Deriving Specifications from Requirements: an Example

Michael Jackson
AT&T Bell Laboratories
and MAJ Consulting Ltd
101 Hamilton Terrace
London NW8 9QX England
jacksonma@attmail.att.com
mj@doc.ic.ac.uk

Pamela Zave
AT&T Bell Laboratories
Room 2B-413
Murray Hill NJ 07974
pamela@research.att.com

Abstract

A *requirement* is a desired relationship among *phenomena* of the *environment* of a system, to be brought about by the hardware/software *machine* that will be constructed and installed in the environment. A *specification* describes machine behaviour sufficient to achieve the requirement. A specification is a restricted kind of requirement: all the environment phenomena mentioned in a specification are *shared* with the machine; the phenomena constrained by the specification are *controlled* by the machine; and the specified constraints can be determined without reference to the future. Specifications are derived from requirements by reasoning about the environment, using properties that hold independently of the behaviour of the machine. These ideas, and some associated techniques of description, are illustrated by a simple example.

1 Introduction

Software development is concerned with the construction of machines of a particular kind: those that can be implemented by a general-purpose computer, which then becomes the desired machine. Many problems can be solved by these means [Jackson 94], including problems in process control, message switching, text manipulation, decision support, and other fields. For example, an information system is a machine that models a real world outside itself and produces information about it based on the model; a word-processing system is a machine that offers its user a repertoire of operations on texts held within the machine; a control system is a machine that interacts with its environment to bring about or maintain relationships in that environment. We call the hardware/software to be developed the *machine*, preferring this term to the more common *system*, which we consider to be open to too many interpretations. For example, the term *system* may be used to denote the hardware/software machine; or the machine together with the part of the environment with which it interacts directly; or the machine together with its users and the whole environment.

Although the different kinds of problem, and the appropriate methods, have much in common, we focus in this paper on control systems, and on their functional requirements. They seem to offer the cleanest and most concise illustration of the points that we want to make.

A *requirement* states desired relationships in the environment — relationships that will be brought about or maintained by the machine. The requirement is concerned entirely with the environment, where the effects and benefits of the machine will be felt and assessed: the machine is purely a means to the end of achieving the required effect in the environment.

A *specification* describes the behaviour of the machine at its interface with the environment. Like a requirement, it is expressed entirely in terms of *environment phenomena*. Seen from the machine, a specification is a starting point for programming; seen from the environment, it is a restricted kind of requirement.

A specification is derived from a requirement. Given a requirement, we progress to a specification by purging the requirement of all features — such as references to environment phenomena that are not accessible to the machine — that would preclude implementation. The derivation is made possible by environment properties that can be relied on regardless of the machine's behaviour. These properties

ICSE '95, Seattle, Washington USA
© 1995 ACM 0-89791-708-1/95/0004...$3.50

must, of course, be explicitly described if they are to be exploited.

Such derivation of specifications from requirements is loosely analogous to program refinement [Morgan 90]. In program refinement the purpose is to refine a specification to a program. Program specifications and programs are expressed in the same language, which contains both non-executable elements and executable code. Refinement is complete when all non-executable elements have been removed. The result is a program, because it contains only executable code. The refinement steps must ultimately be justified by appeal to the properties of the computer, as embodied in the semantics of the specification and programming language.

In refining requirements to specifications, we begin with requirements expressed in terms of the environment phenomena. Just as program specifications may contain non-executable elements, so requirements may refer to phenomena that are inaccessible to the machine. Refinement is complete when all references to inaccessible phenomena have been removed. The result is purely a description of machine behaviour. The refinement steps must ultimately be justified by appeal to the properties of the environment.

In this paper we present some elements of a method for describing requirements and for deriving specifications from them. We explain certain distinctions that we regard as essential to a sound treatment, and we show how they guide us in bridging the gap between requirements and specifications. We also show how certain real-time considerations can be handled in a simple and direct way.

We illustrate our points chiefly by means of a very small example. Our intention in using this small example, rather than something more substantial, is to ensure that as little detail as possible is left to the reader's imagination. In presenting the example we rely on finite-state automata and predicate logic as descriptive languages. This choice is meant to simplify the presentation: it should be taken neither as a recommendation nor as an intended contribution.

