

CHAPTER 5

Validating Requirements

Introduction

Chapter 4 discussed a number of modelling formalisms which can be used for creating a functional requirements model. The use of a single formalism or a combination of them, for example entities/relationships, rules, state transition diagrams etc., will result in a model of what the analyst perceives to be the user requirements for a software system. With careful checking, possibly with some help from an automated CASE tool (such as those discussed in Chapter 6), the analyst can become fairly convinced that the requirements model is a *correct* one, where the term *correct* means that the model will not contain illogical or self-contradicting definitions.

There is a problem with the above approach however. *A correct requirements model is not necessarily the right requirements model.* What has been thought to be the user's problem can sometimes be something totally irrelevant, i.e. a situation can occur where time and effort is spent analysing the *wrong problem*. Such a situation is illustrated in Figure 5.1.

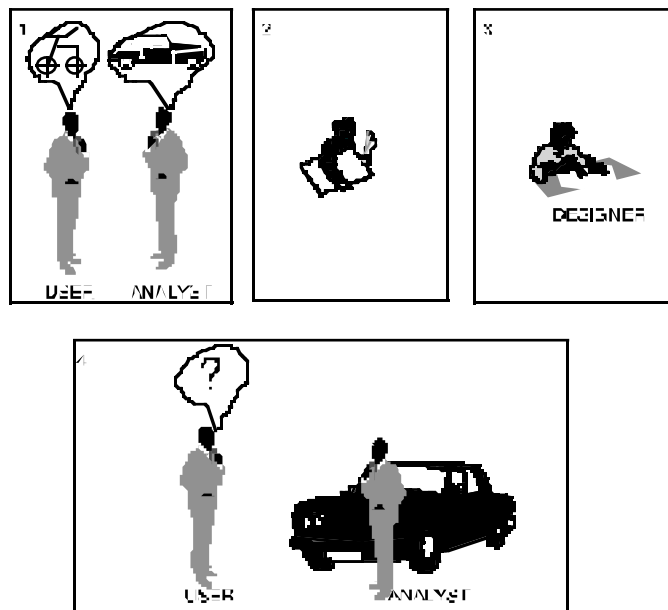


Figure 5.1: Solving the wrong problem.

Chapter 3 identified the main causes for such misconceptions to be:

- difficulty in eliciting the requirements from a user. (In Expert System terminology this is called the *knowledge acquisition bottleneck*)
- difficulty in establishing a common framework of understanding between the analyst and the user, i.e. *making them speak the same language*.

Obviously, wasting precious time and resources in solving the wrong problems is something ill-afforded. It is therefore important to make sure at a fairly early stage that the analysed problem is indeed the user's problem and that the resulting requirements model will be a faithful representation of the user's demand for a computer solution to his/her problem.

Requirements validation is the process of certifying the requirements model for correctness against the user's intention

As such requirements validation helps to *do the right thing* in contrast with the careful following of a modelling approach which helps in *doing the thing right*.

Section 5.1 provides more arguments in favour of an early validation of requirements which will avoid correcting expensive mistakes later on in the software life cycle. Section 2

gives guidelines on issues that should be considered when validating the requirements model. Section 4 introduces techniques for requirements validation, namely prototyping, simulation/animation, the use of natural language in validation, and finally expert system approaches.

Prototyping provides the users with a mock-up version of the software system as a means of acquiring their correct requirements.

Simulation and animation are techniques for 'bringing life' into the requirements model, taking it through different states and generally testing it with under different scenarios in order to see how well it copes in real-life like situations.

Natural language paraphrasing is a technique for translating the requirements model in the user's own language (i.e. natural language) in hope that this will make it easier for the user to judge the validity of the requirements.

Expert system approaches to validation introduce the concept of an *assistant*, i.e. a system containing knowledge about the problem domain which it uses in order to bring to the attention of the human analyst contradictions, omissions, unresolved issues etc. about the requirements model.

5.1 The need for requirements validation

This section stresses the importance of an early validation of the requirements model. Also, validation is distinguished from the more usually practised *verification*. It must be emphasised that in contrast to verification, validation cannot be performed by the analyst and software tools alone; the user's active participation is always necessary.

The IEEE SET Glossary [IEEE, 1983] defines validation as the process of evaluating software at the end of the software development process to ensure compliance with software requirements. The same glossary defines verification as "... The process of determining whether or not the products of a given phase of the software development cycle fulfil the requirements established during the previous phase...".

