Part II

Causal System Analysis

Chapter 3

The Foundations of System Analysis

3.1 Preliminaries: The Importance of Reasoning

The Primacy of Reasoning in Prediction Assessing and ensuring the safety of artifacts and procedures is largely a matter of predicting what may happen in the future. If with perfect foresight one knew that an accident was not going to happen, then one knows with perfect foresight that perfect safety is ensured. But since the future has not happened yet, we cannot report on safety purely using observations. We must reason from the current and past situation in the world to a future situation. We must formulate what we know and attempt to use this information to predict the future as best we can. Safety assessment thus involves reasoning. Precise safety assessment involves precise reasoning. Formal logic is the study of precise reasoning; in some sense, rigorous system safety reasoning must be applied formal logic.

Expressive Limitations of Reasoning However, formal logic as practised by logicians and philosophical logicians and as applied by computer scientists and engineers is not a finished science. In fact, there are relatively few "*settled*" parts of formal logic: the propositional calculus, the predicate calculus, certain so-called "*constructive*" formulations of them, certain forms of tense logic, certain logics with modalities, certain forms of higher-order logic. Important parts of engineering reasoning that have no agreed formulation, or which have demonstrated unsuitabilities for the task which we wish they performed are: arithmetic itself, reasoning about parts and wholes, reasoning about obligations, reasoning about causality.

Computational Limitations of Reasoning Apart from these, there is the simple problem of how to handle reasoning. Reasoning may be simple or complex. It may be readily understandable or obscure to all but a chosen few. It can be very hard to construct reasoning that will determine whether a given assertion is canonically true from, or canonically refuted by, or canonically not decided by, certain other assertions. Even if the formulation of reasoning itself was demonstrably adequate, our ability to reason inside that formulation is limited by the complexity of its combinatorics.

Uncertainty We are very uncertain about many important things. Will this joint hold under this constant pressure for the next two years? Well, our reasoning says it should; most of them have in the past; specially constructed tests come out positive; but some of them, very few of them, have still failed under these requirements. We can't be certain the joint will work, but we think it is very, very likely and we're prepared to place a very high bet on the fact that it will.

Formal logic dealing with uncertain reasoning is often called probability logic. It's very hard, it's complex, and it doesn't tell you much about decisions. Formal methods of reasoning about decisions are also difficult. Still, we have to do it.

The Engineering Task Building safety cases for systems, and performing safety assessments of systems, are examples of this very hardest reasoning. Since the systems are being built and in use, we must use whatever imperfect techniques we have. By some measures, we have been very successful with these techniques. Complex commercial aircraft don't crash every day. By other measures, we have not been. Commercial aircraft still crash for easily avoidable and repeated causes. The complexity of systems is increasing enormously, defying our ability to apply the techniques we knew successfully to assess the safety properties of these new systems. Sometimes reliable and safe designs are replaced, degrading either reliability or safety properties or both in the process. And there are new systems, performing new functions. We have to do something.

Practical Application Doing what we did before, when it worked, can be a good guide. Thus, we follow standards and checklists. Not doing what we did before, when it didn't work, can also be a good guide. Thus, we analyse accidents. Making sure we don't make mistakes that we knew how to avoid is always a good idea. Thus, inspections, reviews, and in general checking our reasoning is a good way of avoiding mistakes in design. Inspections and appropriate maintenance is a good way of avoiding the consequences of unwanted change.

Requirements of Effective Reasoning Reasoning about systems is a large component of safety in design. In order to reason effectively, it is customary to have a language in which one can make assertions, with this language being bound in some way to the "*world*" and its states; to "*objective reality*", to notions of "*truth*" and "*falsity*" of assertions in the language evaluated on "*the actual situation*". I don't know any other way of doing it. Successful engineering involves techniques and procedures which many trained people - engineers - can use. So engineering reasoning about systems must nowadays take the form that reasoning about anything has taken in the last few hundred years. Logic, ontology, objects, properties and relations, assertions concerning them and relations of deduction and logical consequence, probability and probabilistic reasoning, the use of a formal language to make unambiguous assertions amongst trained practioners, assessment procedures for truth and falsity or likelihood of assertions, and so on.

In Any Case, Rigor and More Rigor It is often said that to assess and ensure safety, one must think of everything. It may be added that one should think of everything carefully. Situations should be thoroughly checked. Desire for rigor has led to *formalisms*, ritualised ways of speaking and reasoning that are easily reproducible in standard ways, and that show weaknesses and possible problems. Formal languages enable us to enumerate what we can say, and use of a formal language enables us to specify what we see. We can exhaustively enumerate possibilities and check them, using indirect techniques if exhaustive enumeration is too exhausting. We can identify mistakes we may make in expression and reasoning, in principle. We can identify and correct omissions. And we can do all this in a principled way that allows us to avoid repeats.