2 Designating Environment Phenomena

Our small example concerns the control of a turnstile at the entry to a zoo. The turnstile consists of a rotating barrier and a coin slot, and is fitted with an electrical interface. This mechanical apparatus has already been chosen, and the development job is to write the controlling software. The software will run in a small computer: this is the *machine*. The *environment* is the turnstile mechanism itself and its use by visitors to the zoo. To enter the zoo, a visitor must first push on the turnstile barrier, moving it to an intermediate position from which it will continue rotating of its own accord, returning to its initial position and gently pushing the visitor into the zoo. The turnstile is equipped with a locking device; when locked it prevents the barrier from being pushed to the intermediate position.

The first step is to decide what environment phenomena are of interest (we consider entity classes to be phenomena too). We capture these decisions by writing a *designation set*. Each designation of the set gives a careful informal description by which certain phenomena may be recognised in the environment; it also gives a term by which the phenomena may be denoted in requirement and specification descriptions:

In event *e* a visitor pushes the barrier to its intermediate position ≈ Push(*e*)

In event *e* a visitor pushes the barrier fully home and so gains entry to the zoo ≈ Enter(*e*)

In event *e* a valid coin is inserted into the coin slot ≈ Coin(*e*)

In event *e* the turnstile receives a locking signal ≈ Lock(*e*)

In event *e* the turnstile receives an unlocking signal ≈ Unlock(*e*)

The terms on the right hand sides of the designations are predicates. Push(*e*) is a predicate that is true of *e* if and only if *e* is an event in which a visitor pushes the barrier to its intermediate position. In this small example, all the designated phenomena are unary predicates characterising sets of events. This is not typical: in general, designated terms are n-ary predicates. However, it is fully typical that we choose to refer to the designated phenomena by predicates. Our phenomenology is based on facts about individuals; predicates are regarded as generalisations of such facts, and hence as the appropriate vehicle for denoting phenomena [Jackson 92].

By deciding on the designations that are specific to the environment — Push(*e*), Enter(*e*), Coin(*e*), Lock(*e*) and Unlock(*e*) — we are not only laying down a basis for description. We are also identifying the phenomena in terms of which we will express the requirement and specification. This is an important decision, and must be made consciously and explicitly. It is often claimed that requirements are relative: one person's requirement is another's implementation, and one person's *what* is another's *how*. Without the clear statement that designations provide, it is easy to vacillate about the subject matter of the requirement. Is the requirement really about controlling a turnstile, or is it more generally about admitting and excluding visitors? Or is it about the zoo's profitability? Or perhaps about the profitability of the company that owns the zoo? Might the developers legitimately recommend that entry should be free? Or that the zoo be sold and its real estate redeveloped? Writing a designation set locates the requirement unambiguously in the world.

We must also state explicitly that we are adopting our usual phenomenology of time [Jackson 92, Zave 93]. Like most researchers in formal specifications and requirements engineering, we usually regard events as atomic and totally ordered. We also regard both events, and intervals between successive events, as individuals. Each event begins one interval and ends another. Predicates associated with time-varying phenomena must have interval arguments. The appropriate designations for our view of time are:

e is an atomic instantaneous event	≈	Event(*e*)
v is an interval in which no event occurs	≈	Interval(*v*)
Event *e* occurs before event *f*	≈	Earlier(*e*,*f*)
Event *e* begins interval *v*	≈	Begins(*e*,*v*)
Event *e* ends interval *v*	≈	Ends(*e*,*v*)

These temporal phenomena are general, being recognisable in many different environments. We will assume in this paper that the appropriate assertions about them — for example, that Earlier(*e*,*f*) is a total ordering on events — have been made.

3 Shared Phenomena

If the machine is to interact with the environment, some phenomena must be *shared* by both. Investigation of the turnstile mechanism and its electrical connections shows that Push(*e*), Coin(*e*), Lock(*e*), and Unlock(*e*) are shared phenomena; Enter(*e*) is not shared. (Sharing phenomena does not imply sharing control. Rather, the shared phenomena may be regarded as constituting the interface between the machine and the environment, and control may reside on either side of the interface. We return to this point in Section 4 below.)

By identifying certain events as shared we are choosing to regard them as occurring both in the machine and in the environment. Since events are atomic and instantaneous, this means that we are ignoring any delay involved in transmission of the electrical signals. This decision is reasonable in the context of the turnstile. If we were to decide that the delay is *not* ignorable, we would treat the electrical channel as another part of the environment, distinguishing the events at the machine end of the channel from those at the turnstile end. The shared events would then be those at the machine end of the channel; the events at the other end would not be shared.

