Sadhana, Vol. 19, Part 6, December 1994, pp 941-969. © Printed in India

Design of software for safety critical systems

R K SHYAMASUNDAR

Tata Institute of Fundamental Research, Bombay 400 005, India

Abstract. In this paper, we provide an overview of the use of formal methods in the development of safety critical systems and the notion of *safety* in the context. Our attempt would be to draw lessons from the various research efforts that have gone in towards the development of robust/reliable software for safety-critical systems. In the context of India leaping into hi-tech areas, we argue for the need of a thrust in the development of quality software and also discuss the steps to be initiated towards such a goal.

Keywords. Formal methods; safety-critical systems; software design; synchronous programming paradigm.

"If only we could learn the right lessons from the successes of the past, we would not need to learn from our failures"
C.A.R. Hoare

1. Introduction

Historically and traditionally (Simon 1969) it has been the task of the science disciplines to teach about natural things: *how they are and how they work*. It has been the task of engineering schools to teach about artificial things: *how to make artifacts that have desired properties and how to design*. Webster's dictionary defines engineering as "the application of scientific principles to practical ends as the design, construction, and operation of efficient and economic structures, equipment and systems". By this definition, Computer Science can be viewed as engineering with the "design of programs" as one of the principle activities. Like many other professions, *design* happens to be the core of the engineering profession. It is surprising that one does find a discipline that could be called "philosophy of design" as a counterpart to "philosophy of science" – a well established discipline traditionally. As Herbert Simon argues, that the emergence of the activity of "Design of Programs" in computer science (usually termed *Software Engineering*) has also paved the way for "The Sciences of Design". The major aim of software engineering is to

*An earlier version was presented as an Invited paper at the ISRO Conference on Software Engineering, VSSC, Trivandrum, 29-30 July 1994.

direct the enormous resources of computational power on the silicon chip to the use and convenience of mankind.

Metaphor and analogy can be helpful, or they can be misleading. All depends on whether the similarities the metaphor captures are significant or superficial and ignore the underlying reality. As it stands, significant amount of research criteria for Computer Science has been borrowed from adjacent disciplines of Science, particularly Mathematics. For instance, in mathematics novelty and consistency are the main criteria for measurement of relevance or success rather than applicability[1]. The analogy between programming and traditional engineering disciplines has been very fruitful, and has provided the much needed basis and advantages. However, there are fundamental differences between software and other technologies. Some of the major differences arise due to distinct notions of complexity measures, analysis of reliability, usage of tools, standards etc. in software and other technologies (Hoare & Jones 1989; Parnas *et al* 1990). In that sense, analogy between software and other engineering disciplines breakdown on the following fronts (Hoare & Jones 1989; Parnas *et al* 1990).

1. Complexity measure: Software and hardware differ in the measures of complexity, be it design, development or usage.

2. Methods of achieving reliability: It appears that we should not have any difficulty in achieving reliability while designing software. The reasons are that basic raw materials for programs (registers, bytes, disks, tapes etc.) are almost unbounded and programs work in a controlled environment; there is no need to worry about defective components, friction, unskilled labourers, natural catastrophes like storms, earthquakes etc. This is not the case for the following two main reasons:

   (a) The most important way of achieving reliability in a product is *testing* under extreme conditions (perhaps taking into account a factor of safety) such as temperatures, pressure, voltage (essentially continuous variables) etc. However, in the case of software testing there is no analogous procedure of testing; establishing that the program works for the boundary values is no guarantee that it works for values in between. In some sense, the methods of extrapolation and interpolation that come to rescue in the traditional testing does not help at all as far as program designs are concerned. Lack of *continuity of behaviour with respect to input* in the sense of the traditional engineering products adds an additional dimension to the problems.

   (b) The discipline of software design is still immature and anyone who can type on a keyboard appears to have the impression that he/she can program. This has led to an attitude that there is no need to pay due atten-

[1] One may recall a quotation from Christopher Strachey in connection with the setting of school at Oxford in 1974: *"It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it do not have any clear understanding of the fundamental principles underlying their work. Most of the abstract mathematics and theoretical work is sterile because it has no point of contact with real computing. One of the central aims of the Programming Research Group, as a teaching and research group, has been to set up an atmosphere in which this separation cannot happen ..."* .

tion to the underlying mathematical abstractions. Design is developed on the fly, without much careful analysis and review.

3. Modular structure: The notion of *modules* in engineering is more often used in the sense of *independent* units spatially separated; that is, modules are not likely to interfere in the functioning of other modules. However, the situation differs when it comes to programming as there are no such spatial separations. In fact, even assuming the best *modular* approach, such a separation may not be possible due to efficiency considerations leading to practices such as code sharing techniques.

4. Tools: A programmer has to use the tools such as compilers (for programming languages), editors, environments etc. on which he does not even have a clear understanding (perhaps, they may even have bugs!). Above all he has to deal with software manuals which are more often than not highly unsatisfactory.

5. Evaluation: In spite of the fact that a large number of design methods have been proposed and a large number of systems has been built and used, one hardly finds evaluations that carefully demonstrate the success/failure of a system or of a method used in designing a system.

This paper is an attempt to draw lessons from the various research efforts that have gone into the development of reliable software for safety-critical systems.

The paper is organized as follows: Sections 2 and 3 provides overviews of formal methods and reactive/real-time systems respectively; this is followed by a systematic method of designing reactive systems in §4. Section 5 discusses case studies in the development of safety critical systems using formal methods followed by a discussion on the lessons drawn from the above experience in §6. Section 7 discusses one of the successful paradigms for reactive programming, namely *the synchronous paradigm*. In §8 we discuss the notion of *safety* in the design of safety-critical systems. In the concluding section, we discuss the steps needed to be taken in India for the development of reliable reactive systems in particular and reliable software in general.

2. Formal methods of software development

A method is said to be *formal* if it has a sound mathematical basis provided by its specification languages. Its main function is to check the consistency/completeness of the designers intentions and check whether it is realizable/implementable. It also provides a means of verifying whether the implementation meets its requirements and establish properties of the system without actually running the system. Of course, it is important that the method addresses the pragmatics of the designers; in fact, the success of any method also depends on how successfully the method addresses the various pragmatic considerations. Quite often, it is argued that the use of formal (mathematical!) methods is mandatory for improving the quality of software. The starting point for any formal method is the need for specification. Some of the important reasons for the need of specification are:

1. It serves as a contract (Lamport 1983) between the user and the implementor of the module. This serves to clearly articulate separate the following concerns:

   (a) Implementor: The implementor's responsibility is to meet the requirements imposed by the specification. Thus, there is no need for the implementor to know how the rest of the system works.

   (b) User: The user can use the module as a black box either in the use or the development of programs using this module.

   (c) Test: It precisely brings out testing criteria of the implementation.

2. As a communication among the implementors of the system.

3. Support the development of multi-version software.

Most of the problems that arise either during the development stage or during the use of a program can be attributed to inaccuracies, ambiguities and incompleteness of the problem/solution. The use of mathematical (formal) methods of specification guards against ambiguities and inaccuracies and a properly chosen method also enables to overcome inconsistencies and incompleteness issues.

There have been a plethora of methods for the development of software. Principal criteria used for the classification of software development methods have been (Place *et al*1990):

1. Representation: The foremost task is to represent the designer's intent. Various representations are feasible based on:

   (a) State-based/event-based specifications.

   (b) Style of specification such as declarative or model-based.

   (c) Abstraction features such as concurrency, nondeterminism.

   (d) Handling properties such as safety or liveness properties.

2. Transformation: Having represented the intentions at some level, the next task is: how do we transform the specification into another one that is more detailed than the one we have already, preserving the correctness? This is a very crucial step which also reflects largely on safety or the reliability of the method. The questions one asks are:

   (a) Does it support compositionality?

   (b) Does it support *rigorous*[2] derivation?

   (c) Is the refinement (or reification) completely mechanical?

   (d) Is there a pragmatic interface of the tools with the users?

3. Validation: This part corresponds to raising the level of confidence of the system relative to an environment using various testing, simulation and verification techniques/tools. With reference to verification, questions such as consistency, completeness, equivalence, safety and liveness become pertinent.

