To appear in Automated Software Engineering

3 August 1999

Behavioural Conflicts in a Causal Specification

Jonathan Moffett
Department of Computer Science, University of York
Heslington, York YO1 5DD, UK

jdm@cs.york.ac.uk

Andrew Vickers
Praxis Critical Systems
20 Manvers Street, Bath BA1 1PX

ajv@praxis-cs.co.uk

Abstract

Inconsistencies may arise in the course of specification of systems, and it is now recognised that they cannot be forbidden. Recent work has concentrated on enabling requirements descriptions to tolerate inconsistency and on proposing notations that permit inconsistency in specifications. We approach the subject by examining the use of an existing causal language, which is used as a means of specifying the behaviour of systems, to specify, identify and resolve behavioural inconsistencies. This paper is an exploration of the kinds of inconsistency that can arise in a causal specification, how they can be discovered and how they can be resolved. We distinguish between inconsistencies in the structure of a specification, which are assumed to have been removed previously, and inconsistencies in behaviour which, being dynamic in nature, we describe as conflicts.

Our approach concentrates on the identification of conflicts in the specified behaviour of a system. After summarising the causal language, we describe a classification of behavioural conflicts and how they can be identified. We discuss possible methods of resolution, and propose a simple process to aid the identification and resolution of conflicts. A case study using the causal language illustrates our approach.

1. Introduction

1.1 Background and Aims

It has long been recognised that inconsistencies may arise in the course of specification of systems and that the goal of a single unified view of a system is difficult to achieve. The traditional view was that inconsistencies must either be forbidden, or removed at as early a stage as possible.

However, it became recognised that the various stakeholders in a system might have different, and possibly inconsistent, viewpoints [1], and that requirements descriptions must tolerate, and even embrace, inconsistency. Gabbay [2] proposed that inconsistency should be made

respectable. Subsequently his work has been built on by Nuseibeh and others [3, 4], and they have proposed notations that permit inconsistency in specifications.

We approach the subject from a different direction. An earlier paper [5] proposed the use of a causal language as a means of specifying the behaviour of systems. We pointed out that one of the natural means by which the customers of complex systems express themselves is in terms of causation. We proposed a causal language with two important characteristics:

- It should enable users to specify behaviour using a simple causal notation. Our long-term aim is to integrate causal statements into a specification style based on a restricted natural language;
- The notation should be sufficiently precise to be interpreted in terms of any of several temporal logics, and would therefore be amenable to formal reasoning.

One characteristic of this notation is that it is possible for a causal specification to contain mutually inconsistent causal statements. This paper is an exploration of the kinds of inconsistency that can arise in a causal specification, how they can be discovered and how they can be resolved.

1.2 The Principle of Causality

In order to identify behavioural conflicts we need a unifying principle, and we use the principle of causality – nothing happens without a cause[1]. If things "behave" only if they are caused then by describing all the behaviour of a system in terms of causal statements we can capture the entire behaviour of a system using a single notation, thus easing the task of analysis. It should be recognised, of course, that there is a cost to be paid by eliminating all other descriptions of how a system can change, because we may be significantly reducing our expressive power. One of the purposes of this paper is to demonstrate one of the benefits to be put onto the positive side of the balance: a systematic approach to describing and finding behavioural conflicts in a system.

Further leverage can be obtained from the principle of causality: an approach to the resolution of some conflicts. Implied in the principle of "nothing happens without a cause" is a hierarchy of behavioural imperatives, discussed in section 4.3.3, which can be used as guidance on how to resolve conflicts between causal statements.

1.3 What Are Behavioural Conflicts?

There are at least three possible levels of inconsistency in a system:

1. The system description is inconsistent so that it is impossible to describe what happens in it. Typically this is because the entity model has not been completed successfully. We refer to inconsistencies of this kind as **structural inconsistencies**. Behavioural conflict

[1] It will become apparent below (section2.3) that the principle that we use is actually slightly stronger than this; our notation only recognises causation which has an immediate effect. There can only be a gap between cause and effect if this gap is filled by a chain of intermediate direct causes.

analysis will only be possible for those parts of the system for which there is a structurally complete and consistent entity model[2].

2. There is a consistent entity model but the behavioural statements are in conflict. This is the level, **behavioural conflicts**, at which we are addressing this paper. Behavioural conflicts at this level have one characteristic in common: the conflicts are such that a working system cannot be constructed because it is at some point in contradiction. For example:
   - The same condition is simultaneously caused and prevented;
   - The predicates for the state of the system will be in contradiction if the causal statements are obeyed.

   We describe and analyse these behavioural conflicts in more detail below in section 4.

3. Conflicts that occur in a working system, which we call **operational conflicts**, e.g. deadlock, conflicts with safety and security policies, inefficiencies, incorrect output, and all the other ills that beset working systems. Although some of these may perhaps be usefully addressed using a causal specification, we have not attempted to do so in this paper.

We therefore distinguish between inconsistencies in the structure of a specification, and inconsistencies in behaviour which, being dynamic in nature, we describe as conflicts. Our approach concentrates on the identification of conflicts in the behaviour of a system. Inconsistencies in its structure are assumed to have been removed previously by constructing an entity model; in the example that we use in this paper we use an object model that is adapted from [6], but other approaches could also be appropriate.

The scope of this paper is, therefore, those conflicts in a specification with a consistent object model, which prevent the designer from defining the behaviour of the system without contradiction or ambiguity[3].

1.4 Paper Contents

The remainder of this paper is structured as follows. Section 2 gives a summary of our causal notation. Section 3 uses fragments from an example lift system to illustrate our use of the causal notation in practice and provide some material with which to perform our analysis of conflicts in the specification. Section 4 uses material from the example to demonstrate and analyse the classes of conflict that we have encountered, and discusses approaches to their resolution. Section 5 discusses a sample of the related work in this field. Finally section 6 discusses the results of the work and reaches some conclusions.

2. Causal Notation

This causal notation introduces a restricted vocabulary of causal words which are intended to be used with existing formal notations. We are not attempting to introduce a new logic, but a

---

[2] We have not addressed the case where the entity model is incomplete, i.e. underspecified, but not inconsistent.

[3] Thomas [7], discussed in section 5 below, points out that while there is non-determinism in a specification, the possibility of undesirable interactions cannot be totally eliminated.

notation with an interpretation in terms of existing well-developed temporal logics. In this paper we use extracts from a formal specification for encoding the specification. The specification language Z [8] is used with explicit time.

This section summarises the formal definition of the causal model that is described in more detail in [5]. In section 2.1, we describe the properties of causation. Section 2.2 describes the building blocks of our approach: Time, Events and Conditions; and section 2.3 defines causal expressions.

