t+d) \wedge d \geq 0] \quad \text{(DR3)}$$

The above definition (DR3) says that whatever the value of the fluent $f1$ at interface $p1$ at time t, the fluent $f1$ at the interface $p1'$ at time $t+d$ has the same value, and vice versa.

3. PROBLEM ANALYSIS

Having fitted the problem of muting/unmuting to the Commanded Behavior frame in Section 2, we now show how to specify the Mute/Unmute feature. Methodologically, we will perform informal cause-to-effect and effect-to-cause reasoning in order to derive the feature specification. As a result of fitting the muting/unmuting problem to a known frame, we will also examine the typical concerns of the frame, and revise the specification if necessary. We then reason about the correctness of a specification informally using the *adequacy argument*. Finally, we discuss how problems of complex features may also be fitted to known composite frames, and how specifications of those features are derived.

DOI: 10.1002/spe

3.1. Specifying a simple feature

There are several approaches to finding specifications for machines such as the Mute Controller; for example, [14–17] suggest systematic techniques for obtaining specifications from requirements. We highlight the general idea of deriving specifications from requirements in the PF approach before discussing the application of a lightweight approach with explicit temporal reasoning in Section 4.

In the PF approach, requirements are always expressed in terms of requirement phenomena at the interfaces such as *p_S* and *o_UR*, while specifications are always expressed in terms of specification phenomena at the interfaces such as *c1_MC* and *d1_IU*. Therefore, finding a specification for a machine is in effect a systematic rewrite of the descriptions of the requirement phenomena into appropriate descriptions of the specification phenomena. Refining descriptions of the reference phenomena such as those at *o_UR* typically requires cause-to-effect reasoning, and refining descriptions of the constraint phenomena such as those at *p_S* typically requires effect-to-cause reasoning.

For example, the Mute/Unmute requirement refers to the phenomena at *o_UR* by saying:

When the user presses the mute button on the TV remote control, ...

If the remote control is functioning properly, *MuteIsPressed* becomes true for an instant, which fires the event *signalMuteButton* according to the description of User $\sim$ R Control in Table I. Therefore, the requirement can be rewritten in terms of the event at *j1_UR* as follows:

When the event signalMuteButton is fired, ...

If the remote control generating the event *signalMuteButton* was near the TV, the description of Infrared Unit in Table I allows this event to be rewritten in terms of the event at *d1_IU* as follows:

When the event toggleMute is fired, ...

Refining the constraint phenomena *p_S* requires a different reasoning approach. In this case, we have to ask what must happen at *i1_SU*, and *c1_MC*, in order that there is no speaker sound at *p_S*. According to our domain description, to mute or unmute the sound, Sound Unit should fire either *stopAudioStream* or *startAudioStream* at *i1_SU*. Generating those events at *i1_SU* is determined by whether *MuteOn* of Sound Unit is true or false. *MuteOn* in turn is set true or false by the *setMuteOn* and *setMuteOff* events, respectively, at *c1_MC*. We can therefore conclude that *setMuteOn* will lead to the TV speaker producing no audible sound, and *setMuteOff* will lead to the TV speaker producing audible sound. The constraint phenomena at *p_S* which say:

...muting and unmuting of the TV sound should be toggled.

can now be rewritten as:

...the event setMuteOff should be fired if MuteOn is true, and the event setMuteOn should be fired if MuteOn is false.

This completes our search for the specification of the Mute Controller machine. By conjoining the descriptions of machine phenomena at *d1_IU* and *c1_MC*, we obtain the specification for the Mute Controller, which says:

When the event toggleMute is fired, the event setMuteOff should be fired if MuteOn is true, and the event setMuteOn should be fired if MuteOn is false.

DOI: 10.1002/spe

3.2. Addressing concerns

Fitting a feature to a known problem pattern enables us to apply past experience of solving Commanded Behavior problems to the muting/unmuting problem. These concerns serve as a checklist of issues to consider when analyzing a particular feature specification.

For example, the behaviors of human agents in a Commanded Behavior frame are not necessarily causal: they are *biddable* [7]. If the TV takes some time units to mute or unmute the sound, an obvious concern to consider is *disobedience*: what happens if the user presses the mute button several times very quickly? Stakeholders may decide to address the concern in a number of ways: for example, by weakening the requirement ('it does not matter whatever happens if the user presses the mute button too many times too quickly'), or by writing a stronger specification ('once the user command is received, ignore repeated commands for 5 time units'). In the latter case, the specification may be rewritten as follows:

> *When the event toggleMute is fired and it has not been fired for 5 time units, the event setMuteOff should be fired if MuteOn is true, and the event setMuteOn should be fired if MuteOn is false.*

Physical domains, though considered causal, may have *reliability* concerns: what if the faulty Infrared Unit does not faithfully toggle the muting when the event *signalMuteButton* is fired? In that case the Mute Controller will fail to satisfy the requirement. If this is a critical requirement, stakeholders may want to have other ways of muting/unmuting the sound.

There is an *identity* concern: what if the mute signal is generated by a device other than the TV user with a remote control? Clearly, there is a possibility of interferences: the sound may be muted when the users want them unmuted and vice versa. The developer will have to ask: Is that acceptable to the stakeholders?

There is also an *initialization* concern: should the TV have its sound muted or unmuted when the TV is switched on for the first time?

There are several benefits in considering known concerns when analyzing specifications: (i) since these concerns are informed by past experience, perhaps by experience of design errors, repeat of similar design mistakes can be prevented, (ii) since these concerns are relatively simple and intuitive, the search space for potential errors can be narrowed, and (iii) questions raised by these concerns can lead to a better understanding of the specification and its limitations. This is the basis for our claim that we are applying an engineering approach.

3.3. Adequacy argument

The adequacy argument is typically a causal argument linking the reference phenomena of a requirement to its constrained phenomena. For example, an informal adequacy argument for the Mute Controller specification runs as shown in Figure 4. The argument shows that the specification for the mute/unmute feature is sound (i.e. correct with respect to our descriptions). In Section 4, with the help of the reasoning tool, we will attempt to show that the specification is also complete (i.e. there is no other plausible explanations in our descriptions for, for example, the sound being turned on and off).

It is, however, important to note that the casuality link from the phenomena at *o_UR* to the phenomena at *p_S* holds only under certain conditions of the domains involved. These conditions include (i) the remote control device functions correctly and (ii) the remote control is near the TV.

DOI: 10.1002/spe

o_UR	When the user with the remote control presses the mute button, ...
o_UR–j1_UR	since the remote control is functioning correctly, it generates the signal for the mute button, and ...
j1_UR–d1_IU	since the control is near the TV, the infrared unit picks up the signal, and notifies that the muting should be toggled, and ...
d1_IU–c1_MC	since the machine, depending on whether the speaker is already producing sound or not, generates the events for muting and unmuting when notified that the muting should be toggled, and ...
c1_MC–i1_SU	since the sound unit knows exactly whether the sound signals are passing to the speaker or not at any time, and since it starts and stops sound signals to the speaker correctly in response to the mute/unmute events it receives, and ...
i1_SU–p_S	since the speaker mutes or unmutes the sound depending on whether the sound signals are coming in or not, ...
p_S	the speaker will be muted if it was not mute, and unmuted if it was muted, thus satisfying the requirement.

Figure 4. Informal adequacy argument for the Mute Controller specification.

Furthermore, there are also limitations on how we can interpret such a causality link. For example, the description assumes that it is the user with the remote control that produces the control signal for muting, and say, not a cat that happens to step on the remote control.