The methods can broadly be categorized based on the class of the underlying specification languages. Specification languages can be broadly classified as:

[2] This is used to mean that it is not necessary to perform proof; however, if needed it could be performed.

1. Model oriented (Concrete types),
2. Property oriented (Abstract types), and
3. Type oriented (dependent types).

VDM, Z, Raise are typical examples of model-oriented languages; LARCH, ASL, CLEAR, OBJ EML etc. are typical examples of the class of property-oriented systems and Calculus of Constructions is a typical variety in the type-oriented category. Further categorization is possible based on the support of programming languages/styles and paradigms such as object-oriented, concurrency, functional etc. We will not go into details on the specification languages further in this paper.

VDM and Z have been in industrial use for quite some time. VDM originated from design methods to develop concrete data structures from abstract specifications and hence, supports directly development. On the other hand, Z supports development of *requirement specifications* (discussed in the sequel) and thus, has properties such as conjunction, disjunction and also negation for describing constraints. Schemas in Z provide a pragmatic support for refining specifications. These formalisms have been used for specifying large software systems. However, these formalisms do not directly support features such as concurrency, communication features and also the specification of liveness properties and time. Temporal logic based systems are property-oriented systems and have the power to specify the behaviour of systems in a declarative way; consistency check can be achieved by building models which corresponds to building finite state machines. It is here, many of the restrictions appear. Even here, concurrency does not appear directly and "real-time" could be added to arrive at various classes of real-time temporal logics. One of the difficulties with these formalisms is the state explosion problem that one encounters when dealing with finite-state formalisms. These formalisms have been widely used for specifying requirement specifications. The lack of concurrency and the constraints one needs to specify in the context of reactive systems has also lead to look for formalisms, particularly algebraic so that one can do simulation and also arrive at verification tools based on various notions of bisimulation. We discuss the latter approach for the development of reactive systems in the sequel.

3. An overview: Real-time reactive systems

Computers are being increasingly used in a variety of applications ranging from home appliances or laboratory instruments to process control systems, flexible manufacturing, flight control and tactical control in military applications. In fact, their use has become essential due to stringent service requirements and availability of inexpensive hardware. For example, flexible manufacturing is a special kind of real-time application where the behaviour of each manufacturing machine can be adapted *instantaneously* to continuously changing working conditions while still satisfying a global optimality criterion. In flight control systems, *real-time* automatic maneuvering is used for significant reduction of fuel consumption and also for tactical control over the target. Needless to say that safety and reliability are extremely important for such systems since a failure may result in economic, human and ecological catastrophes. The term *Embedded systems* (has been almost synonymous with real-time systems) have become popular through Ada[3]. The term *Embedded* in Embedded

[3] Ada is a trademark of the US Department of Defense (AJPO).

systems refers to the fact that these systems are embedded in larger systems whose primary purposes are not general computations; the main purpose is to provide the needed support to achieve the overall objective of the system. One of the common concepts that counter a majority of the process control *embedded* systems is that of providing continual feedback to an unintelligent environment. The continual demands of an unintelligent environment cause these systems to have relatively rigid and urgent performance requirements, such as real-time response requirements and *fail-safe* reliability requirements. It seems that this emphasis on performance requirements is what really characterizes embedded systems, and causes us to be more aware of their roles in their environments than we are for other types of systems. Table 1 provides an informal classification of systems, based on properties that show up at the requirements level.

Table 1. An informal classification of systems.

Type	Characteristics	Examples
Embedded systems	• Special purpose (application) • Absolute performance requirements	• Industrial process control systems • Flight guidance systems
Data-processing systems	• Special purpose (application) • Relative performance requirements	• Batch business programs • Online data base systems
Support systems	• General-purpose • Relative performance requirements	• Operating systems • Software development tools

Systems such as airline-reservation systems should probably be viewed as a combination of these types. In addition to the performance requirements, which have been already established as a major distinguishing factor, embedded systems are especially likely to have stringent resource requirements. These are requirements on the resources (mainly physical in this case) from which the system is constructed. This is because embedded systems are often installed in places (such as satellites) where weight, volume, or power consumption must be limited, or where temperature, humidity, pressure and other factors cannot be as carefully controlled as in the traditional machine room.

The interface between an embedded system and its environment tends to be complex, asynchronous, highly parallel and distributed. This is another direct result of the *process control* concept, because the environment is likely to consist of a number of objects which interact with the system and each other asynchronously in a parallel fashion. Furthermore, it is probably the complexity of the environment that necessitates computer support in the first place (consider an air-traffic-control system!). This characteristic makes the requirements difficult to specify in a way that is both precise and comprehensible.

Finally, embedded systems can be extraordinarily hard to test. The complexity of the system/environment interface is one obstacle, and the fact that these programs

often cannot be tested in their operational environments is another. It is not feasible to test flight-guidance software by flying with it, nor to test ballistic-missile-defense software under battle conditions.

Embedded systems are often used in critical applications where real-time/reactive response is essential. The main characteristics of the embedded systems are summarized below:

1. The primary purpose is to provide the needed support to achieve the overall objective of the system rather than general computations.
2. The system tends to be large, complex and can be extraordinarily hard to test.
3. The environment that the system interacts with is nondeterministic. That is, most of the times, there is no way to anticipate in advance the precise order of external events.
4. High speed external events (perhaps in parallel), must be able to affect the flow of control in the system easily.
5. The real-time behaviour must be controllable, e.g. the requests must be handled within certain time limits.
6. The system is a coordinated set of asynchronous distributed units.
7. The mission time is long. The system not only must deal with ordinary situations but also must be able to recover from some extraordinary ones.

4. Design and development of reactive systems

The process of design may be viewed as an iterated transformation of a conceptual abstract functional description through refined levels of description till the emergence of an implementation. Between the concept and implementation (realization of the concept in hardware or software given the physical resources), there can be various stages such as Service requirements, Functional requirements, Architectural requirements, Performance requirements, Detailed Design etc. In fact, there is no agreement about the precise meaning of a stage and also it is not really clear as to where one stage ends and the next begins. However, informally one understands the various ranges from concept to implementation, and we refer to each stage as a *specification*. Assuming, one has a *good view of customer's/designer's intents*, the following two stages (Pnueli 1986) generally play a vital role in the development of systems:

1. Requirement specifications: This is perhaps the closest to the conceptual understanding of the problem (and thus, the earliest stage). In a sense, this could be viewed as the basis for a contract between a customer who orders the system, and a representative of an implementation team who is supposed to construct the system. The important parts of a requirement specifications are the *static part*, which identifies the interface between the system and its environment, and the *dynamic part*, which specifies the behaviour that the customer expects to observe on the identified interface. It may be noted that the identification of the interface depends on various architectural issues. In

short, the requirement specification concentrates on the observable behaviour between the customer and the various components of the system without the knowledge of the implementation (and hence, the internal structure) of the components of the system.

2. System specifications: This has to achieve the "how" part from the "what" part and the architectural considerations. It has basically the following concerns:
   - Architecture of the system: The major concern here is to decompose the system into a set of (hierarchical!) subtasks.
   - Mapping the logical architecture into the physical architecture.
   - Interactions among the sub-tasks.

In a broad sense, system specifications are almost executable. However, it may be noted that a system specification need not represent an actual implementation.

If both a *requirement (service) specification* and an *implementation specification* have been constructed for a system, it is possible to validate the implementation specification by confirming that it satisfies the service specification. This ability is very valuable as the implementation specification is often quite complex and prone to error, while the service specification is much shorter and simpler.

4.1 *What's the concern of requirement specification?*

By specifying the minimum required externally visible behaviour, and leaving all other aspects to lower levels of description, one can obtain a more general specification that reflects the necessary requirements of the system. A specification that is oriented towards one implementation may discourage or even preclude other equally valid implementations. Thus, this stage can be viewed as capturing either *the behaviour of the customer and the system as seen by an observer who can see the components (customer included) as a black box* or *the properties the system should satisfy.*

In a real-world, it must be evident that when a system is being developed it is generally the case that the requirements would have to be augmented either because an incompleteness was detected in the informal description of the customer/designer or the mistake/unsatisfactory performance was detected after the system was designed. Thus, the two most important concerns in this stage are

1. Consistency: A spectrum of examples can be seen where requirements contradict each other (it could be in the safety requirements or between the safety and eventual requirements). The *consistency* ensures that there is some execution model satisfying the requirements.