The Application of Rigor Rigor can be applied in two main ways when dealing with artifacts. It can be applied in reasoning about the artifact itself, and it can be applied to the management and other human behavior in the environment in which the artifact is placed. Both are important; regarded indeed as essential in modern system safety. We shall restrict ourselves here to the first: reasoning about the artifact, its design, its properties and its environment.

3.2 Formal Causal System Analysis

Artifactual systems are built by human beings according to causal principles. Parts are designed in order to have certain influence on other parts: this influence is causal. Analysis of the operation of these systems must therefore be a form of causal analysis. The methods hitherto used in analysis of the safety properties of systems are mostly forms of causal analysis, but without any specific or rigorous notion of what constitutes a cause or causal factor. This means that they are ultimately fouded on intuitive judgement of cause, and this intuition must be built up by consensus and experience in the engineering community. The fact remains, however, that a method based on an undefined and unclarified notion is not truly rigorous. The acid test of objectivity is that the criteria for judgement are explicit and can in principle be used by third parties that are not privy to the socially-learned intuition to check the results of reasoning. Intuition may speed things up, but it should not provide the fundament if alternatives are available.

We illustrate system analysis methods. These methods

- enable causal analyses of artifacts,
- are based on an explicit formal notion of causal factor, which experience has shown can be assessed "in the field" with relative accuracy with a modicum of training,
- are intended for safety analysis of designs: Causal System Analysis (CSA),
- are intended also for causal analysis of accidents: Why-Because Analysis (WBA).

3.3 What is a System?

Examples Things called systems are varied. Sociologists speak of social systems [Luh91]; political scientists speak of system-theoretic influences [Jer97]; other sociologists speak of complex technical systems [Per84, Sag93]; aircraft builders speak of physical systems and subsystems; ecologists of predator-prey systems or environmental cycles of substances; and of course there are computer systems.

What Do They Have In Common? I propose that systems contain *objects* which engage in *behavior*. This behavior, through the objects which constitute a system, may be influenced considerably by the behavior of objects which are not part of the system, through their relations with objects that are in the system. These objects are said to belong to the *environment* of the system. Besides this, there are other objects which have no perceptible influence on and no important relations with system objects, and vice versa. We say that these objects belong to the *world*.

For example, if I consider my bicycle to be a system with all its components (complete with rider), then the environment would include the streets and paths I am riding on and the immediate influences on their state, such as the weather or an overflowing river. Since I am in Germany, the Great Wall of China belongs to the world, not to the system or the environment.

One way of picturing objects divided in this way is as a Venn diagram, such as Figure 3.1, in which points represent objects.

Such a diagram may be misleading, in that it shows world objects and system objects sharing a boundary. "Sharing a boundary with" may be (misleadingly or not) identified with a relation, and system objects by definition only have

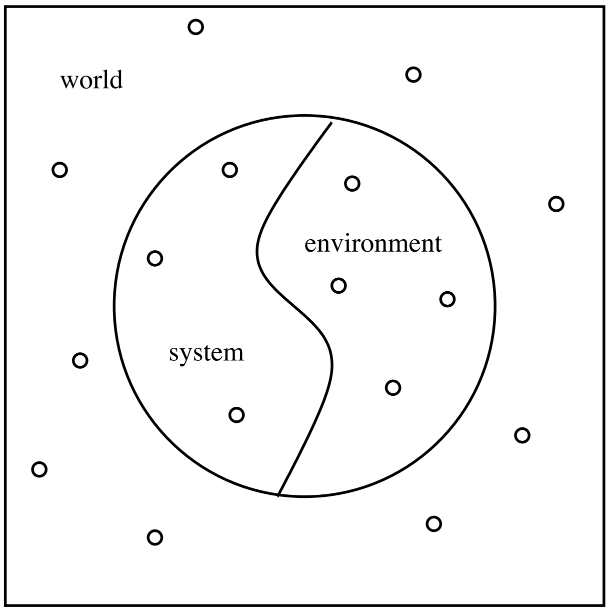

Figure 3.1: The World of Objects

relations with environment objects. Hence a diagram such as Figure 3.2 more closely visually represents what we are aiming at.

