# Correctness in System Engineering

Peter B. Ladkin

Technische Fakultät, Universität Bielefeld, D-33501 Bielefeld, Germany
ladkin@techfak.uni-bielefeld.de
http://www.techfak.uni-bielefeld.de/~ladkin/

Version of April 22, 1995

**Abstract.** A philosopher has argued that there can be no such thing as a provably correct system. Some software reliability and safety experts believe that no system can have a software reliability of 1. These claims are related, and they are both mistaken, which we prove with a simple counterexample. But is this the only kind of counterexample? An interesting question arises: *which* types of systems admit of correctness proofs? Setting the boundary becomes important. I commence an analysis of what systems may consist in, and what the logical relations are amongst various concepts concerned with systems, such as specifications, correctness and failure. The goal is to put system engineering on a logically rigorous foundation. Discussing two more examples, I conclude that some useful systems may, not just in principle but in fact, be proved correct.

## 1. Introduction

Most scientists and practitioners interested in software safety, software quality or formal methods will know a debate that has continued without resolution for years. The debate is concerned not just with the right way of approaching some particularly intractable problem, but about the foundations of system engineering itself. Can there be such a thing as a physical system which we can *prove* correct? The answer is yes; I shall prove it easily. I analyse what kinds of things systems are, in the course of a discussion about correctness and failure. This leads to a clarification of why, and for which kinds of systems, it may be possible to provide correctness proofs.

*Correspondence and copy requests to*: ladkin@techfak.uni-bielefeld.de

### 1.1. Synopsis

In Section 2 I present a simple example of a provably correct system, and the proof. 'The system' is concretely physical – it is a counterexample to any claim that no physical system may be proved correct. It is also a counterexample to the related claim of Fetzer [Fet88], that it is in principle impossible to prove (implemented) programs correct, and a claim of software safety specialists that no software can ever have a reliability of exactly 1. The counterexample motivates a further argument that whether you can prove a system correct depends not only on the physical system but also on what kinds of properties you're trying to assert of it.

I discuss, and answer, various social and technical objections to the counterexample. A significant one is that *satisfying a formal specification* and *being correct* are not necessarily the same thing. While true in general, I show later in Section 4.2 that in this particular case they are indeed the same thing. I argue in Section 7.2 that for certain mathematical computations they may also be the same thing.

Considering how the counterexample works leads in Section 3 to clarifying the notion of system correctness. An argument based on the 'Russell Criterion' shows that either 'correctness' is a predicate whose logical form includes both a physical device and a non-physical 'purpose' as arguments; or the specification is an integral part of the system itself. This identifies the most prominent weakness of a Fetzer-like claim, namely that since the system purpose is involved in assessing the truth of correctness, whether one can prove a system correct (with respect to its purpose) depends also on the logical form of the assertion of system purpose. When it's a predicate that can be satisfied by anything, as in the simple counterexample, then of course correctness can be proved! In the subsequent section, an analysis of the ways in which assertions of failure are related to parts of systems, I conclude that the second option is easier: to treat the specification as part of the system. So, rather than correctness being a predicate with various arguments of different types, one a physical thing, one a type of proposition, I conclude that the surface syntax of correctness assertions is correct: the correctness predicate is unary and the system itself is a conglomeration of heterogeneous components, of which one is a physical thing and another a type of proposition, and so on.

I try to elucidate in Section 4 the relation between the '*supposed-use*' of a system, and the '*formal specification*'. For any system which is to be proved correct, there must be a provable match between the supposed-use and formal spec (as one of many such proofs). Can there be any non-trivial such use-spec pairs? The answer is yes, noted in Sections 4.2 and 7.2.

Given that at least *some* physical-system/property pairs may be proved correct (and some, we presume, may not be), the interesting question now becomes: where exactly *is* the boundary? One of Fetzer's claims is that non-logically-valid predictions of the future must allow the possibility of being false. Such propositions help set the boundary. But this one is mistaken: I use an airplane example in Section 5 to argue that some non-logically-valid predictions may indeed be inviolable. I look at the other side of correctness, namely system *failure*. I claim that under certain circumstances we locate system failures entirely in certain parts of a system and not others, and this is predictable from the outset. In the airplane example, if the safety requirement is violated, we look for the cause not in software, but in hardware, and I maintain that there is no circumstance in which it makes any sense for this case to inspect the software to identify the fault. This entails that the safety property claimed of the software does not allow the possibility of being false. It is nevertheless not a logically-valid property[1] and thus is a counterexample to the (modified) assertion of Fetzer.

Although the purpose of this example is to refute a claim of others concerning the boundaries of provability, I offer a few ideas concerning how the example may be explained, based on the notion of *speech acts*. I suggest that we tacitly perform *commissives*, that is, a tacit inviolable agreement, that certain parts of a system, under certain circumstances, may never come under suspicion in case of failure. The reader may note that whether or not these ideas on commissives are fruitful, no other argument in the paper depends on their verity.

I offer a formalisation of some of the properties of the concepts of fault and failure in Section 6, then I return to the possibility that, for certain sorts of systems, the supposed-use and the formal spec may be seen to match, by considering a theorem of Boyer and Yu concerning a GCD calculation in Section 7. Matching specification and supposed used is a necessary condition for proving a system correct (although not a sufficient one). By analysing what can and what is to be proved in this example, I conclude that such systems may, in principle, be proved correct up to physical hardware failures – that is, their implemented software may indeed be proved correct.

## 1.2. Some Social Issues with Technical Consequences

Although the main point of this paper is technical, there are also issues of rhetoric (in its technical sense) which seem to have affected the technical debate. Section 8 contains some observations concerning the rhetorical style often employed in technical (but non-mathematical) debates of this sort in computer science.

The debate over Fetzer's argument has been particularly murky. A common reason given is that the two (or more) sides lack mutual comprehension. This may be true. However, one could use that feature to advantage, like analytical philosophers, more clearly to define and analyse arguments in public for each position, thus increasing knowledge of the issues. Mathematicians have constructed their particular science with publicly available criteria for judging the correctness or falsehood of claims made. Both groups gain from their respective procedures, but the history of the provability debate amongst computer scientists follows neither paradigm. Of the many published comments concerning [Fet88], only [DR89b] called for further technical discussion of the issues, in a note concerned primarily with the social matters. The only technical discussion with any depth appeared in [Bar89].

Among those who believe (correctly) that Fetzer's claim is false, some wrote *ad hominem* refutations, and others simply chose not to join in the debate, despite their strong beliefs for or against. By attempting analytically to settle the question of theoretical system correctness, I was led to try to clarify the concept of system and the role of specification and verification in system design. Such opportunities may be lost if one forgoes technical analysis in favor of other reactions.

One noticeable feature of the debate to date has been the lack of examples produced and discussed by the participants. Philosophers and mathematicians produce and discuss examples as a major part of their working methodology. Just as it is very hard to produce a working

[1] I call a property or predicate *logically valid* or *logically true* if everything has that property, that is, if any proposition formed from that predicate by instantiating its argument place is true. Similarly, I say that a predicate $P$ *logically entails* another, $Q$, if every assertion formed by instantiating the argument place of $P$ logically entails the assertion formed by instantiating the argument place of $Q$ with the same argument.

computer program without a compiler and a few easy test cases, I suggest it is hard to produce a correct philosophical or logical argument without grounding it in good examples. I use three examples as a basis for analysis. The reader may judge for herself how this helps.

## 2. An Example, Some Corollaries and a Discussion

First, an example of a system which is provably correct. The example is simple and clear. Its purpose is to provide a counterexample to claims of Fetzer and others, and to ground the arguments which follow.

### 2.1. The System S and a Trivial Theorem

Here is a specification of a system S:

*The system S may behave in any manner at all:*
*all behaviours are compatible with this specification.*

An implementation of S:

$$\mathtt{S} = \textit{My refrigerator.}$$

Here is the theorem stating the correctness of this implementation:

**Theorem:** S implements the specification.
**Proof:** We need to show that all possible behaviours of S satisfy the specification. Let b denote an arbitrary system behaviour.[2] By definition, b satisfies the specification. **QED**

Those who prefer formality may consider the specification $\neg(P(\mathtt{S})\ \&\ \neg P(\mathtt{S}))$, in which $P$ is any unary predicate symbol that may be interpreted as any unary predicate. We leave the proof of correctness in a classical or constructive logic to the reader. Let $\Pi$ be this specification: formally, define $\Pi$ to be the predicate $\neg(P(-)\ \&\ \neg P(-))$, where '$-$' is to be replaced by the same denoting term in both occurrences.

### 2.2. First Corollary: Program Verification

Fetzer argued at length that program verification is impossible in principle [Fet88]. A Fetzer-like argument proceeds something like this. Proofs are things concerning abstract objects like 1s and 2s, not physical things such as 1 apple or 2 oranges. Systems (and 'programs') are physical things. It's impossible that one could *prove* properties of the set of future behaviours of a physical thing, that is, that one could guarantee properties of such behaviours with absolute certainty. This resembles a well-known theme of David Hume [Hum78]. According to Fetzer, implemented programs have causal properties, namely that they cause their outputs, and outcomes of causal relations cannot be known with certainty. Actual physical behaviours of a system are not the kind of thing that allows such things as proofs, just as my cup of tea is not the kind of thing that can have wishes, desires or moral obligations (in terms due to Gilbert Ryle [Ryl63], one would be making a *category mistake*).

[2] whatever that might be. We do not need to commit ourselves. b might include some description of S's environment E and E's behaviour also.

The theorem above provides a counterexample to Fetzer's conclusion, thus relieving us from having to examine his argument in detail at this point. See [Lad95b] for a review of Fetzer's papers.

### 2.3. Second Corollary: Software Reliability

There is a view widely-held by system reliability specialists that one can never have a *software reliability of exactly 1* [ALLM93]. We don't have to know exactly what is meant by this expression. We just have to know that (a) if a system can be proved to never, ever admit of any unwanted behaviour, and (b) if also it includes a bit of software (whatever that may be), we may conclude that (c) its "software reliability is exactly 1".

The theorem in Section 2.1 establishes (a) for the system `S`. To establish (b), assume for the purposes of this argument that my fridge has a microprocessor in it running a bit of code.[3] We may conclude (c) that the implementation `S` has a "software reliability of exactly 1". Therefore this implementation of `S` is a counterexample to the claim of [ALLM93].

### 2.4. Some Objections

Here are some objections, which have occurred in debate, and answers to them.

- The example is completely trivial.

That's ideal for a counterexample. The purpose is to refute certain claims, which it does. The refutation is all the more clear, the simpler the counterexample. There are few places in which to question the argument.

- The argument can easily be modified to exclude examples like this with "logically-true" specifications.

Maybe, or maybe not, but the problem does not then go away. The wedge has been driven, and so the issue is altered. The question now becomes *what kinds of specification/system pairs allow an a priori demonstration of correctness*? We now know, contrary to Fetzer, that there are at least some. We argue below that systems with specifications of the sort in Section 2.1 are not the only kinds of systems for which it may be possible to provide a rigorous absolute correctness proof. A careful statement delineating for which systems it is possible to provide absolute correctness proofs would be a valuable scientific contribution to system engineering. I shall only be able to provide suggestions.

- This is a strawman. Nobody really believes that *all* systems could not be proved correct.

One may refer to the works and correspondence cited to see that this statement is untrue. (My experience is that few people are aware of the existence of counterexamples until it's pointed out.)

- In *interesting* and *useful* cases, software is not 100% reliable and systems may not be proved.

[3] Now if only the light would go on when I opened the door.

Maybe, maybe not. But this is no argument against getting the basics right, before proceeding to grander questions. 'Interesting' and 'useful' refer to qualities of software about which I don't yet know how to be precise. I argue below that mathematical software of certain kinds, for example a GCD algorithm, may lend itself to being proved correct. One may argue that this sort of mathematical software is both interesting and useful.

- Stating '`S` is correct' and stating '`S` implements the specification' are two different things. Therefore the proof above doesn't prove that `S` is correct.

This is a plausible claim, in that there are some systems `S` for which this is true. However, it is false for the example `S` above, as we show in Section 4.2. One reason (the objection runs) that the two statements above might be different, is that a specification might be incorrect. I need to show that in the case of `S` above, this cannot happen. I shall provide an argument that `S` *is correct* if and only if `S` *implements its specification*, for this particular system and specification above. To do that, I shall investigate further what '`S` is correct' might possibly mean, in Section 3 and subsequently.

- The notions of *proof* when applied to systems, and of *correctness* of systems, are both too vague to admit a precise counterexample.

First, there is a notion of proof, since Frege, which is not at all vague, but on the contrary has a precise technical definition. Second, note that *proving system* `S` *correct* is shorthand for *proving that system* `S` *is correct*, which is in turn shorthand for *proving the following proposition: System* `S` *is correct*. In other words, we're not 'proving systems', we're proving *propositions about* systems. The usual mathematical, logical, or philosophical notion of proof is involved.

