ISSN 1744-1986

Technical Report N° 2005/08

Composing Problems: Deriving specifications from inconsistent requirements

Robin Laney
Michael Jackson
Bashar Nuseibeh

6th May 2005

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk

Composing Problems:
Deriving specifications from inconsistent requirements

Robin Laney Michael Jackson Bashar Nuseibeh

Dept of Computing, Open University,
Walton Hall, Milton Keynes, MK7 6AA, UK
Email: r.c.laney@open.ac.uk, jacksonma@acm.org, b.nuseibeh@open.ac.uk

ABSTRACT
In this paper we demonstrate an approach to system development based on problem decomposition and subsequent (re)composition of sub-problem specifications. We illustrate the work using Problem Frames, an approach to the decomposition of problems that relates requirements, domain properties, and machine specifications. Having decomposed a problem, one approach to solving it is through a process of composing solutions to sub-problems. In this paper, we show that by formalizing system requirements and domain properties using an Event Calculus, we can both systematically derive machine specifications and solve composition problems. We add a *prohibit* predicate to the event calculus, that prohibits an event over a given time period. This allows a sub-solution to be formalized in a way that provides for run-time conflict resolution. We develop our earlier work on Composition Frames, an approach to composing inconsistent requirements, by adding systematic support and factoring out domain-dependent details. Throughout the paper we use a simple case study to illustrate and validate our ideas.

1. INTRODUCTION
Given a good decomposition of a problem into sub-problems, a range of benefits would arise if we were able to provide a solution by solving the sub-problems in isolation and then composing the partial solutions to give a complete system [20, 36, 37, 6, 26]. The benefits include scalability (due to working with simpler sub-problems), traceability (since sub-problems map directly to their solutions), and easier system evolution (since changes to sub-problems can be addressed by modifying corresponding solutions or compositions). Problem Frames [20] provide a well-founded approach to the decomposition of problems, and provide an explicit representation for relating requirements, domain properties, and machine specifications.

The composition problem is important, since solving it supports a better separation of concerns between requirements analysis and the design phase: the ability to compose solutions allows us to take a set of decomposed requirements, provide individual solutions to discrete requirements and then address the overall system requirements by recomposing solutions. This is consistent with an iterative approach to development [28,17].

The composition problem raises a number of questions: Are the requirements to be composed consistent? 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 machines on the problem domain? 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 to show how to address the composition problem for inconsistent requirements. We show how by expressing requirements and domain properties in a version of the Event Calculus [23,34] we can systematically derive machine specifications in a way that facilitates the resolving of inconsistencies. The Event Calculus, a form of temporal logic, is used in this work, as it is conveniently complementary to Problem Frames' use of events. The Event Calculus has previously been used in software development for reasoning about evolving specifications [13,33], and distributed systems policy specifications [3]. Inconsistency resolution is deferred to run time, in a way that has some similarities with [10,11]. We use Composition Frames, introduced in [26] to express the requirements of composition. Composition Frames show the merging of two Problem Frames. They add a Composition Controller to mediate between the machine specifications that address different sub-problems. Composition Frames allow us to provide arguments showing their satisfaction. They model composition in order to deal with unwanted effects: that is interference of overlapping reactions to events. The use of an Event Calculus for requirements and machine specifications allows us to be more systematic about the way in which a Composition Controller deals with inconsistency. Its use ensures that machine specifications are specific about events that should not occur as well we those that should, thus removing the need to embed lots of domain dependent details in a Composition Controller. We add a $prohibit(\alpha,\tau1,\tau2)$ predicate to the Event Calculus, that prohibits an event α over a time period between times $\tau1$ and $\tau2$. It is the use of this predicate in machine specifications that facilitates a systematic approach to run-time conflict resolution.

The paper is organised as follows. In section 2 we present a simple problem decomposition whilst giving a brief introduction to Problem Frames, and also introduce the Event Calculus. In section 3 we present our approach to addressing the composition problem. We begin by showing how to express domain properties and requirements in the Event Calculus. We then derive 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. We specify some generalized compositions and show how to apply them to our running example. In section 4

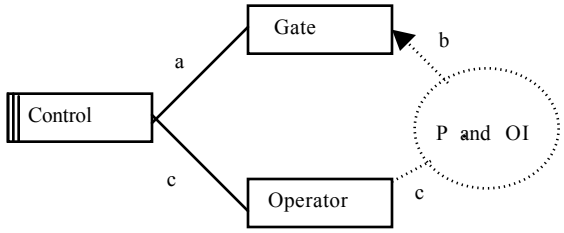
we compare our work with other approaches. In section 5 we discuss some lessons about the composition of requirements, of solutions, and their relationship. We conclude in section 6 and present future work.

2. BACKGROUND

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

2.1 Introductory Example

Throughout this paper we will use an example that involves specifying the requirements for the control of a sluice gate. The example is based on material in [20] which explores a number of further aspects of the problem that do not concern us here. The sluice gate has a motor that can be controlled by pulse events (up, down, off) to move it up or down, and a sensor that indicates when it is fully open or shut by generating events (Top, Bottom). The problem is to control the sluice gate by moving it up and down subject both to pre-programmed control (P) that ensures it is open for 10 minutes in each three-hour period whilst also allowing for intervention by a human operator (OI). The human operator may wish to raise, lower, or stop the gate at any time. The problem is represented in the problem diagram in figure 1.



a: C!{up, down ,off}
G!{Top,Bottom},
c:O!{raise,lower,halt}

b:G{Open,Shut}

Figure 1. Problem Diagram

The diagram shows us the relationship between a *machine domain* (Control) that implements a solution, *problem domains,* that is entities in the world that the machine must interact with (Gate and Operator), and the *problem requirements* (the dotted oval). The requirements are to be met by providing a suitable machine. The machine domain (indicated by double lines on its left side) is a domain that is to be designed and for which we will provide a specification. The lines between domains labelled a, b, c represent interfaces between those domains, i.e. shared phenomena by which they interact. In the example, the phenomena are given as sets of events. The prefixes, such as C!, show which domain controls a set of events (in this case the machine Control). Ultimately, we wish to show that given a particular machine specification, the requirements are met. This requires us to have appropriate descriptions of the domains. In the above diagram we show that requirements P and OI are to constrain the operation of the gate (the dotted arrow), they reference the behaviour of the operator (the dotted line). We will work with the following specification of P and OI.