The System Boundary Although the entire universe can be considered as a single system (the collection of objects = everything; relations and properties = all relations and properties), this is not the system mostly considered by engineers. One mostly considers smaller parts of the universe; hence one may make a distinction between those objects that belong to the system and those that do not. This distinction leads to the notion of the *system boundary*, namely the distinction between what objects and behavior are to be considered part of the system and which not. The system boundary may correspond to something real, or it may simply be some kind of metaphor. Which is the case will depend very much on what kind of thing the system is.

Teleological Systems and Others Define a *teleological system* to be a system with a purpose or goal. This purpose may be the elicitation of certain behavior, or the attainment of a certain state, of system or environment or both (but not of "world" since the constituents of world are outside the mutual influence of system and surroundings, by definition). Artifacts are typical examples of teleological systems. A car is designed with the purpose in mind to transport people in a particular manner. A computer system is designed with the purpose of performing certain sorts of calculations in a certain manner. Examples of non-teleological systems are environmental systems such as an industrial effluent

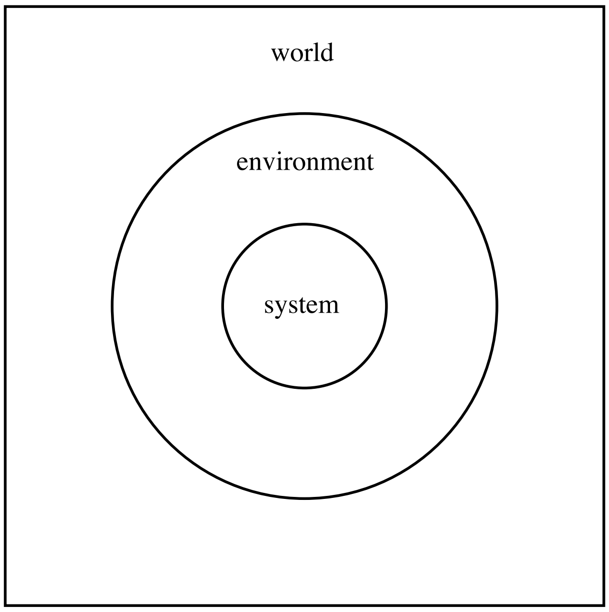

Figure 3.2: The World of Objects As It Should Be

cycle or a predator-prey system. The international political system is also an example of a non-teleological system, although individual component systems, the governments of countries, are teleological. We shall be dealing mostly with teleological systems: those for which goal states or goal behaviors of system and environment can be identified. Constructing teleological systems is the primary goal of engineering.

The Boundary Assumption For Teleological Systems In a teleological system, goals are somehow stipulated. One characteristic method of stipulating the boundary between teleological system and environment is to consider which features of the *universum* (the total world under consideration, including, as possibly proper parts, system and environment) one can more or less control, and which not, and to make the decision on this basis. Call the assumption, that the decision to place the boundary is made more or less consistently with this control criterion, the *boundary assumption*.

Examples of the Boundary For example, the state of a runway surface may be controlled to some degree: one can clear it of debris, or excess water or snow, and direct traffic elsewhere until such time as it has been accomplished. In contrast, one has little or no control over the weather at the airport and hence the dynamic conditions of the air through which landing and departing aircraft fly. Under this criterion, it would be appropriate to consider the runway condition part of the system, and the weather not, and the (expected) behavior of the system varies accordingly. Although one is obliged simply to wait until bad weather changes for safer flying conditions, it would be an inept (or impoverished) airport manager who simply waited for the snow to melt from hisher runway.

3.4 Objects and Fluents

What Are Objects? Objects are, roughly, anything which may be denoted by a noun or noun phrase. More broadly, anything which may be the value of a quantifier [Qui64]. So, you and I are objects, real numbers are objects, the water enclosed inside notional boundaries specified as straight lines between three fixed geographical points is an object. But vanity is doesn't seem to be an object, and neither does humor, nor willpower, and neither is the value of a specific memory location in a computer over time, since this value is constantly something different.

Fluents These quantities can be considered to be objects if one performs certain operations known to logicians and ontologists (people who worry about what objects are, and what objects there are). We thus introduce *fluents*, which are *things which take values over time*. Using the notion of fluents, the number of things we can consider objects can increase considerably. If we think binary numbers are objects, then the value of a memory location over time is a fluent taking binary numbers as values. If we think that the exercise of vanity corresponds to excitation of certain parts of the human brain, then we can consider Fred's vanity to be that function over time which measures this excitation in an appropriate way. And of course now that we have these fluents, we can define further fluents that take these fluents as specific values; and so on iteratively. In short, with common objects and fluents, someone who wishes to talk about systems can indeed talk about as many objects as heshe wishes.