Is the term *correct* vague when applied to systems? I argue in Section 4.2 that although this term may be vague in some uses, it's not vague when applied to the system `S` of Section 2.1. In fact I claim that the only reasonable meaning in this case is that `S` *implements the specification*. In the meantime, to consider whether this point is basically sound, the reader might ask herself: if not this, then what *else* could correctness of the `S` in Section 2.1 possibly consist in?

- This counterexample is unrelated to any of the real activity of system verification, and it's not going to convince anyone who wasn't already convinced, that systems can be proven correct.

These considerations might be of social-psychological importance, but they are irrelevant to the logic of the argument. When refuting a claim technically, only logical relevance counts. (This is not to discount the importance of persuading people. Correct technical reasoning should be the appropriate way of persuading scientists, mathematicians and philosophers of the truth of a claim.)

- (*An extreme position*) We are wasting our time when we vent 'opinions' when we could be advancing the science of verification.

This is another social claim which is irrelevant to the validity of the technical claim. I also urge caution when conflating word-arguments with 'opinions'. Arguments made using the written word, when they are precise, have the same logical status as arguments made using mathematical symbols. 'Opinions' are generally taken to be points-of-view with which legitimate disagreement is possible. The technical arguments made in this paper are not intended to be of that kind (they may nevertheless be wrong if they contain mistakes).

Claims were made by Fetzer and others which are provably false. It is important to produce a counterexample when one exists. And by putting the arguments on a more technical level, it should be possible to define more carefully the principles underlying the engineering of systems. Social-psychological issues are addressed further in Section 8.

### 2.5. A Further Incorrect Argument

In line with the the idea that all counterexamples to im-provability must be trivial, an objection has been made to the argument of this paper that if a specification is to say anything non-trivial about a physical system, it must have *empirical content*, and, bending to Humean argument, the truth of statements having empirical content may not be determined by *a priori* argument.

The notion that statements can have "empirical content" is seductive, however, the "content" of a statement may or may not play a role in determining its truth. A common view of logic is that it is the science of determining the truth or falsity of statements in virtue of their *form* (rather than their content) [Dum73]. Let $P(a)$ be a statement concerning an object $a$. Suppose $P$ has "empirical content". Nevertheless, we may determine the truth of $\neg(P(a) \,\&\, \neg P(a))$ without reference to its content. There is a further question whether specifications of which we may determine the truth without reference to empirical content must be similar to logical truths. This leads to arguments similar to those considered by Kant, who in answer to Hume classified truths into analytic and synthetic, into *a priori* and *a posteriori*, and argued that there were synthetic *a priori* truths. Such issues are relevant but not trivial. A part of the argument in Section 5.1 may be related.

A similar objection concerns the impossibility of predicting things about the future with certainty. This falls to a similar refutation. The specification *Tomorrow it will be true that* $\neg(P(a) \,\&\, \neg P(a))$ is no less provably true for its referring to the truth of a (logically true) subsentence at a future time.

## 3. Concerning Systems, Correctness and Failure

We can nevertheless gain from analysing flawed arguments. Fetzer relied on what may seem to be a categorical difference between abstract mathematical ideas (which admit of proof but have no causality), and actual physical systems (which have causal properties) on the other. In light of this contrast, can we say how the counterexample works? It seems we are able to conclude things about the future satisfactory behaviour of my fridge, without seeing the fridge, or even testing it. How so?

The analysis below of the concepts underlying this example is long, and might seem laborious. My purpose is to establish the reasons and reasoning behind each step, so that the series of arguments is laid out as carefully as can be, and the steps understood and discussed individually. This is one way in which standard definitions developed – when this knowledge becomes received, e.g., as calculus is nowadays 'received knowledge' in engineering. Applications of logic to engineering are still in their youth, compared with calculus.

The conclusion of this section will be that the specification (the 'requirements specification') is best considered as part of the system itself, rather than as an argument to the *Correct* predicate.

### 3.1. A Proposition About Correctness

This section establishes the following proposition:

- *Either:* system correctness is a relation, two of whose components are, first, a *physical system* such as my fridge, and second, an *assertion of what its use is supposed to be*[4] (as contrasted with simply a unary predicate applying to my fridge);
- *or:* correctness is indeed a unary property of a system, but some concept of *what its use is supposed to be* is an integral part of the notion of a system, and therefore *the system* is not entirely a physical thing, but has parts which may be linguistic, logical or mathematical, depending on how the *supposed-use* is expressed.[5]

The first alternative has also been argued using examples, in the context of programs, by Thimbleby [Thi93]. I won't at first attempt to choose between the two alternatives, although after more discussion a preference for the latter will present itself. in Section 3.4. It suffices at first to establish the point that you can't really tell what a system is without also having some idea about how it's supposed to behave. This highlights an influential misconception entertained both by Fetzer and by some of his critics. I want to establish that when we're talking about system correctness, there aren't just physical bits and pieces to consider, there are also more abstract entities like propositional assertions.[6] The proposal does not preclude that there might be further parameters of the correctness predicate (for example, 'the environment', other system specifications).

The argument for the proposition relies on a principle of logical form due to Frege, and a criterion I'll dedicate to Russell.

**Frege Principle:** The *logical form* of a syntactic object (e.g., a term, predicate, or assertion) expresses its *logical type*. Logical form is a concept of syntax, logical type one of semantics. For example, an assertion has the logical type of a complete proposition, namely $\{\mathbf{True}, \mathbf{False}\}$.[7] The logical type of a predicate is given by enumerating the arguments to the predicate and their logical types such that when the predicate is given values for these arguments, an assertion is formed. Furthermore, the value of a predicate is well-defined for every possible combination of well-defined values from the logical types of the arguments. A term denotes, and its logical type is that of the objects it denotes;

[4] I shall call this the *specification*. The relation might be binary, but it's more common to see more than one system specification, for example a requirements specification and a design specification. The relation may therefore become more complicated, involving fit between various levels of specification and its intended use. We won't pursue this further here.

[5] I distinguish between *logical* and *mathematical* entities because not everyone agrees that mathematics is logic, even though most agree that if anything is *a priori*, both logic and pure mathematics are. One common argument that mathematics is not logic concerns the posited existence of an infinite set, which is not held to be a logical assertion, but seems to be necessary for much mathematics, although see [Myc89, Myc92, Myc95].

[6] For example, what I am suggesting seems to strain against the vocabulary used by Parnas [Par90], who refers to everything that is not in the physical hardware as 'documentation'. According to the proposition established in this section, some of the 'documentation' is either part of the system, or is an integral part of talking about correctness. It seems more helpful therefore clearly to distinguish between those parts of the documentation that might be essential in this regard, and those other parts that aren't. We shall do so subsequently.

[7] I shall be using an analysis adapted to classical propositional logic. I propose that a similar analysis may also be given with constructive, intuitionistic, logic.

**Russell Criterion:** Suppose the truth of an assertion is functionally dependent upon a particular feature of the world. Then, the logical type of the assertion is functionally dependent on the logical type of the feature. The logical type of that feature is denoted by its logical form. In the logical form of that assertion, either the logical form of the feature appears as an explicit argument, or there is some collection of arguments, each of which is itself functionally dependent on the feature, such that the variation in value of those arguments with the value of the feature results in the truth value of the assertion varying in conformance with its functional dependence upon the feature.

The Fregean Principle led to the current notation for logic. The Russell Criterion says that if there's a functional dependency of assertion $P$ on something with logical form $F$, then either $F$ appears as an explicit argument, or there's some list $A_1, \ldots, A_n$ of arguments to $P$ such that the denotations of $A_1, \ldots, A_n$ are all functionally dependent on the denotation of $F$, and their values, when "plugged in" to yield the value of $P$, yield the right value of $P$ given the value of $F$. Using this criterion, the argument proceeds as follows:

Suppose my fridge implements the following specification $\Psi$:

$\Psi$: `S` *keeps things inside it cold*

Suppose *correctness* is a unary property that holds, or not, of system `S`, my fridge. That is, a logical form of a correctness assertion is *Correct*(`S`). I omitted to say earlier that the fridge has no plug to obtain electricity. So it doesn't keep things cold.[8] So it's not correct. That is, $\neg$*Correct*(`S`). But there is a proof in Section 2.1 that the system *is* correct, that is, *Correct*(`S`). Taken at face value, this would violate Leibniz's Law that an object does not have two contrary properties at the same time.[9] The apparent logical form of the correctness assertion, its *surface syntax*, which we had written *Correct*(`S`) with the given logical type, does not seem to reflect the functional dependence on the specifications $\Pi, \Psi$.

These considerations show that the correctness-of-`S` assertion is functionally dependent on the requirement – in the first case, $\Pi$: that any behavior at all is allowed; in the second, $\Psi$: that it keeps things cold. Using Russell's Principle, we may conclude that there is some parameter or parameters of the logical type which reflects this functional dependency on $\Pi$ and $\Psi$. There are four possibilities:

- it must be explicit in the predicate *Correct*;
- it must be explicit in the term `S`;
- it's not reflected explicitly in either and needs to be added;
- it's reflected implicitly and the surface syntax is appropriate.

The surface syntax of the expression *Correct*(`S`) may be fairly close to being a correct logical form (after all, its form misled Fetzer and continues to mislead some computer scientists). We can enquire what the appropriate modifications are that we can make to this expression to account for the functional dependency on the specification. We had thought that *Correct*(`S`) has type $\{\mathbf{True}, \mathbf{False}\}$, the type of assertions. According to the fourfold possibility above, there are four situations to consider:

[8] I confess, in truth my fridge does plug in, contrary to what I just supposed, but nonetheless it doesn't keep things cold.

[9] Proposition $P$ is a contrary of proposition $Q$ if and only if $P \rightarrow \neg Q$. Predicate $P(-)$ is a contrary of predicate $Q(-)$ if and only if $\forall x : P(x) \rightarrow \neg Q(x)$. A similar definition may be given for predicates with multiple arguments. A property and its negation are contraries.

**two different predicates:** the two different assertions are formed by applying two different predicates (despite the surface identity) $Correct_1$ and $Correct_2$ to the term $\mathtt{S}$, to generate the two different correctness assertions, $Correct_1(\mathtt{S})$ for $\Pi$ and $Correct_2(\mathtt{S})$ for $\Psi$;

**two different terms:** the two different assertions are formed by applying the predicate *Correct* to two different terms, to generate the two different correctness assertions $Correct(\mathtt{S}_1)$ for $\Pi$, and $Correct(\mathtt{S}_2)$ for $\Psi$;

**both of the above:** it's not appropriate to consider this possibility unless we know that both of the above proposals hold up;

**a different logical type:** the logical type of $Correct(\mathtt{S})$ is actually

$$specifications \rightarrow \{\textbf{True}, \textbf{False}\}$$

reflecting the functional dependency on the specification.[10]

In the first two cases, we accept that the surface syntax and semantics is roughly correct (while mistaken in details) and correct the error syntactically, and in the final case we accept the surface syntax as given and alter the semantics accordingly.

**parametrising the predicate:** The predicates $Correct_1$ and $Correct_2$ are functionally dependent upon the specifications $\Pi$ and $\Psi$, and they have closely related meanings. The Russell Criterion suggests that we should be able to obtain these two predicates from a single correctness predicate with a parameter for the specification, i.e., $Correct_1(\text{-})$ has the form $\mathcal{C}(\Pi, -)$, and $Correct_2(\text{-})$ has the form $\mathcal{C}(\Psi, -)$. The two different correctness assertions thus have the forms $\mathcal{C}(\Pi, \mathtt{S})$ and $\mathcal{C}(\Psi, \mathtt{S})$. This may be interpreted as saying that the specification is a frequently-overlooked second argument to the correctness predicate;

**parametrising the term:** Suppose instead we retain the predicate *Correct* as it is, and choose to modify the term $\mathtt{S}$, to obtain $\mathtt{S}_1$ and $\mathtt{S}_2$. If we were to parametrise the term similarly to the way in which we parametrised the predicate, we would obtain the two terms $\mathtt{Sys}(\Pi)$, respectively $\mathtt{Sys}(\Psi)$; and the two correctness assertions $Correct(\mathtt{Sys}(\Pi))$ and $Correct(\mathtt{Sys}(\Psi))$.

**changing the type:** If the type of the expression $Correct(\mathtt{S})$ is $specifications \rightarrow \{\textbf{True}, \textbf{False}\}$, the question now becomes: what type do the syntactic components of the expression have?

**absorbing the type:** the meaning of the predicate or the term already is functionally dependent upon the type – it remains to clarify the type.

What are the reasons for preferring one of these solutions over another? Before I consider them, I must clarify what is meant by 'absorbing the type', that a functional dependence on the specification is already somehow 'included' in either term or predicate.