P – The gate should be opened for the first ten minutes of each three-hour period

OI – The gate should respond to raise, lower and halt commands issued by the operator.

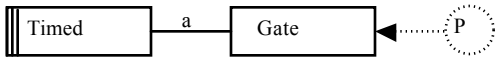
It is clear that there is some potential inconsistency between the requirements P and OI. For example, what if the Operator wants to lower the gate when the Timed machine is raising it? This inconsistency is an example of *divergence* [24]. In section 3.3 we propose ways to weaken the conjoined requirements, resolving any inconsistencies, in a manner that allows us to satisfy them. Our two domains are described as follows:

Gate: The Gate, when stopped, will react to an up event by moving upwards, unless already fully open. When stopped, it will react to a down event by moving downwards, unless already fully closed. When it receives an off event it will turn its motor off, stopping any motion. When it detects that it is fully open it will generate a Top event. When it detects that it is fully closed, it will generate a Bottom event.

Operator: The operator will generate raise commands to request the gate moves upwards, lower commands to request the gate moves downwards, and halt commands to request the gate stops.

The domain descriptions given above, along with the interface topology, gives us a framework for checking whether a given machine specification meets our requirements. One immediate difficulty is how we arrive at that specification. The Problem Frames approach involves decomposing a Problem Diagram by identifying sub-problems (projections of the original problem) that match well-known diagram forms, known as basic Problem Frames. These are problems in a form that are both relatively simple and well understood to the extent that we can expect to address them directly. In our example, we can decompose the problem into the pre-programmed and operator intervention sub-problems as shown in figures 2 and 3.

The Problem Frame for requirement P is of a well-known form called a *Required Behaviour Frame*, drawn from a catalogue of five basic Frames identified in [20].



a: T!{up, down, off}
G!{Top,Bottom}

b:G{Open,Shut}

Figure 2. Problem Frame for Req. P

We need to ensure that the specification for the machine called Timed, along with the above description of the Gate, is sufficient to establish that the requirement P is satisfied. The obligation to demonstrate this is known as the *frame concern*, and the case for it holding needs to be made either formally or informally depending on context. In Section 3.3 we show a way to do this based on deriving the machine specification from formal descriptions of the requirement and gate.

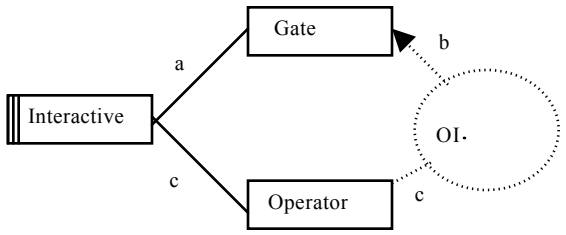


Figure 3. Problem Frame for Req. OI

The Problem Frame for requirement OI is similarly of a well-known form and is called a *Commanded Behaviour Frame*. Again, in section 3.2, we will discharge the frame concern and derive the machine specification through a refinement style process.

At this stage, we have taken our problem and decomposed it into two sub-problems. In itself this process is useful as it gives us a better understanding of the problem domain. We have also provided solution specifications for the sub-problems. But how do we compose the two machines to be specified, Interactive and Timed, in order to produce a specification for the machine named Control in figure 1 that meets the frame concern of that diagram? To address this question for more than the simplest of disjoint requirements, we need to deal with the inconsistencies between requirements. We assume that the two machine specifications are likely to be composed in such a way that their implementations can be run on a single physical machine.

2.2 The Event Calculus

The event calculus [23,34] is a logic system grounded in the predicate calculus that was first introduced in [23]. It has been used in [34] as a way of reasoning with inconsistency in software requirements. One view of the calculus [34] is as a way of relating scenarios (in terms of events that happen), program specifications, and "fluents" that describe properties of a system at a given time. Our approach varies from this schema in that we replace the scenarios with event sequences specifying machine semantics, and we replace the program specifications with domain descriptions. Fluents are used to describe requirements. This provides a conveniently direct mapping between the Event Calculus and the Problem Frames approach. A more generic view is that the calculus relates the events that happen to rules about consequences of events and outcomes.

We will work with a version of the calculus adapted from Shanahan [34] that is intended to be simple whilst fully supporting the contribution of section 3. In particular, we choose not to deal with concurrent events, on the basis that we will assume machines for sub-problems are implemented sequentially, and the composition controller introduced in section 3.3 will be implemented using a merge operation. The table below has also been adapted from [34].

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($\tau 1,\beta,\tau 2$)	Fluent β is terminated between times $\tau 1$ and $\tau 2$
Trajectory($\beta 1,\tau,\beta 2,\delta$)	If Fluent $\beta 1$ is initiated at time τ then fluent $\beta 2$ becomes true at time $\tau+\delta$

We now need axioms to tell us when a fluent holds, based on the events that have happened, initial conditions, and rules about how fluents change their values. We follow [34] in assuming all variables are universally quantified except where otherwise shown.