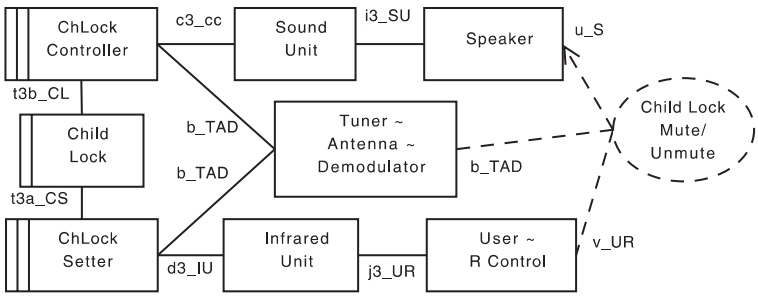
For this reason, the notion of the specifications being sound and complete is bound by the completeness and consistency of the problem world domain descriptions we are working with, and by our interpretation of these descriptions. One can reasonably argue that there is always a part of the problem world missing in our domain descriptions which may turn out to be significant for the specification to be sound and complete. Expertise and experience of domain experts, when systematically recorded and exploited, may provide a degree of certainty for success in dealing with this issue.

3.4. Specifying complex features

We now consider the problem in a child lock feature. With the child lock feature of the TV, owners of the TV can selectively 'lock' specific channels in order that they are not accessible by certain viewers. Typically when the child lock is enabled for a particular channel, the screen should be blanked and the sound turned off when the TV is tuned to that channel. In this section we consider, for clarity, only the part of this feature related to controlling the TV sound. In practice, users access this lock through a menu item protected by a PIN, which although related, is a separate problem, and is not discussed here.

The PD for this feature is described in Figure 5. It is interesting to note the difference between this diagram and the PD for the muting/unmuting feature in Figure 2. Child lock settings are persistent—once the lock is enabled, it should remain enabled until it is disabled—and the problem fits a composite frame. This means that there are two subproblems in this feature: one concerned with the issue of the user setting the child lock on/off for a particular channel (ChLock Setter), and the other concerned with the issue of muting/unmuting according to the status of the lock and the channel to which the TV is currently tuned (ChLock Controller).

DOI: 10.1002/spe



t3b_CL:CL!{ChannelID, LockEnabled}
t3a_CS:CS!{ChannelID, LockEnabled}
i3_SU:SU!{startAudioStream, stopAudioStream}
b_TAD:TAD!{CurrentChannelID}
v_UR:UR!{pressChLockButton}
c3_CC:CC!{setMuteOn, setMuteOff}
d3_IU:IU!{toggleChLock}
u_S:S!{NoSound}
j3_UR:UR!{signalChLockButton}

Figure 5. Problem diagram for the child lock feature.

3.4.1. Requirement

The requirement of this simplified child lock feature is informally stated below:

If the user has enabled the child lock for a particular TV channel, when the TV is tuned to that channel, the TV sound should be muted on that channel, until the lock for the channel is disabled.

3.4.2. Problem world domains

Many of the problem world domains in Figure 5 are the same as those in Figure 2. The new domain Child Lock in Figure 5 records whether child lock is enabled—indicated by whether *LockEnabled* is true or false—for each channel identified by a channel ID. Tuner∼Antenna∼Demodulator has the property that indicates the ID of the channel to which the TV is currently tuned (*CurrentChannelID*).

3.4.3. Specifications

By following the informal approach discussed in Section 3.1, we can first obtain the specification of the ChLock Setter described below:

When the event toggleChLock happens, LockEnabled should be set to true for the channel ID of the current channel if LockEnabled was previously false; LockEnabled should be set to false for the channel ID of the current channel if LockEnabled was previously true.

Similarly, we can describe the specification of the ChLock Controller as follows:

If the ID of the currently tuned channel is LockEnabled, the event setMuteOn should be fired. If the ID of the currently tuned channel is not LockEnabled, the event setMuteOff should be fired.

DOI: 10.1002/spe

3.4.4. Addressing concerns

As in the previous example, *disobedience*, *reliability*, *identity* and *initialization* are obvious concerns.

4. FORMAL ANALYSIS OF A SIMPLE FEATURE

We now show how to use the EC in the PF approach as an aid to make the descriptions precise, perform rigorous analysis when necessary, and provide a basis for automated analysis. Methodologically, we first formalize the requirements, domain descriptions and the specification of a feature, in this case, the Mute/Unmute feature. Then we outline the structure of the formal argument, which, although more detailed, closely resembles the informal adequacy argument. The correctness of the specification is then proved using tool-assisted logical deduction. Since we give full explanations of all formulae we use, we hope that the discussions can be followed without having to read the formulae in detail. First we show how to formalize the descriptions of requirements, problem world domains and machine specification of the muting/unmuting problem in Figure 2.

4.1. Formalizing requirement

We begin by formalizing the requirement for the mute/unmute feature given in Section 2.2.1. Translation of NL statements into the EC is quite straightforward.

The formula below (R1a) says that if the user presses the mute button and if the sound is on when the button is pressed, the system should switch off the sound quickly. The variable *mutedelay* in the formula defines how many time units 'quickly' means.

$$HoldsAt(NoSound, t) \leftarrow Happens(pressMuteButton, t1) \wedge$$
$$\neg HoldsAt(NoSound, t1) \wedge t1 < t \leq t1 + mutedelay \quad \text{(R1a)}$$

The following formula (R1b) says that if the sound is off when the button is pressed, the system should switch on the sound quickly.

$$\neg HoldsAt(NoSound, t) \leftarrow Happens(pressMuteButton, t1) \wedge$$
$$HoldsAt(NoSound, t1) \wedge t1 < t \leq t1 + mutedelay \quad \text{(R1b)}$$

4.2. Formalizing relevant domain properties

We also formalize the necessary properties of problem world domains described in NL in Section 2.2.2 and Table I as shown below:

$$Initiates(pressMuteButton, MuteIsPressed, t) \leftarrow HoldsAt(ControlWorking, t) \quad \text{(D1)}$$

$$Happens(autoTermMIP, t) \leftarrow HoldsAt(MuteIsPressed, t) \quad \text{(D2)}$$

DOI: 10.1002/spe

$$Terminates(autoTermMIP, MuteIsPressed, t) \tag{D3}$$

$$Happens(signalMuteButton, t) \leftarrow HoldsAt(MuteIsPressed, t) \tag{D4}$$

$$Initiates(signalMuteButton, MuteRequired, t) \tag{D5}$$

$$Happens(autoTermMR, t) \leftarrow HoldsAt(MuteRequired, t) \tag{D6}$$

$$Terminates(autoTermMR, MuteRequired, t) \tag{D7}$$

$$Happens(toggleMute, t) \leftarrow HoldsAt(MuteRequired, t) \tag{D8}$$

$$Initiates(setMuteOn, MuteOn, t) \tag{D9}$$

$$Terminates(setMuteOff, MuteOn, t) \tag{D10}$$

$$Happens(stopAudioStream, t) \leftarrow HoldsAt(MuteOn, t) \wedge \neg HoldsAt(MuteOn, t-1) \tag{D11}$$

$$Initiates(stopAudioStream, NoSound, t) \tag{D12}$$

$$Happens(startAudioStream, t) \leftarrow \neg HoldsAt(MuteOn, t) \wedge HoldsAt(MuteOn, t-1) \tag{D13}$$

$$Terminates(startAudioStream, NoSound, t) \tag{D14}$$

The descriptions say that when the user presses the mute button, the property *MuteIsPressed* of the remote control unit becomes true, if the control is working properly (D1), and the property *MuteIsPressed* remains true for a time unit (D2 and D3); if *MuteIsPressed* is true, the event *signalMuteButton* is generated by the remote control unit (D4), which sets *MuteRequired* of the infrared unit to true (D5), and *MuteRequired* remains true for one time unit (D6 and D7); when *MuteRequired* is true, the event *toggleMute* is fired (D8); when the event *setMuteOn* is fired, *MuteOn* becomes true (D9); when the event *setMuteOff* is fired, *MuteOn* becomes false (D10); on property *MuteOn* becoming true, the event *stopAudioStream* is fired (D11), which stops the speaker producing audible sound by making *NoSound* true (D12); and similarly on property *MuteOn* becoming false, the event *startAudioStream* is fired (D13), which makes the speaker produce audible sound by making *NoSound* false (D14). Notice that we have at least one Initiate and one Terminate predicate for each fluent, suggesting a degree of completeness of the predicates.

