This page intentionally left blank.

FOUR DARK CORNERS OF REQUIREMENTS ENGINEERING

Pamela Zave

AT&T Research
700 Mountain Avenue, Room 2B-413
Murray Hill, New Jersey 07974, USA
+1 908 582 3080
pamela@research.att.com

Michael Jackson

AT&T Research and MAJ Consulting Limited
101 Hamilton Terrace
London NW8 9QX, UK
+44 171 286 1814
jacksonma@attmail.att.com

16 July 1996

Abstract: Research in requirements engineering has produced an extensive body of knowledge, but there are four areas in which the foundation of the discipline seems weak or obscure. This paper shines some light in the "four dark corners," exposing problems and proposing solutions. We show that all descriptions involved in requirements engineering should be descriptions of the environment. We show that certain control information is necessary to sound requirements engineering, and we explain the close association between domain knowledge and refinement of requirements. Together these conclusions explain the precise nature of requirements, specifications, and domain knowledge, as well as the precise nature of the relationships among them. They establish minimum standards for what information should be represented in a requirements language. They also make it possible to determine exactly what it means for requirements engineering to be successfully completed.

FOUR DARK CORNERS OF REQUIREMENTS ENGINEERING

1. Introduction

There was a time when the epigram "requirements say what the system will do and not how it will do it" summarized all of requirements engineering. That time is long past. Research in requirements engineering has now produced a body of knowledge including terminology, methods, languages, tools, and issues acknowledged to be critical.

Despite this progress, we have identified four areas in which the foundation of the discipline seems weak or obscure. This paper attempts to shine some light in these dark corners, exposing the problems and proposing some solutions. We are led to these conclusions:

(1) All the terminology used in requirements engineering should be grounded in the reality of the environment for which a machine[1] is to be built.

(2) It is not necessary or desirable to describe (however abstractly) the machine to be built. Rather, the environment is described in two ways: as it would be without or in spite of the machine, and as we hope it will become because of the machine.

(3) Assuming that formal descriptions focus on actions, it is essential to identify which actions are controlled by the environment, which actions are controlled by the machine, and which actions of the environment are shared with the machine. All types of actions are relevant to requirements engineering, and might need to be described or constrained formally. If formal descriptions focus on states, then the same basic principles apply in a slightly different form.

---

[1]Throughout this paper we use "system" only to refer to a general artifact that might have both manual and automatic components, such as an "airline-reservation system." Whenever we are referring to the computer-based artifact that is the target of software development, we use the more precise term "machine."

(4) The primary role of domain knowledge in requirements engineering is in supporting refinement of requirements to implementable specifications. Correct specifications, in conjunction with appropriate domain knowledge, imply the satisfaction of the requirements.

The four areas of inquiry have not been chosen randomly. Together they explain the precise nature of requirements, specifications, and domain knowledge, as well as the precise nature of the relationships among them. They establish minimum standards for what information should be represented in a requirements language. They also make it possible to determine exactly what it means for requirements engineering to be successfully completed. The final section of the paper summarizes these recommendations and elaborates how they can be applied to the wide range of systems and needs served by requirements engineering.

The purpose of this paper is not to promote any particular notation, method, or tool, still less to propose new ones. Its purposes are to clarify the nature of requirements engineering, to show the importance of certain information that is often absent or implicit, and to solve some persistent foundational problems. These results can be used to evaluate and improve current and future approaches to requirements engineering.

This work is limited to the formal aspects of requirements engineering. It does not cover the sociological aspects of requirement engineering, such as how to communicate with customers or resolve their conflicts.

There are many additional motivations and ramifications not covered in this paper [Jackson 95]. Although this paper uses many brief and partial examples, the same ideas have also been used to work through a complete case study [Jackson & Zave 95]. In this paper the ideas are more fully related to ongoing research in requirements engineering.

2. *Grounding formal representations in reality*

Any formal representation uses primitive terms with no inherent formal meaning. In requirements engineering, the meaning of these terms lies in the real world, and the validity of any formal assertion relies on it. Here is a hypothetical dialogue between two requirements engineers working on an information system for a university:

> Able: Two important basic types are *student* and *course*. There is also a binary relation *enrolled*. If types and relations are formalized as predicates, then

$$\forall s\ \forall c\ (enrolled(s,c) \Rightarrow student(s) \wedge course(c)).$$

Baker: Do only students enroll in courses? I don't think that's true.

Able: But that's what I mean by *student*!

If a student is a person who is enrolled in a course at this university, then Able's assertion is nearly vacuous. If a student is a person who has matriculated at the university and not yet graduated or withdrawn, then Able's assertion is a strong constraint on who can take courses at the university.

The only way to establish the meaning of a primitive term is to provide an informal explanation of it. This explanation must be clear and precise, it must be written down, and it must be maintained as an essential part of the requirements documentation. We call such an explanation a "designation."[2]

It is well known that pinning down the real-world meaning of a primitive term can be extremely difficult—it can be fraught with subtleties and ambiguities. Everyone with experience in requirements engineering has, in some situation, used a term freely for a long time before noticing that its meaning is unclear. Yet very few requirements methods address this problem by enforcing the presence of a designation for every primitive term.

A notable exception is the A-7 method. In an early version [Heninger 80, Parnas & Clements 86], for every input or output data item a form must be filled out, and the form requires a "description" entry which is a designation of the data item. A later version [Parnas & Madey 95, van Schouwen et al. 92] says that "the association between the physical quantities of interest and their mathematical representations must be carefully defined," and claims to have demonstrated that "the use of prose in specifications—a major source of ambiguity and imprecision—can be limited to the environmental-quantity descriptions only."

Another good example is the textual convention for Z recommended by Wordsworth [Wordsworth 1992]. Each schema has text preceding it to explain (designate) the variables declared in it, and text following it to explain (paraphrase) the assertions made about those variables.

The remainder of this section points out additional benefits of the use of designations. The examples show how designations provide new perspectives on old problems, and answer some not-so-obvious questions.

The difference between assertion and definition: The difference between assertion and definition is well

[2] In logic, a designation is called an "interpretation." We avoid the word "interpretation" because it is highly overloaded in computing. It also carries the unfortunate connotation that the logic is real and important, while any correspondence it might have to the world is casual and incidental.

known [Brachman et al. 83]. However, the concept of a designation provides a slightly new perspective on it.

The assertion:

$$\forall s\ (student(s) \Leftrightarrow \exists c\ enrolled(s,c))$$

presupposes a designation of *student* (perhaps as a person who has matriculated at the university and not yet graduated or withdrawn) and a designation of *enrolled*. The assertion constrains the real world by expressing a relationship between two real-world phenomena. It might be true or false, and should be validated before it is used. The definition:

$$student(s) \stackrel{def}{=} \exists c\ enrolled(s,c)$$

precludes a designation of *student*. It extends the formal vocabulary without constraining the real world in any way. It might be useless or misleading, but it cannot be false.

In some logical systems the assertion and the definition would be considered equivalent, and in most logical systems the same inferences could be made from both of them. They are different because of designations; if designations are ignored then there is no real difference between them.

Goal regress: Requirements engineering is about satisfying goals [Dardenne et al. 93]. But goals by themselves do not make a good starting point for requirements engineering. To see why, consider a project to develop a computer-controlled turnstile guarding the entrance to a zoo [Jackson & Zave 95].

If the engineers are told that the goal of the system is to deny entrance to people who haven't paid the admission charge, they may decide that the goal has been stated too narrowly. Isn't the real goal to ensure the profitability of the zoo? Should they consider other ways of improving profits, such as cutting costs? What if there is more money to be made by closing the zoo and selling the land? And what is the goal of profit? If the goal of profit is the happiness of the zoo owner, would religion or devotion to family be more effective? Obviously there is something wrong here. Almost every goal is a subgoal with some higher purpose. Both engineering and religion are concerned with goal satisfaction; what distinguishes them is their subject matter.

The engineers should be told, in addition to the goal, that the subject matter is the zoo entrance. This information should take the form of designations of phenomena observable at the zoo entrance, such as visitors, coins, and the action of entering the zoo. These designations circumscribe the area in which alternative goal-satisfaction strategies can be considered, at the same time that they provide the basis for formal representation of

requirements.

Identity: The concept of identity is known to be an important aspect of formal representation schemes [Khoshafian & Copeland 90]. But formal notions of identity and distinguishability are useless if they do not correspond reliably to the real world. Designations help pin down their real-world meanings.

Consider, for example, a specification of a bounded buffer in temporal logic [Wing 90]. This particular specification includes the constraint that for each message *m* currently placed on the input channel and for each previously placed message *m'*, *m* and *m'* are not identical. Wing's discussion says, "This property is . . . an assumption of the environment. This assumption is essential to the validity of the specification."