HoldsAt(f,t1) ← Initially(f) ∧ ¬ clipped(0,f,t1)

(EC1)

HoldsAt(f,t2) ← Happens(a,t1) ∧ Initiates(a,f,t1) ∧ t1 < t2 ∧ ¬ clipped(t1,f,t2)

(EC2)

HoldsAt(f2,t3) ←
Happens(a,t1) ∧ Initiates(a,f1,t1) ∧ Trajectory(f1,t1,f2,d) ∧
t2 = t1 + d ∧ t1 < t2 < t3 ∧
¬ clipped(t1,f1,t2) ∧ ¬ clipped(t2,f2,t3)

(EC3)

Clipped(t1,f,t2) ↔ ∃ a,t
[Happens(a,t) ∧ t1 < t < t2 ∧ Terminates(a,f,t)]

(EC4)

These rules are a way of stating that a fluent holds if: it held initially and nothing has happened since to stop it holding; an event has happened to make the fluent hold and nothing has happened since to stop it holding; or, an event happened that caused some fluent to hold, that in turn, after a period of time caused this fluent to hold, and again nothing has happened since to stop the second fluent holding.

In order to address the frame problem of [27], we again follow Shanahan in adopting the *common sense law of inertia*. That is we assume fluents do not change value unless something happens to cause this.

3. COMPOSING REQUIREMENTS AND SOLUTIONS

We now address the provision of machine specifications to meet the requirements in figures 2 and 3. Having done so, we go on to consider how to compose both our requirements and machine specifications to satisfy the frame concern of figure 1. In order to do this, we need to deal with inconsistencies between the requirements, as mentioned in section 2.1

In section 3.1 we formalize our requirements and the description of the gate domain by translating them into the event calculus described in the last section. We then derive machine descriptions in section 3.2 by refining our requirements using the gate domain semantics. In this way, we are establishing the frame argument, which alternatively would typically be done by arguing from machine descriptions through the causal semantics of the gate domain to establish that the requirements are met. We then show in section 3.3 how our machines can be composed to produce a system meeting our overall requirements, albeit weakened to resolve inconsistencies, by introducing a composition controller.

3.1 Formalizing Requirements and Domains

The natural language specification of the gate domain in section 2.1 can be translated into the event calculus as follows:

Initiates(up,MovingUp,t) ← HoldsAt(Stopped,t)	(G1)
Initiates(down,MovingDown,t) ← HoldsAt(Stopped,t)	(G2)
Initiates (off,Stopped,t)	(G3)
Initiates (top,Open,t)	(G4)
Initiates (bottom,Shut,t)	(G5)
Trajectory(MovingUp,t,Open,sufftime)	(G6)
Trajectory(MovingDown,t,Shut,sufftime)	(G7)
Terminates(down,Open,t)	(G8)
Terminates(up,Shut,t)	(G9)
Terminates(off,MovingUp,t)	(G10)
Terminates(off,MovingDown,t)	(G11)

Rules 1-3 relate to how the gate can be set in motion or stopped. Rules 4-5 are interesting in that causal relationship between the event and the setting of the fluent are the reverse of how the predicate would normally be read. Operationally this corresponds to the fact we can observe the events, as reflected in our original Problem Frame. Rules 6-7 describe how the gate eventually ends up open or shut if movement in a given direction is continued for long enough. Finally, Rules 8-11 go beyond the natural language specification to give us a fuller formalisation.

The natural language specification of the requirements in section 2.1 can be translated into the event calculus as follows. To deal with the timed machine, it is helpful to first refine the natural language specification P:

Making Sure the Gate is Open at Start of three hour period and

Keep Gate open for first ten minutes of three hour period

Formally, this translates to:

$$HoldsAt(Open,t1) \wedge \neg clipped(t1,Open,t1+10) \wedge t1 \bmod 180 = 0 \quad (P)$$

Turning to the interactive machine, the requirement OI can easily be directly translated as follows. The → below should be read as stating that we want to generate the event on the right when the event on the left happens.

$$\exists d.\ Happens(raise,t) \rightarrow HoldsAt(t+d,MovingUp) \wedge$$
$$\exists d.\ Happens(lower,t) \rightarrow HoldsAt(t+d,MovingDown) \wedge$$
$$\exists d.\ Happens(halt,t) \rightarrow HoldsAt(t+d,Stopped) \quad (OI)$$

We can now use the formal descriptions to derive machine specifications for the Timed and Interactive machines.

3.2 Deriving Machine Descriptions

We begin by considering the timed machine. Informally, we note that there is no conflict between the two sub-clauses (HoldsAt and Clipped clauses), corresponding to the two clauses of the informal requirement P, so we will refine them separately. We will use the symbol ⇔ to show that a lhs expression is refined to an rhs expression.

Making Sure the Gate is Open at Start of three hour period

In order to refine the first sub-clause we note that the event calculus gives us three options for dealing with a fluent expressed using HoldsAt.

$$HoldsAt(Open,t1) /\backslash t1 \bmod 180 = 0 \Leftrightarrow$$
sub-clause based on EC1 ∨ sub-clause based on EC2 ∨ sub-clause based on EC3

We will develop each sub-clause separately, and explain their meaning informally at each stage. We begin with the first sub-clause, which deals with the case where the gate was initially open and nothing has changed.

$$Initially(Open) \wedge \neg clipped(0,Open,t1) \wedge t1 \bmod 180 = 0$$
$$\Leftrightarrow (EC4)$$

$$Initially(Open) \wedge \neg \exists\, a,t\ (Happens(a,t), \wedge Terminates(a,Open,t) \wedge (0 < t < t1) \wedge t1 \bmod 180 = 0$$
$$\Leftrightarrow (Unify\ with\ G\ 8)$$

$$Initially(Open) \wedge \neg \exists\, t\ (Happens(down,t) \wedge Terminates(down,Open,t) \wedge 0 < t < t1) \wedge t1 \bmod 180 = 0$$
$$\Leftrightarrow (Unify\ Terminates\ clause\ with\ G8)$$

$$Initially(Open) \wedge \neg \exists\, t\ (Happens(down,t1) \wedge 0 < t < t1) \wedge t1 \bmod 180 = 0$$

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 machine specifications, we introduce a new predicate into our event calculus with the following meaning and related axiom.

