Appendix 2

Reactive systems

This appendix describes what is meant by the term "reactive system".

Appendix 2 Reactive systems

The traditional view of a computer program is that of a black box, with input preceding processing, and processing preceding output, after which the program terminates. This is known as a *transformational system*, because the relation of the input to the output is sufficient to completely characterise the behaviour of the program. A program can therefore be viewed as a realisation of a mathematical function from an initial to a final state in the deterministic case, and as a mathematical relation between initial and final states in the nondeterministic case. Such programs also have start and end points in time. Computation is performed in batch mode, with no reference to the operating environment.

Reactive systems [Pnueli 86b] are those which cannot be completely characterised in terms of the relation between input and output, i.e. they are not adequately described by a *relational* or *functional view*. In other words, transformational programs are *data-driven*, in that it is the flow of data which controls the systems behaviour, which contrasts with reactive systems which are *control* or *event driven.* In other words, reactive systems are interactive by nature. Typically they will receive some initial input, but then continue to interact with their environment during the course of their execution, by both sending values (output) to and receiving new information (input) from the environment. Reactive systems also include those which are real-time, embedded[1], concurrent and distributed. Systems which are concurrent are reactive whenever individual processing component interacts with the other components in the system.

Examples of reactive systems fall into two categories: those which terminate with some final result but exhibit other characteristics of reactive systems, and those which do not terminate. Systems such as real-time and distributed systems (e.g. process control systems, avionics systems and airline reservation systems) fall into the first category, and omnipresent systems such as operating systems, telephone and communications controllers, industrial robots and chemical plants come in the second. Clearly the functionality of non-terminating systems cannot be represented by a relational view.

A nonprocedural programming model does not provide an easy framework within which to specify reactive systems, since there is no mechanism for directly describing a changing environment. It is necessary to introduce some form of mechanism for delaying with interaction with the environment, such as treating input and output as streams defined by functions.

[1] systems with a computerised component that is part of the equipment, and used in its functioning.

At a more abstract level, the rôle of a reactive system is to provide and maintain an interaction with its environment, not just by manipulating data but by reacting to many different kinds of events, signals and conditions. This means that the description and specification of the system must be in terms of its ongoing behaviour, rather than purely in terms of the functionality of the program. For such systems, the relational (i.e. functional) view must be superceded by the more complex *behavioural view*. This poses new problems for the system designer since, for example, the traditional approach to correctness proving, that of showing total correctness as the result of partial correctness combined with proven termination, is no longer valid. Termination in a reactive system is typically either undefined or some form of exception condition - the desired behaviour is no termination.

The difference between a transformational system and a reactive system can be discerned from the specification of the system: the former only requires a description of what is to be done, whilst the latter also requires (control) information on how to do it, i.e. concerning the system on which the computation is to take place. There are therefore two necessary components in the specification of a reactive system: a relational description of the flow of data and the processing performed upon it, along with a precise dynamic behaviour description[2]. The former, also known as *functional decomposition*, consists of describing what is done to the data in the system. The latter consists of describing how and when (i.e. under what conditions) the data is processed, and other events triggered.

Formalising the behavioural description has often been omitted, with the result that many behavioural specifications of complex reactive systems take the form of verbose natural language documents, complete with the inevitable inconsistencies and ambiguities. A behavioural description language should be able to capture the semantics of communication, recursion, nondeterminism, divergence, abstraction and deadlock behaviour [van Glabbeek 86, p.10].

There are several difficulties associated with the behavioural specification. Firstly, the programmer needs some sort of hierarchical construction to allow for modularity in coding - in a state-based behavioural description this means there must be sub-states and a notion of depth. This allows for separation of concerns, by permitting some explicit implementation-dependent details to be omitted from the behavioural specification. Secondly, concurrency

[2] there are two major approaches to the implementation of a solution to this question. In one, an initially simple (and therefore correct) specification, giving the desired behaviour clearly, possibly at the expense of explicitness, is gradually transformed into an implementable form. In the other approach, two languages are used: an expressive specification language that is used to specify the "what" of the program, and a prescriptive programming language that is used to specify the "how" of an implemented system. The theory underlying the first approach is that of *equivalence* between successive refinements. *Verification* (a program p *satisfies* a specification *s*) underlies the second approach

may need to be represented in the behavioural description[3]. This must be done without setting up states for every permutation of events, because then there may be an exponential blow-up in the number of states[4]. Thirdly, there are often unstated assumptions made about the nature of the environment, such as the expected range and variation of monitored components of the environment.

It should be noted that a behavioural specification can only yield a full system specification when combined with a functional decomposition. In other words, a characterisation of the behaviour of the system must take into account not only the system itself, but also the environment within which it operates.

What particular approach is taken may depend on how the problem is specified and what problems the application domain require to be particularly addressed. Several "formal methods" have been developed for specifying systems, so-called because they are more susceptible to formal verification techniques than systems specified using everyday language. They include Statecharts [Harel 87a] [Harel 87b, pp.297-300], Esterel [Berry & Cosserat 85] [Boussinot 86], VDM [Bjorner & Jones 78], LOTOS [Brinksma 87] [Turner 87]) and Z [Hayes 85]. Distinguishing characteristics include whether they use a graphical or textual form of representation, whether they can be directly implemented or refined in a stepwise manner, to what extent they can express concurrency, and so on.

One means of partitioning specifications is by identifying whether they are sequential or concurrent. The development of sequential programs is now reasonably well understood, but parallel environments still require further techniques to be developed. There is certainly a close correlation between the sequential-concurrent partition and the functional-reactive partition, but the correlation is not perfect, since some sequential programs are reactive. Consequently the partitioning of systems into transformational or reactive is generally more helpful, since this is defined by the rôle of the system, rather than the particular implementation chosen for the system.

One might choose not to treat the classification of systems as transformational or reactive as a partitioning of systems, but rather as taking different views of the same system. The relational appellation corresponds to taking a semantic view of the operation of the system, i.e. concentrating on what the function of the system is. This is a useful way of considering systems if programming is viewed as a goal-oriented activity, but it is not always

[3] here we are talking about an idealised form of concurrency, where events are considered as atomic actions, so that there is no possibility of two events happening simultaneously. This approach to concurrency involves sequentialising the execution into an interleaving of the events. Possibly this approach is only appropriate for the requirements specification stage, and not for the system specification being constructed here.

[4] and in a non-terminating reactive system, there might be an infinite number of states.

possible, particularly in the type of system which was outlined earlier. In such cases, an operational view of the system is more appropriate, which concentrates on the internal workings and relations between events occurring in the system, without necessarily assigning the semantic significance (i.e. saying why the event occurs, what it means) which is a feature of the functional view. If one chooses to view a program as a formalisation of the desired behaviour of the system, then clearly the reactive view of a system is more implementation-oriented than the relational view, since there are many programs which can implement the abstract description of the functionality of the system, only a subset of which will also fulfill the behavioural constraints.

In summary, constructing a functional view of a computational process corresponds to interpreting the computation as a transformational (relational) system, whereas describing the abstract behaviour of the system corresponds to interpreting the computation as a reactive (interactive) system.