Since the term *message* is ambiguous, it is not clear what this assumption means or how to satisfy it. If *message* refers to the message content, then the same bit pattern cannot be transmitted twice. This interpretation seems stronger than intended or necessary, but without a designation there is no way to rule it out.

A good designation of the term *message* might include information about message creation and usage:

> A process calls the procedure `send-msg` for the purpose of transmitting data to some other process. A message is the result of a particular call to `send-msg`, and contains the data to be transmitted.

Now it is clear that each call to `send-msg` creates a distinct message, regardless of content. Furthermore, if a call to `send-msg` includes placing the message on the input channel of the bounded buffer, then the buffer's messages *m* and *m'* are guaranteed to be distinct because each is inevitably the result of a distinct call.

3. Implementation bias

Requirements are supposed to describe what the desired machine does, not how it does it. More precisely, requirements are supposed to describe what is observable at the interface between the environment and the machine, and nothing else about the machine. To say anything else about the machine is regarded as implementation bias.

State is used frequently in specifications and specification languages. Very few machines are simple or specialized enough to be specified entirely without state. This causes a problem: specifying a machine in terms of its states appears to introduce serious implementation bias, because its states are internal and not directly observable at the interface between the machine and its environment.

The next subsection reviews some previously proposed solutions to this problem. Then we propose a

completely different way of looking at requirements specification that eliminates the problem altogether. A third subsection expands on the differences and their consequences.

3.1. Previous approaches to the problem

Property-oriented specification techniques [Wing 90] seek to avoid the use of explicit internal state by stating the minimal constraints on the machine state in the form of assertions. Algebraic and axiomatic techniques are the major techniques in this category. Despite eloquent justifications of this approach [Liskov & Zilles 75], many people consider it difficult to use, and model-oriented techniques are gaining in popularity.

Model-oriented specification techniques represent the system state as abstractly as possible, using mathematical structures such as sets, relations, and tuples. Thus model-oriented specifications run a greater risk of implementation bias. In attempting to escape this dilemma, Jones defines implementation bias as follows [Jones 90]:

> A model-oriented specification is based on an underlying set of states. The model is biased (with respect to a given set of operations) if there exist different elements of the set of states which cannot be distinguished by any sequence of operations.

For example, a telephone switch may allow a subscriber to enter telephone numbers from which he is protected—calls to the subscriber from those numbers will not go through. If the telephone numbers blocked by a subscriber appear only in this requirement, then representing them as a set would be unbiased by Jones's definition. Representing them as a list would be biased by the same definition, because a list would contain ordering information that could not be observed by using any machine function.

Jones's recommendation is to use only states that are unbiased by his definition. But trouble arises because the definition is *with respect to a given set of operations*. For real machines, the requirements change and expand frequently. It is possible that a new function will be required of the telephone switch, allowing the subscriber to play back his blocked numbers in the order entered. After this change, a set representation will be inadequate and a list will be necessary. Thus Jones's recommendation brings freedom from implementation bias into conflict with maintainability.

Lamport's solution to the problem of implementation bias focuses on the exact relationship between the specification and an implementation that satisfies it [Lamport 89]. In his method, the machine has an *unspecified*

state space S. Each state component c_i used in the specification is actually a function $c_i: S \rightarrow V_i$ from the state space to a set of values V_i.

The formal meaning of any specification is a temporal-logic formula in which the state components c_i are existentially quantified variables. The specification is satisfied by any implemented machine with any state space whatsoever, provided that there exist functions c_i on the state space that satisfy the temporal-logic formula. The specification does not bias the implementation because a "specification state" $(c_1(s), c_2(s), \ldots, c_n(s))$ can look completely different from an "implementation state" $s \in S$.

Also note that, since the c_i can be many-to-one functions, many implementation states can map to the same specification state. This accommodates the realistic situation in which a single specification action is implemented by dozens or even thousands of separate implementation actions.

3.2. Requirements exist only in the environment

A portion of the real world becomes the "environment" of a development project because its current behavior is unsatisfactory in some way. The developers propose to build a computer-based machine and connect it to the existing environment in such a way that the behavior of the environment becomes satisfactory. Although we are accustomed to think of machine inputs and outputs, it is important to realize that those inputs and outputs are phenomena *in the environment*. If they were not part of the environment, then they could not possibly connect the machine to the environment, or affect the behavior of the environment.

From this perspective, *all* statements made in the course of requirements engineering are statements about the environment. The primary distinction necessary for requirements engineering is captured by two grammatical moods. Statements in the "indicative" mood describe the environment as it is in the absence of the machine or regardless of the actions of the machine; these statements are often called "assumptions" or "domain knowledge." Statements in the "optative" mood describe the environment as we would like it to be, and as we hope it will be when the machine is connected to the environment. Optative statements are commonly called "requirements." The ability to describe the environment in the optative mood makes it unnecessary to describe the machine.

This perspective avoids the problem of implementation bias because *no statements are made about the proposed machine*. The specification state is not a description of the state of the machine, but rather a description of

the state of the environment. The specification might be compromised by poor designations or invalid indicative statements, but it cannot overconstrain the implementation.

Some of our examples are given in Gries and Schneider's equational logic [Gries & Schneider 93]. Time and temporal properties are encoded in logic as follows [Zave & Jackson 93].

We assume that actions are atomic and sequential. They are represented by distinct individuals satisfying the predicate *action(a)*. The binary predicate *earlier*(a_1,a_2) establishes a nondense total order (with an initial member) on actions.

Because the action sequence is nondense, we can postulate a unique *pause* between each adjacent pair of actions during which the state has changed in response to the first action, cannot change again until the next action, and can be observed. These pauses are formalized as individuals satisfying the predicate *pause(p)*. *begins(a,p)* means that action *a* begins pause *p*, i.e., precedes it immediately in the temporal sequence. *ends(a,p)* means that action *a* ends pause *p*, i.e., succeeds it immediately in the temporal sequence.

Note that a specific *earlier* predicate, a specific *begins* predicate, and a specific *ends* predicate combine to represent a single behavior or trace. In other words, each model that satisfies the specification represents one behavior that satisfies the specification. Because most specifications are satisfied by many behaviors, they are also satisfied by many different models.

As a simple example of indicative and optative properties, consider a single-customer banking environment with designations of three action types: *deposit(a,m)* means that *a* is an action in which amount *m* is deposited, *withdrawal-request(a,m)* means that *a* is an action in which a withdrawal of amount *m* is requested, and *withdrawal-payout(a,m)* means that *a* is an action in which amount *m* is paid out as a withdrawal. Deposits and withdrawal requests are performed by the customer, while withdrawal payouts are performed by the machine (control of actions will be discussed in detail in the next section).

The current account balance is part of the state of the environment—it is a legal debt owed by the bank to the customer. Formally, *balance(b,p)* means that during pause *p* the balance is amount *b*. An indicative property of this environment is that, at any time, the balance is equal to the sum of the amounts of all the previous deposits, minus the sum of the amounts of all the previous withdrawal payouts:

$$(\forall\ b,p \mid : balance(b,p) \equiv$$
$$(b = (+m \mid (\exists a \mid : deposit(a,m) \wedge earlier(a,p)) : m) - (+m \mid (\exists a \mid : withdrawal\text{-}payout(a,m) \wedge earlier(a,p)) : m))\ )$$

This property uses the uniform quantification of equational logic, in which

$$(*i \mid R(i) : P(i))$$

denotes the accumulation of values $P(i)$, using operator $*$, over all values i for which predicate $R(i)$ holds. The "accumulation" of $\forall$ is a conjunction, the "accumulation" of $\exists$ is a disjunction, and the "accumulation" of + is an arithmetic sum. We have also extended *earlier* to cover pauses as well as actions.

An important optative property of this environment is that a withdrawal request leads to a withdrawal payout, provided that the requested amount does not exceed the current balance:

$(\forall\ a,m,p,b \mid withdrawal\text{-}request(a,m) \wedge ends(a,p) \wedge balance(b,p) \wedge b \geq m$:
$(\exists a' \mid : withdrawal\text{-}payout(a',m) \wedge earlier(a,a')))$

A requirements method should provide for specification of both indicative and optative properties, and make a clear distinction between them. For example, the recent version of the A-7 method [Parnas & Madey 95, van Schouwen et al. 92] features two relations *NAT* and *REQ*. *NAT* documents the constraints on the values of environmental quantities imposed by nature and previously installed systems; it is in the indicative mood. *REQ* documents further constraints on the environmental quantities to be imposed by the machine; it is in the optative mood.

3.3. Consequences

The idea that a requirements specification should include a model of the environment has been familiar since the early 80s [Balzer & Goldman 79, Jackson 78, Jackson 83, Lehman 80, Zave 82]. To the best of our knowledge, we are the first to propose that requirements should contain *nothing but* information about the environment.