2. Completeness: The main concern here is: Does the stipulated requirements ensure *what the customer wants* or the requirements have to be strengthened to avoid *undesirable behaviour*?

Thus, logic-based formalisms would be appropriate for this stage. There have been a spectrum of formalisms such as variants of temporal logics (Pnueli 1992) suitable for this style. These logic-based formalisms provide a good mechanical support for checking the above two properties (though the general problem is undecidable).

4.2 *What's the concern of system specification?*

As mentioned already, the major concern of this stage is to arrive at a model satisfying the requirements. That is, this stage provides an abstract model relative to an environment satisfying the requirements. As it provides a model, it is necessary for this stage to be concerned with abstract implementation operators such as non-determinism, concurrent and hierarchical that are concerned with the architecture the user has in mind for building the system. In other words, it is desirable that the specification is executable and simulatable from which further custom-made implementations can be derived. By definition consistency does not play a role except that one would have to ensure that there are non-vacuous domains of answers for the system. It must be noted that it is not necessary that there should be one-to-one correspondence between the requirements and the logical units of the system. The property of completeness also does not play a role except at successive refinements of the system specifications where some units may be omitted.

Traditionally, system specifications have been expressed as interacting state machines; such an approach inevitably suffers from over specification as the state machines represent an implementation. If the application is such that only one implementation is envisaged, an implementation oriented specification may be acceptable; but other applications, for example communication protocol specifications, envisage many distinct implementations. Formalisms such as CCS, CSP (Hoare & Jones 1978), (Berry 1992) are some of the formalisms that have been used as successful formalisms for system specifications.

4.3 *What does verifying specifications mean?*

The notion of verification of specifications largely depends on the language frameworks used for writing specifications. It is usual to write specifications either in the axiomatic or the algebraic frameworks. Frameworks used for writing specifications can be broadly categorized into (Pnueli 1986):

1. Two language framework: This corresponds to the usual axiomatic specification wherein one language is used for specifying the properties of the program and another language is used for specifying the abstract model/the program itself.

2. Single language framework: Here, the same language is used for specifying the properties as well as the model or the implementation.

Verification in two language framework: One of the widely used methods of capturing the properties of the program are through state changes (Lamport 1983). As usual hierarchical specifications forms the basis of mastering complexity. Thus, the main question is to ensure that the *lower level specification* is a refinement of the *immediate higher level* in an iterative manner. To make it more precise, consider a specification at the lower level, say S_i described as

$$S_i : \text{there exist state functions } g_1, \cdots, g_s \text{ such that } B_1, \cdots, B_n$$

and a higher level specification S_{i-1} given by,

$$S_{i-1} : \text{there exist state functions } f_1, \cdots, f_r \text{ such that } A_1, \cdots, A_m$$

To show that the lower level specification is a refinement of the higher level, we must find expressions $F_i(g_1, \cdots, g_s)$ such that each A'_k obtained from A_k by performing substitutions

$$f_i \leftarrow F_i(g_1, \cdots g_s)$$

follows logically from the axioms $B_1, \cdots, B_n$. Now consider a still lower level specification

$$S_{i+1} : \text{there exist state functions } h_1, \cdots, h_t \text{ such that } C_1, \cdots, C_p$$

This can be shown to be a correct refinement of S_i by finding substitutions

$$g_j \leftarrow G_j(h_1, \cdots, h_t)$$

which yield formulae B'_l that can be proved from $C_1, \cdots, C_p$. It follows that the formulae A''_k obtained from A_k by the substitutions

$$f_i \leftarrow F_i(G_1(h_1, \cdots, h_t), \cdots, G_s(h_1, \cdots, h_t))$$

can be proved from $B'_1, \cdots, B'_n$, which in turn can be proved from $C_1, \cdots, C_p$. Thus, by transitivity, it follows that S_{i+1} is also a refinement of S_{i-1}. Hence, if each layer is correct, one can show that the lowest level of specification implements the highest level of specification.

The crux of saying that the *refinement is correct* becomes simple (with little leaps of intuition) when both the levels are assertions about the same model as *state refinements in that case can be structured appropriately to preserve the intended intuitions*. In case, the model on which assertions are made in the sequence of refinements are not close to machine languages, then one has to establish that *there is a correct implementation of the language*. It is here that a sound and a complete semantics of the language plays a vital role. It is possible to use variants of temporal logic (Lamport 1983) as the vehicle for specification (translation) at every stage of refinement and to go down to the levels of concrete high level programming languages or even lower level programming languages. For this purpose, one could use the mechanical theorem provers to advantage. Thus, establishment of the correctness of refinement corresponds to proving that the lowest level of specification implements every level in the hierarchy.

Verification in single language framework: The basis of verification in this framework is *equivalence of programs or specifications*. That is, one usually specifies a simpler program and shows that the two programs are behaviourally equivalent. The equivalence establishes that the implemented program behaves like the other program; that is, it is as good or as bad as the simpler program. Thus, a refinement relation $\sqsupseteq$ can be captured for a hierarchical sequence $P_0 \sqsupseteq P_1, \cdots \sqsupseteq P_m$ moving from the simplest naive specification to a concrete implementation P_m.

4.4 *What style to choose?*

It is evident from the above discussion that the two styles are complementary to each other and both stages play a vital role in the design cycle. In fact, the two stages play an important role irrespective of the language frameworks. In the context

of two-language frameworks the distinction is apparent. In the context of single-language frameworks the use can be seen if one visualizes the coarse level specifications/programs with which the fine-grain solution will be checked for equivalence. For the success of the method of two stages it is important to choose an appropriate specification language for each stage. Let us now take a brief look at the choice of specification languages.

The requirement specification essentially states *what system* is to be developed at *what costs*, and under *what constraints*. In other words, it is used

1. as a vehicle of communication,
2. as a scope for modifiability,
3. to constrain target systems, and
4. for accepting or rejecting final products.

Thus, it is necessary that any candidate specification language for this purpose should satisfy these requirements in general. From the point of view of (1)-(2), it is clear that the language must be understandable and modifiable; the latter property asks for the *conjunctive property*. From the point of view of (3), it follows that the language must be precise, unambiguous, internally consistent, and complete. Furthermore, the requirements specification should be minimal, i.e., define the smallest set of properties that will satisfy the users and originators. Otherwise, the specification may over-constrain the target system, so that some of the best solutions to design problems are unnecessarily excluded. For instance, at this level there is no need for the specification of nondeterminism and parallelism. Property (4) enforces the requirement of formal manipulatability (if verification is to be used) or testability (if testing is to be used) on the specification language.

Linear temporal-logic based formalisms (Pnueli 1986) are conjunctive and satisfy the needs for specifying the needed safety and eventuality properties to a large extent. However, as temporal logic suffers from its orientation towards eventuality rather than immediacy or quantization, one would have to use real-time temporal logic formalisms. Another important consideration in the selection of the particular variant has been the availability of model checkers for the language on hand so that one can attempt at automatic verification of properties of finite state systems.

For system specification purposes, abstract notations such as CCS, temporal logic of actions, refinements of temporal logic and concrete languages such as CSP, Esterel etc. are applicable. The choice will also depend upon how close the specification is to be with respect to the implementation and also the type of verification and the tools one would be interested for verification purposes.