## 3.2. Absorbing the Type

A system has parts. It's often observed that one part of a computational system is hardware, and another is software, and that the system is functionally dependent on both its hardware and software (I assume, rather than argue for, this assertion here). The predicate *Part(P,S)*

[10] Actually, I should take the type to be $T \times specifications \rightarrow \{\textbf{True}, \textbf{False}\}$ where $T$ represents a list of other functional dependencies remaining yet unacknowledged. However, it seems that these extra parameters could be handled similarly to the way I'm about to deal with the dependency on specifications, so I shall omit $T$ from the discussion.

says that $P$ is a part of entity $S$. The assertions that hardware and software are both parts can thus be expressed as $Part(hardware(S), S)$ and $Part(software(S), S)$. The study of how parts and wholes may relate is known as mereology, and has a history of formality [Sim87]. Three axioms used in most formulations of mereology are that that something is always a part of itself:

$$Part(P, P)$$

that if $P$ is a part of $S$ and *vice versa*, then $P$ and $S$ are identical:

$$Part(P, S) \,\&\, Part(S, P) \;\Rightarrow\; P = S$$

and that $Part$ is transitive:

$$Part(P, S) \,\&\, Part(S, T) \Rightarrow Part(P, T)$$

Thus, $Part$ is a reflexive, antisymmetric, transitive relation, i.e. a non-strict partial order. I shall follow [Sim87] and denote it $<$ in infix notation. The corresponding relation of *proper Part*, which is irreflexive and transitive, is denoted by $\ll$. (They are related by the properties $x \ll y \Leftrightarrow x < y \,\&\, x \neq y$ and $x < y \Leftrightarrow x \ll y \;\vee\; x = y$.) Parthood is not generally a linear order – my arm is part of me, as is my other arm, but neither is part of the other.

It's widely accepted that two different parts of an entity can be considered together as one part, different from either component [Sim87]. This larger single part is called the *mereological sum* of the two components, and I denote the operation of forming the mereological sum by $\oplus$. What kind of thing is $P \oplus S$, exactly? One way of helping to define it implicitly, also the notion of part itself, is by means of axioms. Some common axioms are that the mereological sum of something with itself is just itself,

$$P = P \oplus P$$

that mereological sum is commutative and associative,

$$P \oplus S \;=\; S \oplus P$$

$$(P \oplus S) \oplus T \;=\; P \oplus (S \oplus T)$$

that mereological sum is an upper bound (in the sense of $<$) to each argument, which follows from commutativity and

$$P < P \oplus S$$

and also increasing in both arguments: a mereological sum of parts of $P$ and $S$ is itself a part of $P \oplus S$,

$$p < P \,\&\, s < S \;\Rightarrow\; (p \oplus s) < (P \oplus S)$$

Since there are many things that may have parts of $P$ and $S$ as parts, $P \oplus S$ is usually conceived as some smallest entity which has all the parts of $P$ and $S$ as parts. The axioms above already entail that all the parts of $P$ and $S$, as well as their mereological sums, are parts of $P \oplus S$. We can require that $P \oplus S$ is smallest by saying that the parts of the mereological sum are precisely these and no more:

$$(P < S \oplus T) \;\equiv\; (P < S) \vee (P < T) \vee \exists s \exists t (s < S \,\&\, t < T \,\&\, P = s \oplus t)$$

Some of these properties will be used in Section 3.4.

If we extend the notation, we may express the property that an entity is the mereological sum of its parts by

$$P = \oplus(p \,|\, p < P)$$

However, in order to make sense of this operation with an undetermined, maybe infinite, number of arguments, it is easiest (and common) to treat the prefix $\oplus$ as a unary operator on classes. But this requires that some set theory be introduced, including perhaps the existence of an infinite class, so that one can treat an unbounded class of parts, and it may be preferable to see what can be done without it.

The Russell Criterion does not require that if an entity is functionally dependent on another, that the logical form of the first must expressly include a term denoting the second. The criterion would be absurd if it were to! For example, nonempty subclasses can reasonably be considered parts of classes [Lew91], and a class can be considered to be functionally dependent on its class of subclasses (by the axiom of extensionality for classes). There are exponentially many subclasses in the size of a class, not to speak of infinitely many for infinite classes. We would never be able to define the logical form of any term denoting a class with infinitely many parts if we were expressly to include denotations for each part in the term itself.

Since it seems to make little sense to include the parts of an entity as explicit arguments to its logical form, one can consider the possibility that I called 'absorbing the type', which retains the 'naive' logical form, *Correct*($\mathtt{S}$). Namely, the specification is considered to be a *part* of either of the entities – predicate or term – occurring in the assertion. If so, which entity could it be considered part of? Considering parts of a predicate requires considering what a predicate is – for example, a predicate could be identified with the class of all entities (or tuples of entities in the case of a multi-argument predicate) which satisfy that predicate, that is, of which the predicate is truly predicated. If we make this move, and we also have a notion of what parts classes have, as in [Lew91], then we are able to say what parts predicates have. This is a fairly involved move, dependent on the notion of *class*. In contrast, in the case of concrete physical entities, the intuitions about what are parts and what are not are much more firmly grounded. For example, we noted that a system has 'natural' parts consisting of *hardware* and *software*.

### 3.3. Choosing Between the Options

The four choices are:

**a:** The specification is an argument to the correctness predicate $\mathcal{C}$, which thus has type

$$\textit{specifications} \times \textit{systems} \rightarrow \{\mathbf{True}, \mathbf{False}\}$$

**a′:** The specification is a part of the correctness predicate;

**b:** the specification is a strict argument of the system parameter, which then has type *specifications* $\rightarrow S$, where $S$ is some type to be clarified, and *Correct* thus has type $S \rightarrow \{\mathbf{True}, \mathbf{False}\}$.

**c:** the specification is a part of the system, just as hardware and software are parts of the system.

Options **a′** and **c** preserve the 'naive' form of the correctness assertion and the logical types of its subexpressions. However, **a′** rests upon an analysis of what the parts of a predicate might consist of. It is a false option. Here is the argument dismissing it.

Objects have parts. One way in which we could express the 'parts' of a predicate would be to reinterpret it as an object of some sort and then consider its parts. The standard way to interpret a predicate as an object is to interpret it as a set-theoretic object, accomplished by considering it as a set of ordered argument-value pairs. Given that the parts of classes are subclasses (but the converse does not necessarily hold [Lew91]), all the parts of the predicate considered as a class are subclasses of it. Reinterpreting, the parts are logically strictly stronger predicates, having the same logical type as the predicate itself. Thus, were a specification to be a part of the predicate *Correct*, it would have the logical form of a unary predicate of systems, with logical type *systems* $\rightarrow$ $\{\mathbf{True}, \mathbf{False}\}$. That is, we would be able to apply a specification to a system to obtain an assertion: namely, $\Pi(\mathtt{S}), \Psi(\mathtt{S})$. This has a natural interpretation: to assert that a system is correct is actually to assert that it satisfies the specification, which is to assert the specification of the system. And $\Pi(\mathtt{S})$ is **True**, $\Psi(\mathtt{S})$ is **False**. This is plausible, but before being seduced by this consequence of the argument, note that we have argued that a specification is a part of the predicate *Correct*, and that 'part' entails 'subclass of'. That reasoning must hold for every specification. Assuming that any specification is in principle permissible, we may choose the logically-true specification $\Pi$, which is universal, and which also satisfies $\Pi < \mathit{Correct}$, and conclude that $\Pi = \mathit{Correct}$, that *Correct* is also the universal predicate of its type: every assertion $\mathit{Correct}(S)$ of any system $S$ must be **True**. This is implausible: it renders every assertion of correctness vacuously true. I conclude that this proposal doesn't allow us to make any sense of option $\mathbf{a}'$, and so I consider it no further.

Some division of systems into parts such as hardware and software are recognised and in habitual use already. Thus option **c**, unlike option $\mathbf{a}'$, is plausible,

Consider option **b**. It entails that the expression `Sys` has type

$$\mathit{specifications} \rightarrow \mathit{systems}$$

which entails that *Correct* has type

$$(\mathit{specifications} \rightarrow \mathit{systems}) \rightarrow \{\mathbf{True}, \mathbf{False}\}$$

In categories of logical types, it is well-known that one can move from something of type $(A \times B) \rightarrow C$ to something of type $(A \rightarrow B) \rightarrow C$ (by a 'Currying' transformation, named after Haskell Curry), and *vice versa*, by means of natural transformations from the category of types into itself. No matter which category of types one prefers, these operations are admissible. Hence we can conclude that the type of *Correct* is isomorphic to the type

$$(\mathit{specifications} \times \mathit{systems}) \rightarrow \{\mathbf{True}, \mathbf{False}\}$$

which is exactly the type of $\mathcal{C}$ of option **a**. Thus there is little argument of a logical nature that would dispose us to choose between options **a** and **b**. We may freely choose option **a** as representative of both, solely for aesthetic reasons.

We thus need to consider only options **a** and **c**. I have established the proposition claimed at the beginning of this section. Now to use it, in developing an argument for preferring option **c** out of an analysis of the kinds of arguments made by system engineers.

Before doing so, there is a particular interpretation of option **a** to note. One can take the operator $\mathcal{C}$, which takes specifications and systems and produces a truth value, simply to be the *apply* operator: $\mathcal{C}(\Theta, S) \stackrel{\triangle}{=} \Theta(S)$. (*Apply* is a well-studied logical operator from $\lambda$-calculus and combinatory logic, where it is represented by concatenation). When considering option **a**, it may make sense to interpret $\mathit{Correct}_1(\mathtt{S})$ as $\Pi(\mathtt{S})$ and $\mathit{Correct}_2(\mathtt{S})$ as $\Psi(\mathtt{S})$. Nevertheless, further discussion will now show that this is a less desirable option than that of **c**.

### 3.4. Analysing Arguments

Suppose for the sake of argument we consider a *system* to consist of *hardware* plus *software* (whatever they may be – see Section 4.1), and possibly also a specification (as in option **c**). Option **a** then says that a correctness predicate has type

$$\mathit{specifications} \times (\mathit{hardware} \oplus \mathit{software}) \rightarrow \{\mathbf{True}, \mathbf{False}\}$$

Option **c** says that

$$\mathtt{S} = \mathit{specifications} \oplus \mathit{hardware} \oplus \mathit{software}$$

and therefore that a correctness predicate has type

$$(\mathit{specifications} \oplus \mathit{hardware} \oplus \mathit{software}) \rightarrow \{\mathbf{True}, \mathbf{False}\}$$

Are there good reasons to prefer one option over the other?

Observe that hardware and software are mutable: that is, one can (with some work) implement many software functions in hardware if one chooses (this is done with special processors, such as those used in avionics [MS95]), or, more obviously, one can implement any given hardware function in software (as is done with RISC processors). Thus, for any given *system design*, there is a choice of what functionality to implement in software and what in hardware. In my view, this bolsters the argument that hardware and software can be considered as parts of a system: they are used mutably to implement various parts of the system design, and different parts of the design have the same logical type (whatever this may be – something like a description of functionality). However, this argument cannot be made for the specifications $\Pi$ and $\Psi$. A specification is in general some form of predicate with an argument of type *system*. It may or may not look like a 'description of functionality', and it does not appear intuitively to be mutable with hardware or software, in the manner in which these latter are with each other.

One may judge this analysis abstruse and having little to do with practical system engineering. In fact, it may be used in analysing arguments such as the following,[11] whose analysis will lead us to prefer option **c**:

*"This computational system failed. Its hardware didn't fail.*
*But the system is composed of hardware and software.*
*Therefore the software must have failed."*

The surface logical form of the argument as presented is:

$$\begin{array}{l} \mathit{Failed}(S) \\ \neg\mathit{Failed}(\mathit{hardware}(S)) \\ \underline{S = \mathit{hardware}(S) \oplus \mathit{software}(S)} \\ \qquad \mathit{Failed}(\mathit{software}(S)) \end{array}$$

The argument cannot be seen to be valid in this form because the conclusion does not follow from the premises. So, either the logical form of the premises doesn't exhibit enough to enable the inference, or there must be missing premises. The goal of a logical analysis is to analyse the forms of the assertions, and to fill in missing premises if necessary. We shall do so.

[11] This argument was suggested as a typical example of industrial failure reasoning by [And93], who, it should be remarked, does not endorse it!

The classification of failures in critical computer systems has been studied [Lut93a].[12] Over 95% of these failures were neither failures in the hardware, nor failures of the software to meet its specifications. They occurred in circumstances which were unaccounted for in the specifications and may reasonably be regarded as failures in the specification. Parnas counts specification in the category of 'documentation' [Par90]. As noted in Section 3.1, specifications seem to have a different logical type from that of software and hardware.

Should we include 'documentation' as *part* of a system? Parnas's proposal has the feature that what he calls 'documentation' intuitively seems all to have more or less the same logical type – that of assertions or descriptions. However, the role that different parts of the 'documentation' play in a failure analysis are different. For example, suppose the functioning of the hardware and software of a system is described by a detailed specification which I shall call the *design*. Suppose the documentation describing a design does not correctly represent the design. No actual system *failure* could be attributed directly to the mismatch between the actual design and the design document. To see this, suppose the design of the system allowed unwanted system behavior. It follows that either the design allows behavior that is inconsistent with the requirements, or the requirements allow behavior that is unwanted anyway. In the first case, the design does not match requirements, and in the second, the requirements don't match actual use. These are the failure conditions. The document that (correctly or incorrectly) describes the design plays no logical or causal role in explanation of the system failure behavior (although it might play a sociological role in explaining how the faults came to be overlooked by the design team in the first place), and it plays no necessary role in correcting that behavior (although it might play a role in the social process by which the system becomes modified to rectify its behavior). So I resist the suggestion to lump these different assertional components of system-building under the same rubric, 'documentation'.