3.5 State, Events and Behavior

Behavior Objects have behavior. We shall consider behavior to be how the properties of an object and its relations with other objects change over time. More generally, behavior is how a collection of objects (including the constitution of the collection itself) changes over time.

Change Consider a complete description of what properties objects have and what relations they have to each other. Such a description is for many reasons impossible to obtain, but let us not worry about that yet. This description could well be true at a certain moment of time. Some time later, this description could well be false. This is what we refer to as change over time.

State We shall say that such a complete description of objects with their properties and relations to each other at a moment of time is a *state description*, and the actual configurations of objects, properties and relations it describes is called a *state*. Figure 3.3 shows an example of a system state. The fluent x takes the value 2; the object *Valve1* has the property *Open* (we take this to mean that *Valve1* is *open*); the quantity of reactant (we take *Quantity*, denoted in subsequent figures also as Q, to be the function which gives as value the quantity of its argument; its argument is *reactant*) is 100 units.

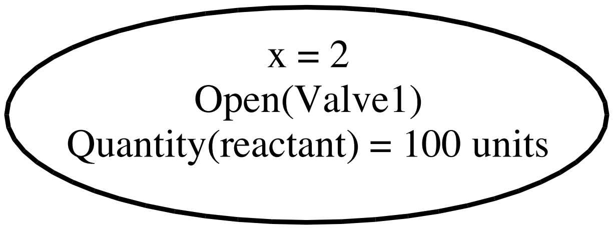

Figure 3.3: A System State

One limitation: it is important for technical reasons that the objects, properties and relations noted in a state contain no explicit reference to time and change themselves.

Justification of This Notion of State One may well ask, why this apparently somewhat restrictive notion of state? Why cannot state be anything at all? The answer is, that it is not as restrictive as it may at first seem, for the following reasons.

- One cannot know the future, hence if one wants the state to include determinate predications, one cannot include predictions of the future in the current state. This rules out including future temporal properties in a description of state;

- Temporal properties in the past can be included by the simple device of "history variables": one introduces a fluent which retains all the information about the past that needs to be retained by the system, or needs to be known by someone reasoning about the system. An "audit trail", if you like. The use of history variables in formal system description techniques is ubiquitous.

- Set theory formulated in first-order logic (also called "first-order set theory", or "Zermelo-Frankel set theory" after two of its founders) suffices for describing the component structure of systems, as well as all the mathematics one needs to describe the discrete or analog behavior of the system.

Some mathematicians and some computer scientists do not like first-order set theory for various reasons. To accomodate the wish to avoid set theory, one can instead provide equivalent descriptions in *higher-order logical type theory*. There are very few properties known, if any, that cannot be accomodated in one or the other of these formal languages.

Comparing States Given two states (or two state descriptions), we may compare them to see what is the same and what is different. We shall consider *change* simply to be what is different, and a description of change to be a specification of the differences (sometimes it will also be important to specify what is the same in the comparison, sometimes not). Figure 3.4 shows an example of change as a comparison between two states. *Valve1* is not open in the first state, and in the second it is open. The other predicates have not changed: The fluent x still has the value 2; the quantity of reactant (denoted by the predicate Q with the argument *reactant*) is still 100 units.

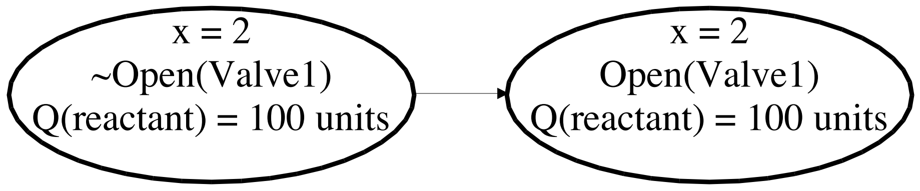

Figure 3.4: A State Change

Events and Event Types The change illustrated in Figure 3.4 does not describe much about either state. If I were part of the system of which the objects mentioned in the states are also part of, it doesn't say what clothes I'm wearing or where I am or what time it is. It potentially describes many individual system changes, namely all those in which the specific objects described change in the way described. Let us call an individual change, which occurs at a specific time, an *event*. Then the state change in Figure 3.4 describes many events, namely all those in which the objects change as specified. We say it describes an *event type*. An event belongs to this type just in case it is an event in which x remains 2, *Q(reactant)* remains at 100 units, and *Valve1* changes from open to closed.

