16 Verification and Validation

Verification and Validation

Introduction

This theme concerns itself with verification and validation. Informally speaking verification and validation are activities related to assuring that the system constructed or under construction behaves well. Verification and validation (often abbreviated V&V) are activities which should be carried out in parallel with the analysis, design and implementation activities. Still V&V is a neglected area in current system engineering. Often we see that the bulk of V&V activity boils down to testing the final product. This is often too late to change fundamental design decisions.

In "Verification" and "Validation" (p.16-3) we introduce the reader to the terms and make a distinction between them.

In V&V and the maturity of companies (p.16-5) we use our classification of company maturity to describe how V&V activities enter into the engineering process. More than other new techniques V&V requires skilled people and dedicated engineers. One cannot expect superb results of advanced techniques if the project itself is not mature enough to cope with the advanced techniques.

In Verification and Validation and the core descriptions (p.16-7) we highlight the fact that V&V activity often relates to the analysis of descriptions. Often we want to establish the consistency between two or more different perspectives represented by different descriptions.

In Philosophy of V&V (p.16-19) we go to the fundamentals of verification and validation. What are the requirements for success? This chapter is mostly intended as background reading which should give some perspective to the system engineering process and its chances of success.

In Walkthrough (p.16-23) we are again very practical. Walkthroughs, where humans scrutinize other engineers work in a systematic way, have shown considerable success with moderate efforts. It requires some training, but the level of maturity is very moderate.

In Strategies for Verification and Validation (p.16-30) we summarize our findings in concrete recommendations for how V&V should be performed.

"Verification" and "Validation"

Unfortunately the terms "verification" and "validation" are used in the literature with varying definitions. We shall use a definition proposed by Boehm [16]:

- *Verification*: to establish the truth of correspondence between a software product and its specification (from the Latin *veritas*, "truth").
- *Validation*: to establish the fitness or worth of a software product for its operational mission (from the Latin *valere*, "to be worth").

Informally, these definitions translate to:

- Verification: are we building the product right?
- Validation: are we building the right product?"

By this definition, validation is directly related to the purpose of the system from the users and owners point of view, while verification is more of a means to achieve quality by ensuring correspondence between the descriptions developed at the various stages in the systems engineering process.

Other scholars argue that when one wants to validate a system, the distinction between verification and validation disappears because in order to validate, the purpose and the user needs have to be described (formally) and we are back to verification. Our position is that the distinction is a conceptual one which keeps the world of symbols (the descriptions) separate from the world of things (the systems at work).

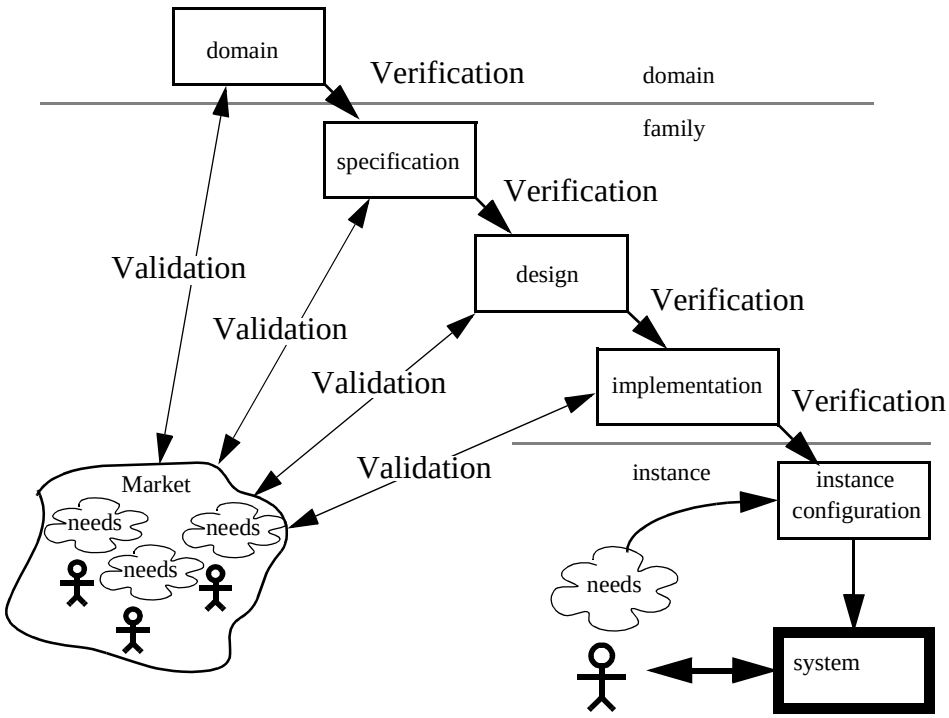
This distinction is illustrated in Figure 16-1 "Verification and Validation" (p.16-4).