4.3. Formalizing the specification

In the entailment defining the relationship between the requirement (R), properties of the problem world domains (W), and the specification (S), W, S $\models$ R, we have now defined (W) and (R) for this particular subproblem. That is, we now have $[(D1) \wedge \cdots \wedge (D14)]$, S $\models$ $R1$. The question then is: What is the (minimal) specification of S so that this entailment holds? There are two main ways to find such specifications: one we may call *correctness by construction*, and the other *correctness by proof*. In [14], we discuss an example of the former, where we use a systematic refinement technique to derive specifications, supported by an automated tool [18]. In this paper, we follow the latter

DOI: 10.1002/spe

approach. Therefore, we will simply formalize the informal specification given in Section 2.2.3 before proving it.

The following partial specification (S1a) says that the Mute Controller should generate the event *setMuteOn* when the event *toggleMute* is fired and *MuteOn* is false.

$$Happens(setMuteOn, t1) \leftarrow Happens(toggleMute, t1) \wedge \neg HoldsAt(MuteOn, t1) \tag{S1a}$$

Similarly, the following partial specification (S1b) says the Mute Controller should generate the event *setMuteOff* when the event *toggleMute* is fired and *MuteOn* is true.

$$Happens(setMuteOff, t1) \leftarrow Happens(toggleMute, t1) \wedge HoldsAt(MuteOn, t1) \tag{S1b}$$

4.4. Adequacy argument

It is now possible to prove, or give a formal argument for, the soundness and completeness of the specifications. The proof below relies on the uniqueness of names (i.e. no two names refer to the same thing) denoted by the U operator, and completeness of predicates (i.e. circumscription) denoted by the CIRC operator [12].

Suppose that when the user presses the mute button on the remote control at the time point 0, the TV has the sound on, the control is working and it is near the TV. In EC, this can be translated into the following statements:

$$Happens(pressMuteButton, 0) \tag{I1}$$

$$HoldsAt(ControlWorking, 0) \tag{I2}$$

$$HoldsAt(ControlNearTV, 0) \tag{I3}$$

$$\neg HoldsAt(NoSound, 0) \tag{I4}$$

For consistency, we make explicit that the fluents *MuteRequired* and *MuteRequired* are also false at the same time point

$$\neg HoldsAt(MuteRequired, 0) \tag{I5}$$

$$\neg HoldsAt(MuteOn, 0) \tag{I6}$$

First we will structure the formal adequacy argument by constructing a simple proposition for the proof.

Proposition. Let $\Sigma = (D1) \wedge (D3) \wedge (D5) \wedge (D7) \wedge (D9) \wedge (D10) \wedge$(D12)$\wedge$(D14), $\Delta_1 =$(I1), $\Delta_2 =$ (S1a)$\wedge$(D2)$\wedge$(D4)$\wedge$(D6)$\wedge$ (D8)$\wedge$(D11)$\wedge$(D13), $\Omega = U[pressMuteButton, autoTermMIP, autoTermMR, signalMuteButton, toggleMute, setMuteOn, stopAudioStream, startAudioStream] \wedge U$ [*Control*

DOI: 10.1002/spe

Working, *MuteIsPressed*, *MuteRequired*, *ControlNearTV*, *MuteOn*, *NoSound*], $\Gamma = (I2) \wedge \cdots \wedge (I6)$, and $EC = (EC1) \wedge (EC2)$. Then we have,

$$CIRC[\Sigma; Initiates, Terminates] \wedge CIRC[\Delta_1 \wedge \Delta_2; Happens] \wedge \Omega \wedge \Gamma \wedge EC \vDash (R1a) \tag{P1}$$

Proof. In this deductive proof, we suppose *mutedelay* to be 5 time units. As in the informal adequacy argument, we will follow these steps: (1) obtain the specific goal we are proving, (2) relate the reference phenomena to appropriate machine phenomena, (3) deduce the machine action, and (4) relate the machine action to the controlled phenomena. The first step simply prepares for the proof and can be done manually. For the remaining steps of the proof, we use the tool Discrete EC Reasoner [8,9] to encode the left-hand side of Proposition (P1) as a SAT problem and use the solver Relsat [19,20] to find all the possible models. If the solver finds exactly one model showing that *NoSound* becomes true within the time required, the specification is both sound and complete.

The first step is straightforward: (I1), (I2), the value of *mutedelay* and (R1a) yield the goal to prove as $HoldsAt(NoSound, t2) \wedge t2 \leq 5$. The remainder of the proof is generated by the tool, and is annotated and shown in Figure 6.

For the second step, the tool first deduces, from (i) the event *pressMuteButton* happening at time point 0, (ii) the fluent *ControlWorking* holding at time point 0, and (iii) the (EC2) rule, that the fluent *MuteIsPressed* is true at time point 1. Then from (D2) and (D4) the tool obtains the events *autoTermMIP* and *signalMuteButton* at time point 1. The fluent *MuteRequired* becomes true in the next time point as a result of the event *signalMuteButton*, (D5) and (EC2). The fluent *MuteIsPressed* is then terminated by *autoTermMIP*. As a result of the fluent *MuteRequired* becoming true, the events *toggleMute* and *autoTermMR* are generated at the same time, according to (D6) and (D8).

```
0                                  <-- time point
  ControlNearTV().                 <-- a given fluent
  ControlWorking().
  Happens(PressMuteButton(), 0).   <-- occurrence of a given or deduced event
1
 +MuteIsPressed().                 <-- a fluent becoming true
  Happens(autoTermMIP(), 1).
  Happens(signalMuteButton(), 1).
2
 -MuteIsPressed().                 <-- a fluent becoming false
 +MuteRequired().
  Happens(ToggleMute(), 2).
  Happens(SetMuteOn(), 2).
  Happens(AutoTermMR(), 2).
3
 -MuteRequired().
 +MuteOn().
  Happens(TurnOffSound(), 3).
4
 +NoSound().
P                                  <-- end of proof
```

Figure 6. Tool-generated proof of P1.

DOI: 10.1002/spe

For the third step, the tool immediately deduces the event *setMuteOn* from the event the event *toggleMute* and the given fact that *MuteOn* was false, according to (S1a).

For the fourth step, the tool deduces that the fluent *MuteOn* becomes true in time point 3, which fires off the event *stopAudioStream* according to (D11). As a result, the fluent *NoSound* becomes true according to (D12) and (EC2) at time point 4. Since this is the only model found by the solver, the specification is both sound and complete. The other case of the TV initially having the sound off can be proved in the same fashion. □

5. EVOLVING FEATURE SPECIFICATIONS

Suppose that, having solved the muting/unmuting problem, it came to light that a variation, albeit a small variation, of the muting/unmuting functionality is required. The requirement of this new feature of 'beep when mute button is pressed' is as follows:

When the user requests TV to mute, *the TV should beep before muting or unmuting.*

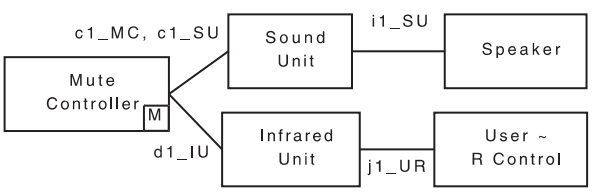
When the Mute Controller machine has already been implemented, it is desirable not to have to change its specification directly.