Some attention must be paid to the above definition. First, validation is defined as taking place only after the software has been developed. Second, verification is always carried

with respect to some requirements. The questions raised as a result of these definitions are as follows:

- is it sufficient to validate software only after it has been developed?
- against what can requirements be verified, i.e. what are the 'requirements for the requirements model'?

Statistics have proved the answer to question 1 to be 'no'. A price has to be paid if software validation is left until late. The longer software validation is postponed for, the more expensive (in terms of testing, debugging and redevelopment) it becomes to correct mistakes. In a survey carried by Boehm in the 70s [Boehm, 1980] it was revealed that up to 50% of the bugs in a software systems were results of errors in the requirements. Moreover, the cost of fixing a bug which was a result of an error in requirements definition was up to 5 times the cost of errors in designing or coding. Actually, such findings are justifiable, considering that most software systems are developed today using a 'waterfall' lifecycle approach (see Chapter 2), and the reasons are as follows. Suppose that an error occurs during the requirements specification phase. This will result in some part of the requirements model to be erroneous and the rest part to be correct. As the development of software progresses, some part of the design will be based in part on the erroneous requirements and in part on the correct requirements. Such design *will not* be partially wrong; it will be completely wrong. Similarly, code which will be derived partially in erroneous and partially in correct design will be wrong. This 'snowball' effect can have catastrophic consequences. A single error in the requirements specification may cause the need to re-design, re-code and re-test a large part of the software system Therefore:

Validating at the requirements specification phase can help to avoid fixing expensive bugs after the software has been developed.

In a similar manner, the answer to the second question asked at the beginning of this discussion, ('is there some other model against which the requirements model can be verified?'), is, again, 'no'. There is no such a thing as the requirements' requirements. Whilst code can be verified against the design and design itself can be proved correct with respect to the requirements, there is no way we can formally verify the later. If such a thing as a 'requirements' requirements' model existed it would had been inside the users' heads. In reality however, users do not know what they want, neither (sometimes) what is best for them, nor (unless they are computer experts themselves!) what is feasible. This leads to

another conclusion:

It cannot be proved formally that a requirements model is correct.

What can, however be achieved instead is a well-justified *belief* that the solution specified in the requirements model is the right one for the user. This can only be achieved by the set of techniques known as requirements validation which can be now defined as

The processes which establish and justify our (and the user's) belief that the requirements model specify a software solution which is right for the user's needs.

5.2 Guidelines on what to validate in a requirements model

Requirements validation should be thought as more of a "debugging" activity rather than a "proving correct" one. The aim of requirements validation, is not to prove the requirements correct; but instead to identify and correct all the errors (inconsistencies, omissions, incorrect information), now rather than later on when the software will be designed and coded. In Chapter 3 validation was defined to be a process which proceeds in parallel with other processes of Requirements Engineering i.e. elicitation and specification. Indeed, validation is an 'ever present' activity which takes place every time a piece of requirement is acquired, analysed or integrated with the rest of the requirements model. Validation of requirements is also mainly an *unstructured* process. This means that it is not possible to apply an algorithmic procedure which will obtain a validated requirements model. However, there is a set of tasks that are performed during validation which apply equally well to all methodologies and formalisms used to construct the requirements model. The purpose of these tasks is to identify the existence of a number of desirable properties in the requirements model, namely:

- internal consistency
- non-ambiguity
- external consistency

- minimality
- completeness
- redundancy

The above properties of requirements models are discussed in the sequel.

Internal Consistency

A requirements model is *internally consistent* if no contradicting conclusions can be derived from it. Consider as an example a requirements model for a payroll application. If some requirement in the model states that "employees who earn less than 10000 should not be taxed" and at the same time a different requirement states that "for all employees tax should be deducted from their salaries" then the model is inconsistent. It must be noticed here that it is not necessary for both contradicting requirements to be explicit in the model; one (or both of them) may be implied from other requirements. This of course makes the validation task difficult since it is not always possible to check all the implications of the requirements model.

Faced with the problems of checking consistency, some researchers have suggested that a requirements model should be specified using *mathematical logic* [Dubois et al, 1987]. In logic we can view requirements as *logic sentences* which can be either *true* or *false*. Complex requirements statements can be constructed from more simple ones using logic connectors such as *and*, *or*, *not* and *implies*. An example of a complex requirement formed from two simple ones is as follows:

"a book is available for borrowing"

"a book is out of the library"

"a book is available for borrowing" *or* "a book is out of the library".

Being able to manipulate the requirements in a logical way when possible, has a number of advantages. A *theorem prover* is a program which can infer new logic sentences by combining existing ones, using rules of *inference*. A theorem prover applied to a logic requirements model can help the analyst discover certain types of contradictions.