The underlying basis of shared phenomena is shared individuals: the event individuals appear in both the environment and the machine. But this is not enough. It is also necessary that the facts about those individuals, generalised in the predicates, are shared. Push and Coin events are clearly distinguished in the environment. But if, perversely, they were identically signalled by the turnstile, then they would still be shared individuals, but the distinction captured in the two predicates would not be accessible to the machine.

Similar considerations apply to shared state phenomena. In a lift control system, the sensors at the floors may be shared individuals. For the information from the sensors to be accessible to the machine, the facts that particular sensors are associated with particular floors, and that a particular sensor is On or Off in a particular time interval, must also be shared.

4 Control of Phenomena

We must also determine where *control* of the shared phenomena resides. Investigation — confirming everyday expectations of turnstiles — shows that Push and Coin events are *environment-controlled*, while Lock and Unlock events are *machine-controlled*. Push and Coin events are environment-controlled because they are *initiated* by the environment. Approaches based on the identification of agents [Feather 87, Johnson 88, Feather 91] would identify agents in the environment rather than in the machine for these events: if there are no visitors to the zoo, no Push or Coin event will ever occur, regardless of the machine's behaviour. Conversely, Lock and Unlock events are initiated by the machine, which sends electrical signals to the turnstile. Regardless of the behaviour of the environment, no Lock or Unlock event will occur unless the machine causes it. Environment phenomena that are not shared are necessarily environment-controlled. (Machine phenomena that are not shared are, of course, of no interest in requirements or specifications. They are significant only in programming.)

Control of an event is the power to perform it spontaneously, but only when it is not precluded by other constraints on its occurrence. Some environment-controlled events may be constrained by environment properties; the machine can exploit these constraints to prevent the events from occurring. For example, Push and Enter events are environment-controlled; but, as we shall see, the machine can prevent their occurrence by locking the turnstile. Coin events are also environment-controlled, but their occurrence, by contrast, can be neither prevented nor stimulated by the machine.

Control of state phenomena is associated with control of events. To say that the environment in a lift scheduling problem controls the state of the floor sensors is to say that the environment controls those events that cause the sensor states to change. The lift scheduling machine can access the sensor states, but only the movement of the lift car in the environment can change them.

In our view, control of events is always unilateral: it is never shared. We consider shared control to be unrealistic: it is rarely found in the real world [Feather 87, Abadi 93]. If some kind of event is sometimes initiated by the machine and sometimes by the environment, we separate it into two kinds by designating the machine-controlled and the environment-controlled events as different phenomena. In some cases shared state phenomena may be changed either by the machine or by the environment.

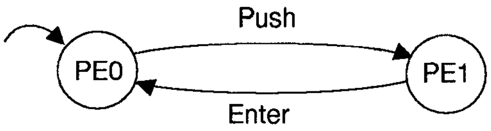
5 Indicative Environment Descriptions

In developing requirements we are interested in two distinct kinds of environment description. The first kind describes the properties we would like the machine to bring about or maintain. These descriptions are in what grammarians would call the *optative* mood: they express our wishes. The second kind describes the properties that the environment has, or will have, regardless of the behaviour of the machine. These are in the *indicative* mood: they express what is the case whether we wish it or not.

We avoid descriptions of mixed indicative and optative mood. This separation allows the mood of a description to be determined by its context rather than by its contents. We adopt this approach for two reasons. First: reliance on internal syntactic distinctions. whether formal or informal, between the two moods would cause great linguistic difficulty and would exclude many languages from effective use. Second: when a system has been successfully built and installed the optative descriptions become indicative — the wishes come true. It would be very inconvenient if the descriptions themselves then had to be rewritten. The contextual information on which we rely is, so far, quite informal; but in a practical development environment it should be formalised. In this paper we distinguish the moods of descriptions by giving indicative descriptions names of the form INDn, and optative descriptions names of the form OPTn. We also use definitions of new (undesignated) terms. Definitions may appear in indicative descriptions, where they may rely on the truth of the indicative assertion. They may also appear in separate, purely definitional, descriptions, whose names are of the form DEFn.