The proposed shift in perspective might not seem to make much difference. For example, we have mentioned two properties of the banking system: the balance is the sum of the deposits minus the sum of the withdrawal payouts, and a withdrawal request leads to a withdrawal payout if the balance is large enough. We have said that the first property is indicative, the second is optative, and the balance is part of the environment state. But the formal representations of these properties would fit equally well into a more conventional setting in which the balance is considered to be machine state, and both properties describe requirements on the machine.

One consequence of the difference concerns the popular subject of "domain knowledge" [Iscoe et al. 91, Jarke et al. 92]. If specification state is considered to be machine state, then the problem of implementation bias forces the specification to be minimal—there must be nothing that is not necessary to carry out the currently proposed machine

functions. When specification state is understood to be environment state, however, then we are free to collect and record interesting information about the environment even before we are sure it will be needed. This freedom is clearly necessary for collecting libraries of reusable information about important portions of the real world. Given the thin spread of domain knowledge [Curtis et al. 88], any true and relevant indicative statement is a contribution.

Another consequence of the difference concerns designations. Obviously, an implemented machine often maintains internal state that mimics the current environment state. The imitation is usually imperfect, however, because there may be delay, errors, and factors in the real world that the machine does not know about.

For example, an inventory-control machine for a warehouse gets reports of *receive* and *ship* actions, each having a *bin* and *quantity* attribute. From these reports the machine maintains an internal model of the contents of the warehouse. However, due to theft, breakage, mishandling, update delays, clerical errors, and other misfortunes, it is extremely likely that the quantity actually present in a warehouse bin at a particular time is not exactly the same as the quantity indicated by the machine's model.

Section 5 discusses how errors and unknown factors can be managed (Bubenko has treated the subject of update delays [Bubenko 83]). In the meantime, we can point out that components of the machine state cannot be designated. Designations refer to the real world, and the machine state may have no direct correspondence to the real world.

In practice, effective specification usually requires many auxiliary terms that would be awkward to designate, and that may correspond more closely to the eventual machine state than designated terms. These can always be introduced and grounded in reality by defining them using designated terms. For example, if needed for the requirements of the inventory-control machine, the *expected-quantity-in-bin* can be defined using the designated terms *receive(a,s,b)*, meaning that a is an action in which quantity s for bin b is received by the warehouse, and *ship(a,s,b)*, meaning that a is an action in which quantity s from bin b is shipped from the warehouse:

$$\textit{expected-quantity-in-bin}(q,p,b) \stackrel{def}{=}$$
$$(q = (+s \mid (\exists a \mid : \textit{receive}(a,s,b) \wedge \textit{earlier}(a,p)) : s) - (+s \mid (\exists a \mid : \textit{ship}(a,s,b) \wedge \textit{earlier}(a,p)) : s))$$

4. Control of actions

In requirements engineering we are always concerned with two actors or agents: the environment and the

machine. In any situation with two or more agents, it is always of primary importance to understand issues of control. If a requirements language or method is deficient in representation of actions or control—as many are—then it is necessary to augment it or compose it with other formal languages better suited to the job.

The first subsection presents two expressive capabilities that are needed for requirements engineering, and shows how they are used. The second subsection comments on their presence or absence in several well-known requirements languages or methods; their absence causes a variety of difficulties.

4.1. Two necessary expressive capabilities

A sufficiently expressive requirements language must provide for the declaration of a finite collection of "action types." These action types partition all possible actions.

Each action type is either "environment-controlled" or "machine-controlled," indicating which agent controls, performs, or takes responsibility for actions of that type.[3] If actions of a type are sometimes environment-controlled and sometimes machine-controlled, then the type can be divided into two subtypes.

Each action type is also either "shared" or "unshared." If it is shared, then it is part of the real world that is shared between (belonging to, observable by) both the machine and environment. If it is unshared, then it is private to the environment and unobservable by the machine.

Obviously an action cannot be both machine-controlled and unshared. This leaves three possible combined action categories: environment-controlled/unshared, environment-controlled/shared, and machine-controlled/shared. A requirements language must be able to express which category each action type belongs to. The importance of this information should be self-evident. The implemented machine must perform the machine-controlled actions and not the environment-controlled ones. The implemented machine can observe and make use of its knowledge of the shared actions, while the unshared actions are beyond its reach.

The second necessary expressive capability is the full ability to state constraints on actions in all three categories. The requirements engineer should be able to make assertions about all actions in both the indicative and

[3]The term "machine-controlled" may seem to indicate a departure from the principle that requirements say nothing about the machine, but the lapse is only terminological. We would prefer to use "active" (actively controlled by the environment) and "passive" (passively done to the environment), where a passive action must be machine-controlled because it is not performed by the environment, and there is no other relevant agent to perform it. The "active/passive" terminology is properly centered on the environment, but unfortunately the meaning of these words is much harder to remember.

optative moods. If a kind of property can be expressed about actions in one category, then it should be possible to state the same property of actions in other categories.

To show why the second expressive capability is necessary, we wish to give examples of almost all the possibilities suggested above. In some formalisms, however, it can be difficult to separate different kinds of properties, and difficult to determine exactly which actions are constrained by a property. So, for the purpose of giving illustrative examples, we shall use a formal language in which both of these are straightforward.

Buchi automata are equivalent in expressive power to linear-time temporal logic. In linear-time temporal logic any property can be expressed as a combination of safety and liveness properties [Alpern & Schneider 87]. Safety properties can be paraphrased as "something bad must not happen" and liveness properties can be paraphrased as "something good must happen."

Alpern and Schneider show that the structure of a Buchi automaton reveals whether it expresses a "pure" safety property, a "pure" liveness property, or a combination of the two. A nontrivial safety property is specified by an automaton with no nonaccepting states, but with at least one state having no out-transition on at least one action type in its vocabulary. It asserts that in a state with missing out-transitions, actions of the missing types do not occur. In other words, it constrains certain action types by prohibiting their occurrence in certain states. A nontrivial liveness property is specified by an automaton with no missing out-transitions, but with at least one nonaccepting state. It asserts that, whenever a nonaccepting state is reached, actions must eventually occur to cause a transition to an accepting state. In other words, it constrains certain action types by requiring them to occur in certain states.

We shall now discuss two software-development projects and many of their relevant properties. Most of these properties are formalized as Buchi automata, which shows whether they are safety or liveness properties, and which action types they constrain. The properties are paraphrased briefly and classified in Table 1.

First consider a real-world environment consisting of a gate, a physical turnstile guarding the gate, a coin slot, and the customers who wish to enter the gate. The desired machine is a computer-based controller for the gate complex.

There are five important action types in the gate complex. A *pay* action occurs whenever a customer puts enough money in the coin slot to be allowed to enter the gate. A *push* action occurs whenever a customer pushes the

turnstile and thereby starts it moving. An *enter* action occurs whenever a customer pushes the turnstile through to its home position, thus entering the gate. These three action types are environment-controlled; pays and pushes are shared with the machine, while enters are not. In addition, there are machine-controlled *lock* and *unlock* actions. Five properties of interest are specified formally in Figure 1.