4.5 *Software maintenance and formal methods*

The need for modifications to old programs cannot be under estimated. Some of the changes are necessitated by the need to adapt the program onto another system, the need to arrive at a new system with some changes (perhaps incremental) in the specification of some of the components or discovery of errors. Often it is not possible to write an entirely new program due to considerations of time and cost. It is here the choice of the formalism, documentation and the ingenuity of the programmer plays a crucial role. If the formalism chosen is *compositional (or modular)* then the

task would be proportional to the changes required. At least if the programmer had properly articulated (and documented!) those aspects of the program that could perhaps be subjected to future modifications, then the task becomes manageable. In fact, modifications indeed could be effected economically and reliably if one had chosen proper mathematical (formal) method of development (Hoare & Jones 1986). If the structure of the program P is the same as the structure of the specification S, then it is sufficient to ensure that the modified component meets the modified specification. But it is not always possible to preserve the structure of specification in the design of the program. This is so because a specification is often most clearly structured with the aid of such logical operators such as negation and conjunction, which are not available in an implemented programming language. Nevertheless, mathematics can come to rescue. If the program P has an *approximate inverse* P^{-1}, defined in the same way as for the *quotient*, then it is possible to calculate the new proof obligation of the program. For details the reader is referred to (Hoare 1989).

4.6 *Validation*

Correctness is the process of establishing either the equivalence of two objects or refinability of one object from the other. Thus, we can say that a system or an implementation is correct if it meets the *designers' intentions*. Thus, unless the the designers' intentions are formalized, the notion of correctness becomes vacuous. The notions of verification in the single- and two-language frameworks have been already discussed in the previous sections. One of the important processes in the development of a system is to show that the system indeed satisfies the *Designer's intentions*. This process is often referred to as *validation*. It must be noted that *Designers' intentions* is not completely formal. Thus, validation is not a complete *formal process*. Thus, for validating a system, one uses test scenarios and uses simulation etc. For the latter, if the specification is executable (like system specifications) then one can validate the system easily. It is important to note that the process of validation is an informal process; the purpose is to ensure that the specification conforms to the intents.

5. Case studies of formal methods in the development of safety critical systems

In the following, we take a brief look at some representative industrial scale software systems for which formal methods have been applied for design, validation, review of the system. We briefly discuss the experience in the use of formal methods in the design, development or review of large software systems in the context of the following areas:

1. Railway Signaling Systems.
2. Nuclear Power Plants.
3. Aviation Control.
4. Medical Systems.

The first example illustrates the study of the development of Paris Metro Signaling system where formal methods were deployed right from the beginning and resulted in substantial economic benefits. The second example illustrates the deployment of formal methods for gaining the level of confidence in an already developed system. The third example illustrates how a simple formal model aids in the understanding of the design by engineers and users as well and thus, enables to remove inconsistencies. The last example illustrates the need for a formal analysis of systems particularly medical life-critical systems for gaining the confidence of regulatory agencies and users alike. Many other case studies have been reported by Gerhart *et al* (1994).

5.1 *Paris Metro signaling systems*

In the Paris Metro signaling system, formal analysis was used to determine that the separation time between two trains on the Paris Metro system could be reduced from 2 minutes 30 seconds to 2 minutes while maintaining safety requirements. More importantly, the successful deployment of the signaling system removed the need for building a new third railway line, thus saving billions of dollars in costs. There was a serious concern about the predictability and reliability of large amount of software and the associated cost in testing the system exhaustively because of safety considerations. This case study is a positive example of the major benefits to be gained by the deployment of formal methods throughout the design and development of the system.

The developers started with Hoare's method of proving correctness of programs (note that the development of the system started in early 80's!). Using this technique three sets of proven software were developed. The question that arose was: *Are these sets consistent?* Consistency of validated sets of software were approached as follows:

1. Top-to-bottom re-specification and refinement.
2. Bottom-to-top re-specification and verification.
3. Match the two at some level close to the code.

Such a revalidation by proof re-engineering validated both their method and design. The tools used in the design and validation (revalidation) of the system were broadly the verification-condition generators and the tools around the B Method (Abrial *et al* 1991).

The success of the project was not due to a particular methodology for formal methods used or the advanced nature of tools deployed. Such methods and approaches are supported in most verification systems, and more advanced and powerful theorem provers with sophisticated interfaces are already available. The important message learnt was the value of combining formal methods with prototyping, simulation and testing. From the point of view of the use of formal methods in the development of software, the experience shows:

1. Formal specifications help developers understand the requirements clearly.
2. Formal methods play a major role in developing confidence in the system.

The experience of the case study has been reused by the GEC Alsthom Transport Corporation to sell similar related technology and expertise in other projects such

as SNCF, France and Controlled Deceleration Control, Calcutta, India. In these projects, the application of formal methods is viewed as a success, and the resulting software is considered to be efficient and of good quality. Formal methods have now become a part of the GEC Alsthom's software strategy.

5.2 *Darlington Nuclear Generating Station*

In contrast to the Paris Metro signaling system in which formal methods were used in all stages of the design and development, the Darlington case study is an example to show how the formal methods helped in enhancing confidence in the functionality of a system that was developed without using any formal methods. This suggests that formal methods can be useful in analyzing an existing design and implementation, a kind of reverse engineering.

The Darlington station is a four-reactor nuclear plant in Toronto, Canada. The reactor had two fully computerized shutdown systems; the first one drops neutron-absorbing rods into the core; the second injects liquid poison into the moderator. The systems are safety-critical and require high levels of confidence. The shutdown system had been designed and developed using conventional software engineering practices. It had gone through unit testing, integration testing, validation testing, in-site trajectory-based random testing, software assessment, and software hazard analysis. However, the review of the software before licensing uncovered discrepancies and raised doubts as to whether the software implemented the requirements correctly. There were also some ambiguities and doubts about the requirements themselves. Since this was the first such system using computerized shutdown system, the Atomic Energy Control Board of Canada was not willing to grant the license to operate the plant because of these doubts about the reliability of the shutdown systems.

Each month's delay in getting a license was costing Ontario Hydro $20 million in interest payments for the whole nuclear generating station. The Ontario Hydro was not keen on redesigning the software, and in order to reduce the licensing impasse, a compromise was reached. The shutdown software was to be formally inspected:

1. Formalize informal requirements by generating specification tables.
2. Use the existing code to develop program-function tables for it.
3. Demonstrate that the code is consistent with the specifications.

In other words, the approach was primarily a reverse-engineering exercise, in the sense that code already existed when the formal specifications were written. The method called *Software Cost Reduction*(SCR) originally developed at the US Naval Research Labs was adopted for the inspection.

The above three steps were done by three independent teams. Deriving specifications from informal requirements consisted of arriving at mathematical formulas which would show the effect anticipated; deriving program-function tables corresponded to arriving the effect of code procedures. Most of this analysis was done by hand; proofs were done by hand with little use of automated tools. Microsoft Excel was used to produce these tables. Finally, the proof of consistency consisted of manually comparing the specification and program-function tables, and transforming the tables. The last team also reported the discrepancies to the other two teams.

The experience of gaining confidence through such an approach was felt worthy by the developers. The main lessons learnt can be summarized as follows:

1. The analysis of requirements clarified the specifications and removed some safety-related ambiguities. In particular, it became easier to understand the operational behavior.

2. Without the formal methods, there would have been no proofs of correspondence between specification and program-function tables. The formalisms became handy in the review extensively and raised the level of the confidence of the system. It also helped identify some problems with timings.

The result of this exercise was that the Atomic Energy Control Board of Canada felt more confident in granting operational license for the plant.

5.3 *Aviation control systems*

The third case study concerns the Traffic Alert and Collision Avoidance (TCAS) system being developed by the US Federal Aviation Administration (FAA). All aircrafts having more than 30 seats are mandated to have such a system installed to avoid midair collisions. The goal has been to develop requirements and design of TCAS Logic and TCAS Surveillance, which is to provide traffic advisories and recommend vertical maneuvers in the event of an impending collision.

There existed a natural language specification of TCAS which included a pseudo-code description of CAS Logic. In 1990, FAA and others had become concerned about the informal requirements specifications especially because errors had been uncovered through the simulation of the pseudo-code; the specification had to be revised quite a few times. FAA wanted to further clarify the TCAS requirements and obtained improved confidence in the system.

The TCAS methodology is directed at process-control systems designed to maintain an acceptable relationship between a system's inputs and outputs in the presence of disturbances to the process. Leveson (1986) and her students at University of California, Irvine, have developed a notation based on Harel's Statecharts for specifying such systems. At the beginning, they wrote most of the specifications of TCAS, and domain experts reviewed them, but later the domain experts took over. The experience shows the the Statechart-like specification has made it easy to specify the transition logic of the TCAS subsystems and has provided a good satisfaction and a comprehension of the specifications for the designers. Engineers and others not trained in formal methods can review and modify the specification. In particular, the use of formal methods eliminated the development of an informal natural language specification.