We begin here with two indicative properties. The first of these properties is that Push and Enter events alternate, starting with Push. A visitor can not Enter without first Pushing; the next visitor can not Push until the first has Entered. This property is described in a Finite-State Automaton:

(IND1)

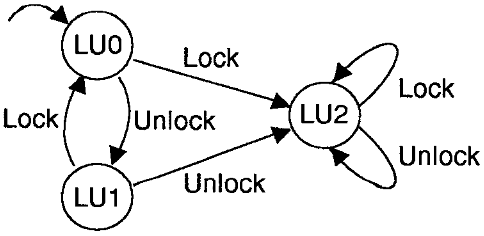


The state names PE0 and PE1 do not refer to designated phenomena: they are *defined* in this indicative description. The description *asserts* only a constraint on the ordering of Push and Enter events. It could be falsified by observation of the environment — for example if the sequence <Push,Push> were found to be possible. The property asserted is purely a safety property: the description would still be true if no Push or Enter event ever occurred.

The second indicative property is that if Lock and Unlock events alternate, starting with Unlock, then a Push event can occur only after an Unlock and before the next Lock. This too is a safety property, but its description needs a little care. We do not know, and therefore must not describe, what will happen if Lock and Unlock events do not alternate in the stated way. Possibly the turnstile mechanism will break; perhaps events not fitting the pattern will be ignored; perhaps the mechanism will become permanently locked or permanently unlocked.

So we make this description in two stages. In the first we define three states of the mechanism. LU2 is the state reached when the alternation has been broken. LU0 and LU1 are the two alternating states in which the alternation has been (so far) maintained:

(DEF1)



This description is purely definitional. It has an outgoing arc in each state for each kind of event, and so imposes no safety constraint on the event occurrences. Nor is it intended to express any liveness property: there is no implication that the initial state, or any other, will not persist indefinitely. The states LU0, LU1 and LU2 do not appear in the designation set. Nothing in this description DEF1 could be falsified by observation of the environment.

These definitions can now be exploited to assert a safety property:

(IND2) $\forall\, e,v \bullet ((\text{LU0}(v) \wedge \text{Ends}(e,v)) \rightarrow \neg\, \text{Push}(e))$

This description asserts that if LU0 holds in interval *v*, and *v* is ended by an event *e*, then *e* can not be a Push event: in other words, Push events are impossible in state LU0. The assertion could be falsified by environment observation — for example, if a Push were found to be possible before the first Unlock. The description exploits the definition of the states, both to assert the safety property concerning Push events and to limit the assertion to the known cases. If we later discover that Lock or Unlock events not fitting the alternating pattern will be ignored, we can add further definition and description to capture the resulting properties without changing or contradicting what we have already said. This kind of technique is essential to effective separation of concerns.

6 Requirements

It is the customer's prerogative to determine the requirements. Essentially, there are two simple requirements: that no-one should enter without paying; and that anyone who has paid should be allowed to enter.

Our customer does not require that payments alternate with entries: that would inconvenience school teachers in charge of groups of children. So the first requirement is simply that entries should never exceed payments. Assume that we have defined predicates Push#(*v,n*), Enter#(*v,n*) and Coin#(*v,n*), meaning that the count of Push, Enter and Coin events respectively preceding interval *v* is *n*. (Like the states PE0 and PE1, and LU0, LU1, and LU2, these are not newly designated environment phenomena: their definitions are based purely on the previously designated phenomena.) The first requirement can then be stated:

(OPT1) $\forall\, v,m,n \bullet ((\text{Enter\#}(v,m) \wedge \text{Coin\#}(v,n)) \rightarrow (m \leq n))$

The second requirement is that visitors who pay are not prevented from entering the zoo. Strictly interpreted, this requirement is unimplementable: they may be prevented by other visitors ahead of them in the queue, or by a police cordon, or by their own inability or unwillingness to perform the Push action that must precede the Enter event that admits them. Intuitively, it means that *the machine* will not prevent their entry. For now, we can state this requirement very informally as:

(OPT2) $\forall\, v,m,n \bullet ((\text{Enter\#}(v,m) \wedge \text{Coin\#}(v,n) \wedge (m < n)) \rightarrow$ 'The machine will not prevent another Enter event')

Later we will make it precise in the form of a specification of the machine behaviour. Like many requirements, this requirement seems very difficult to formalise solely in terms of phenomena that are important to the customer [Johnson 88]. A precise statement must await refinement in terms of the turnstile mechanism.

7 Specifications

A requirement describes a desired relationship among environment phenomena; a specification describes a desired behaviour of the machine in the environment. To be a specification, a requirement must observe at least these rules:

(a) All environment phenomena mentioned in the requirement are shared with the machine. That is, the specification is located entirely at the interface between the machine and the environment.

(b) All phenomena required to be constrained are directly machine-controlled. That is, the implementor will not need to reason about environment properties to achieve execution or inhibition of events: the machine can execute, or refrain from executing, the actions directly.