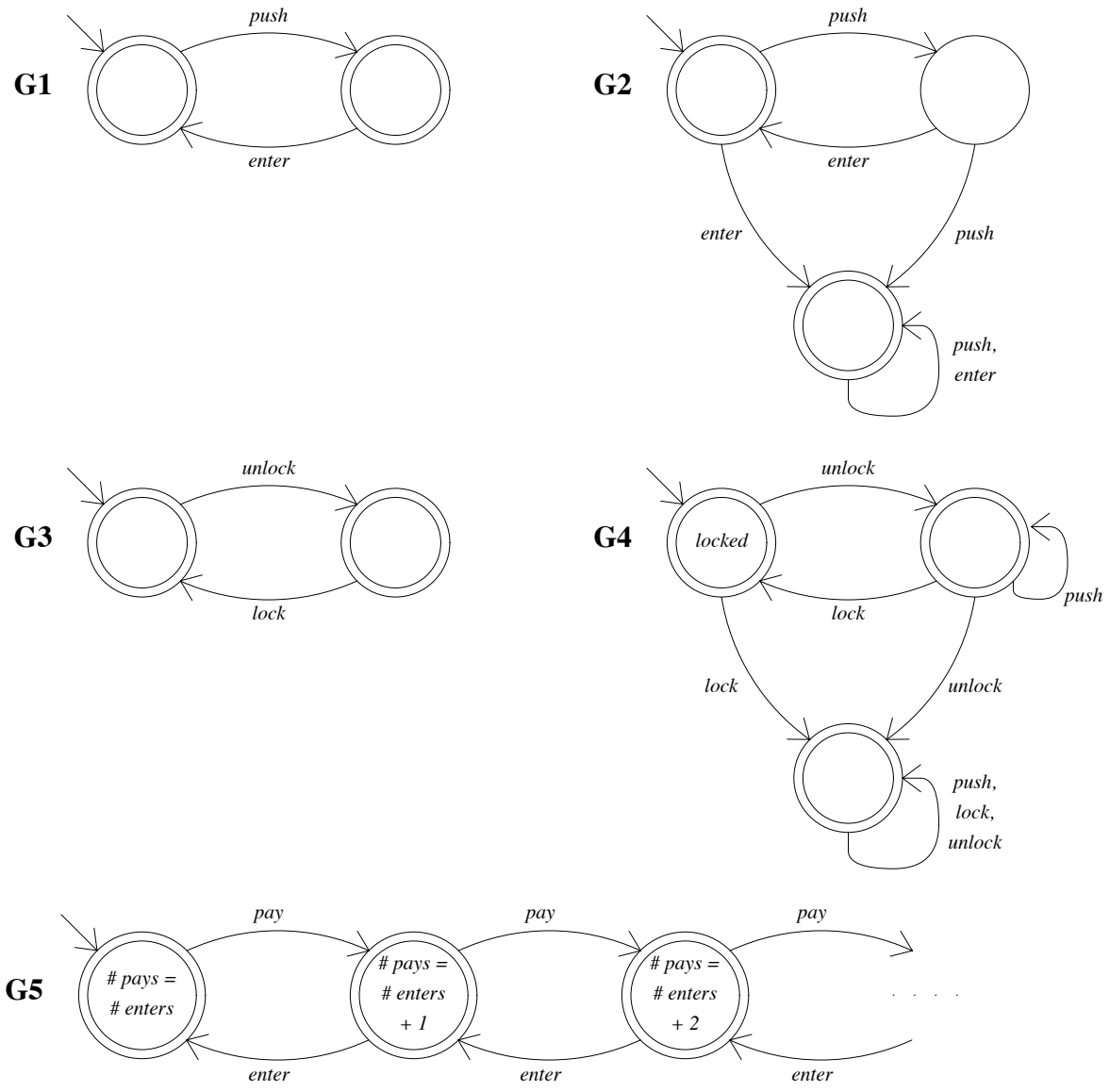

Figure 1. Some properties of the gate complex.

G1 is an indicative safety property, expressed in terms of the action types *push* and *enter*. It says that the turnstile is constructed so that pushes and enters alternate strictly, beginning with a push. Since both enters and

pushes are constrained, G1 appears in two different places in Table 1, one reflecting the fact that it constrains enters (environment-controlled, unshared actions) and one reflecting the fact that it constrains pushes (environment-controlled, shared actions).

To eliminate the possibility that a customer pushes the turnstile and then stops without entering the gate, the turnstile is also constructed with a hydraulic mechanism so that a push always leads to an enter. This indicative property (G2) is a liveness constraint on *enter* actions.[4]

The manufacturers of the turnstile hardware have designed it under the assumption that the controller will perform lock and unlock actions in a strictly alternating sequence. Because of this fault-intolerant design, an improperly controlled turnstile may break, enter an undocumented state, or enter a randomly chosen state. The only way to avoid problems is to ensure that (G3) lock and unlock actions alternate strictly, as the hardware expects them to do. This safety constraint on locks and unlocks is optative because these actions are just electronic signals emitted by the machine, and nothing in the environment imposes any order on them.

As long as the alternation of locks and unlocks is maintained, after a lock action, the turnstile is *locked*, and remains locked until the next unlock action. The turnstile is designed so that whenever it is locked, push actions cannot occur (G4). This is an indicative safety constraint on push actions, and it is the main property that enables the turnstile to do its job of guarding the gate.

The most important requirement on the gate complex is that people are forced to pay for the privilege of entering. Forcing the alternation of enters and pays would be too restrictive, however, because a teacher should be able to pay at once for a whole classful of students, and then lead them in without interruption. The correct property is that the accumulated number of enters never exceeds the accumulated number of pays. This is a safety constraint on enter actions (G5).

For another set of examples, consider a lift (environment) and its controller (machine). The lift environment has a number of designated action types and properties. Some of the properties are formalized as Buchi automata in Figure 2.

A *begin-group-waiting[floor,direction]*[5] action occurs when the first person approaches the lift at the

[4] By itself, G2 appears to be a liveness constraint on both enters and pushes. When G1 is also considered, however, we see that the only legal exit from the nonaccepting state in G2 is an enter.

[5] Lift action types are more complex. They have arguments or attributes, shown in brackets.

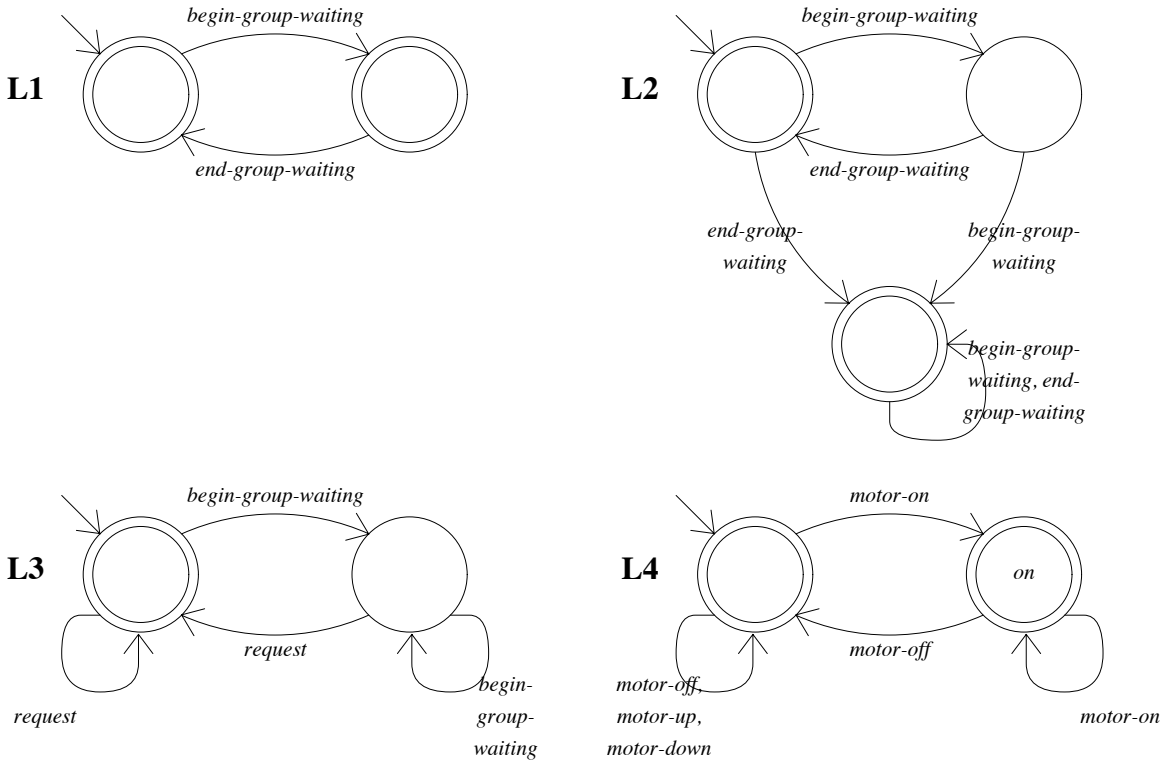

Figure 2. Some properties of the lift complex. Each diagram represents a set of distinct automata, one for each distinct value of *[floor,direction]*.

indicated floor, with the intention of traveling in the indicated direction. "First" means that no previously arrived person is waiting at that floor for travel in that direction. An *end-group-waiting[floor,direction]* action occurs when the members of a waiting group (all at the same floor, intending to travel in the same direction) enter the lift. Both these actions are environment-controlled and unshared. It is easy to see that, by their nature, *begin-group-waiting* and *end-group-waiting* must alternate (L1).

The primary lift requirement is that (L2) each *begin-group-waiting* action leads to a corresponding *end-group-waiting* action. Note that L1 and L2 are syntactically similar to G1 and G2, but G2 is indicative while L2 is optative. G2 is enforced by the turnstile hardware, while L2 can only be enforced by an effective lift controller.

A *request[floor,direction]* action occurs when a waiting person pushes the indicated request button. Because we cannot expect the lift to serve a group without knowing that it is waiting, we must assume about the environment that (L3) a *begin-group-waiting* action leads to a corresponding *request* action. This is an indicative liveness

constraint on requests. Note that requests and *begin-group-waiting* actions need not alternate. For example, an impatient person might push the button more than once.

Among the action types controlled by the machine are *motor-on, motor-off, motor-up, motor-down, request-light-on[floor,direction],* and *request-light-off[floor,direction]*, where the last two control up/down indicator lights at each floor. The motor may break if the controller attempts to change its direction while it is running, in which case there must be an optative safety constraint that (L4) motor-ups and motor-downs do not occur while the motor is on. This is similar in purpose to G3.