Non-ambiguity

Non-ambiguity is another important characteristic for which a requirements model should

be checked. Non-ambiguity means that a requirement cannot be interpreted in more than one way. Unfortunately, ambiguities can be easily introduced in a requirements model, especially when natural language is used. The following example will show the potential disastrous effect that ambiguous requirements can have. Imagine a model of requirements for a software controlled furnace. One particular requirement states

when the furnace temperature reaches 200 °C or the environment temperature falls below 5 °C and the water valve is turned on then the oil valve must be turned on.

The above requirements statement is ambiguous because it can be interpreted in two different ways, i.e. as

when the furnace temperature reaches 200 °C and the water valve is turned on, the oil valve should be turned on. When the environment temperature falls below 5 °C and the water valve is turned on then the oil valve should be turned on
or
when the furnace temperature reaches 200 °C the oil valve must be turned on. When the environment temperature falls below 5 °C and the water valve is turned on, the oil valve must be turned on.

Whilst there seems to be little differences between the two interpretations, in fact the second one is wrong and if implemented in a software system could have devastating consequences. According to the second interpretation, the oil should start flowing in the furnace when its temperature reaches 200°C. This however, ignores the requirement that the furnace's temperature must be controlled by a cooling mechanism. According to the second interpretation, this cooling system would never come into operation, allowing a possible overheating and explosion of the furnace.

The above is only one possible example of the risks brought by ambiguous requirements. One of the most important jobs of the analyst, is therefore to eliminate ambiguities from the requirements model using his common sense, rules of logic, by clarifying ambiguous specifications in conjunction with the users.

External Consistency

External Consistency is the agreement between what is stated in the requirements model

and what is true in the problem domain. In a study by the Naval Research Laboratory, documented in [Basili and Weiss, 1981] it was revealed that 77% of all the requirement errors for the A-7E aircraft's flight program were non-clerical. Forty nine percent (49%) of these errors were incorrect facts. Every effort must be therefore put on ensuring that what is said in the requirements model is in accordance with the problem domain. Most of the facts appearing in the requirements model will come from interviewing the user's or studying the literature about the problem domain etc. (See Chapter 3). If the facts about the domain are volatile (i.e. change frequently-something that usually happens when for example the software is going to be embedded in a new piece of hardware, yet to be developed) then every attempt should be made to keep the facts in the requirements model up to date.

Minimality

Minimality is an important feature which should be present in our requirements document. The opposite of minimality is called *over specification* which is simply the tendency to include more in the requirements model than it is necessary. Usually, overspecification is an attempt to prescribe a design solution at the same time as we specify the requirement. This is, however, wrong because it constrains the possible solutions which the designers can choose from. Example of overspecifying the problem of a heating control system is given below.

When the furnace temperature exceeds 200 °C the master controller module should send a request of the type to the valve controller. The valve controller should use a first-in-first out queue to store the requests it receives from the master controller.

The above statement belongs more to a design rather than to a requirements document as it introduces concepts (modules, queues etc.) and solutions which are not essential to specify the heating control problem. In general, anything that should limit the designer's choices, unnecessarily, should not be included in the requirements module.

Completeness

A requirements model is complete when it does not omit essential information about the problem domain which could result into a system not meeting the user's needs. Completeness is a difficult thing to check in a requirements model, since there is no formal

procedure to do so. A requirements model contains goals to be achieved in a problem domain, as well as rules, facts and constraints that apply to the domain and to the software system that will be operating in the domain. Failure to capture any of these ingredients of a requirements model will have an impact on the correctness of the model.

- The omission of an objective, for example, will result in a model that will not represent all the user needs.
- Failure to capture a rule or fact of the modelled domain, will prohibit the software system from being a correct model of that domain.
- The absence of a constraint from the requirements model can lead to incorrect behaviour of the software system.

Some types of completeness checking can be performed fairly easily, and even be automated using a tool (See Chapter 6). These are usually checks for definition of all the names that appear in the requirements model. If, for example process *xyz* is mentioned in the requirements model, then the tool should be able to check whether the process is described somewhere in the model or not..

The most difficult kinds of completeness checking, i.e. checking for missing goals, facts or constraints can be assisted in a number of ways such as:

- by looking at systems with similar requirements to the one being analysed , we can identify "forgotten" requirements (goals).
- by assuming a hierarchical organisation of the requirements, we can identify requirements which do not have any corresponding "higher level" ones. If the justification for including some particular requirement is missing, then this missing justification could be some other, forgotten, requirement

A good, general way to check for completeness in the requirements model is to use *prototyping* (described in Section 5.4.1). By prototyping the requirements model, obvious omissions (as well of course as inconsistencies, ambiguities etc. can be found by the analyst and the participating users).