2.1 Properties of Causation

The properties of causation are discussed in [9] and [10]. We select here those that are particularly pertinent to this paper:

- Causation is transitive: if *a* causes *b* and *b* causes *c* then *a* causes *c*
- Causal laws may be cyclic: *a* causes *b* and *b* causes *c* and *c* causes *a*. However the condition occurrences which result from those laws are ordered in time and thus cannot be cyclic. This is put into context in section ? .
- We distinguish between sufficient cause and necessary cause
  - If *a* is a *sufficient* cause for *b* then the occurrence of *a* is inevitably followed by the occurrence of *b*.
  - If *a* is a *necessary* cause for *b* then *b* will not occur unless *a* does. This has two negations in it: "will not" and "unless".

We find it easiest to think in terms of sufficient cause rather than using the double negations of necessity. Accordingly, we treat sufficient causation as primary, and throughout this paper, unqualified "cause" refers to sufficient cause.

2.2 Time, Events and Conditions

This section describes the building blocks of our approach to causation: Time, Events and Conditions.

2.2.1 Time

We assume that the instants of time are taken from a given primitive set, which is a strict total order, ordered on (transitive) **precedes**. We use the informal concept of granularity of time, recognising that it may be appropriate to work with a finer or coarser granularity of time, even in different parts of the same specification. No attempt has yet been made to include granularity in our formal specification. Work of this kind is reported in [11]. They describe a temporal logic language, with extensions permitting a temporal universe composed of temporal domains of different time granularities.

2.2.2 Events

We look at executions of the system in terms of the observations of events, each of a named class, made by a single observer, who only observes one event occurring at a time. An event occurrence is a tuple (**event class, time of occurrence).**

Event occurrences are unique; each event occurrence occurs only once, although a number of events of the same class can occur ordered in time – the **StartOf(red)** [traffic light] at 15:05 is

a different event occurrence from, but of the same class as, the **StartOf(red)** at 15:06. For brevity, and where there is no ambiguity, we say "**a** occurs at time **t**" (where **a** is an event class) to mean "an event occurrence of class **a** occurs at time **t** ".

2.2.3 Conditions

We base our concept of conditions upon intervals of the time line, where intervals are defined by their start and end-points.

A **condition class** is the name of a set of intervals, and a **condition occurrence** is one of those intervals, analogous to an instance of an abstract data class. For simplicity we insist that there is no overlap between the occurrences in a condition class. There are two events associated with each condition: **StartOf(***condition***)** and **EndOf(***condition***)** with their obvious meanings.

An event can be regarded as an interval with no duration, and so we regard event classes and occurrences as special cases of conditions. Conditions **hold** during their intervals, and a relation, HoldsAt, between Condition and Time, defines those time points at which a condition holds. A function, abbreviated to "~" in this paper, defines the complementary condition, so that ~c holds at those time points at which c does not hold.

Each condition class defines a set of occurrences with some property in common, e.g. those intervals during which a particular traffic light is red. Each useful condition class is associated with one or more predicates, which express the properties of the system which are true while the condition holds, i.e. during each interval for which there is an occurrence of that condition. These properties are expressed in terms of time-dependent variables

In our specification a time-dependent variable is expressed as a function from time to a value

$variable$: Time $\rightarrow \wp$ *value*

The form of a predicate associated with a condition is:

$\forall$ t:Time $\bullet$ *condition* holds_at t $\Rightarrow$ *variable* (t) = *value*

An example of this is shown in section 3.1.2.

2.3 Causal Relationships and Expressions

Causal relationships are universal relationships between conditions. We refer to them in the example below as **causal laws**, e.g. "striking a match causes fire ". This law does not refer to any particular occurrence of match-striking or fire. However, we may refer to **causal instances**, typically in the past tense, e.g. "the striking of a match (in a particular location) at 12:09 a.m. on 14/1/97 caused the fire there at 12.10 a.m.".

A causal expression has the following format:

[**While** *Compound_Condition*] *condition1 causal_relation condition2*

where we refer to *condition1* (the **causing** condition) as the **subject** and *condition2* (the **effected** condition) as the **object** of the causal expression.

The **While** clause limits the applicability of the causal expression to intervals in which the **While** compound condition holds, e.g. "While there is sufficient oxygen …"

We define four kinds of *causal_relation*:

- **directCauses**: starts a condition or makes an event occur;
- **terminates**: ends a condition;
- **sustains**: prevents a condition from ending;
- **prevents**: prevents a condition from starting or an event from occurring.

We give here the formal definitions of **directCauses** and its generalisation **indirectCauses**. The reader is referred to [5] for the similar formal definitions of **terminates**, **sustains** and **prevents**. It should also be noted that, arising from these definitions:

- **c directCauses d Û c terminates ~d** and **c terminates d Û c directCauses ~d**
- **c sustains d Û c prevents ~d** and **c prevents d Û c sustains ~d**

2.3.1 Direct Cause

We regard direct causation as implying that, unless it already holds, the caused condition starts immediately after the end of the causing condition, where "immediate" is governed by the granularity of time within which the observations are taking place. For example, although there may be a perceptible pause between flicking a switch and the light coming on, we may choose to make observations at a sufficiently coarse granularity of time that we can regard the two conditions as abutting. Formally[4]:

c directCauses d Þ EndOf c = StartOf d Ú EndOf c Î IntervalOf d

Situations 1 and 2 in Figure 1 illustrates this relationship; in situation 1 **d** starts immediately after **c** ends, and in situation 2 **d** already holds at the end of **c**. Situation 3 is inconsistent with **c directCauses d** because it does not show **d** starting immediately after the end of **c**.

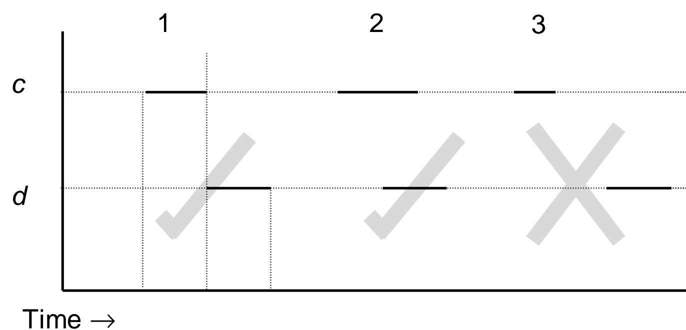


Figure 1 Condition c directCauses condition d

2.3.2 Indirect Cause

We define indirect cause in terms of direct cause. A condition **c** is said to **indirectly cause** condition **d** iff we can find intervening direct causes to make up a causal chain, including the case where the chain has only one link. This is defined recursively:

c indirectCauses d Û
(c directCauses d) Ú
($b: Condition · c indirectCauses b Ù b directCauses d)

---

[4] Ignoring issues of open and closed intervals discussed in [5].

Indirect cause is thus the transitive closure of the **directCauses** operator. Figure 2 illustrates the general causal relationship. The transitivity of indirect cause follows from its definition.