Deriving Behavior Descriptions from State Comparisons Since we may compare two states to see what has changed, we may compare three, one after the other, to obtain a view of progressive change. Or four, or five, or a hundred. We may consider behavior to be a sequence of states such as this, specifying a series of changes. It is important to note that this sequence is *discrete*, that is, each state in it has a definite predecessor and a definite successor. Since we may need to chain together lots of these state comparisons, if we require very great detail or if the sequence goes on for a very long time, we consider that a state sequence may have a huge number or even an infinite number of member states. Thus we shall consider a *behavior* to be *an unending discrete sequence of states.* Figure 3.5 shows such a discrete sequence (at least, the first four states of one).

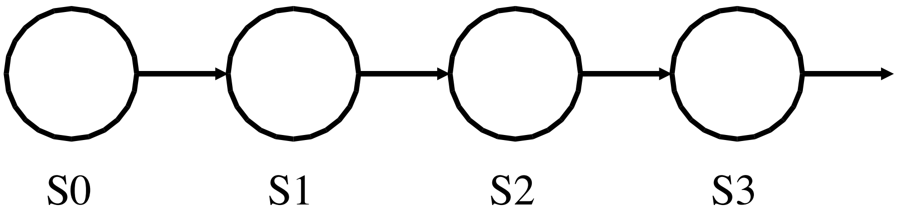

Figure 3.5: A Behavior

Another limitation: for technical reasons, we shall consider that a state sequence that forms a behavior shall always have a *first state*, that is, one that occurs before all others; that itself has no predecessor.

Near And Far State Changes We shall need to distinguish small from large state changes. We shall call small changes *near changes* and large changes *far changes*. These notions are intended to have their intuitive meanings. For example, Figure 3.6 shows a near change, in which the value of the fluent x changes from 2 to 3 and nothing else changes.

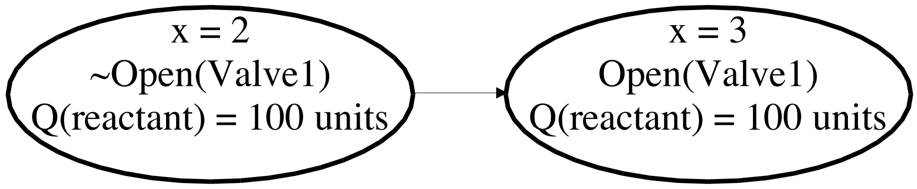

Figure 3.6: A Near Change

Figure 3.7 shows a far change, in which the value of the fluent x changes considerably to 54, and at the same time the quantity of reactant increases threefold.

Near And Far Behaviors We shall also need the notion of near and far behaviors, which is a comparative notion. This notion is a comparison between three behaviors: a behavior B is *nearer to* a behavior A than a behavior C is to A. For example, the behavior in Figure 3.9 is nearer to the reference behavior in Figure 3.8 than is the behavior in Figure 3.10.

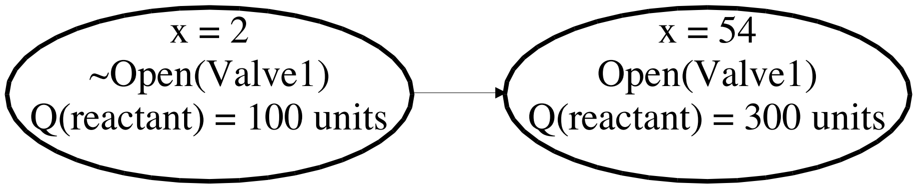

Figure 3.7: A Far Change

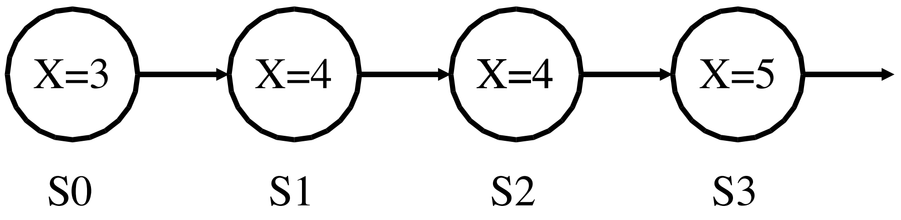

Figure 3.8: A Reference Behavior

The only difference between the reference behavior in Figure 3.8 and its near behavior in Figure 3.9 is that, in state $S1$ of the near behavior, the value of the fluent x is 5 rather than 4.