Formula	Meaning
prohibit($\alpha, \tau 1, \tau 2$)	Event α should not occur between times $\tau 1$ and $\tau 2$

prohibit(a, t1, t2) ← ¬∃ a, t (Happens(a, t) ∧ t1 < t < t2)

In terms of a machine implementing the specification at run-time prohibit(a,t1,t2) needs to be enforced before t1, otherwise it fails. We can now use the new rule to complete the derivation above.

Initially(Open) ∧ ¬∃ t (Happens(down,t1) ∧ 0 < t < t1)
⇔Rule for ¬∃ t (Happens(down,t2) ∧ t1 < t2 < t3) ∧ t1 mod 180 = 0
Initially(Open) ∧ prohibit(down,0,t1) ∧ t1 mod 180 = 0

This case allows the machine to do nothing if the gate is initially empty and stays that way. Also, if the machine is only installed at, for example time 0, it is the only way that our requirement can be met right from the initial operation of the machine.

The second sub-clause deals with the case where the gate is *detected* as open at the start of the three hour period:

Happens(a,t1) ∧ Initiates(a,Open,t1) ∧t1 < t2 ∧
¬ clipped(t1,Open,t2) ∧ t2 mod 180 = 0
⇔ (Unify Clauses with G1)

(Happens(Top,t) ∧ Initiates(Top,Open,t) ∧
¬Clipped(t,Open,t2) ∧ t < t2 ∧ t2 mod 180 = 0
⇔ (G4 is statement of fact)

(Happens(Top,t) ∧ ¬Clipped(t,Open,t2) ∧ t < t2 ∧
t2 mod 180 = 0
⇔ (EC 4)

Happens(Top,t) ∧ ¬∃ a,t1 (Happens(a,t2), ∧
Terminates(a,Open,t2) ∧ (t < t1 < t2) ∧
t2 mod 180 = 0

⇔ (G 8, unify a and down)
Happens(Top,t) ∧ ¬∃ t1 (Happens(down,t2)
∧Terminates(down,Open,t2) ∧ t < t1 < t2) ∧
t2 mod 180 = 0
⇔(G8)

Happens(Top,t) ∧ ¬∃ t1 (Happens(down,t2) ∧
t < t1 < t2 ∧
t2 mod 180 = 0
⇔(Rule for ¬∃ t (Happens(down,t2) ∧ t1 < t2 < t3))

Happens(Top,t) ∧ prohibit(down,t,t2) ∧
t < t2 ∧ t2 mod 180 = 0

This rule effectively says that the gate is open if the gate itself has indicated that this is the case (assumes a reliable gate) and that the status of the gate has not changed since.

The final sub-clause deals with the case where we need to move the gate at the start of the three hour period:

Happens(a,t1) ∧ Initiates(a,f1,t1) ∧ Trajectory(f1,t1,Open,d) ∧
t2 = t1 + d ∧t1 < t2 < t ∧
¬ clipped(t1,f1,t2) ∧¬ clipped(t2,Open,t) ∧ t mod 180 = 0

⇔(Unify clauses with G1, G6)

Happens(up,t1) ∧ Initiates(up,MovingUp,t1) ∧
HoldsAt(Stopped,t1)∧ Trajectory(MovingUp,t1,Open,sufftime) ∧
t2 = t1 + d ∧t1 < t2 < t ∧
¬clipped(t1,MovingUp,t2)∧¬clipped(t2,Open,t)∧t mod 180 = 0

⇔(EC 2)

Happens(up,t1) ∧ Initiates(up,MovingUp,t1) ∧
Happens(off,t1 - 1)∧ Trajectory(MovingUp,t1,Open,sufftime) ∧
t2 = t1 + sufftime ∧t1 < t2 < t ∧
¬ clipped(t1,MovingUp,t2) ∧¬ clipped(t2,Open,t) ∧ t mod 180 = 0

⇔(G1 and G6 are statements of fact)

Happens(up,t1) ∧ Happens(off,t1 - 1) ∧
t2 = t1 + sufftime ∧t1 < t2 < t ∧
¬ clipped(t1,MovingUp,t2) ∧¬ clipped(t2,Open,t) ∧ t mod 180 = 0

⇔(Treat clipped similarly to above)

Happens(up,t1) ∧ Happens(off,t1 - 1) ∧
t2 = t1 + sufftime ∧t1 < t2 < t ∧
prohibit(off,t1,t2) ∧¬ prohibit(down,t2,t) ∧ t mod 180 = 0

In this case the specification shows we need to generate an up event at a sufficiently early time, and a stop event immediately before this. Moreover we need to prohibit any stop events whilst we are raising the gate, and prohibit any down events once the gate is raised.

We now go on to deal with the second main sub-clause of the requirement for the Timed machine:

Keep Gate open for first ten minutes of three hour period

We need to refine the following into a machine specification:

¬Clipped(t1,Open,t1+10) ∧ t1 mod 180 = 0

This expression, can be transformed into the following, by treating the Clipped clause in a similar way to the above:

prohibit(t1,down,t1+10) ∧ t1 mod 180 = 0

We now consider the operated machine. The requirement was:

∃d. Happens(raise,t) →HoldsAt(t+d,MovingUp) ∧
∃d. Happens(lower,t) → HoldsAt(t+d,MovingDown) ∧
∃d. Happens(halt,t) → HoldsAt(t+d,Stopped)
(OI)

If we ignore the fact that the gate may already be moving in the required direction, or it may already be stopped waiting for commands, then we might reasonably be able to refine the above requirement to:

(Happens(raise,t) → Happens(off,t+1) ∧ Happens(up,t+2)) ∧
(Happens(lower,t) → Happens(off,t+1) ∧ Happens(down,t+2)) ∧
(Happens(halt,t) → Happens(off,t+1))

Having derived the specifications of our machines to meet the original sub-problems, we now turn to the question of how to compose them. In the next section we will see that the prohibit(α,τ1,τ2) predicate can be interpreted as generating a prohibit event, whose interpretation by a machine designed to compose solutions to sub-problems permits us to resolve conflicts between machines.