5.4 *Medical therapy systems*

The increasing use of medical instruments for life-support and life-critical systems has made it necessary to ensure a high degree of dependability/reliability. In fact the regulatory requirements from agencies like US Food and Drug Administration has made it mandatory for software developers to establish the safety not only of the software being designed but also establish the safety of the software that has already been developed.

In (Mojdehbakhsh *et al* 1994), a retrofitting software safety analysis is developed for implantable cardiac-rhythm-management systems. The developers and the customers were convinced that the formal analysis successfully identified and mitigated numerous software-safety faults and eliminated several hazards. Some of the safety faults that had been undetected by the reliability analysis (essentially, done through the system fault-tree approach) has been captured through such an analysis.

The use of formal specifications (Ladeau & Freeman 1991) in the development of a bedside instrument used to monitor vital signs in patients in intensive care units and operating rooms has shown the uncovering of several problems and ambiguities in the informal product specification. Several other successful applications have been summarized by Bowen & Stavridou (1993).

6. What have we learned from the use of formal methods?

In this section, we briefly discuss the feedback from the experiences in using formal methods in the development of safety critical systems (Kapur & Shyamasundar 1994). The use of formal methods is pivotal in the design, development, and maintenance of complex systems. They also form a good medium for review and reverse engineering. However formal methods cannot and should not be viewed as a substitute for identification and use of good abstractions relevant for the application domain. It is the proper use of appropriate abstractions that leads to good design, structure and implementation of complex systems. Selection and deployment of relevant abstractions can also reduce the cost and increase the effectiveness of formal methods. In fact, formal methods can be used to assess the selection of appropriate abstractions; a simple case of specifying transitions in the TCAS using predicate calculus has made the specifications more comprehensible to engineers and non-engineers.

Formal methods are likely to be most cost-effective and have a bigger payoff when used in the earlier stages of the system life-cycle. In the later stages of the life-cycle, the use of formal methods is expensive and time-consuming. It is thus more economical to apply formal methods to specifications and designs than to programs. There are at least two reasons. Firstly, sooner the mistakes and errors are found in a specification, better it is in terms of cost effectiveness and avoiding delays in completing a software project. According to Fairly (1985), it is 5 times more costly to correct a requirement fault at the design stage than during the initial requirements, 20 to 50 times more costly to correct it at acceptance testing, and 100 to 200 times more costly to correct the problem once the system is in operation.

Studies show that almost all accidents involving computerized process-control systems are due to inadequate design foresight and requirement specifications, including incomplete or wrong assumptions about the behaviour or operation of the controlled systems, and unanticipated states of the controlled system and its environment (Leveson 1991). In 203 formal inspections of six projects at JPL, Kelly, Sherif and Hops found that requirement documents averaged one major defect every three pages (a page is 38 lines of text) compared with one every 20 pages for code. Two-thirds of defects in requirements were omissions (Rushby 1993). Similar experience is reported from the space-shuttle, Voyager and Galileo projects at NASA. The need for reliable, correct specification cannot be overemphasized.

Secondly, specifications and designs are high-level and can be described function-

ally where formal methods can be applied nicely and efficiently. There is a need for testing specifications to enhance designer's confidence in them and ensuring that specifications have the desired structural properties. Concentrating on consistency and completeness at higher levels of specification goes a long way in the realization of quality software. It is to be noted that several automated tools of deduction are indeed available for this purpose presently.

It was increasingly felt in most case studies that formal requirement and design specifications are essential. They not only assist in early discovery of incomplete and ambiguous specifications, they lead to clarification, crystallize incompatibilities in understanding of the clients and designers thus resulting in better comprehension of client's informal requirements and gains in assurance. The recent work in the context of TCAS and Paris Metro Signaling System is an indication of that.

A new programming paradigm called *synchronous paradigm* (Berry & Gonthier 1992) has provided a good basis for programming reactive systems; some of the languages based on this paradigm are Esterel, Lustre, Signal, and Statecharts. Esterel, one of the earliest languages of this family, has a highly sophisticated environment for design and verification. A striking feature of this system is that the the verified program and executed program are almost the same. Such a property makes it possible to have high levels of confidence in the verification done through the tools based on *bisimulation*. This paradigm will be further explored in the next section.

Formal methods are somewhat more difficult and cumbersome to apply to programs because of having to deal with state and state change. At this stage, one can rely on testing, validation tools and function/procedure tables. Program verification is most advantageous for (i) safety-critical kernels whose ultra-reliability is crucial for the behavior of the rest of the system, and (ii) for hardware chips whose behavior can be described functionally. For an overview of the use of theorem proving in software design and related issues, see (Kapur *et al* (1992) and Kapur (1993).

It should be possible to integrate formal methods to existing software engineering practices in a smooth manner thus enhancing the confidence in software engineering methodologies. Formal methods are most productive and effective when they are combined with other techniques such as prototyping, simulation, verification and testing.

Although having access to automated tools for using formal methods such as theorem provers, simplifiers, etc., would be handy, most studies seem to suggest that formal methods can be used without much automated support. In fact, in most industrial case studies, automated support was available mostly for type checking and syntactic correctness. Given that theorem proving technology has substantially developed over the past two decades, integration of powerful theorem provers and proof checkers could go a long way in mitigating some of the work involved in doing proofs. In the Darlington case study discussed earlier, all tasks (except for documentation) were performed manually, because of which the process of formally inspecting the already developed software became very labor intensive; it was estimated that 35 person-years of effort was expended on the walk-throughs! If there were tools available to perform automated reasoning for walk-throughs as well as to record dependencies among programs through their function tables, the time taken to perform walk-throughs would have been considerably reduced thus saving considerable money. For a discussion of how theorem proving technology should be further advanced for making it better suitable for software design, see Kapur (1993).

Regulatory agencies also appear to be satisfied with the use of formal methods for gaining assurance in product development even if it is not developed using formal methods. This is a kind of reverse engineering where formal methods are used to assure the regulators that software did what it was supposed to do.

1993, 1994) and

Re-usability is critical for cost-effectiveness and acceptance of formal methods. In the case of the Paris Metro project, the reuse of models and theories was helpful in two subsequent projects at GEC Alsthom. GEC Alsthom not only used formal methods to establish that there was no need for an extra railway line, but the developed expertise is being used to market its services in other projects. Object-oriented methodology can be helpful in designing reusable code, designs, theories, proofs, as well as specifications.

7. Synchronous paradigm for reactive systems

In this section, we highlight one of the new paradigms referred to as the *synchronous paradigm* which is being used successfully in the design of reactive systems and hardware systems. The synchronous paradigm is founded on the *perfect synchrony hypothesis*. The hypothesis states that the considered reactive programs respond in no *time* (i.e., the elapsed time is not observable) and produce their outputs *synchronously* with their inputs. In fact, this hypothesis takes roots in classical mechanics, being akin to the Newtonian's instantaneous body interaction principle – which is still useful in practice even though it has limitations at speeds beyond some ranges[4]. Further, perfect synchrony hypothesis can be seen through the assumption of an ideal perfect real-time machine in the works on control theory. Esterel is the first programming language to have been based on this principle.

First, let us take a quick look at the immediate gains for the specification of real-time systems with such an approach. One of the immediate gains of the perfect synchrony hypothesis can be seen by looking at the following paradoxical question in asynchronous languages, such as Ada, supporting specification of real-time:

Is `delay 8 ; delay 6` $\equiv$ `delay 14` ?

There is no clear cut answer to this question in an asynchronous framework. In the semantic theories of CSP-R (Koymans *et al* 1988), and Timed-CSP (PRG 1992), the paradox has been overcome through the notion of executional models like *maximal parallelism* and *maximal progress* respectively. Obviously, such a paradox clearly does not exist under the perfect synchrony hypothesis.