Redundancy

A requirement is redundant if it can also be obtained from some other part of the requirements model, or if it is simply some property or behaviour not wanted in the software system. Multiple definitions of requirements are not desirable features of a requirements model, since ideally, a requirement should be identifiable in one and only one place in the requirements model. Care, however, is required when a requirement is *implied* from other stated requirements. If a requirement is implied (in a logical sense) from other requirements and, at the same time, is stated explicitly, then one of the following conditions apply:

- the explicit statement of the requirement can be removed, as long as there is no fear that the implied requirement will be ignored
- the explicit requirement can remain in the document together with a reference to the requirements by which it is implied. If any of those requirements is subsequently removed, the analyst should re-examine the status of the implied requirement.

Deciding on whether a requirement is redundant or not is an activity which requires an understanding of the users' expectations from the system (sometimes called a "wish list"), the feasibility of the proposed requirements and also assigning priorities to requirements. A way to do so is assigning to each requirement a value from a range, starting from "absolutely essential" through "a nice thing to have in your system" down to "redundant".

5.3 Resources needed for requirements validation

It is fair to describe validation as the *quality assurance* process which can be applied to the requirements model. Quality is a sought after property of every computer artefact e.g. design, code, documentation etc. The need for quality brings forward two issues of practical importance, i.e. the time and resources spent on requirements validation and the quality criteria which can be applied to a requirements model.

In an ideal world, the validation process should take as long as it is required. The requirements model should be checked thoroughly by the analyst (in conjunction with the

users and software tools when necessary) for all the points we mentioned before (omissions, inconsistencies, violation of constraints etc.). The model's behaviour should be exhibited using techniques such as prototyping, animation etc.) and the model should be modified until it corresponds to the user's expectations. Additions, deletions or modifications of requirements should be checked for their impact on the rest of the requirements model.

In the real world, however, the situation is different. Software projects have to meet budgets and deadlines. Requirements validation takes up only a small percentage of the overall project lifecycle and it cannot possibly last forever. Moreover there is usually a tendency amongst the developers to get over with requirements specification in order to move to more 'challenging' issues such as design. Finally, the users do not always have all the time and willingness in the world to collaborate with the analysts in endless requirements validations (inspections, walkthroughs) sessions.

The analyst is thus coming under time and resource constraints to perform a daunting and critical job such as requirements validation. As assistants to the validation task, the analyst can employ a number of tools and practices, i.e:

- logical organisation of the requirements model which show the interdependencies amongst the requirements
- automated tools which help avoiding clerical errors and speed up some aspects of validation
- communication with the user in any opportunity using techniques such as simulation and prototyping
- bringing in experience of analysis of similar systems; reuse of previous requirements models.

Naturally, the duration and effort spent on requirements validation will vary with the modelled application. Also tools and techniques for validation may appear to be more suitable for some kinds of applications than for others. Nevertheless some sort of formal validation procedure is beneficial for all types of applications, since "getting the requirements right" is the single most important step towards a successful software project.

5.4 Techniques for validating requirements

5.4.1 Prototyping

Prototyping (Chapter 2) is the term used for describing the process of constructing and evaluating working models of a system, in order to learn about certain aspects of the required system and/or its potential solution. Prototyping is a common technique in engineering disciplines (e.g. aeronautics) where, a scaled-down model of the artefact (aeroplane, car etc.) under production is first created and used for experimentation in order to arrive at a production model.

In software engineering, prototyping (also called *rapid prototyping*) has been advocated as the paradigm of producing software quickly and cheaply as possible at some stage of the development. The actual uses of prototype vary, depending on the phase of the lifecycle in which it is used, as well as on the particular life-cycle model followed. The software prototype can be throw-away (used only for understanding and assessing solutions, performances, risks etc.) or it can be transformed into the final production system [Balzer et al, 1983]. In order for a model (of any artefact, including software) to be characterised as a prototype, the following must be attainable:

- it must be possible to obtain information about the behaviour and performance of the production system from the prototype. To provide such information, prototypes should be capable of being fully instrumented; the result of such instrumentation is that the execution of a prototype results in the generation of data from which needed information can be inferred.
- prototyping should be a *quick* process. In a well-supported environment, producing a working prototype should take significantly less time and effort than it would take to produce a full-scale artefact.