We have demonstrated in Section 3.1 that the specification of a system plays a necessary role in assessing the correctness of the system. It should not then be surprising to find the specification playing a logical role in attribution of system failure. But the formal version of the failure argument given above does not reflect this. We could introduce the role of the specification in the ways envisaged in Section 3.1, considering the specification either as a part of the system, or to be an explicit parameter of the correctness predicate.

First, there's a distinction to make between 'failed' and 'faulty'. Roughly speaking, a *fault* is a feature of a system, and a *failure* is an event in which the system doesn't behave as desired – a fault causes a failure. A system which has a fault is *faulty*. A system that has failed has exhibited a fault, but a system that has so far not failed may still be faulty. A fault may be there, even though no behavior has yet shown it to be there. Further discussion of these terms is in Section 6. For the present purpose, I shall simply take the predicate *Faulty* to be a contrary to *Correct*, that is

$$Faulty(S) \Rightarrow \neg Correct(S)$$

and replace the predicate '*Failed*' by '*Faulty*' in the argument under discussion.

I now consider rectifying the argument under the options **c** and then **a**.

[12] I'm indebted to John Rushby for this reference.

### 3.5. The Specification as System Part

Under option **c**, in which the specification is considered part of the system, it is no longer true that the system is the mereological sum of hardware with software. The argument becomes

*"This computational system has failed to fulfil its expectations. Its hardware didn't fail. But it's composed of hardware and software, plus a description of its requirements. Therefore the software has failed."*

Formally, this argument is

$$\frac{\begin{array}{l} \mathit{Faulty}(S) \\ \neg \mathit{Faulty}(\mathit{hardware}(S)) \\ S = \mathit{hardware}(S) \oplus \mathit{software}(S) \oplus \mathit{specification}(S) \end{array}}{\mathit{Faulty}(\mathit{software}(S))}$$

The rephrasing is clearly lacking in that its conclusion is too strong. Cognisant of where faults in critical systems may lie, we could expect the argument to read:

*"This computational system has failed to fulfil its expectations. Its hardware didn't fail. But it's composed of hardware and software, plus a description of its requirements. Therefore either the software has failed or the requirements description failed."*

The formal version is

$$\frac{\begin{array}{l} \mathit{Faulty}(S) \\ \neg \mathit{Faulty}(\mathit{hardware}(S)) \\ S = \mathit{hardware}(S) \oplus \mathit{software}(S) \oplus \mathit{specification}(S) \end{array}}{\mathit{Faulty}(\mathit{software}(S)) \vee \mathit{Faulty}(\mathit{specification}(S))}$$

This indicates what premise to add in order to turn this into a correct argument, namely

$$\mathit{Faulty}(S) \leftrightarrow \mathit{Faulty}(\mathit{hardware}(S)) \ \vee \ \mathit{Faulty}(\mathit{software}(S)) \ \vee \ \mathit{Faulty}(\mathit{specification}(S))$$

Likewise, the addition of the premise $\mathit{Faulty}(S) \leftrightarrow \mathit{Faulty}(\mathit{hardware}(S)) \vee \mathit{Faulty}(\mathit{software}(S))$ would have turned the original argument into a valid argument. (If one is worried about the simplicity of this reasoning, note that any propositional argument at this low level of syntactic complexity is bound to look trivial.) But instead of this formula being included as a premise, the assertion was included that $S$ is a mereological sum of parts. This suggests that the missing principle on which the argument is based is that

$$[S = \mathit{hardware}(S) \oplus \mathit{software}(S) \oplus \mathit{specification}(S)] \quad \equiv$$

$$[\mathit{Faulty}(S) \leftrightarrow \mathit{Faulty}(\mathit{hardware}(S)) \ \vee \ \mathit{Faulty}(\mathit{software}(S)) \ \vee \ \mathit{Faulty}(\mathit{specification}(S))]$$

Clearly this generalises. The general principle is that if a system is the mereological sum of parts, and it fails, then at least one of the parts must have failed.

$$S = \oplus\{a_i \mid i \in 1..N\} \ \equiv \ \mathit{Faulty}(S) \leftrightarrow \bigvee_{i \in 1..N} \mathit{Faulty}(a_i)$$

Call this the *part-failure equivalence* condition. The axioms of mereology in Section 3.2 support the tenability of this equivalence scheme, in which $N$ varies, and which has to remain appropriate as systems grow to include more parts. It would be logically incoherent without the properties that the axioms ensure. The axioms ensure that $\oplus$ is an idempotent, commutative, associative operation, as is disjunction; and that if the system $S$ that failed is part of a larger system $T$, $S < T$, then $T$ may consistently also satisfy the part-failure equivalence condition.[13] For example,

$$[(T = P \oplus S \ \&\ S = R \oplus V)\ \&$$
$$(\mathit{Faulty}(T) \leftrightarrow \mathit{Faulty}(P) \vee \mathit{Faulty}(S))\ \&\ (\mathit{Faulty}(S) \leftrightarrow \mathit{Faulty}(R) \vee \mathit{Faulty}(V))]$$
$$\Rightarrow\ [T = P \oplus R \oplus V\ \&\ \mathit{Faulty}(T) \leftrightarrow \mathit{Faulty}(P) \vee \mathit{Faulty}(R) \vee \mathit{Faulty}(V)]$$

So, if the part-failure equivalence holds for $T$, and also for $S$ as a part of $T$, then it also holds for the finer decomposition of $T$ into $P$ and the parts $R$ and $V$ of $S$ (which are also of course parts of $T$). Which version of the part-failure equivalence is used will depend on the parts into which the system is decomposed. Picking irredundant parts is a good idea – for example, there would be no need to include a part $S$ such that $S = R \oplus V$ when $R$ and $V$ are already included.

### 3.6. The Specification As Argument – Considering Trusted Parts

One may hope for a similar analysis for the case in which the specification is an explicit argument. I use the notation $\mathit{specification}_S$ to denote the specification of $S$. Using the predicate $\mathcal{C}(\mathit{specification}_S, S)$, and noting that *Faulty* is a contrary, we may conclude that the expression that $S$ is faulty has the form $\mathit{Faulty}(\mathit{specification}_S, S)$. The original argument is written formally as

$$\begin{array}{l} \mathit{Faulty}(\mathit{specification}_S, S) \\ \neg\mathit{Faulty}(??, \mathit{hardware}_S) \\ \underline{S = \mathit{hardware}_S \oplus \mathit{software}_S} \\ \quad \mathit{Faulty}(??, \mathit{software}_S) \end{array}$$

I shall leave the *Spec* argument place in the *Faulty* predicate undetermined for the moment. The part-failure equivalence principle would read

$$[S = \oplus\{a_i \mid i \in 1..N\}] \ \equiv\ [\mathit{Faulty}(\mathit{Spec}_S, S) \leftrightarrow \bigvee_{i \in 1..N} \mathit{Faulty}(??, a_i)]$$

Alone, this does not suffice to turn the argument into a valid argument, since the specification is no longer a part of the system and so the modified part-failure equivalence principle doesn't provide the needed premise: taking the original argument and replacing $S = \mathit{hardware}_S \oplus \mathit{software}_S$ by

$$[\mathit{Faulty}(\mathit{Spec}_S, S)] \ \Leftrightarrow [\mathit{Faulty}(\mathit{Spec}(??, \mathit{hardware}_S)\ \vee\ \mathit{Faulty}(\mathit{Spec}(??, \mathit{software}_S))]$$

yields only the same conclusion – the original conclusion (and faulty because too strong) to the argument. With this form of the predicate, the easy way of saying that the specification itself might be at fault seems to elude us. Maybe a deeper analysis will suffice.

One of the first questions is: what should be the arguments replacing '??' in the assertions $\mathit{Faulty}(??, \mathit{hardware}_S)$ and $\mathit{Faulty}(??, \mathit{software}_S)$? If they are identical to $\mathit{Spec}_S$, the specification of the whole system, then one should search for a reasonable intuitive interpretation of what the assertion means, that the hardware failed to fulfil the system specification. The hardware is only part of a system, and the specification could use terms that refer to parts

[13] Note, however, that there are certain fault-tolerant system designs in which a subsystem may fail but the whole does not, so the parts considered to fulfil the equivalence must be selected with care.

that aren't in the hardware, and these terms would either have to remain uninstantiated, or be quantified over.

For example, suppose your bicycle has only a front wheel, and its specification states, amongst other things, that it shall have a front and a rear wheel. Consider the front wheel as a part, and the term denoting the rear wheel as uninstantiated. This cannot form an assertion that the front wheel succeeds or fails to fulfil the specification, since there are uninstantiated terms. And no such question can be asked of the 'missing' rear wheel, because there is no such thing! Consider now the alternative, that the term standing for the rear wheel becomes existentially quantified. *There exists a rear-wheel such that ...* . But in fact there isn't, so the specification is unfulfilled (is false). But the specification for the front wheel intuitively is supposed to be true! – the front wheel does fulfil its obligations as a part of the system. So this option doesn't provide us with an appropriate assertion either. We remain without a suggestion as to what the argument should be or how it should be interpreted in an assertion $\mathit{Faulty}(??, \mathit{part\text{-}of}_S)$

Since *Faulty* is a contrary to $\mathcal{C}$, one could rather ask the converse question: what would be the missing parameter in an assertion of correctness, $\mathcal{C}(??, \mathit{hardware}_S)$ and $\mathcal{C}(??, \mathit{software}_S)$? In this form, one is tempted to say that an assertion of correctness of part of a system could consist in saying that the part satisfies that part of the specification *restricted or relativised to itself* in some manner other than by quantifying over the variables not in the domain of the part. In order to complete the argument we're investigating in a manner similar to the way in which we were able to complete it for the form $\mathit{Correct}(S)$, we need to be able to say that a failure in the system must be a failure in a part of the system to satisfy the specification 'relativised to itself'. This is not so clearly possible – it's folklore that one can have components which work wonderfully as components but which have complex undesired interactions when composed together. I shall nevertheless attempt it in Section 3.7.

No matter what we may decide for the missing arguments to *Faulty*, a major problem in system analysis is how to compose specifications [AL93]. One outcome of the research into composition may be that one will be able to define the design of a system as the composition of the specifications of its components. This helps in a failure analysis, as demonstrated by [Gen84] and [Rei87] (with a correction in [GSW89]) for relatively straightforward hardware components. However, until we have a definition and clear idea of the properties of such a general composition, we will not know whether, for example, the composition contains more information than is contained in the system specification (e.g. the composition logically implies the system specification but is not logically implied by it).

It seems thus that there are significant problems to be solved before one can add the 'missing premises' to the argument interpreted according to option **a**:

1. We cannot obtain the appropriate conclusion as under option **c** with similar reasoning;
2. The first argument to the *Faulty* predicate remains unclear.