The next most important gain is that the notion of *physical time* can be replaced by a simple notion of *order* among events; the only relevant notions are the simultaneity and precedences among events. Thus, physical time does not play any special role; it will be handled as an external event, exactly as any other event coming from the programming environment. This is referred to as the *multiform time*. As an example, let us consider the two following requirements:

[4]It may be recalled that Einsteinian physics does not void the utility of Newtonian physics. Both are useful approximations that may be beneficially used for modeling and analysis of systems that fall into well defined ranges, where the main distinguishing parameter is how close are the typical speeds in the system to the speed of light. The same observation holds for the discipline of application of formal methods for the specification, design and verification of reliable reactive systems.

```
The train must stop within 10 seconds

The train must stop within 100 metres
```

Conceptually, these two requirements are the same; one could argue as to which is specific or which is more general. In systems having an internal clock and handled by special statements, conceptually similar requirements will be expressed in different ways. However, in languages following the above hypothesis, there is no notion of internal clock and hence, will be expressed by similar precedence constraints. In other words, a reactive system is completely event driven and we can consider the life (history) of a system to be divided into *instants* that are the moments where it reacts; or the history of a system is a totally ordered sequence of logical instants. Thus, one can speak of the ith instant of a program. Event occurrences which happen at the same logical instant are considered simultaneous, those which happen at different instants are ordered as their instants of occurrences. Apart from these logical instants, nothing happens either in the system or in its environment. Finally, all the processes of the system have the same knowledge of the events occurring at a given instant.

First, let us see informally how we can specify a vending machine. The behaviour of a typical Biscuit Machine can be described as follows:

1. It waits for insertion of 5 cents, ejecting a packet of biscuits or attempting to eject when there are no packets left,

2. The serviceman may open the machine by inserting the service key. After being re-stocked and the removal of the service key, the vending machine resumes where execution was interrupted.

3. The above process is repeated except if the power goes down, in which case it is possible to be cheated of 5 cents.

First, let us describe the basic behaviour considering only requirement (1) assuming that there is enough stock of biscuits and there is no power tripping. In this case, the observable events are waiting for `5 cents` and ejecting the pack ot biscuits after receiving `5 cents`.

```
loop
   await ?5_CENTS; (* wait for 5_cents *)
   if   AVAILABLE then "EJECT"
forever
```

The synchrony hypothesis avoids questions such as: What happens if one puts several `5 cents` since for an input there is a *deterministic instantaneous* reaction.

Considering requirement (2), we can describe the behaviour by

```
loop
   await ?5_CENTS;
   if   AVAILABLE then "EJECT";
   else
        await  INSERT_SERVICE_KEY;
        "EJECT";
   endif
forever
```

It is easy to observe that if the local actions are assumed to be instantaneous then the above description gives the intended behaviour; note that *instantaneous* does

not mean that order of events in the same instant can be ignored; in fact, the order of events should necessarily be preserved- the only thing is that time separation between events is not observable. For taking into account the third requirement, we need some sort of a preemption. This can be done by introducing a guard to watch the execution. The behaviour can now be described by,

```
loop
  do                                  (* Begin of a guarded statement *)
     await ?5_CENTS;
     if   AVAILABLE then "EJECT";
     else
          await  INSERT_SERVICE_KEY;
          "EJECT";
     endif
   watching ?powerdown                (* GUARD *)
   end
 forever
```

In the above code, do *stat* watching S corresponds to preemption of the execution of the it body on receiving S. It can be observed that in the above example, we have been able to develop the program by taking requirements one at a time (that is, conjunctive). In fact, it appears that this feature holds in general. Informally, one can see that there is an in built priority in the specification or interrupts in the implementation. Later, one can see that Esterel framework provides for a nice integration of interrupts and exceptions.

For a language to support reactive specification, we can see that it should support broadly the following features:

- Sound mathematical semantics: The very need of safety requirements of the applications are enough to convince oneself that the behaviour of a program must have a sound and a unique meaning. Such a requirement calls for a sound mathematical semantics.

- Determinism: In any reactive system, one should be able to predict the behaviour relative to the sequence of sets of events. Obviously, this is a must. One should carefully distinguish between the need of nondeterminism in specification and the requirement of deterministic behaviour with respect to possible set/sequence of input stimuli.

- Concurrency: Most applications involve naturally concurrent and communicating components and thus, the language should support its specification at a logical level. It must however be pointed out that one should carefully distinguish between compile-time concurrency and physical run-time concurrency. It is of interest to note that logical concurrency makes it easier to write programs but does not necessarily correspond to the architecture of the executing machine. For an interesting discussion on parallelism as a structuring technique, the reader is referred to [MuSe 92]. However, the physical concurrency reflects the underlying machine's architecture. In the context of distributed programs, physical parallelism is often associated with code *distribution* that correspond to execute some sub-programs on distinct machines communicating through a network.

- Verification: From the reliability point of view, it is absolutely essential that it should be possible to verify the properties of the programs. Since the task of verification is quite complex, it is necessary to have automatic tools to assist the verification of the program.

7.1 *Languages, validation and verification*

There are various languages that support synchronous paradigm. The principal languages are Esterel (imperative), Lustre (data flow), Signal (equational), Statecharts (graphical) etc. Esterel and Lustre are synchronous, and deterministic where as Statecharts is not necessarily deterministic; Signal is not completely reactive (due to the oversampling operator). Esterel is one of the earliest languages of this family and supports a powerful programming environment. The main tools can be classified into:

1. Simulation and development tools: These tools execute compiled automata of Esterel programs instants by instants. It can be linked with a *C* standard library that allows interactive simulation through the keyboard. It also supports a graphic simulation through the X-window library. Recently, it supports an integrated environment called AGEL. The advantage of these tools is that one can validate the system and visualize the working of the program (and the reactive modules) clearly.

2. Verification tools: The Esterel program is compiled into an automata. Now, one can use various techniques for verifying the finite-state automata and also the tools that are used for verification using process calculi. The tools such as AUTO/AUTOGRAPH used for verification of process calculi have been integrated in the environment and thus, enables one to verify using notions such as bisimulation and observable criteria.

One of the main advantages of the system is that the verified code is close to implementation and hence, the proof becomes very reliable. In fact, it is because of this reason, the designer of Esterel, G. Berry claims *What You Prove Is What You Execute*. The language is being widely used for the development of reactive systems and the synthesis of hardware.

7.2 *Illustrative specifications*

In this section, we illustrate the development of an Esterel program from a typical specification; we only give fragments of code for lack of space and use loose syntax.

Slotted ALOHA

This example illustrates how the features such as *broadcast* and *logical concurrency* are helpful in arriving at concrete implementations.

In this protocol, the satellite acts as a repeater, rebroadcasting messages received from independent ground stations. The principal features of the protocol (Tanenbaum 1981) are captured below:

1. The satellite broadcasts a clock signal dividing time into discrete intervals, or slots; it is this feature that tries to avoid collisions.

2. A ground station which is ready to transmit must wait for the next slot before broadcasting.

3. If only one station transmits during a particular slot, then the satellite will receive the message in tact, and will rebroadcast it to all ground stations. If two users broadcast simultaneously, the satellite will rebroadcast the sum of two incoming signals, resulting in garbage. It is assumed that each packet contains a checksum strong enough to permit the receiver to detect all collisions, so damaged packets can be discarded.

4. The transmission is in terms of packets of the same length (the throughput increases with such restrictions).

5. Another important property of satellite packet broadcasting is that the sender can listen for his own packet, *one round-trip time after sending it.* Since the sender can tell from the bounced message whether or not a collision has occurred, there is no need for explicit destination to source acknowledgements. If the packet was garbled, the sender learns of the problem simultaneously with the receiver and can take appropriate action without having to be told.

In the following, we describe an Esterel development of the specification; for purposes of brevity we describe the main aspect of the specification leaving the formal declarations of events and signals.