Use of Prototyping in Requirements Validation

In Chapter 2, the process of Requirements Engineering was said to have a lifecycle, consisting of three major phases, namely those of *elicitation*, *specification* and *validation*. Prototyping can assist in all three phases of Requirements Engineering. It has already been discussed (Chapter 3) how prototyping techniques can help in acquiring (eliciting) and

formalising the requirements. This Section is more concerned with the application of prototyping to validation, i.e. *to the process of certifying the requirements model for correctness against the user's intention.*

Since, by necessity, validation is a process in which users is very heavily involved, the prototype model must provide the user with meaningful information. There are two kinds of such information, namely *behavioural* and *structural*. A behavioural prototype models what the prototyped software system is supposed to do. In this respect, a behavioural prototype is a black-box model of the software system which exhibits responses to stimuli. It can be said that a behavioural prototype models the *functional requirements* (see Chapter 4) of the software. In contrast, a structural prototype models how the system being prototype will accomplish its black-box behaviour. A structural prototype is thus a 'glass-box' model which exhibits aspects of the internal structure and organisation of the system being prototyped, and in this respect it can be said that a structural prototype models the *non-functional requirements* (see Chapter 4) of software.

Techniques for prototyping

A requirements model can be created using a variety of different languages and formalisms as discussed in Chapter 4. However, in order to be used for prototyping, the modelling formalism (language) must exhibit a number of properties as follows.

- it must allow the inference of information 'hidden' in the requirements model
- it must have the notion of 'state' as well as 'state transition'.

The first property (inferencing) essentially means that not only someone must be able to look at what is stated in the requirements model, but also to examine the implications of such statements. One way to make this possible is to use some *logic* language together with an inference mechanism to create the prototype. A very suitable language for this purpose is *Prolog* and an extensive literature on prototyping using Prolog exists now [Budde, 1984]. The major advantage of Prolog is that (by virtue of being a programming language) allows the execution of the requirements specification. As a Very High Level Language (VHLL) Prolog permits incompleteness in the specifications and does not force the use of strong typing or procedural control structures. In addition, it is possible to transform the initial requirements prototype into an efficient production system, without having to leave Prolog.

Amongst the disadvantages of the Prolog (or more generally, Logic programming) approach to prototyping is that

- there is no graphical notation associated with it, and
- users do not always find logic specifications easy to understand

The first problem can be overcome by using some graphical formalism (e.g. an entity-relationship model) as the front-end and having Prolog translating E-R definitions to equivalent Prolog structures in the background.

The second problem can be overcome by paraphrasing Prolog definitions using some sort of 'IF THEN ELSE' rules and pseudo-English.

The following example (taken from a library specification problem) illustrates the use of Prolog in prototyping requirements specifications.

Part of the natural language requirements model for the library case is as follows:

In a university library there are two kinds of users, namely staff and students. Staff can perform two kinds of activities, namely check out a book for a student who wants to borrow it and check-in a book returned by a student. Once a book is checked out it is not available for borrowing until it is checked-in again. A student cannot have more than ten books borrowed at any time.

The above requirements specification translated to Prolog looks as follows.

user(X) :- staff(X)*or* student(X).

check_out(Book, Borrower) :- student(Borrower)*and not* checked_out(Book) *and not* borrowing_limit_exceeded(Borrower) *and* assert(checked_out(Book)) *and* update_borrowing_record(Borrower, borrowed).

check_in(Book, Borrower) :- student(Borrower) *and* book(Book) *and* retract(checked_out(Book)) *and* update_borrowing_record(Borrower, returned).

update_borrowing_record(Borrower, borrowed) :- borrowing_record(Borrower, Number_of_Books) *and* New_Number_of_Books is Number_of_Books + 1 *and* retract(borrowing_record(Borrower, Number_of_Books) *and* assert(borrowing_record(Borrower, New_Number_of_Books)).

update_borrowing_record(Borrower, returned) :- borrowing_record(Borrower, Number_of_Books) *and* New_Number_of_Books is Number_of_Books - 1 *and* retract(borrowing_record(Borrower, Number_of_Books) *and* assert(borrowing_record(Borrower, New_Number_of_Books)).

borrowing_limit_exceeded(Borrower):- borrowing_record(Borrower, Number_of_Books) *and* Number_of_Books = 10.

It can be noticed in the above Prolog requirements specification follows closely the English specification, making at the same time some additional statements which are usually taken for granted in the natural language specification (such as, for example, that if a person wants to check out something then that person must be a student and the thing to check out must be a book).