(c) All required constraints on events are expressed in terms of preceding events or states in preceding intervals. That is, the conditions for executing, or not executing, an event can be evaluated in a suitably defined current state and do not involve reasoning from a subsequent state.

The two requirements stated in the descriptions OPT1 and OPT2 express the customer's intention, but they are not specifications. Both are expressed in terms of Enter events, which are not shared: so they break rule (a).

To realise OPT1 the machine must either compel Coin events or prevent Enter events. Coin events are shared phenomena, but they are environment-controlled. If, then, we interpret OPT1 as requiring the machine to enforce Coin events, it fails as a specification by rule (b): it requires constraints on phenomena that are not machine-controlled.

OPT1 also constrains the state in every interval, including those that are still in the future. When the machine executes, or refrains from executing, any event, it must ensure that OPT1 will hold *afterwards.* A requirement based in this way on a future state, even if refined to a form in which it infringes neither rule (a) nor rule (b), can not be a specification by rule (c).

Our strategy for obtaining a specification from a requirement is to make explicit use of the indicative environment properties. Denoting the requirement, specification, and environment properties by R, S, and E respectively, for a given R and E we seek S such that

$$S, E \vdash R$$

Satisfaction of the requirement can be deduced from satisfaction of the specification together with the indicative environment properties.

Considering OPT1, we know of no environment property by which the machine could ensure the occurrence of Coin events. Therefore it must instead act to prevent Enter events. We must rely on the safety properties described in IND1 — the alternation of Push and Enter events; and in IND2 — the impossibility of Push events occurring after certain sequences of Lock and Unlock events. Our specification will require the machine to perform Lock and Unlock events so that certain Push events, and hence the undesired Enter events, do not occur.

The first step is to obtain a form of OPT1 that does not involve Enter events. From the indicative description IND1 we can immediately derive:

(IND3) $\forall\ v,m,n\ \bullet$
$((\text{Enter\#}(v,m) \wedge \text{Push\#}(v,n)) \rightarrow (n{-}1 \leq m \leq n))$

That is: at all times Push#-1 ≤ Enter# ≤ Push#. This property allows us to obtain OPT1a, expressed in terms of Push# (whose definition depends on shared phenomena), rather than of Enter# (whose definition depends on unshared phenomena):

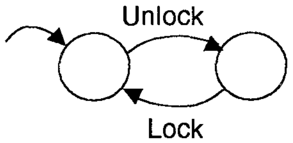
(OPT1a) $\forall\ v,m,n\ \bullet$
$((\text{Push\#}(v,m) \wedge \text{Coin\#}(v,n)) \rightarrow (m \leq n))$

OPT1a is a strengthening of OPT1. The requirement OPT1 is, informally, that at all times Enter# ≤ Coin#. OPT1a specifies the stronger condition that at all times Enter# ≤ Push# ≤ Coin# (the first part of the inequality being guaranteed by IND3). The strengthening is inevitable. If Push# > Coin# were ever allowed to hold, the environment possesses no properties by which Enter# > Coin# could then be prevented: once a Push has occurred the subsequent Enter can not be stopped; and a further occurrence of Coin can not be enforced.

To satisfy the requirement OPT1, then, the machine must ensure that Push# *never exceeds* Coin#. By an obvious piece of reasoning necessitated by rule (c), we refine this to the requirement that when Push# *already equals* Coin# the machine must prevent a *further* Push at least until after a further Coin event. How can Push events be prevented by the machine?

IND2, together with the definitional description DEF1, constrains Push events *provided that the alternation of Lock and Unlock events is maintained*: in the absence of this alternation we can say nothing. So we require the machine's behaviour to satisfy the following safety specification:

(OPT3)



If the machine behaviour has this property, we can be sure that LU2 will never hold. In any interval either Pushes are impossible because LU0 already holds, or LU1 holds and the machine can reach LU0 by causing a Lock event.

Now we can refine OPT1 (in its strengthened form OPT1a). The refinement is to a safety property and a liveness property. The safety property is:

(OPT4) $\forall\ v,e,n\ \bullet$
$(((\text{LU0}(v) \wedge \text{Push\#}(v,n) \wedge \text{Coin}(v,n) \wedge \text{Ends}(e,v)) \rightarrow \neg\ \text{Unlock}(e))$

If LU0 holds and Push# equals Coin#, the machine must not unlock the turnstile. Push events are impossible while LU0 holds, so the turnstile will eventually be unlocked only after another Coin event, as we might expect.