The satellite can be modelled as the parallel combination of two processes CLOCK and REFLECT,

```
SATELLITE :: CLOCK || REFLECT
```

The CLOCK process emits a PIP signal every TM time units (or ticks).

```
CLOCK :: loop
            await TM ticks;
            emit PIP;
         forever
```

The REFLECT process is able to receive packets on CHAN_UP and broadcast the messages on CHAN_DOWN. Collisions can happen on the packets being received on CHAN_UP. It is ready to receive packets in every slot (after a PIP is emitted by the CLOCK process). In the specification given below, we have used APP to denote the append operation which appends an element to a list and HD denotes the operator HEAD of a list. Let m be the size of the packet.

```
REFLECT :: loop
             await PIP;
             PKT:=EMPTY;
             repeat m times
              do
               await CHAN_UP;
               APP(PKT, ?CHAN_UP) ;
             end repeat;
             repeat m times
             do
               await tick;
               VAL:= HD(PACKET);
               emit CHAN_DOWN;
```

```
        end repeat;
      forever
```

The GROUND_STATION is able to send a packet of fixed size during every slot on CHAN_UP. After sending the packet it awaits for the reception of the same on CHAN_DOWN. If the message received is garbled due to collisions (which can be detected by the *strong checksum* property), then the GROUND_STATION retransmits; this is repeated till a successful transmission takes place. Note that the input signal START also aids in avoiding collisions.

```
GROUND_STATION :: RETRANSMIT:=FALSE;
            loop
            await PIP; (* await for the clock signal from the clock *)
            await START;
            (* await for the input signal at the ground_station,
               to instantiate transmission of a packet *)
               if not (RETRANSMIT) then
               CNT_ PKT:= NEW_ PKT
            else CNT_ PKT:= OLD_ PKT
            RETRANSMIT:=FALSE;
              i:=1;
            repeat m times (* m is the size of the packet *)
              do
            [emit CHAN_UP(CNT_ PKT[I]) || await tick];
            I:=I+1;
             end;
             trap T in
             do
            [await CHAN_DOWN || await tick];
            if NOT_VALID(?CHAN_DOWN) then
               begin RETRANSMIT:=TRUE;
                     exit T;
                     end;
            watching m ticks (* Receiving  continues till the complete packet
            is received or it knows that the message is garbled *)
            end
            end trap
            forever
```

7.3 *Application hybrid and timed systems*

Programming languages based on the *perfect synchrony paradigm* have proven useful for programming reactive systems. One of the main reasons for its success is that it permits the programmer to focus on the logic of reactions and makes it possible to use several automata-based verification systems for correctness proofs. Further, the correctness proofs of programs follow their implementation very closely and hence, are more robust and reliable. However, the application is limited to clocked systems. If we look at complex reactive system specifications such as process control systems and robotic applications, the need for features such as (i) asynchronous events (events that can happen arbitrarily close to each other), (ii) integration of discrete and continuous components (continuous components may cause continuous change in the values of some state variables according to some physical law), and (iii) explicit clock times, become apparent.

Further, the use of *hybrid systems* (Kesten & Pnueli 1992) is becoming very extensive. Hybrid systems are systems that combine discrete and continuous computations. Hybrid system model contains activities that modify their variables continuously over intervals of positive duration in addition to the familiar transitions that change the values of variables instantaneously, representing the discrete components. Many systems that interact with a physical environment such as a digital module controlling a process or a manufacturing plant, a digital-analog guidance of transport systems, control of a robot, flexible manufacturing systems etc., can benefit from the study of hybrid models.

In (Berry *et al* 1993), a paradigm referred to as *Communicating reactive processes* (CRP) that provides a unification of perfect synchrony and asynchrony has been presented. In (Shyamasundar 1993), an extension of CRP, referred to as *Timed* CRP has been envisaged that

- models *continuous* computations and thus, provides a convenient formalism for specifying hybrid systems, and
- models asynchronous systems operating in dense real-time domains.

Let us now consider the basic idea for specifying hybrid systems that combine discrete and continuous components possibly with the need to reference clocks explicitly. For consistency, it is necessary to have a consistent assumption about the progress of the computation as the system evolves. In the timed CRP, it has been possible to provide such a consistent assumption through the clocked semantics of CRP and the interpretation of the clocks in terms of the **exec** primitive. One of the interesting features is that hybrid systems without explicit references to clocks can be specified by a subset of timed CRP consisting of just the statements of ESTEREL and the **exec** primitive. The full version of timed CRP can specify dense asynchronous systems with explicit references to clocks (cf. (Shyamasundar 1994)).

8. Role of safety in the design of safety-critical systems

With the advances in technology, computers are being increasingly used to monitor and/or control complex time-critical physical processes or mechanical devices where a run-time error or failure could result in loss of property, injury or even death. Such systems are usually termed *safety-critical* systems.

Safety is associated with the notion of risks such as loss of property, injury, death or damage to the environment. In other words, safety requirements are concerned with making the system *mishap-free* whereas reliability is concerned with making the system failure-free (Leveson 1986, 1991). Software safety is part of system-safety (Leveson 1991). Ensuring system-safety involves:

1. Identifying hazards and assessing the risks involved.
2. Designing ways to avoid or control them.

Thus, it is essential to arrive at system-fault tree from which one has to arrive at safe-system keeping in mind that all the hazards cannot be completely eliminated.

8.1 *How much is safety worth?*

An increasing number of computerized safety-critical systems are currently being deployed in such areas as transportation and nuclear power production, or will be largely deployed tomorrow in medical computing, automotive electronics, etc. Critical applications such as nuclear control systems, flight control systems, life-support systems have extreme safety requirements. For instance, FAA and NASA have established a requirement of less than 10^{-10} safety-critical failures per hour throughout a 10-hour flight, a level roughly equivalent to one failure per million years of operation.

For hardware component faults, it is possible to achieve these low failure rates by use of highly reliable microelectronics, together with replication and adaptive majority voting. The primary factors contributing to unreliable operation are design faults, possibly in the hardware but more probably in the software. Software faults present the greater risk of system failure, because only relatively simple functions are mechanized in hardware while the most complex parts of a systems are implemented in hardware. The statistical evidence that software is the current bottleneck in achieving dependability of Information and Communication Technologies, together with the recognition that probabilistic assessment of software reliability to levels commensurate with safety requirements (e.g. 10^9/h or 10^5 per demand) is currently out of reach, has led to highly labor intensive approaches for the development and validation of operational safety-critical software. Be they undertaken via traditional software engineering approaches or via mathematically formal approaches, orders of magnitudes of effort dedicated to the development and validation of such software are in the range of 10 man-years per 1000 lines of code, for software ranging from a few thousands to a few tens of thousands lines of code.

As the pervasiveness of software induces a clear tendency to complexifying the functions it is expected to fulfill, producing dependable software for critical applications of sustainable cost requires the identification and formulation of abstractions which are at the same time rigorous and representative of both the informatics constructs and the environment where the corresponding computing systems are intended to operate. At the same time, recent unfortunate examples have shown that computerized systems which were not initially felt to be safety-critical, and thus not built according to high costly standards, have endangered human lives upon failures, be they a relatively modestly complex apparatus such as the Therac 25 radiation therapy system, or distributed systems such as the communication system of the London Ambulance Service. Moreover, nation-wide failures of large computing and communication systems which cannot be built at the above cost, such as the outage of the inter-city phone system or the Internet collapse in the USA, can have indirect safety-related consequences, be they caused by accidental or malicious events.

In the case of embedded systems, most of the times it is quite difficult to provide a behavioural decomposition even if one ignores the need for the decompositions to be the basis for system design. In other words the separation of concerns turns out to be extremely difficult. For example, even small real-time systems such as a tactical embedded system for an aircraft might be simultaneously maintaining a radar display, calculating weapon trajectories, performing navigation functions etc. In these kinds of systems, one sees that

- the code implementing the various tasks is mixed together such that it is

difficult to determine which task(s) a given part of the code performs, and

- the timing dependencies between code sections are such that changing the timing characteristics of one section may affect whether or not many otherwise unrelated tasks meet their deadlines.