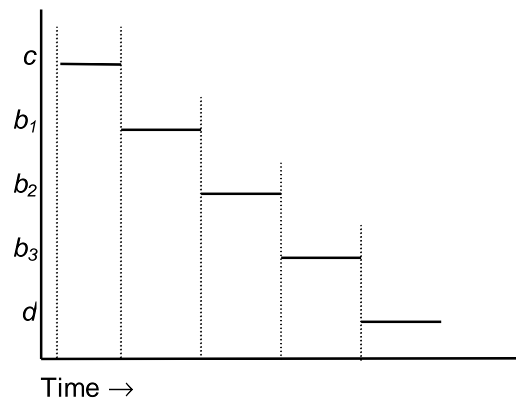


Figure 2 Condition c indirectCauses condition d

3. Example – Lift System

Before expounding the classes of behavioural conflict, we introduce an example system, which provides the reader with an introduction to the notation and us with material to discuss behavioural conflicts. Our example is a lift system, whose description is derived from a textual description that was designed for the requirements engineering module of a Masters course at the University of York. A set of informal requirements for the elevator control system was defined. To avoid duplication, they are set out in section 3.3, where we describe how each one is met by our design.

A full causal specification[5] was written in the Z language and type-checked. For brevity and readability, in this section we only introduce the signatures of variables, and any predicates which are relevant to conflicts are introduced at the appropriate point in section 4.

3.1 Background and Object Model

It is proposed to install a new passenger lift in a new building. An informal object model of the system has been defined. It is illustrated in Figure 3 and described below.

[5] Available as a report from the authors, on request.

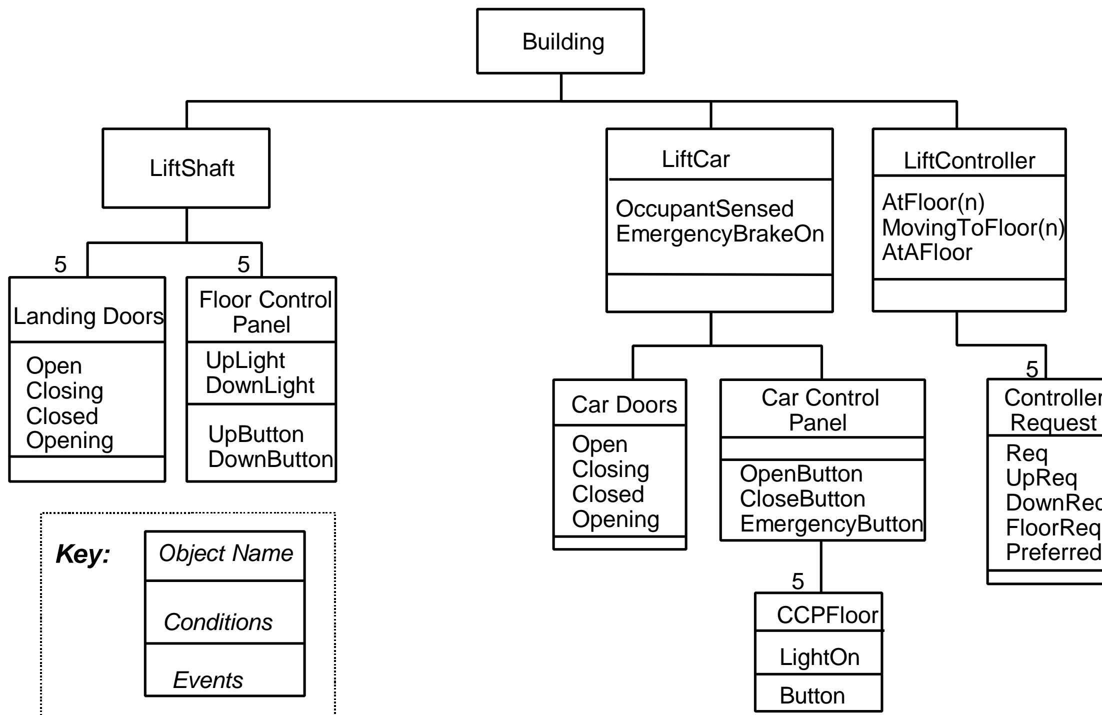
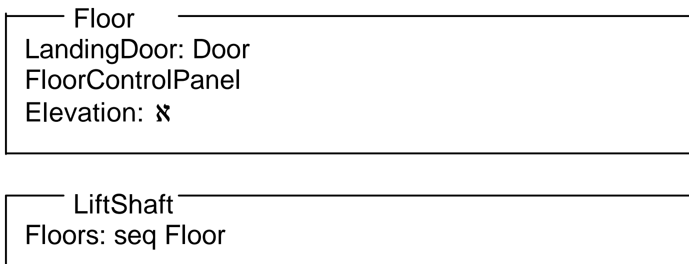


Figure 3 Lift System

3.1.1 The Lift Shaft and Landings

Lift Shaft and Floors

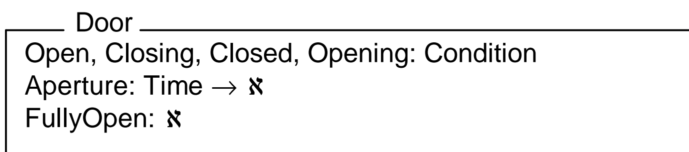
The lift shaft extends over five floors (0 - 4), each of which has a landing door and a control panel. Each floor has an elevation above some fixed point.



Landing Doors

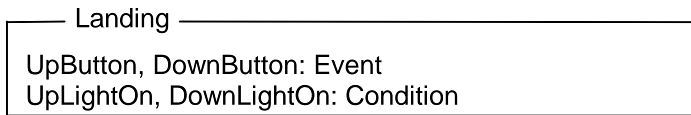
On the landing of each floor there is a door giving access to the lift shaft. Each door is opened and closed by a motor:

- Four conditions are associated with the door: Open, Closing, Closed, Opening
- Aperture is a measure of how far the door is open. FullyOpen is its value when Open holds.



Control Panel on Landings:

On the landing of each floor, near the lift door, there is a control panel containing Up and Down buttons and lights:



UpButton and DownButton are events which signal when they are pressed, and the UpLightOn and DownLightOn are conditions which hold when the lights are on.

3.1.2 Lift Car

The passengers are carried in a lift car which is moved by a motorised system. The schema that defines the lift car is given after its components have been defined.

Car Door

There is a door on the car, giving access to the floor landings. It is defined identically to the landing doors.

Control Panel in Car

The car has a control panel in it with:

- Buttons and lights for each floor destinations;
- Emergency stop button;
- Door open button;
- Door close button;

As for the control panels on landings, buttons are defined as events and lights as conditions.