The liveness property is that the machine must perform a Lock event in certain states. The relevant states are defined by a predicate on intervals:

(DEF2) $\text{ReqLock}(v) \triangleq$
$(\text{LU1}(v) \wedge \exists\ n\ \bullet\ (\text{Push\#}(v,n) \wedge \text{Coin\#}(v,n)))$

The liveness property is that if ReqLock holds — that is, if the turnstile is unlocked and Push# equals Coin# — the machine must perform a Lock event in time to prevent a further Push (and thus a further Enter) event.

If we were to adopt the reactive systems hypothesis (the commonly adopted assumption that the machine will react to each stimulus from the environment before the next stimulus occurs), we would say simply that in state ReqLock the machine must perform a Lock event. But there are important real-time considerations here. We will return to this point — and state the liveness property exactly — in the next section.

The refinement of OPT2 is somewhat analogous to that of OPT1. The machine must ensure that the indicative safety property IND2 does not prevent Push events when there is a coin in credit. Again there is both a safety property and a liveness property. The safety property is:

(OPT5) $\forall\ v,e,m,n\ \bullet$
$((\text{LU1}(v) \wedge \text{Push\#}(v,m) \wedge \text{Coin\#}(v,n) \wedge$
$(m < n) \wedge \text{Ends}(e,v))$
$\rightarrow \neg\ \text{Lock}(e))$

If LU1 holds — the turnstile is unlocked — and there is a coin in credit, the machine must not lock the turnstile. The condition in which Lock events are forbidden will cease to be true when an excess of subsequent Push events over Coin events uses up the credit.

The liveness property is that the machine must perform an Unlock event in certain states. The relevant states are defined by a predicate on intervals:

(DEF3) $\text{ReqUnlock}(v) \triangleq$
$(\text{LU0}(v) \wedge \exists\ m,n\ \bullet$
$(\text{Push\#}(v,m) \wedge \text{Coin\#}(v,n) \wedge (m < n)))$

The liveness property is that if ReqUnlock holds — that is, if the turnstile is locked and there is a coin in credit — the machine must perform an Unlock event. Again, there is a real-time consideration, and we will state the liveness property exactly in the next section.

8 Real Time

We return now to the point deferred above in discussing the statement of the liveness properties in the refinements of OPT1 and OPT2.

The refinement of OPT2 must be more than a specification that in state ReqUnlock the machine will *eventually* perform an Unlock. It must ensure that that state, in which some visitor has paid but has not yet been enabled to Push, does not persist unreasonably long. We may express this quite directly in an optative description:

(OPT6) Duration[ReqUnlock] < 250

The visitor must be enabled to Push within 250 msecs of paying. OPT6 specifies that the machine must terminate a ReqUnlock state within the time limit. Its only means of doing so, by virtue of DEF3, is to exit from state LU0. By DEF1 and OPT3, that means it must execute an Unlock event. (The environment can not terminate a ReqUnlock state: it can not initiate an Unlock event to terminate LU0; and while LU0 holds it can not initiate a Push.)

The refinement of OPT1 discussed above led us to the specification that in state ReqLock the machine must perform a Lock event soon enough to prevent another Push event: that is the whole point of the requirement. Clearly, we can satisfy this requirement only if the environment guarantees a sufficient real-time delay for the machine to respond. Further investigation of the turnstile reveals that hydraulic damping guarantees delays of at least 750 msecs between a Push and a following Enter, and at least 10 msecs between an Enter and a following Push:

(IND4) Duration[PE0] ≥ 10 $\wedge$ Duration[PE1] ≥ 750

At least 760 msecs will therefore intervene between successive Push events, and the necessary refinement of the liveness part of requirement OPT1 is:

(OPT7) Duration[ReqLock] < 760

The freedom to delay the Lock event is important for smooth and efficient working of the turnstile. The machine may wait in state ReqLock, within the limit of 760 msecs, in order to increase the probability that another Coin event will intervene to cause an exit from the ReqLock state and so make the Lock unnecessary. A machine that does so is preferable to a machine that performs the Lock event immediately.

The preferable machine is not readily specifiable under the reactive system hypothesis. The virtue of the reactive systems hypothesis is that we can avoid real-time considerations in writing requirements and specifications. Everything that the machine must do to satisfy the requirement is assumed to be done fast enough. Or, equivalently, everything that the environment might do to frustrate the requirement is assumed to happen too slowly to do so. The disadvantage is that it becomes very inconvenient to specify that the machine should wait in case another stimulus arrives to countermand the effect of a previous stimulus. Our technique of defining states avoids this disadvantage, and allows us to deal reasonably directly with real-time considerations.