## 3.7. Some Decomposition Principles for Specification-as-Argument

However, let us consider for the moment the second problem solved somehow, and denote the missing first argument to *Faulty* as $\mathit{Spec}_P$ when the second argument is $P$. Consider the *prime facie* plausible assertion that correctness of the whole system implies the correctness of any part:

$$\mathcal{C}(Spec_S, S) \;\&\; P < S \;\rightarrow\; \mathcal{C}(Spec_P, P)$$

This condition is simply not satisfied for certain fault-tolerant systems. Certain significant parts of the system are designed to satisfy their specification even though a particular part of them is faulty. It is therefore more appropriate to consider only 'trusted' parts of a system. These are parts such that, if they fail to fulfil their function, the system may fail to fulfil its function. It would be an engineering decision to say what the trusted parts of a system would be. Suppose, without loss of generality, that the trusted parts are selected by a predicate $A$, and constrain the relative correctness assertion to be

$$\mathcal{C}(Spec_S, S) \;\&\; A(P) \;\rightarrow\; \mathcal{C}(Spec_P, P)$$

This implies the assertion

$$\mathcal{C}(Spec_S, S) \;\rightarrow\; \bigwedge_{A(P)} \mathcal{C}(Spec_P, P)$$

We also need to ensure that the trusted parts compose the system,

$$S = \oplus\{P \mid A(P)\}$$

Supposing we have figured out what the 'composition' operator for specifications of trusted parts is supposed to be (let's denote it by $\otimes$)[14], we can expect that if the system is correct, and the specifications of all the trusted parts are correct, that when the composition of the specifications of the individual trusted parts is taken, the specification of the whole system is satisfied. That is to say, the composition must imply the specification of the whole system. This must hold no matter what the decomposition into trusted parts, so long as the system is the mereological sum of the parts. If not, and the composition does not imply the system specification, then the system specification may be false in circumstances in which the composition is true. That is, even though all the trusted parts fulfil their joint specification, the system may fail to fulfil its specification. This is contrary to our intuition of what the composition operator should satisfy. The condition that the system specification cannot impose a more stringent logical condition on the system than is collectively imposed on the parts is given by:

$$S = \oplus\{P \mid A(P)\} \;\Rightarrow\; (\otimes\{Spec_P \mid A(P)\} \;\rightarrow\; Spec_S(S))$$

Thus, the condition on the overall system can be formulated as

$$\begin{aligned}&\mathcal{C}(Spec_S, S) \;\&\; (S = \oplus\{P \mid A(P)\})\\ &\qquad \Rightarrow\; (\otimes\{\mathrm{Spec}_P \mid A(P)\} \rightarrow Spec_S(S)) \;\&\; \bigwedge_{A(P)} \mathcal{C}(Spec_P, P)\end{aligned}$$

This yields an extra disjunction in the conclusion to the failure argument, namely that

$$\otimes\{\mathrm{Spec}_P \mid A(P)\} \not\rightarrow Spec_S(S)$$

[14] Using a single symbol for the operator, no matter what the arguments, suggests psychologically that we have a uniform rule for computing it. Certainly, if we use it in practice, we must have such a rule. However, for the argument presented here, the operator can be considered generally as denoting the association of the correct system specification, whatever that is, to the arguments given. As such, it must exist – here is a sketch of the argument. Such a *relation* must exist by general logical principles. But there may be two or more system specifications corresponding to a given specification of a collection of parts. (Note that the axioms of mereology entail that there is only one system corresponding to a given collection of parts – namely the mereological sum.) Now use the Axiom of Choice to obtain an operator from this relation.

which is equivalent to

$$\otimes\{\mathrm{Spec}_P \mid A(P)\} \;\&\; \neg Spec_S(S)$$

which says in particular that there is a possibility that the composition of the specifications of the parts are not strong enough to guarantee that the specification of the system as a whole is fulfilled. This is certainly a point that has to be addressed in any verification that separates a system into parts, but this still does not give us the original missing disjunct in the conclusion of the failure-argument, which problem we're still left with.

The ability to formulate the failure-argument simply, and simply to supply the correct conclusion containing a disjunct stating that the specification may be lacking, is a powerful reason for preferring option **c** when reasoning about systems and failure. Option **a** can only become a viable choice if

- we find a clear way to supply the 'missing disjunct' to the conclusion under this option;
- we determine what the terms $Spec_P$ for $P < S$ refer to;
- given any $S = \oplus\{P \mid A(P)\}$, we determine what the composition $\otimes\{Spec_P \mid A(P)\}$ denotes.

When these problems are solved, we may at that time choose to reevaluate the present evidence in favor of option **c**. In the meantime, the current argument establishes that it is preferable to consider the specification as part of the system, as in option **c**.[15]

## 4. Matching

As a consequence of the discussion in Section 3, we can consider the specification, the 'requirements description' to be a part of the system itself. I now turn to the question of what it could mean, that the "requirements description failed". One interpretation is that the actual expectations of system behaviour, the *what-its-use-is-supposed-to-be*, are not correctly reflected in the formal or informal requirements specification. Let's call the former the *supposed-use*, and the latter the *formal-spec*.[16] Using this vocabulary, we can now say that for the requirements specification to fail means that *the formal-spec does not match the supposed-use*. I investigate the technical meaning of 'match' in Section 4.1. For the moment, the intuitive sense suffices.

For example, the supposed-use of a modern transport aircraft such as an Airbus A320 or a Boeing 777 should include that it shall fly without crashing, and always land and take off successfully. We can tell that these are components of the aircraft's supposed use, by supposing (counterfactually, we hope) that some such airplane crashes while the pilots are doing everything they should be doing. We're likely to find ourselves saying 'Well, the airplane shouldn't have crashed in *those* circumstances', thereby stating part of its supposed-use with perfect hindsight. Notice that the supposed-use isn't a thing that is itself correct or not correct. It's posited, it simply *is*. One can point to it, but one cannot say it's right or wrong in itself. In contrast, one *can* say that a formal-spec is incorrect, by matching it against what one knows of the supposed-use and finding discrepancies.

[15] Choosing option **c** does not mean that I assert that the decomposition of a system into three components, $S = hardware_S \oplus software_S \oplus specification_S$ is either complete, or even the most desirable general decomposition. I leave it open that may be other components of a system which it is necessary to consider when the system fails.

[16] Even though, in everyday software development, the formal-spec is often not very formal.

One inconvenient feature of supposed-use is that various aspects of it may only be discovered with hindsight. So one would be truly surprised to find all these expectations which I am calling the supposed-use explicitly written in the requirements specification, the formal-spec, for any device with a moderately complex working environment such as an airplane. But one goal of the requirements-capture process is somehow to express as much of the supposed-use as possible into more technical terminology in the formal-spec, in such a way that formal reasoning may be used to connect these formally expressed requirements with the design and other formal specifications of the system. I recall the evidence that most failures of systems designed for high reliability were traced to a misfit between ultimate user expectations (the supposed-use) and the 'official' requirements specification (the formal-spec) [Lut93a] [Lut93b].

## 4.1. The Logical Relation Between Supposed-Use and Formal-Spec

What do I mean by '*match*'? Systems may be described by formal languages using the syntactic conventions of formal logic (after the Frege Principle and Russell Criterion). A language may contain predicates and terms referring to internal parts of a system, and it may contain predicates and terms referring to the behavior of the system in its environment. What is the intended meaning of assertions in the language? A system implicitly defines a collection of behaviors, namely, all the ways in which the system can behave. Whatever else it may mean, therefore, a formal-spec defines a class of sets of behaviors, namely those sets of behaviors of systems which satisfy the formal-spec. Similarly, a supposed-use description corresponds to a class of sets of behaviors. The languages needed to describe those behaviors adequately may be different.

It is standard in system specification to interpret the definition of a system as defining a single set of behaviors. If a definition of a system defines a non-trivial class of sets of behaviors (i.e. a class with at least two members), then the specification can rightly be regarded as *ambiguous* – there is more than one set of system behaviors that the specification allows – and I shall call it so. The relation of *match* compares a formal-spec with a supposed-use. Suppose both are unambiguous. Subtle relations have been defined, such as the relation of Refinement Mapping [AL91] (coupled with logical implication), or the relations of Modal Process Logic [Bru93]. Both of these methods require that the supposed-use is formally described. They are also not symmetric relations, and an intuitive consideration of *match* would suggest that it should be symmetric (that whatever the order in which its arguments are written, its truth value is the same).

Intuitively, the purpose of 'matching' formal-spec and supposed-use is that we intend that they shall say 'the same thing'. That suggests that they share the same language, and as far as possible denote the same set of behaviors. When can they be said *not* to match? Suppose that a system satisfies its formal-spec. How can the formal-spec not match the supposed-use? Intuitively, only if there is some system behavior $B$ which is outside the range of behaviors allowed by the supposed-use. If the supposed-use is not ambiguous, it defines a unique set of behaviors and the behavior $B$ must be outside this set. If the supposed-use is ambiguous, then it is somewhat harder to define what is meant by the 'range of behavior' allowed by the supposed-use. If $B$ is outside of *all* of the sets in the class of sets of behaviors corresponding to the supposed-use, then it is clear that $B$ is outside the 'range of behaviors' allowed.

I shall be interested in exhibiting examples in which the formal-spec and supposed-use match. I therefore choose my cases to be unambiguous, that is, both the formal-spec and the supposed-use define a unique set of behaviors. How should these behaviors be related? Considering unambiguous formal-spec and supposed-use, one can see that the formal-spec only matches the supposed-use, when there is no behavior consistent with the formal-spec that is inconsistent by the supposed-use. That is to say, the set of behaviors allowed by the formal-spec is a subset of the behaviors allowed by the supposed-use. The relations of [AL91] and [Bru93] both satisfy this condition, although since they are primarily concerned with comparing a design specification (an 'implementation') with a requirements specification (rather than two specifications supposed to be saying the same thing), they include extra apparatus to cope with the difference in language normally to be found in different sorts of system specifications.

If the formal-spec actually logically entails the supposed-use, then the behaviors described by the former are indeed a subclass of the behaviors described by the latter. That is, for unambiguous $Spec_S$ and $supposed\text{-}use_S$,

$$match(Spec_S, supposed\text{-}use_S) \;\Rightarrow\; [\forall T : Spec_S(T) \Rightarrow supposed\text{-}use_S(T)]$$

Furthermore, if $Spec_S$ and $supposed\text{-}use_S$ are written in the same language, and we accept that *match* in this case is symmetric,

$$match(Spec_S, supposed\text{-}use_S) \;\Leftrightarrow\; match(supposed\text{-}use_S, Spec_S)$$

then we can conclude that *match* is a predicate of predicates including the relation of logically equivalence on predicates:

$$match(Spec_S, supposed\text{-}use_S) \;\Rightarrow\; [\forall T : Spec_S(T) \equiv supposed\text{-}use_S(T)]$$

May this implication be reversed? Certainly, in the case in which the formal-spec is *identical with* the supposed-use, they must satisfy both the extensional condition (that there is no behavior allowed by the one that is inconsistent with the other) as well as any intentional conditions that refer simply to differences in syntax. I can see no argument that suggests that identical formal-spec and supposed-use may not match. Further, if one has in mind a supposed-use for a system that any behavior is allowed, that is, a logically-true requirement, then any logically-true predicate should suffice for the formal specification, as suggested implicitly in Section 2.

The relation of matching deserves some further logical investigation. However, the present discussion suffices for establishing the final argument that the example in Section 2 does the job required. That argument consists in asserting, for S the system in Section 2, that $match(\Pi(\mathtt{S}), supposed\text{-}use_{\mathtt{S}})$.

### 4.2. For Some Systems, There Is No Misfit

The objection was considered in Section 2.4, that for the system S of Section 2.1, the specification might not be correct, so that the proposition S *is correct* might not be the same thing as the proposition S *implements its specification*, and we have only proven the latter in Section 2.1. We counter the objection here by noting that S *is correct*.

If S implements its specification but nevertheless S is not correct, then it must be the case, in the vocabulary of Section 4.1, that the formal-spec doesn't match the supposed-use. We considered in Section 4.1 how, in the case of supposed-use chosen to be a logically-true predicate, a formal-spec, namely any logically-true predicate, may be chosen in order to fulfil the criteria of matching. Therefore, they match (from Section 4.1). Therefore, the specification is correct, S implements the specification, and so S *is correct* and we have countered the objection.

These considerations yield some insight into for which systems it might be possible to demonstrate matching between supposed-use and formal-spec. For systems for which the supposed-use is logically or mathematically precise, we may take the supposed-use and formal-spec to be identical. Such is the case with the GCD calculation of Section 7. In any case in which we can establish the logical or *a priori* status of supposed-use, we stand a chance of being able to define a formal-spec which matches, and thereby eliminate one source of error in building such systems.

Note that proving the logical equivalence of the supposed-use with the formal-spec does not, by itself, guarantee the correctness of a system. For example, such a system may be radically altered by an explosion, and the system in its transmuted state may fail to satisfy the formal-spec, and therefore equivalently the supposed-use. But for System S, this does not destroy the correctness property, because its supposed-use is logically true, and nothing whatever that happens to a system can cause it not to satisfy a logically true supposed-use.

### 4.3. This Even Happens With Bicycles

One should note that the phenomenon of matching supposed-use and formal-spec has little or nothing to do with software, although the process of matching formal-spec against supposed-use occurs traditionally in 'software development'. All systems, whether they involve software or not, have some sort of expectations they should fulfil, against which their performance may be measured, and they may 'succeed' or 'fail' (... to fulfil their expectations).

Consider a bicycle, and a new worker in the bicycle-building factory, who is given instructions on how to join components together. Spokes fit into hubs, rims fit on spokes, tires fit on rims, wheels fit here on the frame, they also fit here ...... He makes the bicycle, but it can't be ridden because he has only attached one wheel. The requirements happened to state which bits go in which places, but failed to state that *all* the right bits must go in *all* the right places.[17] He has built a machine to formal-spec, but formal-spec and supposed-use don't match. And there's no software in sight. So we have no plausible reason to call this kind of failure a 'failure concerning software'.

### 4.4. Interlude – A Critique of Two Common Arguments

This analysis sheds some light on a couple of arguments often employed to attempt to show that systems may not be proved correct. I'll address them by the name of their most well-known advocate.

**Fetzertion 1:** An implemented system is a physical thing.

According to the proposition discussed in Section 3.1, this may not be the whole truth. If we include the supposed-use in with the definition of *system* as a formal-spec, then the system is only partially physical. It's also partially non-physical. The expectations of its function are not physical objects, but they are part of the system.

[17] More basically, they fail to state that the bicycle should be rideable.