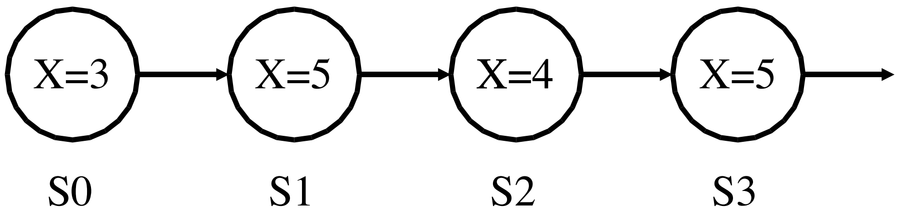

Figure 3.9: A Near Behavior to the Reference

In contrast, the values of x in states $S1$ and $S3$ of the far behavior are very different from the corresponding values of x in those states in the reference behavior, and the value of x in state $S2$ is also somewhat different.

The "Space" of Behaviors Suppose we were to represent behaviors as points in a Venn diagram. Then we could use the distance in the diagram to represent visually the nearness, respectively, farness, of the behaviors from each other, as in Figure 3.11. If we consider the "space" of all possible behaviors as a Venn diagram, and supposing we had a way of measuring nearness and farness on an *ordinal scale* [KLST71] (see Section 8.2 for an enumeration of the explicit properties meant), we could represent nearness and farness of all possible behaviors relative to a given behavior, the "*real world*", as in Figure 3.12.

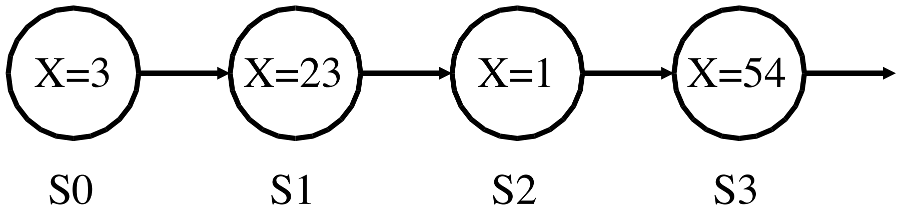

Figure 3.10: A Far Behavior to the Reference

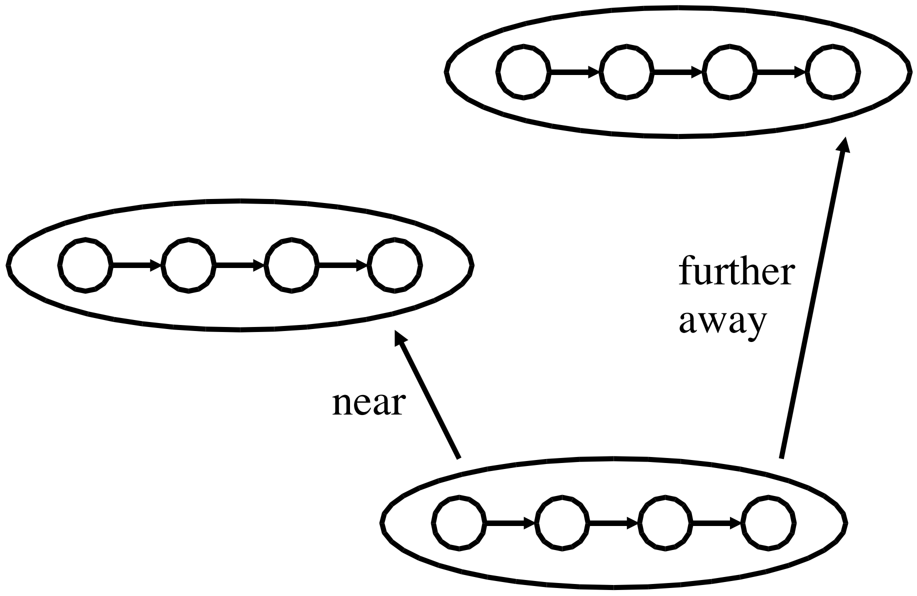

Figure 3.11: Nearer and Further-Away Behaviors

The Purpose of The Definitions The point of this ontology is that

- there are provably complete forms of formal reasoning about these structures; and
- one can describe any "real world" situation adequately using these structures

Since we can describe any situation we may care about, and we can reason accurately and formally about that description, this ontology lends itself to rigorous reasoning about systems, which safety analysis requires.