9 Satisfaction of the Requirement

In the entailment

$$S, E \vdash R$$

mentioned in Section 7 above, the requirement R is OPT1 and OPT2. The specification S is OPT3, OPT4, OPT5, OPT6, and OPT7. The environment properties E are IND1, IND2 and IND4 (IND3 being deduced from IND1). Assuming the definitions DEF1, DEF2, DEF3, the entailment is therefore

$$IND1, IND2, IND4, OPT3, OPT4, OPT5, OPT6, OPT7 \vdash OPT1 \wedge OPT2$$

To prove satisfaction of the requirement is to prove this entailment. OPT1a will be a lemma in this proof.

The derivation steps presented in this paper are, of course, somewhat too informal to constitute a proof of satisfaction. Most notably, some subtleties in the relationship of Push and Enter events were ignored in the refinement of OPT2. The requirement 'The machine will not prevent another Enter event' is satisfied by a specification in which the machine unlocks, or refrains from locking, the turnstile, thus enabling the visitor to perform a Push event, following which the visitor is automatically enabled to execute an Enter event.

10 Related Work

Many researchers in requirements engineering are interested in achieving a fuller understanding of the relationship between requirements and specifications. In this section we compare our ideas to those of four closely related papers.

[Feather 91] is agent-oriented; it envisions systems that are mixtures of human, software, and hardware agents, taking responsibility for various goals and subgoals. We recognize only two agents — environment and machine — and their multiple agents are clearly a decomposition of our two. We emphasise a fixed environment that must be fully accommodated by the machine, while Feather, Fickas, and Helm emphasise a environment that is "designed" along with the machine. Both are legitimate viewpoints for requirements engineering (and we certainly don't intend to limit ourselves to only one of them), but they are irrelevant to this comparison. We are concerned here chiefly with the technical issue of how requirements and specifications differ, and how are they related. For a precise comparison, it is necessary to factor out this difference in viewpoint.

They mention four key transformations by which agent specifications are obtained from requirements or goals:

(a) *Brinksmanship*: identify actions that could cause a constraint to be violated, add components to exert control over these actions, assign some agent to be the controller.

(b) *Spatial split*: split goal responsibility into pieces assigned to separate agents.

(c) *Indirect access*: agent B needs some information it does not have direct access to; agent A gets it and communicates it to B.

(d) *Responsibility accumulation*: assigning multiple responsibilities to the same agent.

Spatial split and responsibility accumulation concern a level of detail that is lower than our scheme — decomposition of the environment and machine agents. We would expect such separations of concerns to be reflected in separate descriptions; but these separations are not needed to explain the difference between requirements and specifications.

Brinksmanship is reminiscent of our rule concerning requirements that are not specifications because they constrain environment-controlled phenomena. But brinksmanship concerns only safety properties, while our rule includes liveness.

Indirect access is reminiscent of our rule concerning requirements that are not specifications because they use unshared phenomena; but it requires the use of an active operational agent to maintain the relationship between the unshared and the shared phenomena. In our view this relationship must be described explicitly, but need not be attributed to an active agent.

In summary, we find our scheme simpler because it does not depend on decomposition of agents, and does not introduce them when not needed. It seems to be more comprehensive because it includes such things as liveness requirements on environment-controlled phenomena. It also seems to be more systematic because it is not an empirically discovered collection, but rather is based on the exhaustive classifications into shared and unshared, and environment-controlled and machine-controlled phenomena.

[Feather 94] extends the work reported in [Feather 91], concentrating on bringing a number of formal techniques to bear on the derivation of specifications from requirements. In particular, Feather exploits a finite differencing transformation, calculation of weakest preconditions, weakening of invariants, and the unfolding of invariants into guarded commands. Use of such formal techniques assists refinement by reformulation of previously stated requirements. It complements the exploitation of indicative environment properties that is a central feature of our approach.

Feather also discusses a form of our distinction between shared and unshared phenomena. Agents in the environment — in his example, railway trains — may be unable to evaluate a predicate that guards one of their actions. For example, a train does not 'know' whether there is another train in the next track segment.