3.3 Composition Frames and Controllers

So, how do we compose our two machines in order to meet the overall system requirements associated with figure 1? Since, as section 2.1 argued, the requirements of the sub-problems are not fully consistent we will not be able to meet the requirements of the problem in figure 1 completely. However, we need to be precise about how any conflicts are to be resolved. In [26] we introduced a family of weakened conjunction operators to address this issue. We proposed the following ways of combining two general requirements R1 and R2. For the example above, R1 and R2 are the requirements for the timed gate and operator controlled gate respectively.

Option 1: No Control
Let R1 $\hat{}_{\{any\}}$ 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.

Option 2: Exclusion
Let R1 $\hat{}_{\{active\}}$ R2 be the requirement that both R1 AND R2 should hold at all times except that when R1 leads to some activity in the world, R2 may be suspended and vice versa.

Option 3: Exclusion with Pre-emption
Let R1 $\hat{}_{\{R1\}}$ R2 be the requirement that both R1 and R2 should hold except that when R1 leads to some activity in the world, R2 may be suspended.

Option 4: Exclusion & Fine Grain Pre-emption
Let R1 $\hat{}_{\{important,\ R1\}}$ R2 be the requirement that R1 $\hat{}_{\{R1\}}$ R2 holds, except that any sub-requirement associated with the phenomenon *important* should be given top priority. For the sluice gate example, *important* might be instantiated as the operator halt event.

The use of a formal temporal semantics has led to machine specifications that show a clear inadequacy in the above operator semantics. That is, a machine may be required to control a domain in order to meet the machines allocated requirement without necessarily generating any activity. For example, the Timed machine needs to be able to ensure the gate remains open. Further issues are also clarified by our formal semantics. Mutual exclusion can be partial, that is a machine may need to prohibit certain events being issued by other machines, without necessarily wishing to prohibit all types of event. For similar reasons, we can also now see that option 4 does not have to be considered as distinct from the other options. We therefore subsume certain aspects of option 4 into options 2 and 3, whilst noting that option 4 is still suggestive of further enhancement to our scheme in terms of prioritizing events orthogonally to requirements and machines. For example, we may wish to prioritize an emergency shutdown event regardless of its source.

Option 1: No Control
Let R1 $\hat{}_{\{any\}}$ 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.

Option 2: Exclusion
Let R1 $\hat{}_{\{control\}}$ R2 be the requirement that both R1 and R2 should hold at all times except that when meeting R1 leads to the need to control some activity in the world, certain events needed to meet R2 may be forbidden even if this leads to R2 failing to be met for some period of time, and vice versa.

Option 3: Exclusion with Pre-emption
Let R1 $\hat{}_{\{R1\}}$ R2 be the requirement that both R1 and R2 should hold at all times except that when meeting R1 leads to the need to control some activity in the world, certain events needed to meet R2 may be forbidden even if this leads to R2 failing to be met for some period of time.

In order to meet these requirements on the composition of sub-problem requirements, and to provide a framework in which we can reason about them, we will use the composition frames and their associated composition controller introduced in [26], as shown in figure 4. However, we will see that the fact we adopted the use of event calculus specifications in section 3 now gives us a much more succinct approach to reasoning about the composition controller semantics that we require. Figure 4, includes the requirements of the composition itself (RC), we will see how to express this shortly. Note, in figure 4, the addition of prohibit(…) events to the interfaces named *a* from figures 2 and 3, giving interfaces *a'* and *a''*. Such prohibit(a,t1,t2) events will be generated on the basis of the prohibit(*e*,t1,t2) predicates in our machine specifications. The composition controller will interpret them in order to resolve conflicts.

We will now begin to specify composition controllers to provide way of composing sub-solutions that meet the requirement composition operator semantics introduced above. We prefix events by the name of the interface that they

appear on. Thus *a':event* represents the binding of an event generated by the Timed machine to a variable named *event*. More specific cases are given by pattern matching with specific event names; e.g., *a':Off*. In this example this is unambiguous and aids comprehensibility.

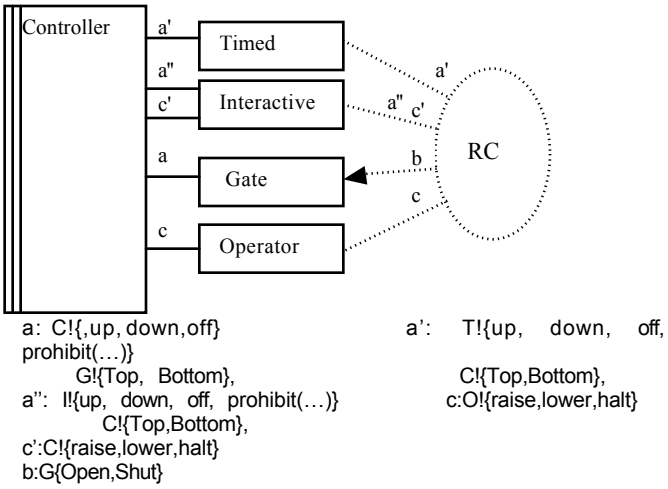


Figure 4: Composition Frame Controller

Unsurprising, it is rather easy to specify a controller matching the semantics of the first version of our operator. The → below should be read as stating that the controller is responsible for generating the event on the right when the event on the left happens.