The remaining lift properties are not worth formalizing as Buchi automata, either because they are too repetitive or too complex. The previous examples should be sufficient to show what phrases such as "do not occur" and "leads to" mean in the brief paraphrases of these properties found in Table 1.

Note from L4 that motor-ons and motor-offs need not alternate—extra commands are simply ignored by the motor. In a totally different situation, a motor-up action could actually be designated as an action of changing the motor direction from down to up (and similarly, from up to down for the motor-down action). In this situation there is an indicative constraint that (L5) motor-ups and motor-downs alternate. L5 is indicative because, given the real-world meanings of motor-up and motor-down, no other behavior is possible—you cannot change a direction from down to up when the direction is already up! Compare this to G3, which is optative because there is nothing in the environment preventing two consecutive *lock* signals (it is just that we will be sorry if the machine performs two consecutive locks and thus breaks the turnstile).

One of the simpler lift requirements is that (L6) a request action leads to a corresponding request-light-on action.

An *arrive[floor]* action occurs when the lift arrives at the indicated floor (arrival at a floor does not necessarily mean that the lift stops there). Arrive actions are shared with the machine by means of sensors. For efficient lift scheduling, there might be a requirement that (L7) foolish arrive actions do not occur, where "foolish" means that they do not satisfy or move toward satisfying any outstanding request.

Further properties of the gate complex (G6,G7) and lift environment (L8) will be added in the next section. By the time we have finished with these examples, all of the spaces in Table 1 will have sample entries except the space for indicative liveness constraints on machine-controlled actions. Although we have no examples of these

particular constraints and believe they do not occur, the general point of the table is established without them.

Any specification method will provide for optative constraints on machine-controlled actions (the lower-right corner of Figure 1) because they describe what the machine must do. But optative constraints on environment-controlled actions (the upper-right corner of Table 1) are also important because many of the true requirements reside there, and are enforced only indirectly by machine actions. Indicative properties (the left half of Table 1) are often the intermediaries through which machine actions enforce optative constraints in the environment. These topics are explored further in Section 5.

	indicative statement (assumption, domain knowledge)		optative statement (requirement)	
constrained action is environment-controlled, unshared (unobservable by machine)	(G2) A *push* leads to an *enter*.	(G1) *Pushes* and *enters* alternate. (L1) *Begin-group-waiting [floor,direction]* alternates with *end-group-waiting [floor,direction]*.	(G5) *Enters* do not outnumber *pays*.	(L2) A *begin-group-waiting [floor,direction]* action leads to an *end-group waiting[floor, direction]* action.
constrained action is environment-controlled, shared (jointly observable)	(L3) a *begin-group-waiting [floor,direction]* action leads to a *request[floor, direction]* action.	(G1) *Pushes* and *enters* alternate. (G4) When the turnstile is *locked, pushes* do not occur.	(L7) Foolish *arrive[floor]* actions do not occur.	(L8) Group waiting at a floor leads to an *arrive [floor]* action.
constrained action is machine-controlled, shared (jointly observable)		(L5) *Motor-ups* and *motor-downs* alternate.	(G3) *Locks* and *unlocks* alternate. (L4) When the motor is on, *motor-ups* and *motor-downs* do not occur. (G7) *Locks* and *unlocks* do not occur when inappropriate.	(L6) A *request [floor,direction]* action leads to a *request-light-on[floor, direction]* action. (G6) *Locks* and *unlocks* occur when appropriate.
	liveness	safety property		liveness

Table 1

4.2. Examples of control issues

A Z specification [Spivey 92] is organized in terms of operations that affect the specification state. There is

no formal notion of control. Some Z specifications include operations controlled by diverse agents, for example Spivey's real-time kernel [Spivey 90]. Since there is no way to say which agent controls which operation, from a formal perspective such specifications are as bizarre as Lamport's example of a queue that spontaneously initiates its own put and get actions [Lamport 89].

More commonly, users of Z assume that the specification state is the machine state and that the operations are the interface between the machine and its environment. Operations are always initiated by the environment, and have "input" and "output" variables for conveying information across the machine/environment interface.

The computed precondition of a Z operation gives the circumstances under which it can be invoked safely (the operation can be carried out and all state invariants can be preserved).[6] The trouble with the common usage of Z outlined above is that there is no mechanism for recording constraints on what the environment can and cannot do. In particular, there is no way of answering the all-important question, "What properties of the environment justify ignoring the possibility that an operation is initiated when its precondition is not satisfied?"

The Software Cost Reduction (SCR) method [Heitmeyer et al. 95a, Heitmeyer et al. 95b] emphasizes constraints on the machine, and has limited capabilities for constraining the environment.[7] These limitations can cause gaps or anomalies.

The problem is illustrated by an SCR specification of a real-time railroad crossing system [Atlee & Gannon 93]. The environment is described in terms of boolean variables such as *Train* (which is true when a train is close to the crossing) and *TrainXing* (which is true when the train is actually going through the crossing). Within some versions of the SCR method it is possible to say that these two variables cannot be true at the same time, but in no version is it possible to say that when *Train* becomes true, it stays true for at least 300 seconds.

As a result of this deficiency, Atlee and Gannon specify this crucial information *as a constraint on the machine*. The specified machine is in mode *BC* when *Train* is true, and in mode *Crossing* when *TrainXing* is true. The machine specification states that a mode transition from *BC* to *Crossing* occurs if and only if *TrainXing* becomes true and the machine has been in mode *BC* for at least 300 seconds.

This is a dangerous specification if taken literally. The *BC* and *Crossing* modes provide the gate-control

[6] Again, this is only the most common interpretation, and others are possible [Jackson 96].

[7] The recent version of the A-7 method [Parnas & Madey 95, van Schouwen et at. 92] and the SCR method have common roots in the early version of the A-7 method [Heninger 80, Parnas & Clements 86], but the two have now diverged and are quite different with respect to the issues discussed in this paper.

algorithm with its information on the location of the train. If a train is faster than expected, then these modes will become decoupled from the train location they are supposed to reflect.

In CSP [Hoare 85], the concept of "shared actions" conflates two concepts that we have kept separate: control and joint observability. Section 5 will show that these two concepts play different roles in refinement of requirements, so that there is practical value in distinguishing them.

Suppose that an environment and a machine are represented formally as two concurrent processes in CSP. Consider the problem of recovering from this representation which action types fall into which of the three categories. It is easy enough to recover which actions are jointly observable by the two processes ("shared" in our sense) and which are private to the environment process, because that is a syntactic property.

Recovering the control information is harder. For each jointly observable action type and each process, there are exactly two possibilities: the process sometimes refuses to participate in actions of that type, or the process never refuses to participate in actions of that type. If one process sometimes refuses an action type and the other process never refuses it, then actions of that type are controlled by the sometimes-refusing process. If both processes sometimes refuse an action type, then actions of that type are jointly controlled, which we believe is a misrepresentation in essentially all real situations. If both processes never refuse an action type, then it is impossible to tell which process controls it. (*Pay* actions in the turnstile system are like this—they are controlled by the environment, but in the environment as described they can occur at any time.)

Thus CSP is almost expressive enough in terms of control of actions, but it might be confusing. When CSP is used for requirements, engineers would do well to augment it with a more explicit statement of their intentions.

Naturally, there are many good examples as well as bad ones. Lamport's method [Lamport 89] has precise identification and flexible treatment of all shared actions, but no unshared actions. The recent A-7 method uses a state-oriented formalism; it has precise identification and flexible treatment of all shared state components, but no unshared state components. Agent formalisms [Dardenne et al. 93, Dubois et al. 93, Feather 87, Feather et al. 91, Jeremaes et al. 86, Johnson 88] have all the recommended expressive capabilities. Each agent has specific actions that it can perform. Typically some agents represent the environment while some represent the machine, and environment agents are not limited to those that interact directly with the machine. If the agent formalism is also a deontic logic [Jeremaes et al. 86], then permissions and obligations are like safety and liveness in our example—it

must be possible to assert them in both the indicative and optative moods.

5. *The role of domain knowledge*

A requirement is an optative property. Let R be the set of requirements for a software-development project, i.e., the set of optative properties whose satisfaction will fully satisfy the customer.

A specification is also an optative property, but one that must be implementable.[8] Let S be the set of specifications for a software-development project. S can be given to a software-development team, which can implement it successfully without recourse to any additional information. S must be such that an implementation of it, connected to the environment, ensures that all the properties of R are satisfied.

The gap between requirements and specifications has long been recognized. The process of bridging this gap is often referred to as refinement of requirements, by analogy with specification or program refinement. Specification refinement is concerned with removing the features of a specification that are not executable by the target implementation platform, and replacing them with features that are executable on that platform. Requirements refinement is concerned with identifying the aspects of a requirement that cannot be guaranteed or effected by a computer alone, and augmenting or replacing them until they are fully implementable.