Lift Car Conditions and Predicates

Variables of Elevation, Speed and Acceleration are declared as functions from Time to Number. Constant numbers define the maximum speed and acceleration for comfort and safety

There are the following conditions associated with the lift car:

- EmergencyBrakeOn: a condition which holds when the emergency brake is on. The brake brings the car to a halt when operated, with a deceleration which is greater than that defined by MaxAccComfort.
- OccupantSensed: a condition which holds when, and can be queried to determine whether there are currently occupants in the car.
- Stationary, InMotion, SpeedComfort, AccComfort: conditions which hold when their associated predicates are true, e.g. AccComfort only holds when the absolute value of Acceleration is less than a defined maximum value for comfortable acceleration:

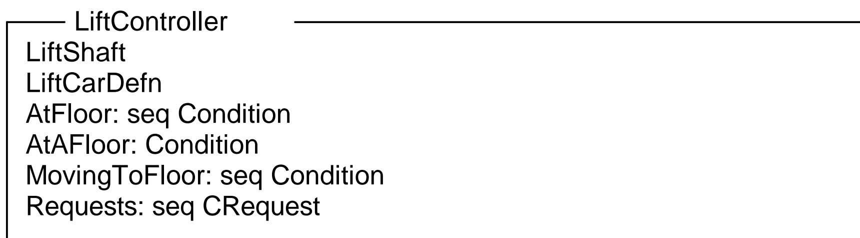
$$\forall \text{ t:Time} \bullet \text{AccComfort HoldsAt t} \Leftrightarrow \\ \quad \text{abs (Acceleration(t))} < \text{MaxAccComfort}$$

3.1.3 Controller

There is a controller module which contains the logic for controlling the system. The state of requests for a floor is defined in the Crequest schema:

- There are three kinds of request for a particular floor: as a result of the UpButton or DownButton having been pressed at that floor, or the relevant FloorReq button having been pressed from within the car.
- One of the main functions of the controller is to decide to which floor the car should go next. PreferredReq holds for only one floor at a time, and defines the next destination. We do not define the algorithm for this.

The LiftController schema includes LiftCar and Liftshaft and it is also aware of the position and movement of the car, and a sequence of Crequest to define the state of requests for each floor:



The LiftController schema includes LiftCar and Liftshaft and it is also aware of the position and movement of the car, and a sequence of Crequest to define the state of requests for each floor:

3.2 The Working of the Lift System

We describe here, using extracts from the formal specification, the essentials of the working of the system. This illustrates the application of the causal language and provides the basis for the investigation of conflicts described in section 4.

Calling the Lift

The lift system is a reactive system, and a cycle of activity is started by a passenger pressing a button on a landing, e.g. the Up button. This directly causes the associated light to come on and a request to be registered with the controller. Also it indirectly causes the lift eventually to arrive at the floor:

Whenever there is a change in the status of a request, the next destination floor may change. This is regularly recalculated by the Controller according to its algorithm, which is not visible at this level of description.

Despatching the Lift

The Controller, having calculated the next destination floor, despatches the lift to that floor.

Arriving at a Floor

When the lift arrives at a floor, the door starts to open, and opening is terminated when the door is fully open. The request to go to that floor is then terminated.

While the lift is at a floor, pressing the Up or Down button on the landing or Open button in the car causes the door to open if it is not open already.

We do not model the act of passengers entering and leaving the lift, but if there are any passengers in it the condition OccupantSensed holds.

Request a Destination

When a passenger presses a floor button, the request to go to that floor is registered and the door starts to close. Closing is terminated when the door is Closed. The controller can then service its next call.

3.3 Requirements for Lift System

A number of informal requirements for the elevator control system were given. The following are relevant to our discussion:

1 The elevator doors should not be allowed to be opened when the elevator is moving or between floors. This is regarded as a safety requirement;

2 When the 'Emergency' button on the elevator control panel is pressed the elevator should be stopped and the doors opened.

4. Behavioural Conflicts

In section 1.3 we placed behavioural conflicts in their context; they are those inconsistencies, in the specification of behaviour in terms of an consistent entity model, that make it impossible to construct a working system.

We assume that the behavioural specification is free of time-independent logical contradictions. Then, behavioural conflicts occur when two or more behavioural goals can, at some point in time, result in a contradiction. We can classify them as follows:

- Application-independent conflicts, in which the possibility of conflict can be seen from the semantics of the causal language, without reference to the application;
- Application-dependent conflicts, in which the conflict depends upon contradictions between predicates in the application;

Since behavioural conflicts are time-dependent, and the applicability of any causal statement is limited by any While clause that qualifies it, the conflicts can only occur during the time intervals in which the While clause holds. Depending upon the clause, it may be possible to evaluate the possibility of conflict statically. If not, timing assumptions will be necessary, as discussed in section 4.1.1, below.

The types of conflict and their means of detection are introduced, before their means of resolution are proposed. At this stage, the means of detection are quite simplistic. We have

primarily focussed our efforts on achieving a means of understanding of the conflict, as this is a necessary precursor to identifying rigorous methodological guidance.

4.1 Application-Independent Conflicts

In a different context [12] we suggested that the starting point for the detection of policy conflicts is that without some overlap there can be no conflict. The most obvious form of overlap is when two statements refer to the same entity. Given two causal statements:

x *causal-relation* y and z *causal-relation* w

there are four potential sources of overlap:

- Overlap of subjects, where x and z are the same condition. This does not typically give rise to conflicts, because it is normal for a condition to cause more than one other condition. There are several examples in section 3. We do not discuss these further.
- Overlap of objects, where y and w are the same condition. These are discussed in section 4.1.1.
- Overlap of both subjects and objects, where x and z are the same, and y and w are the same, but the linking causal relations are different. These are discussed in section 4.1.2.
- Overlap of the object of one statement with the subject of another, i.e. where y and z are the same. These are discussed in section 4.1.3.

We therefore suggest that a first approach to conflict detection should be to scan the specification, making a pairwise comparison of causal statements, to search for the occurrence of potentially conflicting pairs of conditions. These can be examined to decide if they match any of the conflict criteria described below.

4.1.1 Overlap of Objects

We can see the possible conflicting causal statements from the following table. Since there is no significance in the order of stating causal laws, we have omitted the lower diagonal:

	B directCauses C	B terminates C	B sustains C	B prevents C
A directCauses C	No conflict (a)	No conflict (b)	No conflict (b)	**Conflict**
A terminates C		No conflict (a)	**Conflict**	No conflict (b)
A sustains C			No conflict (a)	No conflict (b)
A prevents C				No conflict (a)

Notes:

a) All of the causal statements are idempotent, so no conflicts arise from overlaps on the diagonal.

b) **directCauses** and **prevents** are concerned with the start of the object condition, and **sustains** and **terminates** are concerned with its end, so neither **directCauses** nor **prevents** can conflict with **sustains** or **terminates**.

The conflicts which may arise are **directCauses** versus **prevents,** and **terminates** versus **sustains**.

- A **directCauses** C and B **prevents** C;

These state that the start of C coincides with the end of A, and that the start of C cannot occur while B holds. If B holds at the end of A there is a contradiction between the two statements, since the definition of **directCauses** implies that the start of C coincides with the end of A, which is within the interval of B, and the definition of **prevents** implies that the start of C is not within the interval of B.