The problem context of the new feature is similar to the muting/unmuting feature in Figure 2. Since we prefer not to modify the Mute Controller, it will be taken as a given domain in our new problem context in Figure 7, where the label M denotes the fact that the Mute Controller is an implemented machine. The presence of the implemented machine makes it difficult to fit the beeping problem into the Commanded Behavior frame as it is not clear where the machine should be introduced. In order to fit this problem into a frame, we will first transform this problem structure.

5.1. Transformation rules

If a problem structure does not neatly fit a frame, we may apply some of the following transformation rules in order to make the problem better fit a frame.

1. Given an interface *p_DA* between a machine and a world domain, we may insert a new machine between the two domains as indicated in Figure 8. In doing so, the control of the



c1_MC:MC!{setMuteOn, setMuteOff}
c1_SU:SU!{MuteOn}
d1_IU:IU!{toggleMute}

i1_SU:SU!{startAudioStream, stopAudioStream, makeBeepSound}
j1_UR:UR!{signalMuteButton}

Figure 7. Problem world domains for the new feature.

DOI: 10.1002/spe

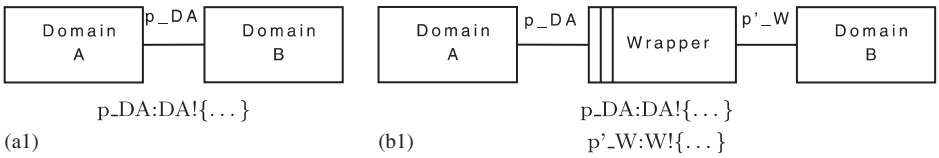


Figure 8. Transformation Rule #1: Domain addition: (a1) before adding a wrapper and (b1) after adding a wrapper.

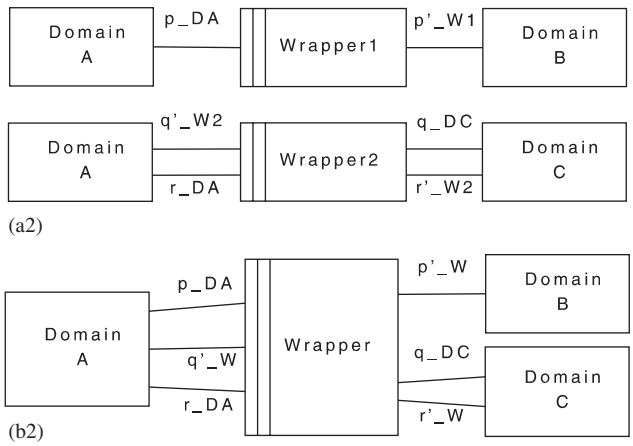


Figure 9. Transformation Rule #2: Merging domains: (a2) before merging and (b2) after merging.

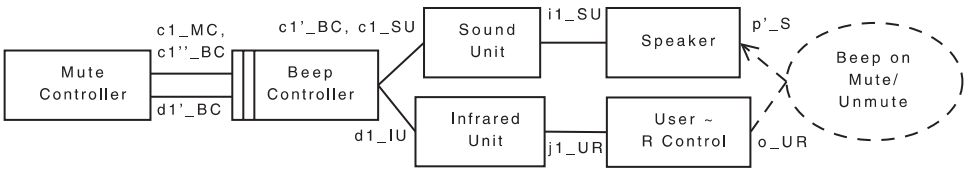
phenomena is preserved: for example, if the phenomena of *p_DA* are controlled by Domain A before the transformation, they are still controlled by Domain A after the transformation. The specification of the Wrapper domain determines if and how the phenomena at the two interfaces *p_DA* and *p′_W* are related.

2. Given two machines, we may merge them while preserving the interfaces (Figure 9). If the two machines are non-interacting—that is, their properties and events are totally independent of each other—then a parallel composition can be achieved; otherwise, composition concerns arise [7].

By applying the first rule, we insert new machines at the interfaces *d1_IU* and *c1_MC* of Figure 2, and then by applying the second rule to merge the two new machines, we obtain the PD for the 'beep when mute button is pressed' feature in Figure 10. The new machine, the Beep Controller, in effect wraps around the Mute Controller, enabling it to manipulate the behavior of the mute/unmuting feature through its input and output at *d1′_BC* and *c1_MC*. For example, by simply instantaneously reporting fluent changes and passing on events from *d1_IU* to *d1′_BC* and *c1_MC* to *c1′_BC*, the Beep Controller can maintain the integrity of the original solution.

However, in order to satisfy the requirement, the Beep Controller has to do more than this reporting. We will first work through this simple example before giving a general discussion of the diverse utility of the wrapper mechanism in the requirements engineering context (Section 5.3).

DOI: 10.1002/spe



c1_MC:MC!{setMuteOn, setMuteOff}
c1'_BC:BC!{setMuteOn, setMuteOff, makeBeep}
i1_SU:SU!{startAudioStream, stopAudioStream, makeBeepSound}
c1"_BC:BC!{MuteOn}
p'_S:S!{Beep+NoSound, Beep+¬NoSound}
d1'_BC:BC!{toggleMute}
d1_IU:IU!{toggleMute}
c1_SU:SU!{MuteOn}
j1_UR:UR!{signalMuteButton}
o_UR:UR!{pressMuteButton}

Figure 10. Problem diagram for the new feature.

Using the approach suggested in Section 3.1, we obtain the specification of the Beep Controller as:

When the event toggleMute at d1_IU is fired, generate the event makeBeep at c1′_BC and report the value of MuteOn from c1_SU to c1″_BC. When either setMuteOn or setMuteOff is generated at c1_MC, pass on the event to c1′_BC after 1 time unit.

This specification treats the specification of the Mute Controller as a black box. When *toggleMute* is fired, the Beep Controller generates the *makeBeep* event at the same time, and reports the fact that *toggleMute* has been fired to the Mute Controller. It is still up to the Mute Controller to generate the *setMuteOn* or *setMuteOff* event, but the Beep Controller makes sure that these events are delayed by one time unit to allow time to beep.

Therefore, we can make our specifications modular by treating the existing components as black boxes and specifying incremental functionality. Although this modularity provides rich traceability between the requirements and the software component, it may also imply additional complexity of specifications. How this trade-off should be managed is an interesting practical issue that is beyond the scope of the paper.

5.2. Formal analysis of the mute controller

In order to show the correctness of this incremental feature, we first formalize the requirement, domain descriptions and the specification.

5.2.1. The incremental requirement

When formalizing the incremental requirement, for space reasons, we focus on muting and ignore unmuting, while noting that formalization of the unmuting requirement is not dissimilar.

$$\begin{aligned}[HoldsAt(Beep,t)\wedge HoldsAt(NoSound,t+1)]\leftarrow\\ Happens(pressMuteButton,t1)\wedge\\ \neg HoldsAt(NoSound,t1)\wedge t1<t\leq t1+mutedelay\end{aligned}\qquad(R1^{\Delta 1})$$

DOI: 10.1002/spe

The requirement ($R1^{\Delta 1}$) says that if the TV is muted (within the required time range) as a result of the user pressing the mute button, the TV should beep once before muting.