3.6 Objects, Parts and Failure Reasoning

Objects with Parts Consider a large chunk of computer code. Say, a program of a few thousand lines. This code *contains* procedures, and instructions. The

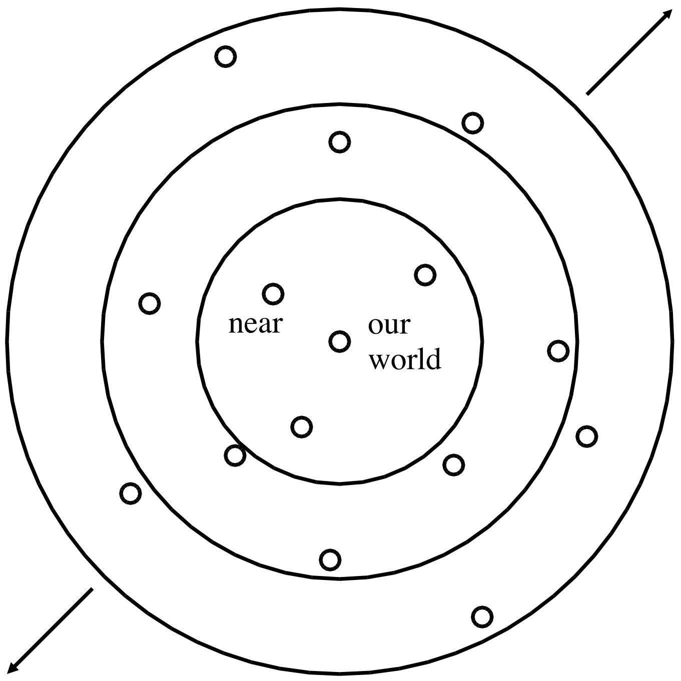

Figure 3.12: All Behaviors, Arranged in "Nearness" Circles

procedures are *part of* the program; the instructions are *part of* the procedures.

Structural Parts The instructions are part of the procedures and the procedures are part of the program. But the progam, when looked at another way, is just a very long string of alphabetical symbols. Any contiguous substring is also part of the program, even though it may start in the middle of an instruction and end in the middle of another one. Probably any collection of contiguous substrings is part of the program also. We want to say that whole instructions and whole procedures are the meaningful parts of the program as far as the operation of the program is concerned, and that individual characters and strings of characters are not, except insofar as they constitute instructions and procedures. Compilers tacitly perform the distinction by having a preprocessor, a lexical analyser, which groups individual characters into *tokens*, which are regarded as the minimal meaningful objects as far as the operation of the program is concerned. We distinguish the cases by saying that instructions and procedural parts and the tokens identified by the lexical analyser (presuming it is correct) are *structural parts* of the program as far as its operation is concerned. Characters will be structural parts of the program if we are considering maybe its storage requirements, or if comparing it with the work of six monkeys sitting at typewriters.

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

Figure 3.13: Correct Failure Reasoning With A False Premiss

Mereology and Fusion There is a logical science, *mereology*, concerning which parts of objects exist. One widely-accepted mereological operation is that of *fusion*, whereby from objects X and Y is formed the object $X \oplus Y$, the '*mereological sum*', which has X and Y as parts, and such that any object with X and Y as parts has also $X \oplus Y$ as a part: the 'smallest' object one can make from X and Y in other words.

Parts and Failure If a system fails to perform its function, a subdivision into parts is often used in order to identify a part that failed to fulfil its function. Take a common example:

"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 shown in Figure 3.13. Unfortunately, although we might want the conclusion to follow from the premises in this particular inference, the conclusion is false. There are examples in which the system failed, the hardware didn't fail, and the software did not fail to fulfil its designed function either. Ariane Flight 501 is an example. The situation, as in most failures of mission-critical or safety-critical systems, is that there was a misfit between the requirements for the design of the system, and the environment in which the system actually operated. A subroutine in the navigation hardware for the Ariane 5 had been reused from the Ariane 4. It needed to operate within certain bounds of its variables, which had been shown for the Ariane 4 not to be capable of overflowing during the flight environment. However, the initial trajectory of the Ariane 5 was different, and checks had not been made to see if the design assumptions for the Ariane 4 navigation routines were still valid for the Ariane 5. They weren't. A variable overflowed, causing a series of events which ended in loss of control and destruction of the vehicle.