The question is how do we go about? As discussed already, a general strategy is to base the design on a formal method. Of course, at this stage, it is necessary to do the hazard analysis by various techniques such as (1) design reviews and walkthroughs, (2) fault tree analysis, (3) failure modes and effect analysis etc. Synchronous paradigm has shown a good promise and has various components such as dependable compilers, simulators, verifiers etc, it is natural that one should base the development around this paradigm. In fact, the tools available on such systems also aid the hazard analysis techniques mentioned above. One of the important factors to be kept in mind is that in the design of reactive systems one should first go about specializing. for the class of systems one is concerned with rather than going about for the generalized system design. In the next section, we discuss some of the measures that should be taken in India that would aid in the development of reliable reactive systems and reliable software in general.

9. An approach towards development of quality software

In recent years, India has been leaping into hi-tech areas areas such as communication, transport, nuclear industry, military tactical systems. Directly and indirectly, there is a large involvement in the development of software for embedded systems. Some of the failures in such projects can be attributed to software faults. Some of the hi-tech projects/systems being planned in India are facing a series of problems due to the fear in the community that a failure in the system will endanger the society or will lead to environmental catastrophes. We would make a major headway in overcoming such bottlenecks if we can

- develop safety standards and regulations in the use of embedded safety-critical software,
- place mandate for the use of formal methods and languages with sound rigorous mathematical basis, and
- agree to apply reverse-engineering techniques for safety-critical systems and evaluate them; this would gain the confidence of the society and also the design errors can be corrected and further, catastrophes can be avoided.

In fact, several countries such as UK have promulgated such standards as a must. Arriving at such standards and regulations will go a long way in the development of science, and technology in India.

One of the foremost things of concern in the Indian context in the need to convince the industry of the use of formal methods. The use of formal methods need to be made cost-effective, which is possible. Related to this point is providing training to professionals in the use of the formal methods. It is here that a major effort is required by academics and software professionals in India.

It can argued that a thrust towards the development of quality software will also have an enormous impact on the economy in the context of software export.

1. Given that manpower in India is much cheaper, it is our contention that strong economic arguments can be made for the use of formal methods for generating safety/quality software. Generation of safety/quality software using formal methods is smart-labor-intensive and not equipment or commercial software intensive. We believe that programmers and software engineers in India (especially from top technical institutions) typically have a stronger background in mathematics and analytic methods than their counterparts in the west, especially the US. They can be trained quickly and with lesser economic cost in the use of formal method and the associated tools.

2. For sophisticated machinery and equipment to be sold to other third world countries, India would be competing with Western countries. It may become essential to develop methods for reliable software. India may develop a slight edge over Western countries on this front because of cheaper cost in producing software due to cheaper labor costs. Also it is estimated that there would be a large demand based on the synchronous paradigm for the development of reactive systems and hardware systems. Hence, investment on the synchronous programming technology will not only aid in the development of systems but also will put India on an advantageous position in the software development (for consumption or export).

3. India can take a lead in developing tools and methodologies for generating ultra-reliable software. It is technically feasible to generate a next-generation environment for designing systems based on formal methods.

   What should be key characteristics and features of a next-generation integrated environment for providing support for formal methods in the life-cycle of system design and development? Many technical issues related to integrating automated reasoning tools, simplifiers, specification analyzers are discussed by Kapur (1993). We would like to emphasize that formal methods should be integrated into the existing methods in a

   (a) Localized way: that is, it should not be the case that formal methods have to be applied to the whole system. It should be possible to test them and experiment with them on parts of a system.

   (b) Reversible way: If for some reason, the use of formal methods has to be abandoned that should be possible in graceful without causing significant delays in the development of the system.

   In other words, the use of formal methods should not interfere with the existing development process, that is, the additional features required for deploying formal methods, if ignored, should not cause major disruption in the life-cycle. It is the careful integration of formal methods with existing methods that include prototyping, testing, structured walk-throughs, validation, hazard analysis, fault-tree analysis, and simulation, etc., which has led to reasonable success in the development of critical software and hardware in practice (Weber-Wulff 1993).

References

Abrial J R, Lee M K O, Nielson D S, Scharbach P N, Sorenson I H 1991 The B-method. *VDM 91. Lecture Notes in Computer Science* 552: 398-405

Berry G 1992 A hardware implementation of pure Esterel. *Sadhana* 17: 95-139

Berry G, Gonthier G 1988 The Esterel synchronous programming language: Design, semantics, implementation. *Rapport de Recherche 842, INRIA 1988, Science of Computer Programming* 19: 87-152

Berry G, Ramesh S, Shyamasundar R K 1993 Communicating Reactive Processes. *20th ACM Symposium on Principles of Programming Languages* South Carolina, pp 85-99

Bowen J, Stavridou V 1993 Safety-Critical Systems: formal methods and standards. *IEE/BCS Software Eng. J.* 8: 189-209

Brinksma E 1992 What is the method in formal methods. *Formal Description Techniques* (eds) K R Parker, G A Rose (New York: Elsevier Science)

Fairly R E 1985 *Software engineering concepts* (New York: McGraw-Hill)

Gerhart S, Craigen D, Ralston T 1994 Experience with formal methods in critical systems. *IEEE Software* : 21-39

Hoare C A R, Jones C B 1989 *Essays in computing science* (London: Prentice Hall International)

Kapur D 1993 Automated reasoning in software design. *CSA Jubilee Workshop on Computing and Intelligent Systems* (New Delhi: Tata McGraw-Hill) pp 201-216

Kapur D, Musser D R, Nie X 1992 The Tecton proof system. *Proc. of the Workshop on Formal Methods in Databases and Software Engineering, Workshop in Computing Series* (eds) Alagar, Lakshmanan, Sadri (Berlin: Springer-Verlag) pp 54-79

Kapur D, Shyamasundar R K 1994 Software for safety-critical systems: Quality and futuristic technologies. Invited Lecture at the Seminar on Science Policy, Jawarharlal Nehru University, New Delhi

Koymans R, Shyamasundar R K, de Roever W P, Gerth R, Arun Kumar S 1988 Compositional semantics for real-time distributed computing. *Inf. Comput.* 79: 210-256

Ladeau B R, Freeman C 1991 Using formal specification for product development. *Hewlett-Packard J.* 6: 62-66

Lamport L 1983 What good is temporal logic. *IFIP*

Leveson N G 1986 Software safety: Why, what and how. *ACM Comput. Surv.* 18: 125-163

Leveson N G 1991 Software safety in embedded computer systems. *Commun. ACM* 34: 34-46

Mojdehbakhsh R, Tsai W T, Kirani S, Elliott L 1994 Retrofitting software safety in an implantable medical device. *IEEE Software* : 41-50

Parnas D, van Schouwen J, Po Kwan S 1990 Evaluation of safety-critical software. *Commun. ACM* 33: 636-648

Pnueli A 1986 Specification and development of reactive systems. *IFIP* : 845-858

Pnueli A 1992 System specification and refinement in temporal logic. *Proc. FST & TCS 92, Lecture Notes in Computer Science* (Berlin: Springer-Verlag)

PRG 1992 Programming Research Group at Oxford. *Timed CSP: Theory and practice, Lecture Notes in Computer Science* (Berlin: Springer-Verlag) 600: 640-675

Place P R H, Wood W G, Tudball M 1990 Survey of formal specification techniques for reactive systems. CMU/SEI-90-TR-5, Software Engineering Institute, Carnegie Mellon University, Pittsburgh

Rushby J 1993 Formal methods and the certification of critical systems. Computer Science Lab Tech. Report SRI-CSL-93-07, SRI Intl, Menlo Park, CA

Shyamasundar R K 1993 Specification of hybrid systems in CRP. *Proc. of the 3rd Int. Conference on Algebraic Methodology and Software Technology (AMAST 93)* University of Twente, The Netherlands; full proceedings in the *Workshops in Computing Series* (eds) M Nivat, C Rattray, T Rus, G Scollo (Berlin: Springer-Verlag) pp 227-238

Shyamasundar R K 1994 Specifying dynamic real-time systems in CRP. *IFIP 94 Congress* Hamburg

Simon H 1969 *The sciences of the artificial* (Cambridge: MIT Press)

Tanenbaum A 1981 *Computer networks* (New York: McGraw-Hill)

Weber-Wulff D W 1993 Selling formal methods to industry. *Formal Methods Europe 93, Lecture Notes in Computer Science* (Berlin: Springer-Verlag) pp 671-678