5.2.2. *Additional problem world properties*

First we introduce a new event *makeBeep* at $c1'\_BC$ in Figure 10 which when fired initiates a property of Sound Unit *BeepOnce* (D15), which holds for exactly one time unit (D16 and D17); when *BeepOnce* is true, the event *makeBeepSound* is fired (D18); and when the event is fired, the fluent *Beep* becomes true (D19) for one instant (D20 and D21)

$$Initiates(makeBeep, BeepOnce, t) \tag{D15}$$

$$Happens(autoTermBO, t) \leftarrow HoldsAt(BeepOnce, t) \tag{D16}$$

$$Terminates(autoTermBO, BeepOnce, t) \tag{D17}$$

$$Happens(makeBeepSound, t) \leftarrow HoldsAt(BeepOnce, t) \tag{D18}$$

$$Initiates(makeBeepSound, Beep, t) \tag{D19}$$

$$Happens(autoTermB, t) \leftarrow HoldsAt(Beep, t) \tag{D20}$$

$$Terminates(autoTermB, Beep, t) \tag{D21}$$

Notice that we have assumed, for the sake of clarity, that the beep lasts a fixed time length, and the requirement does not constraint the duration of the beep. In cases when the TV hardware allows beeps of variable durations, and the requirements constrains the duration of the beeps, they can be handled in the same way as muting and unmmuting.

Finally, we have to resolve how the Beep Controller will report the changes of *MuteOn* at $c1\_SU$ to $c1''\_BC$. In this particular transformation, the Beep Controller will not delay the reporting of the fluent (D22)

$$Report(MuteOn(c1\_SU), t, MuteOn(c1''\_BC), 0) \tag{D22}$$

5.2.3. *Additional specification*

We can now formalize the specification of the Beep Controller as follows:

$$\begin{aligned} &[Happens(makeBeep(c1'\_BC), t) \wedge \\ &PassedOn(setMuteOn(c1\_MC), t, setMuteOn(c1'\_BC), 1) \wedge \\ &PassedOn(toggleMute(d1\_IU), t, toggleMute(d1'\_BC), 0)] \leftarrow \\ &Happens(toggleMute(d1\_IU), t) \end{aligned} \tag{$S1^{\Delta 1}$}$$

DOI: 10.1002/spe

5.2.4. *Adequacy argument*

We now give a formal adequacy argument for the above specification. As in the previous case, the proof relies on uniqueness of names and predicate completion.

Proposition. Let $\Sigma = (\text{D1}) \wedge (\text{D3}) \wedge (\text{D5}) \wedge (\text{D7}) \wedge (\text{D9}) \wedge (\text{D10}) \wedge (\text{D12}) \wedge (\text{D14}) \wedge (\text{D15}) \wedge (\text{D17}) \wedge (\text{D19}) \wedge (\text{D21})$, $\Delta_1 = (\text{I1})$, $\Delta_2 = (\text{S1a}) \wedge (S1^{\Delta 1}) \wedge (\text{D2}) \wedge (\text{D4}) \wedge (\text{D6}) \wedge (\text{D8}) \wedge (\text{D11}) \wedge (\text{D13}) \wedge (\text{D16}) \wedge (\text{D18}) \wedge (\text{D20}) \wedge (\text{D22})$, $\Omega = U[\mathit{pressMuteButton}, \mathit{autoTermMIP}, \mathit{autoTermMR}, \mathit{signalMuteButton}, \mathit{toggleMute}, \mathit{setMuteOn}, \mathit{stopAudioStream}, \mathit{startAudioStream}, \mathit{makeBeep}, \mathit{autoTermBO}, \mathit{makeBeep Sound}, \mathit{autoTermB}] \wedge U[\mathit{ControlWorking}, \mathit{MuteIsPressed}, \mathit{MuteRequired}, \mathit{ControlNearTV}, \mathit{MuteOn}, \mathit{NoSound}, \mathit{BeepOnce}, \mathit{Beep}]$, $\Gamma = (I2) \wedge \cdots \wedge (I6)$, and $EC = (EC1) \wedge (EC2)$. Then we have,

$$\mathit{CIRC}[\Sigma; \mathit{Initiates}, \mathit{Terminates}] \wedge \mathit{CIRC}[\Delta_1 \wedge \Delta_2; \mathit{Happens}] \wedge \Omega \wedge \Gamma \wedge EC \models (R1^{\Delta 1}) \qquad (P1^{\Delta 1})$$

Proof. The proof is deductive and, as in the previous case, *mutedelay* takes the value of 5. We again follow the steps set out in the previous proof. In the first step, we find the goal to prove as $\mathit{HoldsAt}(\mathit{Beep}, 4) \wedge \mathit{HoldsAt}(\mathit{NoSound}, 5)$. For the remaining steps, the tool finds the only model shown in Figure 11.

The two proofs are similar in many ways. The interesting difference is that at time point 2 when the machine action is deduced, the tool also finds the event *makeBeep* being fired at that point. When combined with (D15) and (EC2), it finds the fluent *BeepOnce* to be holding at time point 3. Significantly, the event *setMuteOn_c1* fired at time point 2 does not have any effect on *MuteOn* at time point 3. The event is simply passed on to *setMuteOn_c1prime*, which sets *MuteOn* to true at time point 4, thus, achieving the delay required before muting the sound.

The fluent *Beep* starts to hold at time point 4, as a result of the event *makeBeepSound* at time point 3 and (EC2). *Beep* is terminated at the next time point, as the fluent *NoSound* starts to hold. Therefore, the speaker produces the beep sound before the sound is turned off as required.

```
0                                    |  3
  ControlNearTV().                   |   -MuteRequired().
  ControlWorking().                  |   +BeepOnce().
  Happens(PressMuteButton(), 0).     |    Happens(AutoTermBO(), 3).
1                                    |    Happens(MakeBeepSound(), 3).
 +MuteIsPressed().                   |    Happens(SetMuteOn_c1prime(), 3).
  Happens(AutoTermMS(), 1).          |  4
  Happens(SignalMute(), 1).          |   -BeepOnce().
2                                    |   +Beep().
 -MuteIsPressed().                   |   +MuteOn().
 +MuteRequired().                    |    Happens(AutoTermB(), 4).
  Happens(AutoTermMR(), 2).          |    Happens(TurnOffSound(), 4).
  Happens(MakeBeep(), 2).            |  5
  Happens(SetMuteOn_c1(), 2).        |   -Beep().
  Happens(ToggleMute(), 2).          |   +NoSound().
                                     |  P
```

Figure 11. Tool-generated proof of $P1^{\Delta 1}$.

DOI: 10.1002/spe

Again, the other case of the speaker initially producing no sound can be proved in the same manner. □

In summary, the discussions so far show that we can extend feature specifications by adding further predicates and rules into our descriptions without necessarily having to modify the existing ones directly. This allows us to keep clear mappings between requirements and specifications of features, and makes automation of proofs more manageable.

5.3. Patterns of wrapping

The wrapping machine has diverse utility, and architectural implications. Introducing the Beep Controller is not so different from introducing a component in a pipe-and-filter architecture. In this sense we are working with a model of system development that allows problem and solution structures to influence each other. We now briefly discuss some patterns of wrapping in our approach.

5.3.1. Total wrapping

In the mute and beep controllers example, we have allowed the Beep Controller to manipulate both inputs and outputs of the Mute Controller (Figure 10). This is both a problem space and a solution space decision. It is a problem space decision, because we have decided that, according to the requirement, the Beep Controller needs to manipulate *d1_IU* and *c1_SU*. On the other hand, this is a solution space decision, because, as we shall see, different styles of wrapping lead to different implementation architectures.