- A **terminates** C and B **sustains** C. There is a similar contradiction relating to the end of C.

An example conflict of this kind is shown in the lift car specification:

StartOf(EmergencyBrakeOn) **terminates** AccComfort
OccupantSensed **sustains** AccComfort

Since there is no constraint preventing EmergencyBrakeOn from starting while OccupantSensed holds, there is the possibility of a conflict occurring.

We can generalise the **directCauses** case to **indirectCauses**:

- A **indirectCauses** C and B **prevents** C;

  The reasoning is similar to **directCauses**. There must be some X such that A **indirectCauses** X and X **directCauses** C; if B holds at the end of X there is a conflict, as above.

These possible conflicts need to be detected, and if they can cause an actual run-time conflict it must be resolved before the system is implemented. Jaffe et al [13] point out the difficulties of determining whether they can cause a run-time conflict as this depends on the relative timings of the causing conditions.

4.1.2 Overlap of both Subject and Object

One conflict arising from overlap of both subject and object is straightforward to detect and does not require analysis of the semantics of causal statements: complementary object conflicts, e.g. A **causal-relation** B and A **causal-relation** ~B. This leads to a conflict because, by the definition of "~", B and ~B cannot hold simultaneously.

We also need to consider the cases when the same two conditions are subject and object of two different causal statements, e.g. A **directCauses** B and A **terminates** B. These require more detailed analysis of the semantics of causal statements, which is shown in Appendix A. A summary of our conclusions from the analysis in the Appendix is:

- Since all the causal statements are idempotent, no conflict arises from the duplication of causal statements, e.g. A **directCauses** B and A **directCauses** B.
- No formal logical conflicts arise from any of the combinations examined, which we would expect since the intention in designing them was that their effects should be orthogonal.
- Three of the combinations are potentially useful:
  - StartOf(A) **directCauses** B and A **sustains** B. This is stronger version of **sustains** which has proved useful in an earlier application [5].
  - StartOf(A) **directCauses** B and A **terminates** B. This is stronger version of **terminates**.
  - A **directCauses** B and A **terminates** B: The net effect is to toggle B at the end of A, so that if B does not hold at the end of A then it will be initiated, and if B does hold at the end of A then it will be terminated. This appears to be a potentially useful combination.

- The remaining combinations are less likely to be useful as components of the language, and some are counter-intuitive in their effects, so they are best regarded as being potentially in conflict with the specifier's intentions.
- Further analysis to include the "~" operator (complementary condition – see section 2.2.3) is not necessary since, as described in section 2.3, we can replace causal statements which include this operator with their equivalents, e.g. replace **c directCauses ~d** with **c terminates d**

Overlaps of both Subject and Object therefore also need to be detected, and examined carefully for their effects on the behaviour of the system.

4.1.3 Sequential Conflicts

There are potentially conflicts arising from a chain of causal statements, in particular if there are two cycles rotating in opposite directions, e.g.:

A **directCauses** B and B **directCauses** C and C **directCauses** A
A **directCauses** C and C **directCauses** B and B **directCauses** A

It is unlikely that it is possible to construct a useful set of predicates associated with the three conditions which can consistently apply to both cycles.

A cycle occurs in the lift system:

StartOf(Open) **terminates** Opening
StartOf(Closed) **terminates** Closing
StartOf(Closing) **terminates** Open
StartOf(Opening) **terminates** Closed

This is quite legitimate, and the predicates (the value of Aperture) associated with each condition are consistent. However, a second cycle in the opposite direction would clearly be nonsensical.

Therefore the specification should be scanned for cycles of causes, to ensure that none of them conflict.

4.2 Application-Specific Conflicts

It has been possible to identify the above conflicts without reference to any specific application. However, even if none of these conflicts exist, there may be conflicts which depend upon the application. They arise when causal statements would put the system into two incompatible conditions, e.g. A **directCauses** AtFloor(1) and A **directCauses** AtFloor(2) simultaneously. Two conditions are incompatible if their associated predicates are in contradiction.

A **directCauses** AtFloor(1) and A **directCauses** AtFloor(2) are incompatible because they have contradictory predicates. We sketch the basis for a proof of this inconsistency, introducing known properties of the application to support it:

- For any n, AtFloor(n) holds only when the elevation of the lift is at the same elevation as Floor(n) since

$$\forall t\text{: Time; n: } \aleph \bullet \text{(AtFloor(n)) HoldsAt t} \Leftrightarrow \\ \text{(Elevation(t) = (Floors(n)).Elevation} \wedge \text{Speed(t) = 0)}$$

- The elevation of every floor is different since

  $\forall$ n1, n2: $\aleph$ | n1 $\leq$ 5 $\wedge$ n2 $\leq$ 5 $\bullet$
  (n1 > n2) $\Rightarrow$ ((Floors(n1)).Elevation > (Floors(n2)).Elevation)

- So

  $\forall$ t: Time $\bullet$ $\neg$ (AtFloor(1) HoldsAt t $\wedge$ AtFloor(2) HoldsAt t)

Occurrences of this type of conflict must be found by inspection as they are dependent upon properties of the application, rather than the syntax of the notation. We can now consider two specific examples of application-specific conflict from our lift example that serve to illustrate the principles.

4.2.1 Example 1: Emergency Button Conflict

A more practical example of a conflict is that between requirements 1 and 2 as defined in section 3.3.

The following predicates specify that AtFloor(n) only holds when the car is stationary at the floor n, and AtAFloor holds when AtFloor(n) holds for any n:

$\forall$ t: Time; n: $\aleph$ $\bullet$ (AtFloor(n)) HoldsAt t $\Leftrightarrow$
(Elevation(t) = (Floors(n)).Elevation $\wedge$ Speed(t) = 0)

$\forall$ t:Time $\bullet$ AtAFloor HoldsAt t $\Leftrightarrow$ ($\exists$ n:$\aleph$ $\bullet$ (AtFloor(n)) HoldsAt t)

Requirement 1 states: The elevator doors should not be allowed to be opened when the elevator is moving or between floors, and is expressed by following safety requirements in the specification:

$\forall$ n:$\aleph$ $\bullet$ (~AtFloor(n)) **prevents** (Floors(n)).LandingDoor.Opening
(~ AtAFloor) **prevents** CarDoor.Opening
(~ CarDoor.Closed) **sustains** Stationary

Requirement 2 states: When the ‘Emergency’ button on the elevator control panel is pressed the elevator should be stopped and the doors opened:

EmergencyButton **directCauses** EmergencyBrakeOn
EmergencyButton **directCauses** CarDoor.Opening

These causal statements violate both parts of requirement 1 unless the car is stationary at a floor when the button is pressed.

4.2.2 Example 2: Scheduling Conflict

Another, hypothetical, example of a conflict is a scheduling conflict which we introduce in order to illustrate a different method of resolution in section 4.3.1.