Despite looking like a conventional program, the above Prolog specification conforms to the quality criteria set for a requirements specification; it describes the problem domain, rather than the solution domain. As it is, the above specification says nothing about design decisions, e.g. decisions about the choice of data structures to hold the borrowing record of students. However, by virtue of its executability, the above specification provides the second necessary ingredient for prototyping, namely *states* and *state transition*.

In order to create a state in the above Prolog specification, a number of *facts* must be defined. Facts are supposed to represent a (hypothetical) state in the real library, as follows.

```
student(student1).
student(student2).
book(book1).
book(book2).
{stating that the individual students 'student1', 'student2', as well as the individual books 'book1', 'book2'
exist in the library}
checked_out(book2). {stating that book 'book2' has been checked out'}
borrowing_record(student1, 0).
borrowing_record(student2, 1). {stating the borrowing records of students}
```

The above library state can be transformed to another state by invoking one of the rules describing the activities of checking in and out in the library. If for example, rule 'check_in' is invoked as follows

```
check_in(student2, book2).
```

The new state of the library will be as follows.

```
student(student1).
student(student2).

book(book1).
book(book2).

borrowing_record(student1, 0).
borrowing_record(student2, 0).
```

In this new state, 'book2' is no longer borrowed and 'student2' borrowing record has been amended to show that the student has currently borrowed no books. In a similar way, by invoking other rules the present state can be transformed to a number of other allowable states.

The value of this approach lies on the fact that users are asked to validate something which is more close to their real experience than a static requirements document. Users are more capable of detecting anomalies (in terms of inconsistencies, omissions etc.) when they have a hands-on experience with a dynamic executable requirements model. As an example, consider the above library state in which 'student1' is trying to check-in 'book1' (i.e. a book which is not borrowed!). If the execution of such request (i.e. `check_in(student1, book1)`) is refused by Prolog then this is obviously a positive evidence for the quality of the specification, since a correct specification should not allow the creation of meaningless library states. On the other hand, the execution of the above request by Prolog would mean that there are problems with the specification, since obviously the later does not model the reality in a realistic manner. A possible remedy to this problem would be to redefine the rule describing the activity of 'checking-in'. A precondition could be added to the rule stating that in order for a book to be checked-in it must be currently in a 'checked-out' state, as follows.

```
check_in(Book, Borrower) :- student(Borrower) and book(Book)
and checked_out(Book) /*the precondition*/
and retract(checked_out(Book)) and update_borrowing_record(Borrower, returned).
```

It is up to the analyst to devise a number of suitable test cases which can lead to the identification of flaws in the requirements model.

Apart from running test cases, another prototyping technique known as *static validation* can be used with Prolog requirements specifications. With static validation, the specification is analysed in order to check a number of possible flaws such as missing definitions. In the above case study for example, it could be discovered that the 'check in' activity can never take place because there is no corresponding rule for the 'check-out' activity defined. Static validation can save the user from going through too many test cases, but as dynamic validation is not a sufficient technique on its own. Here it must be noted that static analysis is also supported by many CASE tools for a variety of specification formalisms (data flows, Entity/Relationship models etc.). Such tools will be also discussed in Chapter 6.

In summary, prototyping is a technique whose importance in requirements validation cannot be easily dismissed. However, whether prototyping is achievable at all depends on the modelling formalism and supporting environment used. .

5.4.2 Animation

Animation is a technique which can be effectively brought into use for the validation of real-time systems. Real time systems have a common set of notions such as *processes*, *process synchronisation*, *messages* and *timers*. An important concern in validating real-time systems is that the requirements for concurrent and time-constrained activities are correctly captured. With the advent of new technology such as personal high-speed workstations with high resolution display devices, the graphical representation of concepts such as processes and the dynamic display of their behaviour has become possible.

Animation is a multiple graphical view of a process in action. In an animation environment, the analyst is given the ability to depict graphically all the major objects of the requirements model (processes, timers) and interact with them (using a mouse or pointing device) in terms of messages. For example, the user can change the state of a process from active to suspended and see the effects of that on the rest of the system. The objects have usually some way of graphically showing the state they are in (e.g. by using colour) and in addition numerical information is available on the screen. By animating the processes, the analyst can identify problems of performance and also detect situations such as deadlocks, starvation etc.

In the context of information systems development, a prototype specifications animation approach described in [Laloti and Loucopoulos, 1994] aims to provide a visual environment for validating and symbolically executing conceptual specifications (in terms of entity, process and rule models) of an information system. The term *conceptual prototype* is used by this approach to emphasise the difference between executing the specifications and creating a prototype of the information system. In conceptual prototyping there is no need to make any design decisions prematurely. By animating the conceptual prototype, all the actors of the Requirements Engineering process can understand and inspect the behaviour of the system under development as early as possible in the Requirements Engineering process.