5.3.2. Partial wrapping

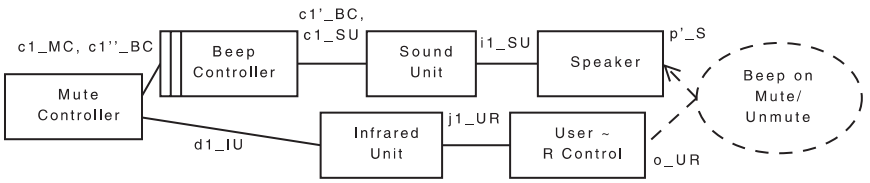
In fact, there are two alternatives to this total wrapping. It is possible to have a wrapper that manipulates either input or output, but not both, of a component.

Output-only wrapper. In this case, the wrapper manipulates only the output part of a component. For example, the alternative PD in Figure 12 suggests that the Beep Controller can manipulate the output of the Mute Controller only. The requirement assumption is that *whenever* the Mute Controller generates the *setMuteOn* event, the Beep Controller should send a *makeBeep* event to Sound Unit ahead of the *setMuteOn* event. This decision has important implications. If the Mute Controller also generate the *setMuteOn* in response to events other than the user pressing the mute button, then the Beep Controller will also generates the *makeBeep* event on such instances. The new axioms in our extended domain theory allow us to describe this wrapper neatly as follows:

$$\begin{aligned} &[Happens(makeBeep(c1'\_BC), t) \wedge \\ &PassedOn(setMuteOn(c1\_MC), t, setMuteOn(c1'\_BC), 1)] \leftarrow \\ &Happens(setMuteOn(c1\_MC), t,) \end{aligned} \qquad (P1^{\Delta 2})$$

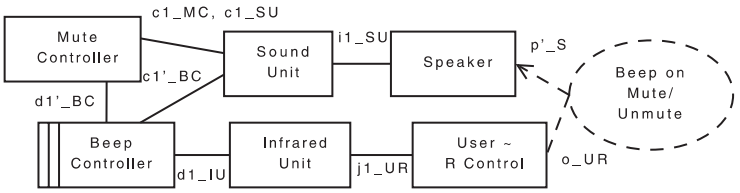
Input-only wrapper. The wrapper, in this case, manipulates only the input part of a component. The alternative PD in Figure 13 suggests that the Beep Controller can manipulate the input to the

DOI: 10.1002/spe



c1_MC:MC!{setMuteOn, setMuteOff}
c1'_BC:BC!{setMuteOn, setMuteOff, makeBeep}
c1_SU:SU!{MuteOn}
c1"_BC:BC!{MuteOn}
p'_S:S!{Beep+NoSound, Beep+¬NoSound}
i1_SU:SU!{startAudioStream, stopAudioStream}
d1_IU:IU!{toggleMute}
j1_UR:UR!{signalMuteButton}
o_UR:UR!{pressMuteButton}

Figure 12. An output-only wrapper for the new feature.



c1_MC:MC!{setMuteOn, setMuteOff, makeBeep}
c1_SU:SU!{MuteOn}
c1'_BC:BC!{setMuteOn, setMuteOff, makeBeep}
o_UR:UR!{pressMuteButton}
p'_S:S!{Beep+NoSound, Beep+¬NoSound}
i1_SU:SU!{startAudioStream, stopAudioStream}
d1_IU:IU!{toggleMute}
d1'_BC:BC!{toggleMute}
j1_UR:UR!{signalMuteButton}

Figure 13. An input-only wrapper for the new feature.

Mute Controller only. It can do so by, for example, delaying the passing of the event *toggleMute* from *d1_IU* to *d1′_BC* by one time unit and sending the *BeepOnce* event to Sound Unit at *c1′_BC* immediately

$$PassedOn(toggleMute(d1\_IU), t, toggleMute(d1'\_BC), 1) \wedge$$
$$Happens(makeBeep(c1'\_BC), t) \leftarrow Happens(toggleMute(d1\_IU), t) \qquad (P1^{\Delta 3})$$

There are several advantages and disadvantages of these different formulations of the Beeping problem. The choice of a particular solution may be ultimately determined by various constraints including those of stakeholders. With these examples, we have demonstrated that the transformation of problem structures opens up these opportunities and allows the developer to explore problem and solution alternatives.

5.3.3. *Other wrappers*

In general, wrappers, like other domains, work with client–server-type communications where a wrapper knows the domain to which it is sending the events (even though the events may be

DOI: 10.1002/spe

intercepted by another domain) but not the domain from which it is receiving events. Therefore, a wrapper may not be able to discriminate between same-typed events coming from different sources. In Figure 13, for example, the Sound Unit does not know whether a *makeBeep* event is coming from the Mute Controller or the Beep Controller.

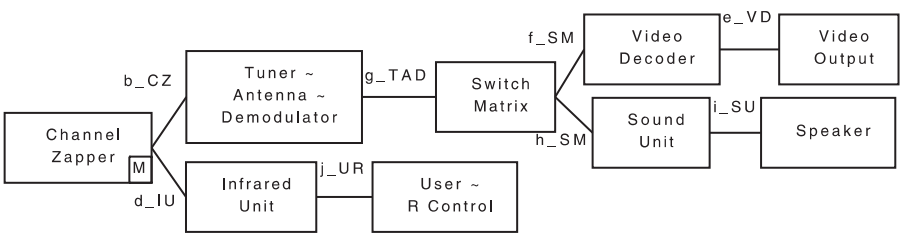
There are cases where the wrapper needs to be sure of the origin of an event to determine a correct course of action. In some cases, it is possible to deduce this information by allowing the wrapper to examine both inputs and outputs of possibly several components. In Figure 10, for example, the Beep Controller 'knows' whether a mute/unmute event from the Mute Controller is in response to a user requested or not. In fact, it beeps, only if it believes that is the case. We call this a *selective wrapper*.

Wrappers can also be used as a kind of composition operator, often to resolve conflicting behavior at runtime. In Section 6, we will make use of this wrapping mechanism when composing several features.

5.4. Noise suppression feature

We now introduce the last TV feature in our discussion. Figure 14 shows a problem context of the well-known feature of changing channels, or 'channel zapping'. The diagram can be understood as follows: when the user presses a channel number of the remote control, the implemented machine, or software component, the Channel Zapper senses it through the infrared unit. The Channel Zapper then instructs the Tuner∼Antenna∼Demodulator to change frequency. Tuner∼Antenna∼Demodulator then searches for the requested frequency and, if a TV station is found it allows the signals to flow 'downstream' through the switch matrix. Video Decoder and Sound Unit obtain this feed from Switch Matrix and produce appropriate visual images and sound on Video Output and Speaker appropriately.

However, there is a problem with simply re-tuning the frequency when the user requests the channel to change [4]. The problem is to do with audio and visual 'noises' that TVs produce when the tuner is not tuned to a station, as it is the case while Tuner∼Antenna∼Demodulator is searching for a new frequency. Therefore, turning the sound off and blanking the screen until a



b_CZ:CZ!{tuneTo(X)} d_IU:IU!{ChangeChannel(C)}
g_TAD:TAD!{AntennaFeed} j_UR:UR!{ZappingSignalsGenerated}
f_SM:SM!{SwitchVideoFeed} h_SM:SM!{SwitchAudioFeed}
e_VD:VD!{VideoSignals} i_SU:SU!{startAudioStream, stopAudioStream}

Figure 14. Problem diagram for channel zapping.

DOI: 10.1002/spe

stable signal is restored are a regular feature of modern TVs. In the following discussion we will concentrate on the issue of audio noise suppression.

5.4.1. Requirement

The requirement, concerning the TV audio, for this feature is stated as follows:

Do not produce audio noise during the channel changeover.

The context of this problem in Figure 14 is not clear because the reference phenomena in the stated requirement are also not clear. To make the problem context clear, we first need to be able to answer the following question: should noise be suppressed only when the user requests the channel to change, or should noise be suppressed at every changeover? Since we assume that the user does not want noise suppression, say, during the TV installation, we take the former to be the case. The requirement is reformulated again as follows:

When user commanded channel changing happens, mute the sound until the audio feed from the switch matrix is stable.