Composition Controller for P $\hat{}_{\{any\}}$ OI	
a':e → a:e	(1)
a":e → a:e	(2)
c:e → c':e	(3)
a :e → a':e	(4)
a:e → a":e	(5)

For this controller, we are simply saying in line 1 above that events from the Timed machine are passed to the gate, and similarly in line 2 for the Interactive machine. Operator commands are propagated to the interactive machine (line 3) and events generated by the gate are propagated to both the Timed and Operated machines (lines 4 and 5).

In order to address the other operators we need some additional, but quite minimal, machinery. Let *P* be a set that hold tuples of form (e,t1,t2,m) which will represent the fact that event e is prohibited by the specification of machine m between times t1 and t2. We allow the → to be guarded by an optional predicate (following 2nd operand after a comma).

Composition Controller for P $\hat{}_{\{CONTROL\}}$ OI		
a':e → a:e,	$\forall t1,t2,m.t1 \le t \le t2 \wedge (e,t1,t2,m) \notin P$	(1)
a'prohibit(e,t1,t2) → *insert((e,t1,t2,T),P)*		(2)
a':e → ignore,	$\exists t1,t2,m.t1 \le t \le t2 \wedge (e,t1,t2,m) \in P$	(3)

a":e → a:e,	$\forall t1,t2,m.t1 \le t \le t2 \wedge (e,t1,t2,m) \notin P$	(4)
a"prohibit(e,t1,t2) → *insert((e,t1,t2,I),P)*		(5)
a:e → ignore,	$\exists t1,t2,m.t1 \le t \le t2 \wedge (e,t1,t2,m) \in P$	(6)
c:e → c':e,	$\forall t1,t2,m.t1 \le t \le t2 \wedge (e,t1,t2,T) \notin P$	(7)
c:e → ignore,	$\exists t1,t2,m.t1 \le t \le t2 \wedge (e,t1,t2,T) \in P$	(8)
a:e → a':e		(9)
a:e → a":e		(10)

In lines 1-3 we deal with the receipt by the controller of unprohibited events from the Timed machine, the prohibit event itself, and prohibited events which are simply ignored, by the controller. Lines 4-6 show that events from the Operated machine are dealt with in a similar way. The next two lines (7-8) relate to whether operator commands are passed to the interactive machine or rejected. Finally, lines 9-10 show that events from the gate are again propagated to both the Timed and Interactive machine.

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 figure 4 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.

The specification makes a number of important assumptions. Firstly we assume the frame is not super-imposed on machines that are already actively in operation. Removing this assumption leads to an initialization problem (of the world and the machine!) which is almost certainly worthy of a problem diagram in itself.

It is relatively trivial to adapt the above composition controller semantics to deal with P $\hat{}_{\{P\}}$ OI. The main difference is that the prioritized machine needs to be able to remove and replace entries in the set *P* that have been placed there by the other machine.

4. RELATED WORK

There are very few other general approaches to decomposing problems rather than solutions. Notable exceptions are the goal-based approaches of KAOS [25] and the NFR framework [7]. However, these two approaches do not make such a direct relationship between requirements and specifications.

Composition of software artefacts has been addressed by a range of aspect-oriented techniques [8]. However with the notable exceptions of [16], [31], and [4], aspect-based approaches, whilst capable of addressing design and implementation issues, do not address requirements and their decomposition. The approaches of [16] and [31] are mainly concerned with reconciling inconsistencies between a range of non-functional requirements and do not fully address decomposition of functional requirements. While [4] does consider requirements and their relationship to design and implementation, it does not focus on the issues of composition.

A number of formal approaches exist where emergent behaviours due to composition can be identified and controlled [1,9]. Our approach differs from these in that we identify how requirements interact and remove non-deterministic behaviour by imposing priorities over the requirements set.

The whole area of inconsistency management [12,14,15] offers a variety of contributions to dealing with inconsistencies in specifications. Robinson, in particular [32], reviews a variety of techniques for requirements interaction management, and Nuseibeh et al [29] 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.

In [11], a run-time technique for monitoring requirements satisfaction is presented. This approach is taken further in [10], 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.

Various approaches [2,35,18] exist based on iteratively capturing scenarios whilst generating behavioural specifications, and resolving inconsistencies as they arise. Another approach is the use of an abductive reasoning framework [13] to detect and resolve inconsistencies, again at an early stage in the development lifecycle. Our work should be seen as complementary to such approaches in that it will allow inconsistencies to be resolved at run-time. Indeed, the abductive approach of Russo et al. [33] uses the Event Calculus, so a tool encompassing both techniques seems highly feasible.

Our work is related to the feature interaction problem, common in the field of telecommunications [36,22]. While less ambitious about the extent to which requirements can be composed, our work is less domain-specific. In [37] 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.

The need to compose frames is identified in [20] which provides a frame called a Composite Frame to this end. However Composite Frames omit any detail of how to reconcile inconsistencies between the requirements of the machines being composed, or how to deal with interference between phenomena.

In [21] 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 an extra machine (in Problem Frame terms). This gives us the potential to deal with a wider range of compositions.

In [19] other concerns of a timed sluice gate are dealt with, in a more formal setting. An example of composition is given, but the conjoining of specifications takes place in a context where the outputs of the two specifications are distinct. As a result, the difficulties dealt with in this paper do not arise.

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

5. DISCUSSION

We have shown how by expressing requirements and domain properties in a temporal logic we can formally derive machine specifications. Furthermore, by adding a *prohibit* predicate to the event calculus and making use of it in machine specifications we have arrived at a generic approach to the composition of solutions to sub-problems: the composition controller specifications contain no domain specific information. The composition is done non-intrusively in the sense that we have made no changes to the specifications of the machines being composed.

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. Thus trade-offs can be made between design and analysis issues.