It has also long been recognized that domain knowledge (or domain modeling, assumptions about the environment, etc.) should play an important role in requirements engineering, but there is uncertainty about what it is for. Because there is uncertainty about what it is for, there is also uncertainty about how much of it to gather [Iscoe et al. 91].

The primary role of domain knowledge is to bridge the gap between requirements and specifications. Requirements that are not specifications are always converted into specifications with the help of domain knowledge. Let K be the relevant domain knowledge, i.e., the set of relevant indicative properties. Then S and K together must be sufficient to guarantee that the requirements are satisfied:

$$S, K \vdash R$$

It is important to note that a precise characterization of the difference between specifications and requirements

[8] We are discussing implementability in principle. A specification that is implementable in principle may not be implementable in practice, for instance because all of the available computers are too slow.

depends on precise location of the interface between the machine and environment, i.e., on knowing what the shared actions are. Many papers in requirements engineering are concerned with the flexibility of the machine/environment boundary [Dardenne et al. 93, Dobson et al. 94, Feather 87, Feather et al. 91, Feather 94, Mostow 83, Swartout & Balzer 82]. Thus we are characterizing the possible results of their explorations, not the explorations themselves.

Some requirements are already implementable, so they can be copied directly from *R* to *S*. Requirements that are not implementable fail to be so for three general reasons. We shall discuss these reasons in the next three subsections, followed by a subsection of comparisons to related ideas.

5.1. Environment constraint

Some requirements are not directly implementable because the only way to satisfy them is to constrain an action that is controlled by the environment. As an example of a requirement with environment constraint, there is a requirement on the gate complex that (G5) enters do not outnumber pays. The Buchi automaton in Figure 1 makes it clear that this is a constraint on *enter* actions, which are controlled by the environment.

To satisfy this requirement, we need several pieces of domain knowledge. It is important that (G1) pushes and enters alternate, so that enters can be prevented by preventing pushes. It is important that (G4) whenever the turnstile is locked, pushes cannot occur. The effects of lock and unlock actions are also important, because they are used to put the turnstile into a locked state. Finally, we also need domain knowledge giving us a lower bound on the elapsed time between two consecutive pushes (because if pushes can follow each other arbitrarily closely, then the machine must be infinitely fast).

The resultant specification [Jackson & Zave 95] consists of properties constraining when *lock* and *unlock* actions must and must not occur (these properties are summarized as G6 and G7 in Table 1). These appropriate actions keep the turnstile locked at the right times, because of the indicative properties, to satisfy G5.

The lift requirement L2 must be refined to several other requirements, including (L8) group waiting at a floor leads to an *arrive[floor]* action. L8 is not a specification because arrive actions are environment-controlled. Satisfaction of this requirement relies on the domain knowledge that certain motor states will cause the lift to move, and on specification of machine actions that control the motor state appropriately.

To summarize, requirements that constrain the environment cannot be satisfied by a machine acting alone.

Such requirements are always satisfied by coordinating specifications with domain knowledge. A requirement with environment constraint should always be verified by a demonstration or proof that specified properties, conjoined with domain knowledge, guarantee the satisfaction of the requirement.

In this context, Buchi automata are an easy notation to understand because they make it clear which actions are constrained. Thus it is always possible to determine syntactically whether a requirement is a specification or not, and to refine it (if the environment allows) to a property that is obviously a specification. For example, in Figure 3 the requirement R constrains both *ec* (an environment-controlled action type) and *mc* (a machine-controlled action type). Provided that the domain knowledge K holds, then R can be satisfied by the specification S, which clearly constrains only the machine.

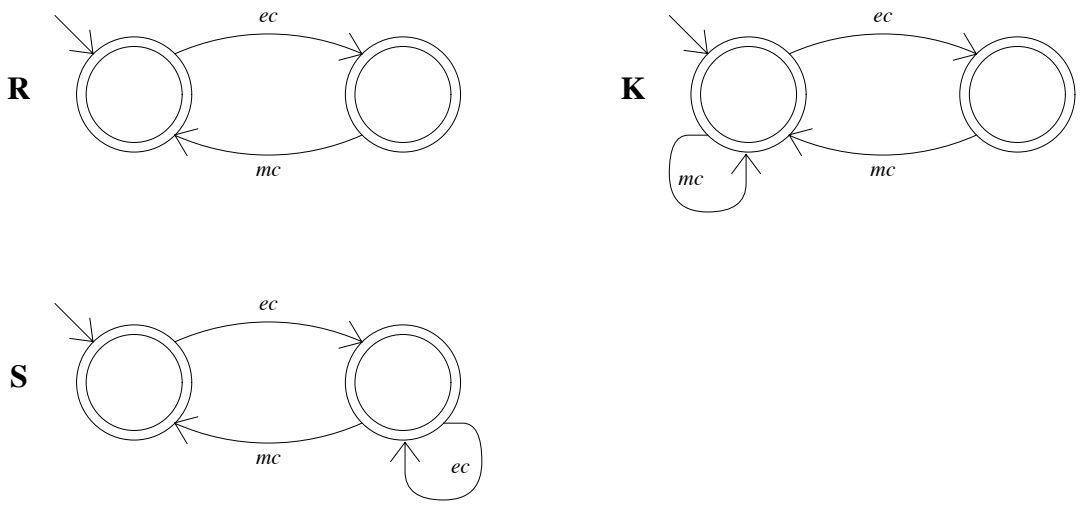

Figure 3. An easy example of refinement.

Unfortunately, in other formalisms it may not be so clear what is being constrained by a requirement. In these situations it may be more difficult to determine when the requirements phase is over, or to be sure that the specification alone contains everything that the implementors need.

5.2. Unshared information

Some requirements are not directly implementable because they are stated in terms of unshared phenomena. In the banking environment (Section 3.2), for example, there is a requirement (a withdrawal request leads to a

withdrawal payout, provided that the requested amount does not exceed the current balance) stated in terms of the account balance. The only shared phenomena in this example are the action types *deposit, withdrawal-request,* and *withdrawal-payout*. Since the *balance* state component (a legal debt) exists in the environment and is not shared with the machine, the machine cannot directly implement this requirement.

Requirements with unshared information are refined using domain knowledge relating unshared information to shared information. For example, the banking requirement can be refined using the indicative property also given in Section 3.2, which states an equivalence between the balance and an expression on *deposit* and *withdrawal-payout* actions. The following is a version of the requirement with the balance replaced by its equivalent action expression. It is implementable (it uses nothing but shared phenomena), and is therefore a specification.

$(\forall\ a,m,p \mid withdrawal\text{-}request(a,m) \wedge ends(a,p) \wedge m \leq$
$\quad ((+m \mid (\exists a \mid : deposit(a,m) \wedge earlier(a,p)) : m) - (+m \mid (\exists a \mid : withdrawal\text{-}payout(a,m) \wedge earlier(a,p)) : m)) :$
$(\exists a' \mid : withdrawal\text{-}payout(a',m) \wedge earlier(a,a')))$

In the presence of the indicative equivalence between the balance and the action expression, it can be proved that the above property satisfies the original requirement.

Another way to achieve the same effect would be to define a new term *expected-balance* as the sum of the amounts of all previous deposits, minus the sum of the amounts of all previous withdrawal payouts. If a requirement referring to *balance* is rewritten as referring to *expected-balance,* then the rewritten form is a specification. In the presence of the definition of *expected-balance,* it is directly implementable. In the presence of information about *balance,* it provably satisfies the original requirement.

As another example of unshared information, the requirement (L8) that the lift must arrive at a floor where people are waiting for it is not implementable. How can the machine know that people are waiting?

Request and arrive actions are shared with the machine. There is already domain knowledge that (L3) the beginning of group waiting leads to a request action. Other indicative statements must say that people continue to wait until the lift arrives at their floor going in their direction, at which point they enter the lift and end group waiting. From these assumptions about human behavior, and the occurrences of requests and arrivals, the lift controller can compute where people are presumed to be waiting for the lift.

In the warehouse example (Section 3.3), if requirements are stated in terms of *quantity-in-bin*, designated as the quantity actually present in the warehouse bin, then refinement is not possible. We know that *expected-*

quantity-in-bin is not the same as *quantity-in-bin*, and there are no shared phenomena from which the machine can compute the actual *quantity-in-bin*.

This problem must be solved by changing and elaborating the requirements. Probably the requirements will be stated in terms of *expected-quantity-in-bin*, which is a defined term representing the best implementable estimate of *quantity-in-bin*. There will probably be extra requirements for shipping failure (when a bin has unexpectedly run dry) and for periodic stock inventories to reduce the difference between *quantity-in-bin* and *expected-quantity-in-bin*. *expected-quantity-in-bin* will be defined in terms of shared, designated actions including *ship, receive,* and inventory adjustments.

Turski has pointed out that real time often appears in specifications because of unshared information [Turksi 88]. If the beginning of a physical process is a shared action and the ending of the process is not, then the machine can estimate the ending time from a description of the physical process including elapsed time.