Visualisation has been applied successfully in programming environments in order to provide an indication of the behaviour of the program. In the context of conceptual specifications, visualisation involves the animation of the behaviour of a system and a visual interface reflecting the results of events upon the graphical - and where appropriate the textual - components of the specification.

The advantage of visualisation over prototyping is that design decisions will not have to be made prematurely during Requirements Engineering when things are still vague. A requirements specification is likely to change many times before proceeding to design and visualisation should help in deriving a succession of specification.

Experiences from the use of visual environments in programming tasks has encouraged researchers in Requirements Engineering to make use of similar techniques, normally referred to as animation techniques, in assisting the activity of validating conceptual specifications [Kramer and Ng 1988; Tsalgatiidou 1988].

Animation of a specification is the process of providing an indication of the *dynamic* behaviour of the system by walking through a specification fragment in order to follow some scenario. Animation can be used to determine causal relationships embedded in the specification or simply as a means of browsing through the specification to ensure adequacy and accuracy by reflection of the specified behaviour back to the user.

5.4.3 Natural Language paraphrasing

The technique of natural language paraphrasing has been devised in order to tackle the problem caused by two conflicting concerns in Requirements Engineering, namely the concern of the analyst to develop a formal requirements model, and the users' need to communicate their requirements in their own terminology. Paraphrasing is a technique which compromises the two concerns by providing a 'user-friendly' version of a formal requirements model. Since paraphrasing sometimes summarises parts of the requirements model it can come to the aid of not only the user but the analyst as well, who can see the specification from a new perspective.

As with prototyping, paraphrasing does not need to be applied to the whole requirements model, as it can instead focus on those parts of the model which need clarification and reconsideration by the user. Naturally, paraphrasing should be an automated activity so that it does not occupy more valuable time than necessary. Such an automatic paraphrasing tool has been developed by a research project [Myers and Johnson, 1988] and is targeting specifications of the language *Gist*. A typical *Gist* specification together with its English paraphrase produced by the tool is shown in Figure 5.2.

<i>Gist specification</i>	<i>English paraphrase</i>
type driver	
subtype of person	All cars are mobile-objects. All drivers are people.
type car	
subtype of mobile-object	Each car can be in the state of running.
WITH{relation running().	
invariant in-motion() =>	
self: physical-object-location isa location	
PROCEDURE start[]	Start is a procedure of car.
DEFINITION	Start a car simultaneously
atomic {	(atomically) does the following. It asserts
insert in-motion().	that the car is in-motion. It updates the
update self: physical-object-location to a location	physical-object-location of the car
}.	to any location.

Figure 5.2: An English Paraphrase of the Car Specification.

5.4.4 Expert System Approaches

This category of approaches includes a number of automated (CASE) prototype tools which assist the validation of requirements. What justifies the characterisation of such tools as 'expert systems' is the knowledge of some aspect of Requirements Engineering process that they embody. This can be either *method knowledge*, i.e. knowledge of how to apply a method for Requirements Engineering (e.g. Structured Analysis- see [Yourdon, 1989]), or *domain knowledge*, i.e. knowledge about the domain which the software system is supposed to model (see Chapter 3). Expert approaches are still at the prototype status but they are certainly expected to have an impact on the functionality of the next generation of commercial CASE tools.

In general, there are three modes at which an expert system tool can act in the requirements validation process, namely

- the *expert*. At this mode the tool is expected to carry the validation process on its own
- the *assistant*. At this mode the tool is asked to assist the analyst validate some

aspect of the requirements, i.e. by checking for some specific kind of inconsistency in the model

- the *apprentice*. At this mode the tool is asked to perform the low-level clerical activities of validation, i.e. keeping track of all the names used in the requirements model, checking for multiple definitions and so on.

The current generation CASE tools function at the level of the apprentice. However, the next generation of CASE tools is expected to show performance at the level of the *assistant* (see also Chapter 6). Below is a scenario showing a typical interaction between a human analyst and a requirements validation assistant tool.

The following is the scenario for the analysis of the library system introduced in Section 5.4.2. In the session the human analyst text is in normal font, the tool's output is in **bold** and the comments are in italic.