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 usually a need to implement an (analogic [20]) model of at least part of the problem domain. Of course arriving at a conceptual model that can subsequently be implemented is often very problematic in itself. In the case of our Timed and Interactive machines the models are very simple. This is partly because the original machine specifications make an assumption that it is safe to ensure the gate is open by moving it upwards for a fixed time. What happens if we start to open the gate from a position other than fully closed and the sluice gate sensor that produces the `Top` event is unreliable. Failing to stop the gate when it reaches the fully open position could burn out the motor. To fully resolve the problems in this scenario, we would need to explicitly model the position of the gate. Composing machines containing such models is more complex, unless they have been originally designed to model changes to domains affected by entities other than themselves. The problem is that 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 two machines A and B and a domain C under their control. In the specification we used the notion of a particular machine being in control of the gate, including passive partial control specified using the *prohibit* predicate. As this is all we rely upon, the same technique should be usable with any machines A and B.

Our Composition Frame deals with two specific Problem Frames, a Required Behaviour and a Commanded Behaviour Frame. It is fairly easy to see that it would generalize to composing two Required Behaviour Frames, or similarly two Commanded Behaviour Frames. We can deal with other basic

Problem Frames in a similar fashion. The basic set of Problem Frames in [20] also includes frames to display information, edit data, and transform data. The interaction issues that need to be dealt with in such cases by the Controller go beyond those illustrated in this paper to include issues such as sequencing and scheduling [20,30].

An alternative semantics for composition is a situation where we know that at any given time a specific requirement should apply; e.g., the sluice gate should be subject to operator control during weekdays but timed behaviour at night and during weekends. At first sight this seems to simplify the design of the Controller, as we do not need to dynamically deal with inconsistencies in requirements. We suggest that this may not be the case, since in general we presumably still need to effect a smooth transition between the Timed and Interactive machine [21]. We would go further and say that systematic analysis of the issues in switching between two machines is an important issue in its own right.

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 [20]. This oversight seems all the more serious when consideration is given to the potential complexities of determining both when a domain begins to react to a machine and when this stops, taking into account in the latter case issues associated with momentum, acceleration and deceleration etc. 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 these concerns at the requirements stage. In particular, our requirements composition operators make explicit the issue of control.

6. CONCLUSIONS AND FUTURE WORK

We have shown how by expressing requirements and domain properties in a temporal logic we can formally derive machine 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 the likelihood of meeting the system requirements. Furthermore, by adding a *prohibit* predicate to the event calculus and making use of it in machine specifications we have arrived at a generic approach to the composition of solutions to sub-problems. The composition controller needs only to be parameterized in respect of the events our application uses, rather than internal implementation details of sub-problem machines. We have illustrated this through the application of our approach to a sluice gate control system.

We have shown how to combine two inconsistent requirements in terms of the operator given in section 3.3. Our 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 (RC) rather than solely in design or implementation terms.

In principle our approach could be extended to deal with more than two requirements. In some scenarios it may be an improvement on compile-time techniques in that practical limits exist, as in principle, global consistency cannot be proved through local consistency checking [29]. 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. Actual validation of scalability though, is still an issue for future work.

As noted above, our approach is strongly related to the typical mutual exclusion problem of concurrent resource usage. Clearly one area for future consideration is a mechanism for queuing rather than dropping events to which we choose to not to respond immediately.

We are currently looking at combining the Problem Frames approach with the architectural approach of coordination contracts [5], giving us the ability to exploit a more developed architectural approach underpinned by a formal semantics [9].

Future work is planned in order 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 3.3 and across a larger set of basic Problem Frames. The set of basic frames in [20] includes 5 such frames, i.e. 10 possible combinations. The latter generalization may not be difficult since we can pattern match on shared domains. On the other hand, particularly 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 in this paper, it may be feasible to automate our approach, supporting its use in larger systems.

6. ACKNOWLEDGMENTS

The authors are grateful for the support of their colleagues in the Department of Computing at The Open University. In particular we have had many interesting conversations with Leonor Barroca, Charles Haley, Jon Hall and Lucia Rapanotti.

7. REFERENCES

[1] R. Allen and D. Garlan, A Formal Basis for Architectural Connectors, *ACM Transactions On Software Engineering and Methodology*, 6(3), 1997, 213-249.

[2] João Araújo, Jon Whittle, Dae-Kyoo Kim: Modelling and Composing Scenario-Based Requirements with Aspects. In *Proceedings of 12th IEEE International Requirements Engineering Conference (RE'04)* (Kyoto, Japan, 6-10 September 2004). 58-67.

[3] A. Bandara, E. Lupu, A. Russo, Using Event Calculus to Formalise Policy Specification and Analysis In *Proceedings 4th IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2003)*, Lake Como, Italy, June 2003.

[4] E. Baniassad and S. Clarke, Theme: An Approach for Aspect-Oriented Analysis and Design, In *Proceedings of the International Conference on Software Engineering*, 2004.

[5] L. Barroca, Fiadeiro J.L., M. Jackson, R. Laney and B. Nuseibeh, Problem Frames: a Case for Coordination, In *Proceedings Sixth International Conference on Coordination Models and Languages*, 2004.

[6] D. Bjørner, Towards Design Calculi for Requirements Engineering and Software Design, In *From Object-orientation to Formal Methods: Dedicated to the Memory of Ole-Johan Dahl*, volume 2635 of Lecture Notes in Computer Science, page 21. Springer-Verlag, August 2003. Editors: O.Owe, S.Krogdahl, and T.Lyche.

[7] L. Chung, B.A. Nixon, E.Yu, J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[8] T Elrad, R. Filman and A. Bader (Guest editors). Special Issue on Aspect Oriented Programming. *Communications of the ACM*, 44(10), October 2001.