On the other hand, taking option **c**, in which we don't include the supposed-use in with the physical parts, although the 'implemented system' may be considered to be physical, proving the system correct involves comparing some non-physical things, namely observations of behaviour, with the supposed-use. This may only be done via the intermediation of *descriptions* of this behaviour and propositions expressing supposed-use. Looking at properties of physical parts alone does not suffice. This leads to the second set of considerations.

**Fetzertion 2:** The future behaviour of physical things cannot be predicted using logical arguments only. Such behaviours are not the kind of things that admit of proof.

In talking about system behaviour, or comparing it with something else, one does not grasp the behaviour directly (since it comes and goes), but through a *description*. This is because one is considering timeless properties of behaviour - one wants to make observations of the behaviour and refer to those observations at some later time. So let's give ourselves more room, and talk about *descriptions of behaviour*, rather than just behaviour.

The Fetzertion rests on the intuition that we predict individual behaviors, rather than properties of behaviors. However, predictions are usually made through a description of some sort. And many behaviors may satisfy even a description thought to be narrow – for example, behaviors which exhibit features which are not talked about in the description (e.g., which are not considered relevant). Predictions generally are descriptions, rather than specifications of individual behaviors.

If we consider descriptions of behaviour which a system may contingently satisfy, such as behaving *this* way rather than *that other* way, then there are undoubtedly some such behaviours which cannot be predicted with certainty. This is roughly what 'contingent' means.

There are descriptions of behavior which are logically true. It may successfully be predicted that a system will satisfy any logically-true specification. Suppose there are descriptions which are in fact always true, even although they might not be logically true, for example, descriptions which assert, amongst other things, that there exists an infinite set.[18] Similar considerations to those for logically-true predicates show that it may be asserted that any future behavior will satisfy such a predicate.

There is also a quirky possibility which one may consider. Suppose one believes in (Leibnizian) determinism, that is, that the future states of the universe are uniquely determinable from any past configuration. Then presumably one can predict the future states even of our system exhibiting 'contingent' behaviour by knowing sufficiently much about the past state of the world. Most philosophers and scientists are not determinists, but nevertheless respectable versions of determinism have been offered. A determinist argument directly contradicts Fetzer's conclusion, even for contingent behavioural descriptions.

Thus, whether 'future behavior' can be predicted with certainty rests on the logical or metalogical status of descriptions under which one considers that behavior, including whether one is prepared to allow 'synthetic *a priori*' assertions, or Leibnizian determinism. It is the kind of question that analysis shows to have less and less precise a meaning the more one considers the possibilities.

---

[18] This is often considered as an example of a non-logically true but still *a priori* assertion in mathematics. Concerning descriptions of behavior, virtually any such description which which rests upon the (classical) mathematics of real numbers will fail if the existence of an infinite set is denied. However, see the works of Mycielski (*op. cit.*) for some ways around this.

## 5. How Predictions May Be Rigorous

We now know *that* certain descriptions of future system behaviour can be proved. But if the only such descriptions that may be proved consist of those that are logically true, then that would only make Pyrrhus happy. There are some more interesting kinds of system descriptions which I claim may be proved to be true. I shall give a contrived example of provably safe engineering, and discuss how it might come to have that feature, in order to argue for the expansion of the class of systems/system properties which may be provable.

### 5.1. An Actual Safety-Critical System Example

It is of crucial importance that the systems of fly-by-wire passenger aircraft such as the Airbus A320 or Boeing 777 are *safe*. I'll sketch a way in which the software (not the whole system this time) of an actual airplane may be *proven* to satisfy its safety criteria (this will involve putting the system to not very interesting a use).

The reason for introducing this example is to clarify further the relation between logical truth, physical systems, and the supposed-use/formal-spec relation. An extra assumption is needed, that the requirements specification contains no *liveness* criteria.[19] This assumption is hardly useful in general – just for the example.

Obtain a *specific* physical airplane. We tow it to the ramp, and commit to leave it there without touching it, moving it, starting it, or altering its state in any way by intervention for a certain length of time, say, five hours. Suppose something bad happens in this five hours (the system fails to satisfy a safety expectation), e.g. a piece falls off and hurts a pedestrian who shouldn't have been there. Do we go looking to retest the software in this case? No. But the more interesting point is, would we *ever* (during this inert period) go looking for the fault in the software? Equally, no. We locate the fault in hardware (or requirements — maybe the incident should have been taken into account by the requirements-capture team), but we don't locate the fault in software *because the software's not doing anything, so it cannot contribute to the state change that caused the violation of the safety property.* This last principle of *software-irrelevance* follows directly from a principle which I shall call the *safety-invariance principle*, introduced below. The software-irrelevance principle corresponds to the following valid argument of temporal logic:

$$\frac{\begin{array}{l}\neg \mathit{Failed\text{-}state}(\mathit{Initial\text{-}State}_S) \\ \Box(\mathit{State}(S) = \mathit{Initial\text{-}State}_S)\end{array}}{\Box\neg \mathit{Failed\text{-}state}(\mathit{State}(S))}$$

where in this case $S$ is instantiated as $\mathit{software}_{airplane}$; $\mathit{Initial\text{-}State}_S$ is a rigid variable (one that does not change its value from state to state) and *State* is a state function.[20] The fact that the software-irrelevance principle corresponds to a valid argument of temporal logic does not by itself entail that the software-irrelevance principle is a metaphysical truth. The semantics used for the interpretation into temporal logic must first be shown to be appropriate, for example, that a state is an appropriate object that may be denoted by a term, that

[19] A safety criterion, after Lamport [Lam77], is a condition that "nothing bad will happen". A liveness criterion, by contrast, is a criterion that "something good will eventually happen". There are formal characterisations of both of these criteria, along with theorems concerning their relation [AS87].

[20] This formulation is in the Temporal Logic of Actions (TLA) of Lamport [Lam94].

the predicate of failure can take an argument that denotes the state, and that the logic is extensional, that is, that the assertion that a state is failed depends only upon the state and not upon the term chosen to denote the state. So, without these principles having been established, a translation of the software-irrelevance principle into a logical truth of temporal logic serves only to show it is consistent with and follows from these general ontological and semantic principles of temporal logic, and not to establish it outright. An argument for the truth of the ontological and semantic principles of temporal logic to talk about action systems has been attempted in [Lad94]. However, principles such as the software-irrelevance principle have been used to justify these principles, rather than the other way round.

### 5.2. How Non-Logical Principles Become Inviolate

Is it possible for the software-irrelevance principle to be violated? It has a general form; it doesn't seem to depend on any particular observation of system behaviour. And it follows from the *safety-invariance principle*, which is that *if a part $P$ of a system $S$ is in a safe state with respect to the system, and the environment doesn't change, and $P$ remains in that state, then $P$'s state remains safe*, and which may be expressed in a logical syntax similarly to the above. This principle itself follows from a more general invariance principle, which is that under the same assumptions that 'nothing changes', a system state does not spontaneously lose or gain properties (I don't spell out this principle in detail, but see [Lad94] for an argument that it is *a priori* if anything is). In this case, the property we are interested in is a *safety* property. An argument for the general invariance principle would follow from the meanings of the terms 'state', 'property of a state' and so forth. It asserts properties of the relation between these terms *that are semantically inviolable*, rather as a logical truth is inviolable. It's semantically inviolable *because we can't see how things could be otherwise.* But the invariance principle is not a logical truth, so how can this be? One possible answer is that we *commit* ourselves to maintaining the principle as inviolable. I now must explain what kind of answer this amounts to.

This explanation holds that any potential violation of the principle must be accomodated by, for example, changing (or 'clarifying') the meanings of the terms involved. For example, we can't *mean* 'environment', 'state', 'state property' *unless* we also agree to accept the invariance principle that comes with them. This kind of approach to clarifying the use of terminology in context is related to a thesis of Duhem that has been argued eloquently by Quine (see e.g. [Qui60]).[21]

So such principles may have the certainty of mathematical truths in part because we *commit* to their continuing truth in the future. We would rather adjust to a potential violation by changing slightly what we had taken 'environment' to denote, than by giving up the invariance principle.[22] A commitment *now* to a principle to hold in the future may indeed be made, and may be kept or violated. But when a commitment is violated, we do not locate the fault in the *meaning* of what is committed to, but in the sincerity of the actor. If I break a promise, then it is inappropriate to say 'Oh, I discovered the promise was wrong', since promises are not the kind of things that are wrong or right (unlike declarative statements).

---

[21] However, Quine would repudiate talk of the 'meaning' of an individual concept or word except as an allegory.

[22] Note that people do this in light conversation frequently, when they set more store by 'not being wrong' than by how things really are. Our suggestion is not philosophically as trivial as this.

They are the kind of things that are kept or broken. The investigation of different kinds of acts such as promises, declarations, commitments, that may be performed with language is known as the study of *speech acts*, originated by Austin [Aus75] [Aus78], and developed particularly by Searle [Sea69] [Sea79], who calls these acts of commitment *Commissives*.

Quine does not recognise certain categorical distinctions among sets of statements we take to be true, such as '*a priori*' truths and 'contingent' truths. He has argued that the truth or falsity of statements has a similar semantic status, no matter how the statements are classified. When there are conflicts or misfits in our ways of using language, something has to give.[23] But we are more committed to retaining some kinds of statements (the logical truths) than others (*The sun will rise tomorrow*) and yet others (*That driver will have an accident if he drives like that all day*), because some kinds of revision cause greater reverberation in our conceptual schemes (*Oh, dear, I suppose the law of non-contradiction is wrong!*) than others (*Oh, dear, maybe the mail didn't arrive today after all!*).

Supposing that collections of statements form an interconnected web like this, with varying degrees of freedom in its various parts, gives us one way of seeing how we may predict that the software-irrelevance principle will not be violated in the future by our parked airplane. We are not making a prediction in the contingent sense, but instead undertaking not to violate the principle; if necessary, to push and pull the web of interconnected statements in a different manner to preserve the principle, adjusting other definitions, terms and truth values in order to take up the slack. (Equally, if this account of statements-as-a-web is mistaken, that does not destroy our argument concerning the inviolability of the airplane safety property, but merely a suggested *explanation* of how it happens.)

## 5.3. Where Do We Locate Faults?

I suggest that we may detect what kinds of commitments we have implicitly made, and what kinds of meanings we implicitly chose for our terms, *by looking where we locate the faults*, as we did for the airplane example. If we had heard that an unspecified safety violation had occurred with the parked airplane, we wouldn't rush to find our software experts, we'd find our hardware and structures experts first, maybe our requirements specialists. This is because we cannot violate the invariance principle. Even if *we* decided that the invariance principle may be violated, the software experts we find, who are themselves committed to the invariance principle, would be unable to understand what we wanted them to do. (Compare with the case of someone who tries to live a life violating the principle of non-contradiction, as may happen in cases of severe schizophrenia. Is there any way we can truly 'understand' such a person's world view?)

Looking where we locate the faults therefore helps us elucidate our commitments to principles such as the principle of invariance. It shows us how, and in what way, we may push and pull the web of meanings, and which parts we choose to remain fixed. This may help us to explain how physical things may be proven to satisfy certain properties. Notice, though, that properties may not be chosen at random to be violated or not. The web imposes constraints, and we can describe those constraints. However, if one accepts arguments similar to those of

[23] e.g. *But we retain a wide latitude of choice as to what statements of the system to preserve and what ones to revise; any one of many revisions will be sufficient to unmake the particular implication which brought the system to grief. Our statements about external reality face the tribunal of sense experience not individually but as a corporate body.* [Qui52]. See also [Qui61].

Quine, ultimately we have only limited means to try to understand the world view of someone who, when the postman doesn't call as expected, chooses instead to give up the law of non-contradiction rather than his belief (to us, false) about the postman having arrived.

### 5.4. Safety and Liveness

A last practical note before we leave this example. Modern industrial methods of software validation, especially on systems such as found in an Airbus [DM93], are particularly concerned with establishing safety properties (if they try to establish any properties at all!). For example, if a system may be shown to be finite-state, safety properties may be checked by investigating each individual state. Liveness properties are much harder to establish (see [Lad95a] for a formal example). For that reason, they may be tested in a less thorough manner during system validation. But our example shows that liveness properties are the most interesting and crucial part of a system specification. Our airplane doesn't fly, so it doesn't transport passengers, make money for its owners, or serve much useful purpose at all (except as a monument to the philosophy of specification).

## 6. Failures, Faults and their Logical Interconnections

In Section 3.4, I noted a distinction between 'failed', that a failure event has occurred, and 'faulty' that there's something not right with the system. A fault causes a failure, and a system in which a failure has occurred is a system which (in the past) has failed. I define Failed($S$) to mean that a failure event concerning system $S$ has occurred. Faulty($S$), as before, is a contrary to correctness.