The Analyst has created a first-cut mode of the problem using the Entity-Relationship analysis method and is asking the Assistant tool to perform certain types of validation

<ANALYST>: Validate model LIBRARY

<ASSISTANT>: **Select one of the following types of validation:**

Syntactic validation

Consistency with Domain Knowledge

<ANALYST>: Syntactic validation

<ASSISTANT>: **I notice that the cardinality of relationship BORROWS from STUDENT to BOOK has not been defined You can do one of the following:**

Define the cardinality of the relationship

Consult KB "libraries" for typical cardinality of relationship BORROWS

At this stage the analyst has the option to supply the missing cardinality or be advised by the library domain knowledge-base on what the cardinality should be like.

<ANALYST>: Consult KB "libraries"

<ASSISTANT>: **The typical cardinality of relationship BORROWS is 0 TO MANY. Action?**

<ANALYST>: ...

Some of the typical prototype tools belonging to the category of expert assistants are *ASPIS* [Pietri et al, 1987] the *Requirements Apprentice* [Reubenstein and Waters, 1991] *KATE* [Fickas, 1987] and the *Analyst Assist* [Loucopoulos and Champion, 1988]. All these systems help not only in validating the requirements but also in tasks such as requirements acquisition, formal specification (see also Chapter 6).

Summary

This chapter tackled the important issue of validating the requirements model. Important

ideas discussed in this chapter include the following

- validation is a process whose importance cannot be easily dismissed. Errors that pass through the Requirements Engineering phase to design and coding can have catastrophic consequences on the software system
- validation of requirements can only be performed against the user's intention; thus the user's participation in the validation process is paramount
- requirements can never been proved correct in a formal manner. They can however be checked for qualities such as consistency minimality, non-redundancy etc.
- validation is a time-consuming process that can be greatly assisted by automated tools
- one of the biggest problems in requirements validation is getting the user understand the formal models created by the analyst. A number of techniques to overcome this problem are *prototyping*, *simulation/animation* and *paraphrasing*
- all the above techniques presuppose the use of suitable modelling languages and environments in which the requirements can come closer to the user's experience through execution, animation and simulation and thus be easier validated
- expert tools are expected to play a significant part in the near future in validating requirements by drawing upon method and domain knowledge.

References

Balzer, R., Cheatham, T. E., Green, C. (1983) *Software Technology in the 1990's: Using a New Paradigm*. IEEE COMPUTER, November.

Basili, V. and Weiss, D. (1981) *Evaluation of a Software Requirements Document by Analysis of Change Data*. In Fifth IEEE Int'l Conference on Software

Engineering, Washington D.C. IEEE Computer Society Press.

Boehm, B. W. (1980) *Software Engineering Economics*. Englewood Cliffs. N.J:Prentice4-Hall.

Budde, R. (1984) (ed). *Approaches to Prototyping*. Springer-Verlag, Berlin, 1984.

Dubois, E. & Hagelstein, J. (1987) *Reasoning on Formal Requirements: A Lift Control System*. Proc. 4th Int'l Workshop on Software Specification and Design, IEEE.

Fickas, S. (1987) *Automating the Analysis Process: An Example*. In Proc. 4th Int'l Workshop on Software Specification and Design, Monterey, CA. IEEE.

IEEE (1983) IEEE Standard 729. *Glossary of Software Engineering Terminology*. IEEE New York.

Kramer, J. and Ng, K. (1988) *Animation of Requirements Specification*, SPE, Vol. 18, No. 8, 1988, pp. 749-774.

Lalioti, V. & P. Loucopoulos (1994) *Visualisation of Conceptual Specifications*. Information Systems, Vol. 19, No. 3, pp. 291-309

Loucopoulos, P. & Champion, R. (1988) *A Knowledge-Based Approach to requirements Engineering Using Method and Domain Knowledge*. Journal of Knowledge-based Systems, June.

Myers, J. J. & Johnson, W. L. (1988) *Toward Specification Explanation: Issues and Lessons*. Proc. 3rd Annual Rome Air Development Center Knowledge-Based Software Assistant Conference, Utica, NY.

Pietri, F., Puncello, P. P. , Torrigiani, P., Casale, G., Innocenti, M. D., Ferrari, G., Pacini, G., Turini, F. (1987) *ASPIS: A knowledge-based environment for software development*. In ESPRIT '87: Achievements and Impact, North-Holland.

Reubenstein, H. B. & Waters, R. C. (1991) *The Requirements Apprentice: Automated Assistance for Requirements Acquisition*. IEEE Trans. on Software

Engineering, Vol. 17, No. 3.

Tsalgatidou, A. (1988) Dynamics of Information Systems Modelling and Verification, PhD thesis, UMIST, Manchester, UK, 1988.

Yourdon, E. (1989) *Modern Structured Analysis*. Prentice-Hall.