An initial specification of requirements for the system might say that pressing a request button directly causes the lift to come:

$\forall$ n:$\aleph$ $\bullet$ (Requests (n)).FloorReq **directCauses** MovingToFloor(n)

This statement would not be recognised as potentially leading to a conflict by the scanning methods recommended above if it were the only one whose object was MovingToFloor(n). However, there may be a number of passengers, whose demands for service conflict. The

conflict arises from the fact that the speed predicates associated with the conditions MovingToFloor(0) … MovingToFloor(4) are mutually incompatible.

4.2.3 Detecting Conflicts through the Use of a Conflicts Database

In order to search systematically for conflicting conditions, we need to identify those whose predicates may be in contradiction. It is possible in principle to automatically create a database of sets of conditions whose predicates are potentially in conflict. This would ensure thorough coverage, however the size of such a database is likely to be quite large.

If the database were limited to pairwise comparisons, its size would be $O(n^2)$, approximately 400 for our example. Unfortunately we cannot limit the inspection to pairwise comparisons. Consider the following:

A **directCauses** B

A **directCauses** C

C **sustains** ~B

Pairwise there is no conflict between the statements. Only together are they problematical, requiring examination of their predicates using knowledge of the application. The inclusion of three-way comparisons would increase the database size to $O(n^3)$, approximately 8,000 for our example. There could also be sets of four or more statements creating application conflicts. Therefore, although the concept of a conflicts database is attractive, it will require further work in order to ensure that it can be kept to a manageable size while still ensuring thorough coverage of conflicts, or efficient search strategies employed. We anticipate that for the moment, the most effective means of identifying such conflicts are likely to be through traditional inspection methods (though armed with an increased knowledge of the types of inconsistency), or supported through the kind of animation toolset described in section 6.3.

4.3 Resolution of Conflicts

Having identified conflicts, it is necessary to decide what to do with them. We work within the domain of safety critical systems, and so for us the most desirable solution in the face of conflict is to eliminate it entirely. This is however not always possible, and so we identify two further ways of managing the conflict. The actual acceptability of approach for any particular context must be judged on a case-by-case basis, and depends on the sector within which the technology is applied. The three possible approaches to resolution of conflicts are:

- Total elimination;
- Dynamic elimination;
- Prioritisation.

Elimination, as opposed to prioritisation, implies modifying the conflicting statements so that the conflict cannot occur. This must not, of course, be done unilaterally but has to be negotiated between the stakeholders of the system.

We illustrate the approaches using the two conflicting sets of statements from the lift specification:

EmergencyButton **directCauses** EmergencyBrakeOn S1

EmergencyBrakeOn **directCauses** CarDoor.Opening S2

These statements cause the Emergency Brake to bring the car to a halt and then cause the car door to open without reference to whether the car is at a floor. However, we also have the statement:

~AtAFloor **prevents** CarDoor.Opening S3

This statement prevents the car door from opening unless the car is at a floor. So there is a conflict if the Emergency Button is pressed at any time that will result in the car halting between floors.

4.3.1 Total Elimination of a Conflict

The conflict may be totally eliminated so that, regardless of the state of the system, it cannot occur. In the Emergency Button example we could achieve this by completely removing statements S1 and S2[6].

We can also eliminate the scheduling conflict (section 4.2.2) in the way that was actually done in the specification. If instead of **directCauses** we use **indirectCauses** the conflict of predicates is removed. Hence the solution that was adopted, which decoupled the registering of a floor request from the decision to despatch the lift to the floor:

∀ n:ℵ • (Requests (n)).FloorReq **indirectCauses** AtFloor(n)
∀ t: Time • (∃ n: ℵ • (Requests (n)).PreferredReq HoldsAt t)
∀ n:ℵ • (StartOf (Requests(n)).PreferredReq) **directCauses** MovingToFloor(n)

4.3.2 Dynamic Elimination of a Conflict

We can dynamically eliminate the conflict by modifying the conflicting statements so that they never apply at the same time. In the example above we could achieve this by completely replacing statement S1 with the following:

While AtAFloor S4
EmergencyButton **directCauses** EmergencyBrakeOn

Since ~AtAFloor and AtAFloor can never hold simultaneously, the conflict is eliminated.

4.3.3 Prioritisation of Conflicting Statements

Alternatively, we can introduce, in advance, methods of resolving the conflict without modifying the conflicting statements, by changing the semantics of the specification language, so that one statement is nullified by the presence of the other. There is then no need to modify the specification.

There are several possible approaches to prioritising conflicting statements, derived from similar conflicts in access control specifications (see, e.g. [14]). Possibilities include:

1. Labelling conditions with explicit priorities, so that e.g. A has priority 1 and B has priority 2, and resolving the conflict on the basis of priorities;
2. Resolving the conflict on a temporal basis, e.g. giving priority to the most recently created condition occurrence;

[6] Actually, we believe that the emergency button was introduced into the requirement by mistake, and this would be our preferred solution.

3. Resolving the conflict on the basis of the source of the requirement, e.g. giving priority to the most senior stakeholder;
4. Assigning priorities to the causal statements themselves, so that prevents has priority over causes and sustains has priority over terminates.

Although the first three methods have been discussed in the access control literature, they clearly add complication to the specification, and it may be difficult to predict their effects.

We prefer the last approach, and in our definition of the causal language we give **prevents** priority over **directCauses**, and **sustains** priority over **terminates**, so that in this example S2 has no effect. We can point to some persuasive arguments in favour of this approach.

Implied in the principle of causality – "nothing happens without a cause" is a hierarchy of behavioural imperatives. The default behaviour of a system is no change:

- Newton's First Law, which applies to the behaviour of physical systems, is that bodies continue in a straight line with uniform speed unless subjected to an external force;
- Typically, digital control systems are programmed to maintain their environment in a constant state unless they receive external inputs requiring change.

Any explicit causal imperative for action has priority over the default, and this is the second level of the hierarchy. This suggests a natural third level of hierarchy for the resolution of conflicts – that the explicit prevention of change has priority over imperatives for action. This is similar to the typical hierarchy that is used to resolve conflicts in access control specifications – if the default is negative, positive authorisations override the negative default, and explicit negative authorisations override positive ones. This gives sufficient power in most access control systems to meet all practical access specification needs.

We therefore propose the following hierarchy, in ascending order of priority:

- The default – nothing is specified and nothing changes;
- Positive statements – A **directCauses** B;
- Negative statements, which will override the positive – A **prevents** B.

A similar, orthogonal, hierarchy is used for **terminates** and **sustains**:

- The default – nothing is specified and nothing changes;
- Positive statements – A **terminates** B;
- Negative statements, which will override the positive – A **sustains** B.