The scheme of [Dubois 89] is based on bilaterally controlled actions, which we consider unrealistic. They are also prone to unnecessary semantic complications, such as the distinction between external and internal (hidden) nondeterminism in [Abadi 93]. Further, Dubois's scheme requires a cumbersome and nonstandard logic.

In many ways, Johnson's work on deriving specifications from requirements [Johnson 88] is the closest to ours philosophically. Johnson's transformation of "removing the perfect knowledge assumption" has exactly the same purpose as our rule about requirements that are not specifications because they use unshared phenomena. Also, his transformation of "defining capabilities" has exactly the same purpose as our rule about requirements that are not specifications because they constrain environment-controlled phenomena. (Incidentally, these transformations have roughly the same purpose as the "operationalization" goals *IsEvaluable* and *IsAchievable* in [Mostow 83]. But Mostow's work focuses on automated problem-solving, and thus assumes — if applied to requirements engineering — that the domain is as malleable as the machine.)

One difference is that [Johnson 88], like [Feather 91], is agent-oriented rather than description-oriented. Also, Johnson describes requirements as being *edited* until they become specifications. Our characterization of requirements and specifications as distinct optative descriptions, linked by the indicative descriptions that cause the specification to imply the requirement, is more general: it embraces other considerations such as the reuse of existing specifications.

We feel that we have added significantly to Johnson's notion of "defining capabilities" by explaining the precise circumstances under which agents have the wrong capabilities (an optative description constrains environment-controlled phenomena) and the precise remedy for the problem (there must be indicative descriptions linking machine-controlled phenomena to the relevant environment-controlled phenomena).

11 Conclusions

We have explained a distinction between *requirements* and *specifications.* Both are expressed in terms of environment phenomena. A requirement is expressed in terms of phenomena and relationships that are of direct interest to the system's customers and users, while a specification is restricted to implementable behaviour of a machine that can ensure satisfaction of the requirement. The gap between the two is bridged by reasoning based on environment properties that can be relied on independently of the machine's behaviour.

This view leads to an emphasis on careful expression of environment properties. We separate *indicative* from *optative* properties — those that can be relied on from those that the system must bring about. We separate *definition* from *assertion*, and *designated phenomena* from *defined terms.* We pay explicit attention to *control*, and express liveness properties in terms of *real time.*

We have illustrated our ideas with a simple control system example. We believe that other kinds of problem will demand application of the same ideas, albeit in different contexts and expressed in different languages. In some cases a structuring of the environment into *domains* will be necessary. Larger problems, of realistic complexity, will additionally demand a decomposition into simple problems, and a recombination of the resulting solutions.

Acknowledgement

Martin Feather read an earlier version of this paper and gave us many detailed and helpful comments.

References

[Abadi 93] Martin Abadi and Leslie Lamport; Composing specifications; ACM TOPLAS Volume 15 Number 1 pp73-132, January 1993.

[Dubois 89] Eric Dubois; A Logic of Action for Supporting Goal-Oriented Elaboration of Requirements; in Proc IWSSD-5; IEEE CS Press, 1989.

[Feather 87] M S Feather; Language Support for the Specification and Development of Composite Systems; ACM TOPLAS Volume 9 Number 2 pp198-234, April 1987.

[Feather 91] Martin S Feather, Stephen Fickas, and B Robert Helm; Composite System Design: the Good News and the Bad News; in Proc 6th RADC KBSE Conference; IEEE CS Press, 1992.

[Feather 94] Martin S Feather; Towards a Derivational Style of Distributed System Design — An Example; Automated Software Engineering Volume 1 Number 1 pages 31-60, March 1994.

[Jackson 92] Michael Jackson and Pamela Zave; Domain Descriptions; in Proc RE'93; IEEE CS Press, 1992.

[Jackson 94] M A Jackson; Software Development Method; in A Classical Mind: Essays in Honour of C A R Hoare; A W Roscoe ed; Prentice-Hall International 1994.

[Johnson 88] W Lewis Johnson; Deriving Specifications from Requirements; in Proc ICSE-10; IEEE CS Press, 1988.

[Morgan 90] Carroll Morgan; Programming from Specifications; Prentice-Hall International 1990.

[Mostow 83] Jack Mostow; A Problem-solver for Making Advice Operational; in Proc AAAI-83, pages 279-283; William Kaufmann Inc, 1983.

[Zave 93] Pamela Zave and Michael Jackson; Conjunction as Composition; ACM Transactions on Software Engineering and Methodology, Volume 2 Number 4 pages 379-411, October 1993.