5.4.2. Problem world domains

This requirement makes clear that the domains User $\sim$ R Control, Infrared Unit, Video Decoder and Video Output are not relevant to the problem. Now that we have some understanding of the problem and its context, we will first attempt to describe it using a PD. However, the problem cannot be fitted to a frame neatly because we cannot introduce a machine into the existing structure easily. After applying the transformation rules discussed in Section 5.1 to the interfaces *b_CZ* and *h_SM*, we describe the problem as shown in Figure 15. The problem now becomes simpler: if the audio feed to the sound unit becomes unstable as a result of the channel changing by the user, the sound should be muted.

5.4.3. Specification

Once described in this way, we can obtain the following specification for the Noise Suppressor.

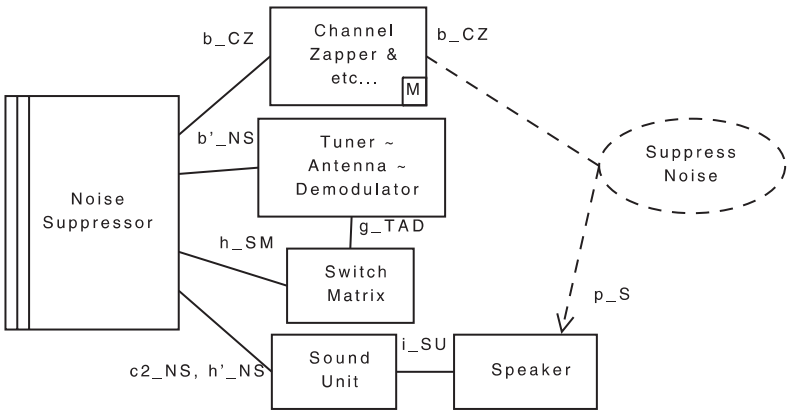
When the event tuneTo(X) happens at b_CZ, the event setMuteOn should be fired at c2_NS immediately, and the event tuneTo(X) should be passed on to b′_NS. The event setMuteOff should be fired at c2_NS when SwitchAudioFeed becomes stable.

We have now shown how to transform an existing design when specifying incremental features, and how the soundness and completeness of an incremental specification can be checked with the help of a logical deduction tool.

6. COMPOSING FEATURES

In our descriptions of the problems so far, we have not taken into account what may happen when the features are composed: they may be inconsistent and produce conflicting behavior at runtime.

DOI: 10.1002/spe



b_CZ:CZ!{tuneTo(X)}
h_SM:SM!{SwitchAudioFeed}
h'_NS:NS!{SwitchAudioFeed}
i_SU:SU!{startAudioStream, stopAudioStream}
b'_NS:NS!{tuneTo(X)}
g_TAD:TA!{AntennaFeed}
c2_NS:NS!{setMuteOn, setMuteOff}
p_S:S!{NoSound}

Figure 15. Problem diagram for noise suppression.

However, we have deferred the composition concerns when decomposing complex problems, and will consider them when solutions to sub-problems are recomposed in order to create a complete system [7]. Thus we have observed the separation of concern principle. This is also in line with what has been called the 'composition-oriented' approach to product-line software development [21].

We note that by deferring composition concerns in our decomposition, and only considering them when we recompose features, we hope to provide a way of dividing complexity and managing responsibility between those who develop features and those who compose them. Since our composition approach is not intrusive, i.e. domains representing features are treated as black boxes or given domains, it can potentially be used to compose features developed by disparate developers.

Suppose that a particular model of TV requires the beep on mute, child lock and noise suppression features. Methodologically, we detect possible interactions between features by first examining whether they have any shared domain among them. If problem structures of any two features have a common domain, we then examine if there is a pair of *Initiates* and *Terminates* predicates for any fluent of the domain, and if the two features can independently generate events with the initiating and terminating effects on the fluent. If so, there is a potential feature interaction problem [14]. In our example, a possible three-way interaction is detected because (i) the Sound Unit domain is shared by the three features, (ii) the Sound Unit domain has a fluent *MuteOn* which can be initiated by the event *setMuteOn* (D9) and terminated by the event *setMuteOff* (D10), and (iii) the events *setMuteOn* and *setMuteOff* can be independently generated by Beep Controller, ChLock Controller and Noise Suppressor.

Having identified the potential interaction between the three features, a compositional wrapper is introduced and the requirement for the composition is formulated by identifying features or events that have higher priority over others. For example, if the user mutes the TV and then changes the channel, presumably it should stay muted after the channel is changed. If the child lock is on, the

DOI: 10.1002/spe

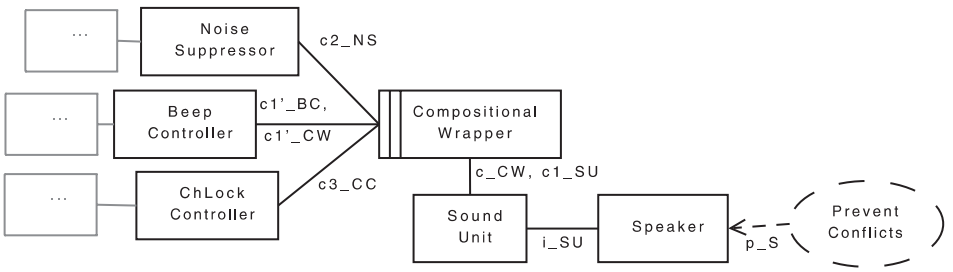


Figure 16. A partial composition diagram for 3-way feature composition.

sound should be muted in all circumstances. The specification of the compositional wrapper is then derived by declaratively describing the priority rules.

Figure 16 shows a partial composition diagram involving the three features, highlighting relevant aspects of the problem.

6.1. Composition requirement

The composition requirement Prevent Conflicts may be stated informally as follows:

> *Child lock has the highest priority*: *if it is on*, *keep the sound muted in all circumstances. If the user wants to mute*, *mute the sound. If the user wants to unmute*, *unmute the sound unless noise suppression has muted. Similarly, if noise suppression wants to mute*, *mute the sound. If noise suppression wants to unmute*, *unmute the sound unless the user has asked to mute.*

The requirement is close to the specification, and therefore we will not formalize the requirement, but specify the compositional wrapper straightaway.

6.2. Specifying compositional wrapper

Specifying compositional wrappers is done declaratively. We will simply state conditions under which certain events are allowed or not allowed. In order to break down the complexity, we will specify it step-by-step. First we say that all *setMuteOn* events from the three features are passed on to the sound unit (C1a, C1b and C1c).

$$PassedOn(setMuteOn(c2\_NS), t, setMuteOn(c\_CW), 0) \quad \text{(C1a)}$$

$$PassedOn(setMuteOn(c1'\_BC), t, setMuteOn(c\_CW), 0) \quad \text{(C1b)}$$

$$PassedOn(setMuteOn(c3\_CC), t, setMuteOn(c\_CW), 0) \quad \text{(C1c)}$$

The strategy to specify *setMuteOff* is that by default they are passed on to the sound unit unless the following conditions for failure are met. Notice that the conditions for failure are defined from the perspective of each of the three features.