There are failures which are not necessarily recognised as faults – so-called 'transient' failures in on-board aircraft systems are of this type: for example a stochastic event such as a cosmic-ray impact caused a bit-flip – and the consequences are often rapidly rectified in the course of normal operation (the affected register or memory location is reset). Failures which are rectified without action on our part within the course of time do not necessarily cause us to conclude that the system itself is faulty. A system which is in a state in which it is experiencing the results of a transient failure I shall call *Transiently-failed* and use the axiom

$$Failed(S) \equiv \mathit{Transiently\text{-}failed}(S) \;\vee\; Faulty(S)$$

It follows that a failure which is not a transient failure is one which is not self-rectified in the course of time. Failures which are not transient failures lead to permanent disablement of the system until they are rectified, and therefore a non-transient failure may be characterised as one which persists. I shall apply the predicate *Faulty* to a system which is exhibiting such a non-self-rectifying problem, and capture this property by the axiom of temporal logic[24]

$$\Box Failed(S) \;\Rightarrow\; Faulty(S)$$

in which $\Box$ is the '*always*' operator of temporal logic: $\Box A$ is true, where $A$ is an assertion, if and only if $A$ is true in all states from now throughout the future. Also, since *Faulty* is a persistent property of systems, I shall take it to satisfy the axiom

[24] The reader may note that I am committed to calling *Faulty* a system which experiences a sequence of potentially unrelated transient failures, one after the other for ever.

$$Faulty(S) \;\Rightarrow\; \Box Faulty(S)$$

From now on, I shall also take *Faulty* as the converse of *Correct*, namely

$$Faulty(S) \;\Leftrightarrow\; \neg Correct(S)$$

We have considered the following valid argument of temporal logic:

$$\frac{\begin{array}{l}\neg \mathit{Failed\text{-}state}(\mathit{Initial\text{-}State}_S)\\ \Box(\mathit{State}(S) = \mathit{Initial\text{-}State}_S)\end{array}}{\Box\neg \mathit{Failed\text{-}state}(\mathit{State}(S))}$$

where $\mathit{Initial\text{-}State}_S$ is a rigid variable and *State* is a state function. I have noted that this principle is supported by an ontology of system states. If systems have states, we may connect the argument to the current definitions of failure by the equivalence

$$\mathit{Failed}(S) \;\equiv\; \mathit{Failed\text{-}state}(\mathit{State}(S))$$

With this definition, we may deduce from the above argument and the other definitions that the following argument is valid:

$$\frac{\begin{array}{l}\neg \mathit{Failed\text{-}state}(\mathit{Initial\text{-}State}_S)\\ \Box(\mathit{State}(S) = \mathit{Initial\text{-}State}_S)\end{array}}{\mathit{Correct}(S)}$$

which yields the conclusion that we drew from intuition, that the problem with the software-inert airplane was a hardware problem. This is a pretty simple example, but it does show how conclusions which we individually draw by intuition may instead be founded on more general principles with wider justification.

I wish to avoid a discussion here of how my terms relate to the standard engineering terms. It suffices that they are roughly consistent with the standard terms, and that the axioms define just the properties of these terms that I shall use in the final example. If the reader does not concur with my choice of terms, she is free to use her own.

## 7. Proving Software Correct

Neither my fridge nor even the software-safe parked airplane are interesting examples of systems, part or all of which are provably correct. Using the tripartition of computing systems into requirements specifications, software, and hardware, I turn to an example of practical consequence.

Is it possible to prove the match between formal-spec and supposed-use for interesting examples? I consider an example due to Boyer and Yu [BY92] [Yu93]. The specification and proof of this example in `Nqthm` is available in the note [BY95], adapted from [BY92]. It shows a specification of a GCD calculator that employs the Euclidean algorithm, compiled into assembly code for an MC68020. The specification of the MC68020, which runs to about 100 pages of formal text, is omitted. Also omitted is the proof that the gate-level implementation of the MC68020 satisfies this specification. There is a proof in Nqthm (Boyer and Moore's computational logic [BM88]) that the given assembly code correctly implements a GCD function on the MC68020. Thus the assertion of correctness is the conjunction of five steps:

$$\mathit{Correct}(\mathit{GCD\ system}) \quad \Leftrightarrow$$

$$\begin{aligned}&\mathit{hardware} \models \textit{gate-level implementation} \quad \& \\ &\textit{gate-level implementation} \vdash_{RM} \mathit{MC68020\ code} \quad \& \\ &\mathit{MC68020\ code} \vdash_{RM} \mathit{GCD\ in\ Assembler} \quad \& \\ &\mathit{GCD\ in\ Assembler} \vdash_{RM} \mathit{GCD\ spec} \quad \& \\ &\mathit{match}(\mathit{GCD\ spec}, \mathit{GCD})\end{aligned}$$

where $\vdash_{RM}$ means that there is a refinement mapping [AL91], an assignment of state functions of the left-hand side to variables on the right-hand side, so that the formula is provable in logic with the state functions of the Refinement Mapping replacing the variables on the right-hand side. Boyer and Yu omit the first two steps, and prove the next two. Let us assume that the second step is true. It is left to see whether *hardware* $\models$ *gate-level implementation* and whether $\mathit{match}(\mathit{GCD\ spec}, \mathit{GCD})$. The first step I shall relate to the assertion $\neg\mathit{Faulty}(\mathit{hardware})$. The next three constitute a proof that *gate-level implementation* $\vdash_{RM}$ $\mathit{GCD\ spec}$ because $\vdash_{RM}$ is transitive [AL91]. What remains is the assertion of *match*.

### 7.1. Proving the GCD *Software Correct

I define *software to be all the specifications/programs in the second through fourth steps above. *Software therefore includes the hardware architectural definition and the description of the gate-level realisation of the hardware design, as well as the assembly code translation of the 'high-level' software. *Software does not include the supposed-use, although it does include the formal-spec. *Software amounts to the entire design of a computing system built to satisfy a given formal-spec. A proof of correctness of *software is a proof that the gate-level behaviour of this design may be interpreted in a canonical way such that it may be proven (i.e. it is a theorem) that this interpretation (which bits are where in which register) satisfies the requirements specification (specific bits represent in a specific way a number which is the GCD of the input). This may be captured in the assertion

$$\mathit{Correct}(\mathit{*software}_{\mathrm{GCD}}) \equiv (\textit{gate-level implementation} \vdash_{RM} \mathit{GCD\ spec})$$

The Boyer-Yu example is not yet a complete proof of the *software correctness of a system implementing this description, since the proof that the actual gate-level design of the chip implements the architecture of the chip (the second step) is omitted. There is no problem of principle to provide it. For the real sceptics, we may note that Hunt has verified a gate-level design of a microprocessor (the FM8502) against its architectural specification [Hun87]. The example could have been targeted to an FM8502 architecture.

In order for us to accept the Boyer-Yu assertion, which I'll take to be

$$\textit{gate-level implementation} \vdash_{RM} \mathit{GCD\ spec}$$

as a theorem, we have to know that it indeed logically follows via the refinement mapping, that is, that if appropriate substitutions are made, that the right-hand side is a logical consequence of the left-hand side. Since the truth of this theorem for this example was obtained by the theorem prover Nqthm [BM88], we must establish that Nqthm is sound (that is, that it only obtains conclusions that logically follow from the premises). Soundness of Nqthm itself is not a trivial thing to establish. But on general principles which I will not develop, I claim that, if not Nqthm itself, then some modification of Nqthm is sound. Let's just suppose we are working with a sound version of Nqthm (the point of the example does not depend on this hypothesis in any crucial way that I can see).

### 7.2. Matching Supposed-Use and Formal-Spec

The Boyer-Yu theorem establishes $Correct(*software_{\mathrm{GCD}})$, that the gate level design of a certain system implemented on an MC68020 behaves in such a way that it satisfies its specification. I shall indicate how this argument may be completed to establish system correctness. First, I argue for the match of supposed-use and formal-spec.

The supposed-use of this system is to produce a result which satisfies a predicate of formal mathematics, which may be regarded as equivalent to a statement in a language with precise semantics (a logical language, with maybe certain extra axioms such as the axiom of infinity). As I argued in Section 4.1, one may take the formal-spec to be identical to this formal statement. Using the properties argued in Section 4.1, it follows that

$$match(GCD\ spec, GCD)$$

and therefore

$$Correct(GCD\ spec)$$

For such precise mathematical uses, it would seem always to be possible to make this move. Precisely which uses of computation allow this maneuver is a topic of research that leads beyond this paper, but I suggest it could be a rich source of progress in system development technology, since the more we know about this, the easier it is to eliminate one potential source of error in system design.

### 7.3. Wondering About the Hardware

It's not likely that anything such as a proof may be given that a given MC68020 physical device matches its gate-level description. My goal is to show that the only weak point in this system is the possibility of hardware failure whose only solution is physical. So far, I know that

$$Correct(GCD\ spec)\ \&\ Correct(*software_{\mathrm{GCD}})$$

The part-failure equivalence condition from Section 3.5 and the definition of *Correct* as the converse to *Faulty* from Section 6 yield

$$Correct(S) \Leftrightarrow Correct(hardware_{\mathrm{GCD}})\ \&\ Correct(*software_{\mathrm{GCD}})\ \&\ Correct(GCD\ spec)$$

To be able to assert $Correct(S)$ from the part-failure equivalence condition, we are missing a proof that

$$Correct(hardware_{\mathrm{GCD}})$$

i.e., that $\neg Faulty(hardware_{\mathrm{GCD}})$ To be able to assert $Correct(S)$ from the five-step conjunction for this example, we lack a proof that

$$hardware \models gate\text{-}level\ implementation$$

How are these two assertions related? To establish that it's plausible that

$$\neg \mathit{Faulty}(\mathit{hardware}) \;\Rightarrow\; (\mathit{hardware} \models \mathit{gate\text{-}level\ implementation})$$

I return to the earlier discussion of commissives. Suppose we are assured through proof that *gate-level implementation* $\vdash_{RM}$ *GCD spec* and that (through an argument yet to be given) *match*(*GCD spec*, *GCD*). If you build a physical system and implement this design on it and the physical system fails during operation, you won't under any circumstances talk to a *software engineer, you'll call someone with a hardware-tester who knows where to look for bubbles in the silicon. In the words of Section 5, the (completed) Nqthm proof along with the assertion of matching entails that *if anything goes wrong, it must be a physical fault in the physical chip*. Note this principle also takes care of the possibility of random destruction of the chip (nuclear explosions nearby), gamma rays altering the values of bits, and other physical disturbances. I conclude that the assertions $\neg$*Faulty*(*hardware*) and *hardware* $\models$ *gate-level implementation* are equivalent. Thus, the five-step correctness assertion is correct, it corresponds to the part-failure equivalence principle, and the Boyer-Yu theorem is the most complex step in proving the correctness, the *real* correctness, of the GCD algorithm.

## 8. The Rhetorical Style of Computer Science

Every scientist must learn rhetorical skills appropriate to her profession. I use the word 'rhetoric' in its classical sense, to refer to styles of argument used to persuade others of the truth, falsity, or relevance of views, theorems, or particular pieces of scientific work. Investigating rhetorical styles may give clues to the progress of an intellectual discipline. Mathematics has evolved its styles of arguments to develop nearly universal protocols for agreement (there are substantial divisions only between classical and constructivist modes of argument – but also substantial connections and intertranslations have been found), and analytic philosophy harnesses the coherent development of disagreement to furthering knowledge. Both disciplines have rigorous rhetorical standards. This does not mean that all attempts, even published attempts, at furthering these disciplines succeed. But it does mean that there is almost universal agreement on failure. Published mathematical 'theorems' are shown to be false. Published philosophical arguments are shown incomplete or incorrect.

Progress in a science may come about through the eventual resolution of technical conjectures on which there is legitimate initial disagreement. This may commit us to the principle that others with whom we disagree should be able eventually to demonstrate that our beliefs are incorrect, or *vice versa*. In contrast, it seems to me that many computer scientists have prefered to give up on the debate concerning proofs of system correctness. If we ignore a problem, we are thereby unlikely to obtain a universally agreed answer. If an answer is required for $P{=}NP$, why not also for the existence of provably functional systems?

I conjecture that the fuzziness of the debate on provably correct systems involves (a) a lack of common meaning to the terms in which the problem is stated; (b) a deficiency in the styles of rhetoric employed in practical computer science; (c) a lack of understanding and sophistication in formal argument; (d) people having an interest in a particular answer for reasons other than its truth or falsity.

None of these are trivial components either of computing science or of computing practice. A lack of common meaning to common expressions, and a deficit in effective methods of reaching agreement lead to a deficiency in the ability to tell truth from falsehood, and these factors, along with hidden interests as in (d), impede our ability as a profession correctly to analyse approaches to problems of major and sometimes life-critical interest. Feature (d) in particular strikes at the heart of progress in science. I have nothing technically relevant to say here concerning (d).

The ideas in this section bear no logical relation to the correctness of any formal claims about the possibility of system verification. Arguments and examples stand and fall on their own terms. Similarly, my discussion of (a) - (c) is logically independent of the argument I have given that there is a provably correct system.