5.3 Future reference

Some requirements are not directly implementable because they are stated in terms of the future. For example, consider the liveness requirement in a switching environment that, whenever the telephone user has dialed the last digit, eventually a *connect-to-ringback* or *connect-to-busytone* or *connect-to-errortone* action occurs. Restated, the tone must be produced when a digit has been dialed and no additional digits will be dialed in the future.

Like requirements with unshared information, requirements with future reference are satisfied with the help of domain knowledge, in this case relating the future to the past. The switching requirement can be satisfied using an approximate kind of domain knowledge: if the user is going to dial another digit, he will almost always do so within four seconds of the last digit.

The resulting specification says that the system must respond after a dialed digit followed by a four-second silence. The silence enables the machine to distinguish the final digit from nonfinal ones, and thus satisfy the original requirement.

5.4. Other views of requirements refinement

Some of these ideas have appeared in the literature before, in a variety of forms. Johnson's refinements called "defining capabilities" and "removing the perfect knowledge assumption" [Johnson 88] are addressing essentially the same gaps as environment constraint and unshared information, respectively, but they are described in very different terms, and without the special emphasis on domain knowledge. Mostow's "operationalization" goals *IsAchievable* and *IsEvaluable* are also similar in purpose [Mostow 83]. But Mostow's work focuses on automated problem-solving, and thus assumes—if applied to requirements engineering—that the environment is as malleable as the machine.

A recent trend in the study of requirements refinement is emphasis on various agents within the environment [Feather et al. 91, Dardenne et al. 93]. It is recognized that these agents often cooperate with the machine and with each other to satisfy unrefined requirements.

Agent-centered refinement is a special case of refinement as we have characterized it. A description of an agent in the environment is indeed domain knowledge, but not all domain knowledge takes the form of agent descriptions. Some domain knowledge simply captures static relationships in the environment.

For example, "indirect access" is a requirements refinement in which the machine needs some information it does not have direct access to; an agent in the environment gets it and communicates it to the machine [Feather et al. 91]. This is the only strategy mentioned by Feather et al. for dealing with unshared information, but there are others. In the lift example, the machine needs to know where people are waiting. It does not obtain this information through the intercession of another agent. Rather, it computes it from a known, fixed relationship between past shared actions (requests and floor arrivals) and the behavior of groups of people waiting for the lift to arrive.

"Soft" requirements as discussed by Yu et al. [Yu et al. 95] are "soft" for two reasons. They may be vague and imprecise, such as requirements that a system be "secure," "reliable," or "easy to use." Until they are precise enough to be formalized, our description of refinement is not relevant to them. Requirements are also characterized as "soft" if the domain knowledge that supports their satisfaction is not absolutely reliable. For example, the lift requirement that a waiting group is eventually served (L2) is satisfiable partly because (L3) someone in the waiting group pushes the request button. But L3 is a highly probable *assumption* rather than an unalterable fact. In those rare cases where the assumption is not true, the requirement will not be satisfied, even if the lift controller is perfect.

In contrast to the requirements refinements discussed above, in which domain knowledge is invoked to bridge

the gap from a requirement to a specification, some requirements transformations are equivalence-preserving. They restructure the available information, often for the purpose of deriving an acceptable specification, without adding to it. Since the notion of an "acceptable" specification depends on the expressive power of the chosen specification language, categorization of these transformations as requirements refinements or specification refinements is somewhat subjective. For example, unfolding of invariants and finite differencing have been used as requirements refinements by Feather [Feather 94], but they are also used as specification refinements by several authors cited by Feather.

6. *Four corners mark a foundation*

So far this paper has focused on four miscellaneous problems of requirements engineering, and suggested improved ways of understanding or managing them. But the problems are related. Their solutions fit together to define and regulate some of the primary concepts of requirements engineering. The results are summarized in this section.

6.1. *Terminology and Applicability*

Here are brief definitions of the major terms we have used:

The *environment* is the portion of the real world relevant to the software-development project.

The *machine* is a computer-based machine that will be constructed and connected to the environment, as a result of the software-development project.

A *designation* is an informal description of the meaning of an atomic formal term referring to the environment.

A *definition* is a formal description of an atomic term, using other defined or designated terms.

A statement (assertion, property) in the *indicative mood* describes the environment as it would be without or in spite of the machine.

A statement (assertion, property) in the *optative mood* describes the environment as we would like it to be because of the machine.

A *shared* action is an action in the environment in which the machine also participates.

An *unshared* action is an action in the environment in which the machine does not participate.

An *environment-controlled* action is an action in the environment that is controlled, performed, or

initiated by the environment.

A *machine-controlled* action is an action in the environment that is not controlled, performed, or initiated by the environment. The intention is that such an action will be controlled by the machine when it is connected to the environment.

A *requirement* is an optative property, intended to express the desires of the customer concerning the software-development project.

A *statement of domain knowledge* or *domain assumption* is an indicative property intended to be relevant to the software-development project.

A *specification* is an optative property, intended to be directly implementable and to support satisfaction of the requirements.

This terminology applies to all software-development projects, although projects differ in emphasis and some projects require imaginative application of the above definitions. The point is that requirements, specifications, and domain knowledge always have the same relationship to each other.

For one example, if the proposed machine is automating a currently existing manual system, then the focus is on the environment as it will be in the future (when the machine is installed), not as it is now. The indicative properties are those that will be enforced by the environment, including new manual operating procedures. The optative properties are those that will be enforced by the machine; they might be the same as the properties of the replaced manual system, but some changes are likely.

For another example, if the proposed machine is an augmentation of an existing ("legacy") machine, then the environment of this particular development project includes the legacy machine. The indicative properties are the properties that currently hold of the system at large (including both its manual and automated parts), and the optative properties are the desired new properties of the environment.

For a final example, if the proposed machine is a new reusable procedure, then the environment of this particular development project is nothing more than the procedure interface. The indicative properties are the guaranteed constraints on the procedure inputs. The only optative property (a specification as well as a requirement, since it is directly implementable) is the desired function from inputs to outputs.

6.2. Notation

Formal representations used in requirements engineering should obey these four rules. If a requirements language is insufficient in this regard, its users should be able to augment or supplement it as necessary.

(1) Each atomic term must be designated or defined. This means that everything said is said about the environment, since all atomic terms are grounded in the environment.

(2) Table 2 characterizes all relevant actions of the environment. Each action type must be identified as belonging to exactly one of the categories in Table 2.

	environment-controlled	machine-controlled
shared	X	X
unshared	X	

Table 2

(3) Whenever necessary to express relevant properties of the environment, actions in all three categories must be constrained.

(4) Each property or assertion must be identified as a requirement, statement of domain knowledge, or specification.

For clarity and brevity, we have phrased all of our arguments in terms of action-based notations. However, the concepts apply just as well to state-based notations as to action-based notations. When a state-based notation is being used, "action" is just another name for a state change [Abadi & Lamport 93].

In a state-based notation, the interface between the machine and environment is represented as shared state components, i.e., components whose values are accessible to both machine and environment. A machine-controlled (shared) component is one whose value is changed only by an action of the machine. An environment-controlled component is one whose value is changed only by an action of the environment. Environment-controlled components can be shared or unshared, i.e., inaccessible to the machine. The simplest state-based formulation of Rules 2 and 3 demands that all state components be identified as belonging to one of the three categories, and that state components be able to appear equally freely in assertions no matter which category they belong to.

Unfortunately, some state components may be changed by both the machine and the environment. When these "mixed" components are present, it appears necessary to introduce explicit actions, so that the machine-controlled and environment-controlled state changes to the mixed components can be distinguished.

For clarity and brevity, we have also used examples in which the formal descriptions are collections of distinct, isolated properties. However, the concepts apply just as well to executable notations such as Statecharts [Harel 87] and RSML [Leveson et al. 94].

The reason is that executable notations and property-oriented notations are not as different, fundamentally, as they might seem. Consider a large Buchi automaton—one with many states and transitions. It might be used as an executable specification, just as a Statechart or RSML specification might be. At the same time, the results of Alpern and Schneider show that it is equivalent to a large number of separate safety ("in state *S*, actions of type *A* cannot occur") and liveness ("in state *S*, an action of type *A* or *B* must eventually occur") properties.

There is no reason whatsoever why an executable notation cannot obey the first three rules; it is only necessary to ensure that its states and actions are environment phenomena, and that no category of action is discriminated against.

There may be a slight problem with distinguishing indicative and optative properties in executable specifications because, as noted above, many distinct properties are grouped together in a unified whole. This problem can be solved by putting indicative and optative properties in separate modules, or by establishing conventions for determining which aspects of a single module are meant to be indicative and which optative.

6.3. Completion

If the five following criteria are satisfied, then requirements engineering, in the strongest sense, is complete. We are guaranteed that the specification is implementable (at least in principle) without recourse to any additional information. We are also guaranteed that if the specification is implemented as a machine which is subsequently connected to the environment, then the requirements will be satisfied.