[9] J. L. Fiadeiro, A. Lopes, M. Wermelinger, A Mathematical Semantics for Architectural Connectors, in *Generic Programming*, R.Backhouse and J.Gibbons (eds), LNCS 2793, Springer-Verlag, 2003, pp. 190-234.

[10] M.S. Feather, S. Fickas, A. van Lamsweerde, C. Ponsard, Reconciling System Requirements and Runtime Behaviour, In *Proceedings of the 9th International Workshop on Software Specification and Design*, 98.

[11] S. Fickas and M. Feather. Requirements Monitoring in Dynamic Environments. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, York, England, Computer Society Press, Mar. 1995.

[12] Finkelstein and I. Somerville (eds), special issue of *the BCS/IEE Software Engineering Journal on "multiple perspectives"*, Vol II(1), Jan 96.

[13] A. 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*, February 2003.

[14] C. Ghezzi, and B. Nuseibeh (eds) special issues on inconsistency management in *IEEE Transactions on Software Engineering*, Vol 24(11), Nov 98.

[15] C. Ghezzi, B. Nuseibeh (eds) special section on inconsistency management in *IEEE Trans. on Software Engineering*, Vol 25(6), Nov 99.

[16] J. Grundy, Aspect-Oriented Requirements Engineering for Component-based software systems. In *Proceedings Fourth IEEE International Symposium on Requirements Engineering (RE'99)*. Limerick, Ireland: IEEE computer Society Press, 7-11 Jun 1999.

[17] J.G. Hall, M. Jackson, R.C. Laney, B. Nuseibeh, L. Rapanotti, Relating Software Requirements and Architectures using Problem Frames, *IEEE Proceedings of RE 2002*, 2002.

[18] D. Harel, H. Kugler and A. Pnueli, Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements, *Formal Methods in Software and System Modeling* (H.-J. Kreowski et al, eds.), Lecture Notes in Computer Science, Vol. 3393, Springer-Verlag, 2005, 309-324.

[19] Ian J Hayes, Michael A Jackson, Cliff B Jones, Determining the specification of a control system from that of its environment, in Keijiro Araki, Stefani Gnesi and Dino Mandrioli eds, *Formal Methods: Proceedings of FME2003*, Springer Verlag, Lecture Notes in Computer Science 2805, 2003, p.p. 154-169.

[20] M. Jackson, *Problem Frames*, ACM Press Books, Addison Wesley, 2001.

[21] M.Jackson, Why Software Writing is Difficult and Will Remain So, in J.Fiadeiro, J.Madey and A.Tarlecki (eds), *Information Processing Letters Special Issue in Honour of Wlad Turski*, 88(1-2), 2003.

[22] M. Jackson and P. Zave; Distributed feature composition: A virtual architecture for telecommunications services, *IEEE Transactions on Software Engineering XXIV(10)*:831-847, October 1998.

[23] Robert Kolwaski and Marek Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67-95, 1986.

[24] A. van Lamsweerde, R. Darimont, E. Letier, Managing Conflicts in Goal-Driven Requirements Engineering, *IEEE Transactions on Software Engineering*, Special Issue on Managing Inconsistency in Software Development, November 1998.

[25] A. van Lamsweerde, Goal-Oriented Requirements Engineering: A Guided Tour, In *Proceedings of the 5th International Symposium on Requirements Engineering (RE'01)*, pp249-261, IEEE CS Press, 2001.

[26] R. Laney, L. Barroca, M. Jackson, and B. Nuseibeh, Composing Requirements Using Problem Frames, In *Proceedings of 12th IEEE International Requirements Engineering Conference (RE'04)*, Kyoto, Japan, 6-10 September 2004.

[27] McCarthy, J. & Hayes, P.J. (1969), Some Philosophical Problems from the Standpoint of Artificial Intelligence, in *Machine Intelligence 4*, ed. D.Michie and B.Meltzer,

[28] B.A. Nuseibeh, Weaving Together Requirements and Architecture, *IEEE Computer* 34 (3), March 2001, pp. 115-117.

[29] B. Nuseibeh, S. Easterbrook and A. Russo, Making Inconsistency Respectable in Software Development, *Journal of Systems and Software*, 58(2), September 2001, Elsevier Science Publishers, pp. 171-180.

[30] L. Rapanotti, J. Hall, M. Jackson, and B. Nuseibeh , Architecture Driven Problem Decomposition, In *Proceedings of 12th IEEE International Requirements Engineering Conference (RE'04)*, Kyoto, Japan, 6-10 September 2004.

[31] A. Rashid, A.M.D. Moreira, and J Araujo. Modularisation and Composition of Aspectual Requirements, In *Proceedings Aspect Oriented Software Development Conference*, 2003.

[32] William N. Robinson, Suzanne D. Pawlowski, Vecheslav Volkov, Requirements interaction management. *ACM Computing Survey*, 35(2) (2003), pp. 132-190.

[33] A. Russo, R. Miller, B. Nuseibeh, and J. Kramer, An Abductive Approach for Analysing Event-Based Requirements Specifications, In *Proceedings of 18th International Conference on Logic Programming*, Copenhagen, Denmark, 29 July-1 August 2002, Springer.

[34] M.P.Shanahan, The Event Calculus Explained, in *Artificial Intelligence Today*, ed. M.J.Wooldridge and M.Veloso, Springer Lecture Notes in Artificial Intelligence no. 1600, Springer (1999), pp. 409-430.

[35] S. Uchitel, J. Kramer and J. Magee. Synthesis of Behavorial Models from Scenarios. *IEEE Transactions on Software Engineering*. Volume 29, Number 2, February 2003.

[36] P. Zave, Feature Interactions and Formal Specifications in Telecommunications, *IEEE Computer* XXVI (8), 1993, pp. 20-30.

[37] P. Zave, M. Jackson, Conjunction as Composition, *ACM Transactions On Software Engineering and Methodology* 2(4), 1993, pp. 371-411.