DOI: 10.1002/spe

The unmuting event from the child lock feature (ChLock Controller) fails if the noise suppression feature (Noise Suppressor) has muted since the child lock feature has muted and the noise suppression feature has not unmuted afterwards:

$$\begin{aligned} &Happens(setMuteOn(c3\_CC), t1) \wedge \\ &Happens(setMuteOn(c2\_NS), t) \wedge \\ &Happens(setMuteOff(c3\_CC), t2) \wedge \\ &\neg Happens(setMuteOff(c2\_NS), t3) \wedge t1 < t < t3 < t2 \rightarrow \\ &Fail(setMuteOff(c3\_CC), t2, setMuteOff(c\_CW)) \end{aligned} \tag{C2}$$

Suppose that the child lock is switched on, and the TV activates the noise suppression feature. While the noise suppression is active, if the user tries to remove the child lock, the unmuting event will fail and the sound will remain muted. However, if the child lock is switched on, and the user muted the sound, the child lock will succeed in unmuting. In other words, the last action has to be muting by the noise suppression feature in order for unmuting of the child lock feature to fail.

The unmuting event from the beeping feature (Beep Controller) fails if the noise suppression or child lock feature has muted since the beeping feature has muted and they have not unmuted afterwards. Let $c23$ be either *c2_NS* or *c3_CC*, then we have:

$$\begin{aligned} &Happens(setMuteOn(c1'\_BC), t1) \wedge \\ &Happens(setMuteOn(c23), t) \wedge \\ &Happens(setMuteOff(c1'\_BC), t2) \wedge \\ &\neg Happens(setMuteOff(c23), t3) \wedge t1 < t < t3 < t2 \rightarrow \\ &Fail(setMuteOff(c1'\_BC), t2, setMuteOff(c\_CW)) \end{aligned} \tag{C3}$$

For example, if the child lock or noise suppression feature has muted the sound since the user has muted, the user's attempt to unmute will fail.

The unmuting event from the noise suppression feature (Noise Suppressor) fails if the beeping or child lock feature has muted since the noise suppression feature has muted and they have not unmuted afterwards. Let $c13$ be either *c1'_BC* or *c3_CC*, then we have:

$$\begin{aligned} &Happens(setMuteOn(c2\_NS), t1) \wedge \\ &Happens(setMuteOn(c13), t) \wedge \\ &Happens(setMuteOff(c2\_NS), t2) \wedge \\ &\neg Happens(setMuteOff(c13), t3) \wedge t1 < t < t3 < t2 \rightarrow \\ &Fail(setMuteOff(c2\_NS), t2, setMuteOff(c\_CW)) \end{aligned} \tag{C4}$$

Similarly, if the user has muted the sound or switched the child lock on since the noise suppression feature has muted, noise suppression feature will fail to unmute.

This scheme of describing requirement priority in composition operator has some advantages over the mechanism we used in [14], in particular: (i) instead of an additional machinery, here we

DOI: 10.1002/spe

use only EC predicates, (ii) our EC predicates is more expressive because we can, for example, delay certain events if necessary, rather than fail them at all times, and (iii) here it is not necessary to specify the length of time during which certain events may fail: for example, it is impossible to say for how long the user may want to keep the sound muted.

7. CONCLUSIONS

In this paper, we have described an engineering approach to specifying a feature-rich TV software system in a modular fashion. Our approach involves an extensive application of past knowledge about problem patterns in software specifications, and formal analysis.

It is an advantage of using the PF approach that we have been able to decompose complex problems in TV features to simpler and familiar problems. Having done this, specifications of these features have been derived and analyzed using a list of known concerns. This enables the developer to capture and reuse past knowledge of solving similar problems. In addition to specifying new features, we have shown how features for incremental requirements can be specified by transforming the existing design and introducing wrappers. The wrappers introduced are non-intrusive in the sense that they treat the existing feature specifications as black-boxes. Guidance on using general patterns of wrappers has been provided.

Having derived informal specifications of new and incremental features, we have shown how features can be specified formally, and sometimes incrementally, using the EC and our extension of it. One of the advantages of using the EC is the general availability of free off-the-shelf tools supporting various types of automated reasoning. Feature specifications in the EC have been analyzed by the Discrete EC Reasoner tool to show their soundness and completeness with respect to their problem world contexts and requirements.

As a result of using this engineering approach, we have derived specifications that are highly modular. For example, there are clear separations between requirements for new features, requirements for incremental features and requirements for their composition, and also their correspondence with respective specifications. This has helped to create mappings between requirements and features, and between problem and solution structures, to support the evolution of a feature-rich software system.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for providing detailed and insightful comments.

REFERENCES

1. Rajlich V. Changing the paradigm of software engineering. *Communications of the ACM* 2006; **49**(8):67–70.
2. Nuseibeh B. Weaving together requirements and architectures. *Computer* 2001; **34**(3):115–117.
3. Turner CR, Fuggetta A, Lavazza L, Wolf AL. A conceptual basis for feature engineering. *Journal of Systems and Software* 1999; **49**(1):3–15.
4. van Ommering RC. Horizontal communication: A style to compose control software. *Software—Practice and Experience* 2003; **33**(12):1117–1150.
5. van Ommering R, van der Linden F, Kramer J, Magee J. The koala component model for consumer electronics software. *Computer* 2000; **33**(3):78–85.

DOI: 10.1002/spe

6. Anton AI, Potts C. Functional paleontology: The evolution of user-visible system services. *Transactions on Software Engineering* 2003; **29**(2):151–166.
7. Jackson M. *Problem Frames*: *Analyzing and Structuring Software Development Problems*. ACM Press, Addison Wesley: New York, Reading, MA, 2001.
8. Mueller ET. *Commonsense Reasoning*. Morgan Kaufmann: Los Altos, CA, 2006.
9. http://decreasoner.sourceforge.net/ [13 February 2009].
10. Kowalski R, Sergot M. A logic-based calculus of events. *New Generation Computing* 1986; **4**(1):67–95.
11. Shanahan M. The event calculus explained. *Artificial Intelligence Today* (*Lecture Notes in Artificial Intelligence*, vol. 1600), Woolridge MJ, Veloso M (eds.). Springer: Berlin, 1999; 409–430.
12. Miller R, Shanahan M. The event calculus in classical logic—alternative axiomatisations. *Journal of Electronic Transactions on Artificial Intelligence* 1999; **3**:77–105.
13. Classen A, Laney R, Tun TT, Heymans P, Hubaux A. Using the event calculus to reason about problem diagrams. *Proceedings of the 3rd International Workshop on Applications and Advances of Problem Frames*. ACM: New York, NY, U.S.A., 2008; 74–77.
14. Laney R, Tun TT, Jackson M, Nuseibeh B. Composing features by managing inconsistent requirements. *Proceedings of 9th International Conference on Feature Interactions in Telecommunications and Software Systems* (*ICFI 2007*), Grenoble, France, 2007; 141–156.
15. Li Z, Hall JG, Rapanotti L. From requirements to specifications: A formal approach. *Proceedings of the 2006 International Workshop on Advances and Applications of Problem Frames*, Shanghai, China, 2006; 65–70.
16. Rapanotti L, Hall JG, Li Z. Deriving specifications from requirements through problem reduction. *IEE Proceedings Software* 2006; **153**(5):183–198.
17. Seater R, Jackson D. Requirement progression in problem frames applied to a proton therapy system. *Proceedings of RE'06*. IEEE Computer Society: Washington, DC, U.S.A., 2006; 166–175.
18. Tun TT, Laney R, Jackson M, Nuseibeh B. Tool support to derive specifications for conflict-free composition. *Technical Report*, *2008/13*, Department of Computing, The Open University, 2007.
19. Bayardo RJ Jr, Schrag R. Using CSP look-back techniques to solve real-world SAT instances. *AAAI/IAAI*, Providence, RI, 1997; 203–208.
20. http://code.google.com/p/relsat/ [13 February 2009].
21. Bosch J. The challenges of broadening the scope of software product families. *Communications of the ACM* 2006; **49**(12):41–44.

DOI: 10.1002/spe