(1) There is a set *R* of requirements. Each member of *R* has been validated (checked informally) as acceptable to the customer, and *R* as a whole has been validated as expressing all the customer's desires with respect to the software-development project.

(2) There is a set *K* of statements of domain knowledge. Each member of *K* has been validated (checked informally) as true of the environment.

(3) There is a set *S* of specifications. The members of *S* do not constrain the environment, they are not stated in terms of any unshared actions or state components, and they do not refer to the future.

(4) A proof shows that:

$$S, K \vdash R$$

This proof ensures that an implementation of *S* will satisfy the requirements.

(5) There is a proof that *S* and *K* are consistent. This ensures that the specification is internally consistent and consistent with the environment. Note that the two proofs together imply that *S, K,* and *R* are consistent with each other.

The most commonly understood use of verification is to show that an implementation satisfies a specification. The completion rules show that verification also has a role to play in requirements engineering. First, verification is necessary to establish the satisfaction result.

Second, some consistency checking is supported by verification. This occurs when there is some redundancy among the various parts to be shown consistent—a natural enough occurrence in large specifications, with reuse, multiple views, etc. In the presence of redundancy, the statements made during the course of requirements engineering are not all independent, and some are implied by others. A proof that some statements imply others necessarily establishes the consistency of the consequences with the antecedents.

Acknowledgments

We are grateful for the helpful comments of Joanne Atlee, Martin Feather, the referees, and the editors.

References

[Abadi & Lamport 93]
Martĭn Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems* XV(1):73-132, January 1993.

[Alpern & Schneider 87]
Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing* II:117-126, 1987.

[Atlee & Gannon 93]
Joanne M. Atlee and John Gannon. Analyzing timing requirements. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis*, pages 117-127. *Software Engineering Notes* XVIII(3), July 1993.

[Balzer & Goldman 79]
Robert Balzer and Neil Goldman. Principles of good software specification and their implications for specification language. In *Proceedings of the Specifications of Reliable Software Conference*, pages 58-67. IEEE Computer Society, 1979.

[Brachman et al. 93]
Ronald J. Brachman, Richard E. Fikes, and Hector J. Levesque. Krypton: A functional approach to knowledge representation. *IEEE Computer* XVI(10):67-73, October 1983.

[Bubenko 83]
Janis A. Bubenko, Jr. On concepts and strategies for requirements and information analysis. In *Information Modeling*, Janis A. Bubenko, Jr., ed., pages 125-169. Chartwell-Bratt, 1983.

[Curtis et al. 88]
Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Communications of the ACM* XXXI(11):1268-1287, November 1988.

[Dardenne et al. 93]
Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming* XX:3-50, 1993.

[Dobson et al. 94]
J. E. Dobson, A. J. C. Blyth, J. Chudge, and R. Strens. The ORDIT approach to organisational requirements. in *Requirements Engineering: Social and Technical Issues*, Marina Jirotka and Joseph A. Goguen, eds., pages 87-106. Academic Press, 1994.

[Dubois et al. 93]
Eric Dubois, Ph. Du Bois, and M. Petit. O-O requirements analysis: An agent perspective. In *Proceedings of the Seventh European Conference on Object-Oriented Programming (ECOOP 93)*, pages 458-481. Springer-Verlag (LNCS 707), 1983.

[Feather 87]
Martin S. Feather. Language support for the specification and development of composite systems. *ACM Transactions on Programming Languages and Systems* IX(2):198-234, April 1987.

[Feather 94]
Martin S. Feather. Towards a derivational style of distributed system design—An example. *Automated Software Engineering* I(1):31-59, March 1994.

[Feather et al. 91]
Martin S. Feather, Stephen Fickas, and B. Robert Helm. Composite system design: The good news and the bad news. In *Proceedings of the Sixth Annual Knowledge-Based Software Engineering Conference*, pages 16-25. IEEE Computer Society, 1991.

[Gries & Schneider 93]
David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, 1993.

[Harel 87]
David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* VIII:231-274, 1987.

[Heitmeyer et al. 95a]
Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*, pages 109-122. IEEE, ISBN 0-7803-2680-2, 1995.

[Heitmeyer et al. 95b]
Constance Heitmeyer, Bruce Labaw, and Daniel Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*,

pages 56-63. IEEE Computer Society, ISBN 0-8186-7017-7, 1995.

[Heninger 80]
Kathryn L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering* VI(1):2-13, January 1980.

[Hoare 85]
C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[Iscoe et al. 91]
Neil Iscoe, Gerald B. Williams, and Guillermo Arango. Domain modeling for software engineering. In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 340-343. IEEE Computer Society, ISBN 0-8186-2140-0, 1991.

[Jackson 96]
Daniel Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, to appear.

[Jackson 78]
Michael Jackson. Information systems: Modelling, sequencing, and transformations. In *Proceedings of the Third International Conference on Software Engineering*, pages 73-81. IEEE Computer Society Press, 1978.

[Jackson 95]
Michael Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles, and Prejudices*. Addison-Wesley, 1995.

[Jackson 83]
Michael Jackson. *System Development*. Prentice-Hall International, 1983.

[Jackson & Zave 95]
Michael Jackson and Pamela Zave. Deriving specifications from requirements: An example. In *Proceedings of the Seventeenth International Conference on Software Engineering*, pages 15-24. ACM, ISBN 0-89791-708-1, 1995.

[Jarke et al. 92]
Matthias Jarke, Janis Bubenko, Colette Rolland, Alistair Sutcliffe, and Yannis Vassiliou. Theories underlying requirements engineering: An overview of NATURE at genesis. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 19-31. IEEE Computer Society, ISBN 0-8186-3120-1, 1992.

[Jeremaes et al. 86]
P. Jeremaes, S. Khosla, and T. S. E. Maibaum. A modal (action) logic for requirements specification. In *Software Engineering 86*, D. Barnes and P. Brown, eds., pages 278-294. Peter Peregrinus, 1986.

[Johnson 88]
W. Lewis Johnson. Deriving specifications from requirements. In *Proceedings of the Tenth International Conference on Software Engineering*, pages 428-438. IEEE Computer Society, ISBN 0-8186-0849-8, 1988.

[Jones 90]
Cliff B. Jones. *Systematic Software Development Using VDM,* Second Edition. Prentice-Hall International, 1990.

[Khoshafian & Copeland 90]
Setrag N. Khoshafian and George P. Copeland. Object identity. In *Readings in Object-Oriented Database Systems*, Stanley B. Zdonik and David Maier, eds., pages 37-46. Morgan Kaufmann, 1990.

[Lamport 89]
Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM* XXXII(1):32-45, January 1989.

[Lehman 80]
M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE* LXVIII(9):1060-1076, September 1980.

[Leveson et al. 94]
Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering* XX(9):684-707, September 1994.

[Liskov & Zilles 75]
Barbara H. Liskov and Stephen N. Zilles. Specification techniques for data abstractions. *IEEE Transactions on Software Engineering* I(1):7-19, March 1975.

[Mostow 83]
Jack Mostow. A problem-solver for making advice operational. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-83)*, pages 279-283. William Kaufmann, ISBN 0-86576-065-9, 1983.

[Parnas & Clements 86]
David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering* XII(2):251-257, February 1986.

[Parnas & Madey 95]
David Lorge Parnas and Jan Madey. Functional documentation for computer systems engineering. *Science of Computer Programming* XXV:41-61, October 1995.

[Spivey 90]
J. Michael Spivey. Specifying a real-time kernel. *IEEE Software* VII(5):21-28, September 1990.

[Spivey 92]
J. Michael Spivey. *The Z Notation: A Reference Manual, Second Edition*. Prentice-Hall, 1992.

[Swartout & Balzer 82]
William Swartout and Robert Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM* XXV(7):438-440, July 1982.

[Turski 88]
W. M. Turski. Time considered irrelevant for real-time systems. *BIT* XXVIII:473-486, 1988.

[van Schouwen et al. 92]
A. John van Schouwen, David Lorge Parnas, and Jan Madey. Documentation of requirements for computer systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 198-207. IEEE Computer Society, ISBN 0-8186-3120-1, 1992.

[Wing 90]
Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer* XXIII(9):8-24, September 1990.

[Wordsworth 92]
J. B. Wordsworth. *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*. Addison-Wesley, 1992.

[Yu et al. 95]
Eric Yu, Philippe Du Bois, Eric Dubois, and John Mylopoulos. From organization models to system requirements: A "cooperating agents" approach. In *Proceedings of the Third International Conference on Cooperative Information Systems,* pages 194-204. Vienna, Austria, 1995.

[Zave 82]
Pamela Zave. An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering* VIII(3):250-269, May 1982.

[Zave & Jackson 93]
Pamela Zave and Michael Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology* II(4):379-411, October 1993.