## 8.1. Contrasting Rhetorical Styles

In mathematics, a general claim may only be refuted by counterexample. A conjecture that says 'there is no X with the following property ... ' or 'every X has the following property ... ' may only be refuted by exhibiting a counterexample to the claim. Likelihood and plausibility arguments are used as guides only to the direction to search for a solution - either to try to prove the theorem or to try to find a counterexample.

In philosophy, a counter-claim to a general claim may frequently be argued in general terms, but once a potential weakness of an argument is located, the discussion soon turns to arguing the merits and demerits of examples. The rhetorical device of the counterexample is well established, as well as the device of the paradigm example.

In contrast, the discussion over Fetzer's paper generated *not one specific example* to test out the opposing views [Aut89a], [Aut89b], [Aut89c], [Aut89d], [DR89a]. A letter written by a collection of eminent researchers suggested mainly that it was irresponsible of the journal to publish the paper [ABG+89]. A letter from other eminent researchers suggested that Fetzer was ignorant, without addressing his argument [PCG89]. Fetzer noted correctly that both issues were irrelevant to his argument. It is likely that such commentary, had it been submitted to a mathematical or philosophical journal, would have been considered eccentric. Nevertheless, some serious attempts were made to address features Fetzer had illuminated, such as the social issues [DR89b], or the relation between formal mathematics and the physical world [Bar89].

So, some of the leading experts in the field to which Fetzer's argument was addressed felt, after due consideration, that an argument from authority was a more effective, and a more acceptable, way of refuting his claim than providing a counterexample.[25] I believe that neither philosophers nor mathematicians would make this particular choice of rhetorical device in a similar case in their own field, for two reasons. Firstly, it does not shed any light on the truth or falsity of the claims Fetzer made. Secondly, it does not shed any light on the quality of his argument (Is it complete? Full of holes? Irrelevant? If so, how and why? Can it be fixed? Is there a counterexample?).

The choice by computer scientists of which rhetorical devices to employ in refutation shows vast discrepancies between the current practice of computer science and the more established fields of mathematics and the philosophy of language, science, mathematics and logic. Yet some fundamental issues in computer science are very close to those considered in these other fields, as demonstrated by Fetzer's article, as well as by the increasing collaboration of neurobiologists, computer scientists, psychologists and philosophers in attempting to explain

[25] One other possibility is that no-one could have thought of a simple counterexample. I find this implausible. I think no-one bothered to look.

certain features of the mind and brain. (Consider, by way of contrast – when have philosophers since Aristotle attempted to shed light on, say, the technology of horticulture?)

We need to assess the effect of our choices of rhetorical device on the development of our science. After all the claims and counterclaims, do we know, or have we been persuaded, whether program verification is in principle impossible, or not? Further, are we provided with any insight into whether our current research or practice in computing and software engineering is foundationally shaky, or fundamentally sound?

### 8.2. Good Engineering Practice

It is often claimed that discussion of broadly social issues such as the rhetorical devices we employ is irrelevant to actual software practice. I find this claim implausible, for the following reasons.

Firstly, those of us who have worked in close liason with industrial software development are aware of the often palpable aversion expressed to the use of the rigorous standards and techniques of formal mathematics and technical philosophy. "*We've tried it, we messed it up, so we *know* it doesn't work*". "*Our engineers can't understand it, *therefore* it isn't any good*". "*It's *absurd* to believe that software can ever have a reliability of 1*". The authors of the 'irresponsibility' reply to Fetzer [ABG+89] implicitly acknowledge that misleading ideas may propagate to the detriment of computing science and practice. So may insufficiently rigorous styles of rhetoric.

Secondly, experts in an essential area of computing practice apparently believe, as a consequence of their other beliefs, something which is demonstrably false (see Section 2.3). We may conclude there is a part of their belief system which causes them to believe a falsehood. We may even erect a not-so-straw man, and have a fictitious such expert claim that he believes this falsehood follows from extensive experience and observations. The falsehood can only occur either because of a logical mistake somewhere, or because certain observations have been misinterpreted, because it is logically impossible to infer a falsehood from truths using logically correct inference rules. Therefore, should not 'good engineering practice' rather dictate that we attempt to find out where that mistake is, or where observations have been misinterpreted?

I propose that rigor serves as important a role in good engineering practice as it does in mathematics, and for similar reasons.

## 9. Summary

I have given a simple example to refute claims that (a) systems may never be proved to satisfy their specifications; and (b) system software may never be taken to be fully reliable. This example, in turn, led to an analysis of some of the concepts and arguments used in assertion of correctness and failure of systems, and a formulation of such principles as the part-failure equivalence condition, which led to the conclusion that a specification should be considered a proper part of a system, even though it's not mutable with software and hardware the way that these two are with each other.

Having established *that* some kinds of properties of the future behaviour of physical systems can admit of rigorous proof, I attempted further to explain *how* properties could attain that distinction. For example, properties may be logically true. Or, if the supposed use allows the possibility of pure hardware faults to occur, we can obtain provable satisfaction if supposed-use and formal-spec match (coined as a term of art), or match modulo some inviolable principle, and also if the full gate-level design matches the formal-spec. This used the notion of commitment to a proposition or principle, investigated by Searle and other speech act theorists, and in a different context by Duhem and Quine. In particular, I gave an informal argument to show that safety properties of physically-realised systems such as in a fly-by-wire transport airplane can admit of proof, if no liveness properties are specified. I offered a formalisation of some of the logical properties of the concepts involved.

The final discussion concerned an example of a computer design (complete with chip) which computes a useful function (a GCD), whose *software was proved correct (modulo a hardware architecture verification which has not yet been performed) by Boyer and Yu. I observed that for this case supposed-use and formal-spec may be taken to be identical. It follows that if a physical system using the appropriate chip implements this design and fails, the failure is in the physical hardware.

These arguments concern the foundations of system engineering and provide a refutation of certain views about those foundations and a vindication of others. My approach is that of philosophical logic. None of the arguments I presented attempted to address the epistomological argument of De Millo, Lipton and Perlis, that even though systems may be correct, it's doubtful for even moderately complicated systems that we could ever come to know this [DLP79].

Finally, I considered the styles of rhetoric currently employed in the science of computing. I indicated how they may fail to help us, for example by diverting us from searching for examples and counterexamples to establish or refute claims, and how therefore they may affect engineering standards.

## Acknowledgements

My thanks to Tom Anderson, Bob Boyer, John Fitzgerald, Cliff Jones, Leslie Lamport, Nancy Leveson, Pete Mellor, Harold Thimbleby, Ken Turner and Friedrich von Henke for some lively email and discussions on these topics. Thanks again to Bob Boyer for giving me the Nqthm GCD proof to use as an Addendum. A significant amount of the work enabling this proof is due to Yuan Yu.

## References

[ABG+89] M. Ardis, V. Basili, S. Gerhart, D. Good, D. Gries, R. Kemmerer, N. Leveson, D. Musser, P. Neumann, and F. von Henke. Editorial process verification (Letter). *Communications of the ACM*, 32(3):287–288, Mar 1989.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Jan 1993.

[ALLM93] T. Anderson, P. Ladkin, N. Leveson, and P. Mellor. Informal discussion. Jan-Feb 1993.

[And93] T. Anderson. Private communication. 1993.

[AS87] B. Alpern and F.B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[Aus75] J.L. Austin. *How to Do Things with Words.* Oxford, Clarendon Press, Urmson/Sbiza edition, 1975.

[Aus78] J.L. Austin. *Philosophical Papers.* Oxford, Clarendon Press, Warnock/Urmson edition, 1978.

[Aut89a] Many Authors. Correspondence. *Communications of the ACM*, 32(3):287–290, 374–381, Mar 1989.

[Aut89b] Many Authors. Correspondence. *Communications of the ACM*, 32(4):506–512, Apr 1989.

[Aut89c] Many Authors. Correspondence. *Communications of the ACM*, 32(7):790–792, Jul 1989.

[Aut89d] Many Authors. Correspondence. *Communications of the ACM*, 32(8):920–921, Aug 1989.

[Bar89] J. Barwise. Mathematical proofs of computer system correctness. *Notices of the American Mathematical Society*, 36(7):844–851, Sep 1989.

[BM88] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[Bru93] G. Bruns. Applying process refinement to a safety-relevant system. Submitted for publication, May 1993.

[BY92] R.S. Boyer and Yu Yuan. Automated correctness proofs of machine code programs for a commercial microprocessor. In D. Kapur, editor, *Automated Deduction - CADE11*, volume 607 of *Lecture Notes in Computer Science*, pages 416–430. Springer Verlag, 1992.

[BY95] R.S. Boyer and Yu Yuan. Proof of a gcd algorithm on an mc68020. In `http://www.techfak.uni-bielefeld.de/~ladkin/`, 1995. Adapted by P. B. Ladkin from [BY92] and comments of R. S. Boyer.

[DLP79] R. DeMillo, R. Lipton, and A. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[DM93] R.D. Dorsett and P. Mellor. The Airbus A320 electronic flight control system. Unpublished Manuscript, 1993.

[DR89a] J. Dobson and B. Randell. Correspondence. *Communications of the ACM*, 32(9):1047, 1050, Sep 1989.

[DR89b] J. Dobson and B. Randell. Program verification: Public image and private reality. *Communications of the ACM*, 32(4):420–422, Apr 1989.

[Dum73] M. Dummett. *Frege: Philosophy of Language.* Duckworth, London, 1973.

[Fet88] J.H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, Sep 1988.

[Gen84] M. R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24:411–436, 1984.

[GSW89] R. Greiner, B. A. Smith, and R. W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, Nov 1989.

[Hum78] D. Hume. *A Treatise of Human Nature.* Oxford, Clarendon Press, Selby-Bigge edition, 1978.

[Hun87] W.A. Hunt. The mechanical verification of a microprocessor design. Research Report CLI-6, Computational Logic, Inc., Austin, Texas, 1987.

[Lad94] P. B. Ladkin. Why use tense logic to describe digital systems?, 1994. In `http://www.techfak.uni-bielefeld.de/~ladkin/`.

[Lad95a] P. B. Ladkin. Formal but lively buffers in TLA$^+$, 1995. In `http://www.techfak.uni-bielefeld.de/~ladkin/`.

[Lad95b] P. B. Ladkin. A review of Fetzer's Very Idea, 1995. To appear in `http://www.techfak.uni-bielefeld.de/~ladkin/`.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering SE-3*, 2:125–143, 1977.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lew91] D. Lewis. *Parts of Classes.* Basil Blackwell, Oxford, 1991.

[Lut93a] Robyn R. Lutz. Analyzing software requirements errors in safety-critical embedded systems. In *IEEE International Symposium on Requirements Engineering*, pages 126–133, San Diego, CA, January 1993. IEEE Computer Society Press.

[Lut93b] Robyn R. Lutz. Targeting safety-related errors during software requirements analysis. In *SIGSOFT '93: Symposium on the Foundations of Software Engineering*, pages 99–106, Los Angeles, CA, December 1993. Published as ACM Software Engineering Notes, Vol. 18, No. 5, Dec. 1993.

[MS95] S.P. Miller and M. Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *Proceedings of WIFT'95, the Workshop on Industrial-Strength Formal Specification Techniques.* IEEE Computer Society Press, 1995.

[Myc89] J. Mycielski. The meaning of pure mathematics. *Journal of Philosophical Logic*, 18:315–320, 1989.

[Myc92] J. Mycielski. Quantifier free versions of first order logic and their psychological significance. *Journal of Philosophical Logic*, 21:125–147, 1992.

[Myc95] J. Mycielski. Equational treatment of first-order logic. *Algebra Universalis*, 33(1):26–39, Jan 1995.

[Par90] D.L. Parnas. Documentation of communications services and protocols. In S. T. Vuong, editor, *Formal Description Techniques II*, pages 277–280. North Holland, 1990.

[PCG89] L. Paulson, A. Cohn, and M. Gordon. Letter. *Communications of the ACM*, 32(3):375, Mar 1989.

[Qui52] W.V.O. Quine. *Methods of Logic*, page xii. Routledge Kegan Paul, 1952.

[Qui60] W.V.O. Quine. *Word and Object.* MIT Press, 1960.

[Qui61] W.V.O. Quine. Two dogmas of empiricism. In *From a Logical Point of View.* Harvard University Press, 2nd edition, 1961.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[Ryl63] G. Ryle. *The Concept of Mind.* Penguin, Reprinted edition, 1963.

[Sea69] J.R. Searle. *Speech Acts: an essay in the philosophy of language.* Cambridge University Press, 1969.

[Sea79] J.R. Searle. *Expression and Meaning: studies in the theory of speech acts.* Cambridge University Press, 1979.

[Sim87] P. Simons. *Parts – A Study in Ontology.* Oxford, The Clarendon Press, 1987.

[Thi93] H.W. Thimbleby. On programming and understanding computers. Unpublished Manuscript, 1993.

[Yu93] Yuan Yu. Automated proofs of object code for a widely used microprocessor. Technical Report 114, Digital Equipment Corporation Systems Research Center, Palo Alto, California, Oct. 5 1993.