This hierarchy does not apply universally. The emergency brake causes greater deceleration than is comfortable:

StartOf EmergencyBrakeOn **terminates** AccComfort

However, we also have the comfort requirement:

OccupantSensed **sustains** AccComfort

Our prioritisation gives **sustains** priority over **terminates**, but almost certainly this is not what the stakeholders require, so in this case the specification will need to be modified to exclude the case of emergency braking:

[while ~ EmergencyBrakeOn] OccupantSensed **sustains** AccComfort

Note that there is an advantage in using prioritisation over explicit removal of conflicts. S4, above, removes the conflict satisfactorily, but it has introduced redundancy into the specification; it for any reason it is necessary to change the statement S3, it will be necessary also to change S4, and anywhere else in the specification that a similar While occurs.

Although we have a view on the hierarchy of causal statements, as stated above, we accept that this is by no means settled – with experience it may be necessary to adopt a domain-specific approach to prioritisation of causal statements.

5. Related work

In this section we give some examples of related work dealing with behavioural inconsistencies and conflicts, from two application areas. The field of inconsistency and conflict is very wide, ranging over a large variety of applications, and straying into philosophy; we have limited ourselves to two areas that directly address behavioural inconsistency.

Feature Interactions in Telecommunications Systems

The most conspicuous example of behavioural conflict that has emerged recently is feature interactions in telecommunications systems. This was the subject of special issues of IEEE Computer [15] and IEEE Communications Magazine [16]. An example is the behaviour of Selective Call Rejection in the presence of Call Forwarding, which may result in undesired behaviour. In [17] Zave discusses formal approaches to managing feature interactions, identifying three main approaches: strong typing of features to prevent accidental interactions; specifying system state invariants, e.g. forbidding states in which there is a cycle of call forwarding; and formal specification methods such as the use of finite state automata.

More recently Thomas [7] presents a formal modelling approach to telecommunications services in which she points out that, although a "feature interaction" operator has been used in the literature it has never been formally defined, and goes some way towards its formal definition. Service features are defined as "first-class" concepts and explicit conflicts between them are modelled as logical inconsistencies and resolved by defining orderings between them, using LOTOS. The other source of potential feature interaction is non-determinism, which is captured in the LOTOS model and by the use of the modal μ-calculus for the analysis of timing properties (see the paper for references to these formalities). This appears to be a promising approach to dealing with the problem described in section 4.1.1 above, i.e. determining whether a potential timing conflict will materialise.

Database Integrity Constraints

Work on the enforcement of database integrity constraints is also relevant to behavioural inconsistency, if we regard a database transaction as a form of behaviour. Sheard & Stemple [18] describe a schema notation which, in addition to those constraints which can be expressed directly through an ER diagram, enables the expression of application-specific constraints. They then go on to show how it can be automatically verified that individual transactions conform to these constraints. Plexousakis & Mylopoulos [19] demonstrate an integrity maintenance technique that modifies transaction specifications by incorporating in them conditions necessary to constraint satisfaction, thus eliminating the need for subsequent verification. Techniques such as these could be useful, in a causal specification, for identifying application-independent conflicts and resolving conflicts of all kinds after identification.

6. Conclusions

6.1 Structural Inconsistencies and Behavioural Conflicts.

As we discussed in the introduction, a persuasive case has been made out by Gabbay, Nuseibeh and others for allowing inconsistencies to reside in specifications. The emphasis of their work has been on allowing inconsistency, with rather less discussion of the necessity for its eventual elimination. It is common ground among all concerned that inconsistencies must be removed from systems before they are implemented, with the exception of any facilities that may be provided for run-time resolution, e.g. by interaction with a human operator.

However, it is not sufficient to treat all inconsistencies at a single level. The behaviour of a system cannot begin to be described until there is a consistent model of the objects in it. Inconsistencies in the model – we might describe them as structural inconsistencies to contrast them with behavioural conflicts – must be resolved before the behavioural conflicts can be dealt with.

6.2 An Approach to Managing Behavioural Conflicts

We have implied an approach to the management of behavioural conflict in the discussion above, and here we suggest an outline process to do this methodically making use of the causal notation:

- The causal specification language itself can contain a mechanism for conflict resolution. We provisionally recommend giving priority to **prevents** and **sustains** over **directCauses** and **terminates**, respectively, but recognise that other methods, such as labelling statements with priorities, may turn out to be more appropriate.
- Structural inconsistencies must be removed before attempting to identify behavioural inconsistencies.
- We have identified four kinds of potential conflict which can be recognised without any knowledge of the application:
  - Overlap of subjects;
  - Overlap of objects;
  - Overlap of both subjects and objects;
  - Overlap of the object of one statement with the subject of another.

  These can be detected systematically by a pairwise comparison of causal statements in the specification.
- We have also described application-specific conflicts which arise from contradiction between the predicates associated with conditions.

  Some, at least, of these can be detected by compiling a conflicts knowledge base, which lists those sets of conditions whose predicates are potentially in conflict with each other because their predicates are related, and systematically checking this for potential conflicts.
- Thought needs to be given to the resolution of conflicts:
  - Where there is a built-in resolution method, such as prioritisation of causal statements, that resolution may not be correct, and may need to be overridden by explicit modification of the specification;

- Where conflicts can be resolved, they should be. Although the notation has the ability to tolerate continuing conflicts, a working system cannot be created until they are removed.

6.3 Future Work: Tool Support

If an approach such as we have suggested is to be useful in practice, then tool support for the detection of conflicts is necessary. At this stage it appears to us that the *resolution* of conflicts is an art rather than a science and, until we can systematise an approach to resolution, tool support in that area cannot be envisaged. With regard to conflict *detection*, we here describe a framework for the tools that we envisage as being useful and practical. There are two main areas (see section 4): application-independent conflicts and application-specific conflicts. In addition, animation of the specification can provide useful insights into the behaviour of the system, including identification of some conflicts.

Application-Independent Conflicts

There are two preconditions for any kind of application-independent conflict

- Overlaps of the kinds described in section 4.1;
- Simultaneity.

We envisage construction of a tool which identifies the overlaps by static analysis of the specification, and reports them to the specifier for manual analysis. It will be necessary for the designer, with some help from animation, to perform the analysis which determines whether the conditions can occur simultaneously and, in some cases, whether the overlaps are in conflict with the specifier's intentions.

Application-Specific Conflicts

As mentioned in section 4.2.3, systematic search for conflicting predicates associated with conditions will require a conflicts database, recording those sets of predicates that are in conflict. Tool support for this can help at two stages:

- Production of a candidate list of conflicting predicates, for manual culling;
- Identification of the causal statements associated with each set of predicates, for analysis by the specifier.

A simple version of the tool, using pairwise comparisons only, would be the first stage. Further work is needed on the problems of n-way conflicts.

Animation

An animation tool is a valuable means of examining the behaviour of a system, and preliminary work has been done on this [20], successfully animating a simple specification by translating it into Prolog, setting initial conditions, running the program and outputting the successive states of the system to a simple graphical interface. Since we take a sceptical view of being able, in the near future, to prove that a specification is free of all conflicts, an effective animation tool for causal specifications will include in its benefits the possibility of examining the system for any unusual behaviour, including conflicts.