If the first two premisses are true, and the conclusion is false, then either the third premiss is false or the reasoning is invalid. We may see from the Ariane example (and others that I haven't quoted) that the reasoning in Figure 3.14 is more appropriate.

$$\frac{\begin{array}{l} \mathit{Failed}(S) \\ \neg \mathit{Failed}(\mathit{hardware}(S)) \\ S = \mathit{hardware}(S) \oplus \mathit{software}(S) \oplus \mathit{ReqSpec}(S) \end{array}}{\mathit{Either\ Failed}(\mathit{software}(S))\,\mathit{or\ Failed}(\mathit{ReqSpec}(S))}$$

(The term *ReqSpec* is used to denote the requirements specification)

Figure 3.14: Corrected Failure Reasoning

$$\frac{\begin{array}{l} \mathit{Faulty}(S) \\ S = \mathit{Part}_1(S) \oplus \mathit{Part}_2(S) \oplus \mathit{Part}_3(S) \\ \neg \mathit{Faulty}(\mathit{Part}_1(S)) \\ \neg \mathit{Faulty}(\mathit{Part}_2(S)) \end{array}}{\mathit{Faulty}(\mathit{Part}_3(S))}$$

Figure 3.15: Correct Failure Reasoning

The Role of Fusion in Failure Reasoning In the reasoning in Figure 3.13, the role of fusion is clearly indicated in the premiss:

But the system is composed of hardware and software.

The conclusion, that the software failed because the hardware didn't fail, was mistaken.

In the otherwise similar premises in the reasoning in Figure 3.13, the role of fusion is indicated in the premiss:

But the system is composed of hardware and software and its requirements specification.

In contrast to the reasoning in Figure 3.13, the conclusion in Figure 3.14, that the software or the requirements failed because the hardware didn't fail, is correct.

The difference between the two cases is the premiss involving fusion, and the incorrectness, respectively correctness, of the conclusion. I conclude that getting the fusion premiss right is an important component of correct reasoning about failure.

It seems that we should like to be able to reason as in Figure 3.15. A little thought shows that this kind of reasoning goes into many failure analysis procedures (software people call this 'debugging'). However, the s$\mathit{oftware}(S) \oplus$ h$\mathit{ardware}(S)$ example above shows that one must be very cautious in asserting that all the parts one thinks one has are all the significant parts of a system.

Is "Documentation" Part of the System? The difference between the reasoning with the false conclusion and that with the true conclusion is the constituents of the fusion in the premisses. In the Ariane example, the failure was actually in the specification of and determination of compliance with requirements. But our solution, including requirements in the fusion, entails the somewhat counterintuitive idea that the requirements specification, which includes or should include the limitations under which the system was designed to function, is actually part of the system itself. That is,

$$S = \mathit{software}_1(S) \oplus \mathit{hardware}_2(S) \oplus \mathit{requirements}_3(S)$$

It may indeed seem strange to include a *specification*, which is a piece of text, in with the physical components of a system. But it is not unprecedented: the system code is considered to be part of the system, and code is text too. How is a requirements specification different from, say, the system code? The system code can be considered a specification also; the system shall behave according to this-and-this instruction.

Adequate Decompositions One could define a decomposition of a system into parts as an *adequate decomposition*, when

(a) the system is the fusion of the proposed parts, and

(b) if the system fails, then one of the parts has failed also.

The purpose of an adequate decomposition is to enable reasoning about failure as in Figure 3.15. The example discussed, and others, show that most common engineering decompositions of systems into parts are not adequate decompositions.

System Accidents and the DEPOSE decomposition The accident sociologist Charles Perrow has argued in [Per84] that "*interactively complex*" systems which are "*tightly coupled*" suffer from a propensity to "*system accidents*", which are accidents caused by the system which cannot be put down to failures or misbehaviors of any of the parts.

If these accidents are considered to be failures of the system, then, according to the above definition, Perrow would be arguing that "interactively complex" and "tightly coupled" systems cannot have an adequate decomposition. There is, of course, no proof of this assertion, and it would be hard to see how there could be. Instead, Perrow could be taken to be arguing that a humanly possible decomposition of an interactively complex and tightly coupled system is unlikely to be adequate.

He has himself proposed a scheme DEPOSE for classifying complex systems into types of components. DEPOSE stands for

- Design
- Equipment
- Procedures
- Operators
- Supplies and Materials
- Environment

While Perrow's classification emphasises essential features, such as the design of procedures and the training and behavior of human operators of the system, which have not traditionally been examined with the same care as the physical components, he does not provide any argument from which it can be concluded either that

- For any complex system T, DEPOSE provides an adequate decomposition, that is,
  $$T = D_T \oplus E_T \oplus P_T \oplus O_T \oplus S_T \oplus E_T$$
  or that
- DEPOSE enables a more thorough categorisation of failure categories than traditional investigative techniques special to each industry.

Nevertheless, Perrow's work has inspired significant contributions to the study of human error possibilities and procedure design in complex system engineering.

Common Decompositions into Component Types Adequate decompositions may exist for certain classes of systems. A discussion of some common or useful classifications of complex systems into component types may be found in [Lad99].