6.4 Conclusion

This paper has shown some of the potential for discovering and resolving behavioural conflicts using a specific notation. Although we believe that the causal notation has advantages, it is clearly only one of many. Work needs to be done to see how far our approach generalises to other notations, and how far its promise will be fulfilled in industrial practice.

Acknowledgements

We acknowledge valuable feedback on this topic from our colleagues and from students on the MSc in Safety Critical Systems Engineering at York, to whom an earlier version of this paper was presented. Jon Hall kindly reviewed, and improved, a draft of this paper, which has also benefited from the comments of the anonymous reviewers.

Appendix A – Conflicts involving Overlap of both Subject and Object

We consider the cases when the same two conditions are subject and object of two causal statements, e.g. A causes B and A terminates B. These require more detailed analysis of the semantics of causal statements.

	A causes B	StartOf(A) causes B	A terminates B	StartOf(A) terminates B	A sustains B	A prevents B
A causes B	No conflict	Note 1	Note 2	Note 3	Note 4	Note 5
StartOf(A) causes B		No conflict	Note 6	Note 7	Note 8	Note 9
A terminates B			No conflict	Note 10	Note 11	Note 12
StartOf(A) terminates B				No conflict	Note 13	Note 14
A sustains B					No conflict	Note 15
A prevents B						No conflict

1 A **directCauses** B and StartOf(A) **directCauses** B: B starts with A if is does not already hold. If B ends during A, then A **directCauses** B will start B again at the end of A.

2 A **directCauses** B and A **terminates** B: If B does not hold at the end of A then A **directCauses** B will initiate B and A **terminates** B will have no effect. On the other hand, if B does hold at the end of A then A **directCauses** B will have no effect and A **terminates** B will be effective in terminating B. The net effect is to toggle B at the end of A

3 A **directCauses** B and StartOf(A) **terminates** B: These will end B if B holds when A starts, and A **directCauses** B will start B again at the end of A.

4 A **directCauses** B and A **sustains** B: If B holds at any point while A holds, B does not end before the end of A, and in any case it is started at the end of A.

5 A **directCauses** B and A **prevents** B: B is not started (by any other condition) while A holds, and starts at the end of A.

6 StartOf(A) **directCauses** B and A **terminates** B: This is strong **Terminates** as defined in [5]. If B does not already hold, it is started at the start of A and is terminated at the end of A if it has not been terminated before.

7 StartOf(A) **directCauses** B and StartOf(A) **terminates** B: This will cause and simultaneously terminate B, i.e. B is an event that is identical to the start of A.

8 StartOf(A) **directCauses** B and A **sustains** B: This is strong **Sustains** as defined in [5]. If B does not already hold, it is started at the start of A and continues at least until the end of A.

9 StartOf(A) **directCauses** B and A **prevents** B: B does not start at the start of A because prevents has priority over causes, and cannot be started (by any other condition) during A.

10 A **terminates** B and StartOf(A) **terminates** B: B is terminated at the start of A and also at the end of A, if it holds at either point

11 A **terminates** B and A **sustains** B: If B holds during A, it is sustained until the end of A, and then terminates.

12 A **terminates** B and A **prevents** B: B is prevented from starting during A, and if it held at the start of A, it is terminated at the end of A

13 StartOf(A) **terminates** B and A **sustains** B: **sustains** has priority over **terminates**, which has no effect.

14 StartOf(A) **terminates** B and A **prevents** B: B terminates at the start of A and cannot be started by any other condition while A holds.

15 A **sustains** B and A **prevents** B: While A holds, if B holds it is sustained, and if it does not hold it is prevented.

References

1. Kotonya, G. and I. Sommerville, *Viewpoints for Requirements Definition.* Software Engineering Journal, 1992. **7**(6): p. 375-38.
2. Gabbay, D. and A. Hunter, *Making inconsistency respectable 1: A logical framework for inconsistency in reasoning*, in *Foundations of Artificial Intelligence Research*, P. Jorrand and J. Kelemen, Editors. 1991, Springer. p. 19-32.
3. Hunter, A. and B. Nuseibeh. *Analysing Inconsistent Specifications*. in *3rd Int. Symposium on Requirements Engineering (RE'97)*. 1997. Annapolis, USA: IEEE Computer Society Press.
4. Nuseibeh, B. *To Be and Not to Be: On Managing Inconsistency in Software Development*. in *8th Int. Workshop on Software Specification and Design (IWSSD-8)*. 1996. Schloss Velen, Germany: IEEE CS Press.
5. Moffett, J.D., *et al.*, *A Model for a Causal Logic for Requirements Engineering.* Journal of Requirements Engineering, 1996. **1**(1): p. 27-46.
6. Rumbaugh, J. and M. Blaha, *Object-oriented modelling and design.* 1991: Prentice-Hall.

7. Thomas, M., *Modelling and analysing User Views of Telecommunications Services*, in *Feature Interactions in Telecommunications Networks*. 1997, IOS Press. p. 163-183.
8. Spivey, J.M., *The Z Notation: A Reference Manual*. 2nd ed. 1992: Prentice Hall.
9. Shoham, Y., *Reasoning about Change: Time and Causation from the Point of View of Artificial Intelligence*. 1987: MIT Press. ISBN 0-262-19269-1.
10. Sosa, E. and M. Tooley, eds. *Causation*. Oxford Readings in Philosophy. 1993, Oxford University Press. 33-55.
11. Corsetti, F., *et al. Dealing with different time scales in formal specifications*. in *IEEE-ACM International Workshop on Software Specification and Design*. 1991. Como.
12. Moffett, J.D. and M.S. Sloman, *Policy Conflict Analysis in Distributed System Management.* Journal of Organizational Computing, 1994. **4**(1): p. 1-22.
13. Jaffe, M.S., *et al.*, *Software requirements analysis for real-time process-control systems.* IEEE Transactions on Software Engineering, 1991. **17**: p. 241-258.
14. Jonscher, D. and K.R. Dittrich, *A Formal Security Model Based on an Object-Oriented Data Model.* 1993. Institut für Informatik der Universität Zürich, TR 93.41, Oct 1993.
15. *Special Issue on Feature Interactions in Telecommunications Systems.* IEEE Computer, 1993. **26**(8).
16. *Special Issue on Feature Interactions in Telecommunications Systems.* IEEE Communications Magazine, 1993. **31**(8).
17. Zave, P., *Feature Interactions and Formal Specifications in Telecommunications.* IEEE Computer, 1993. **26**(8): p. 20-31.
18. Sheard, T. and D. Stemple, *Automatic Verification of Database Transaction Safety.* ACM Transactions on Database Systems, 1989. **14**(3): p. 322-368.
19. Plexousakis, D. and J. Mylopoulos. *Accommodating Integrity Constraints during Database Design*. in *EDBT-96*. 1996. Avignon, France.
20. Chatzikyriakos, E., *Causal Animator.* 1998. University of York, Dept of Computer Science, MSc, .