The scope of validation is system quality. At the end of the day the quality of a system is determined by how well it fulfills its purpose. This will depend on the role it is supposed to play in its environment. In this respect system quality is not an absolute measure, but a relative one that depends on where and by whom, the system is used. Thus, a given system can have good quality in one context and poor quality in another. Quality can be seen as the systems ability to satisfy the needs and expectations of its environment.

The scope of verification is conformance between the various descriptions of the system.

Figure 16-1: Verification and Validation

Open figure



V&V and the maturity of companies

For verification and validation we have formulated five levels of maturity for companies and projects.

1. Test-oriented (p.16-5)
2. Inspection-oriented (p.16-5)
3. Animation-oriented (p.16-6)
4. Analysis-oriented (p.16-6)
5. Synthesis-oriented (p.16-6)

We can relate these levels to the development process in Figure 16-1 "Verification and Validation" (p.16-4) and to the definitions of “verification” and “validation”. What activities can be classified as “verification” and what activities as “validation” on the various stages?

The reason behind the classification is to improve the consciousness of the activities performed in this area. It can also be used as an aid to classifying the project or company relative to the stages.

Table 16-1: V&V Maturity Stages

Stage	Verification	Validation
1. *Test-oriented* (Testing is the most important means to establish the correctness of the system. Testing is performed on the implementation of the system)	When the tests are defined formally and the results of the tests are determined in advance	When testing is done on the ultimate product, but the correct results of the tests are not made explicit prior to the testing. Haphazard “monkey-testing”. Beta-testing at the customer.
2. *Inspection-oriented* (Inspections involve human readers who control the quality of the descriptions)	When emphasis is on the consistency of two related descriptions. When manual coding is used, and the inspection checks whether the coding principles have been followed	When the inspection is on more informal documents and when the emphasis is on assuring that the right system is specified

Table 16-1: V&V Maturity Stages

Stage	Verification	Validation
3. *Animation-oriented* (Animation and simulation are executions of the system based on descriptions on higher abstraction levels than implementation)	When the animation and simulation have explicit goals, i.e. the simulation of SDL diagrams is compared with requirement specs	Prototyping. When the animation is used to achieve a better understanding of the execution of the specified system.
4. *Analysis-oriented* (Formal analysis is used in order to prove statements about the system, or to disclose hidden aspects with a system)	When there is a formal verification that the implementation implies design, or the design implies requirements. When it is formally verified that undesired properties like deadlock are absent.	When formal transformation techniques are used in order to disclose aspects of the system which are complicated to discover by manual means
5. *Synthesis-oriented* (When the implementation can be synthesized from a description of the requirements)	When the specification and the implementation is synthesized from a number of typical examples	Finding techniques to express the right system which are more formal, but also closer to domain knowledge and natural language

Our five levels represent levels of maturity of companies (or projects). Still the reader should appreciate that the higher levels do not make lower levels obsolete. When the company chooses to focus on inspections, this does not make testing unnecessary. Furthermore simulation should not eliminate walkthroughs. Analysis orientation supplements simulation and synthesis is dependent on analysis for its ultimate success.

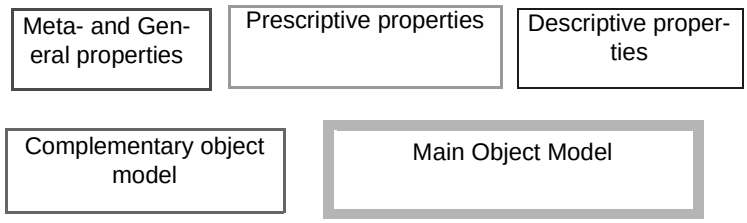
Verification and Validation and the core descriptions

Our methodology relies on a set of core descriptions. Different verification and validation situation can be illustrated effectively as relations between the different core descriptions. A major distinction is the distinction between property models and object models.

As we explain in the theme on properties, property descriptions are often fragmentary, relating only to isolated aspects of the situation. Object models are more imperative and in the end of the day they give a complete prescription for the system in SDL (and programming languages). For the purpose of explaining the approaches to V&V, we sketch the following more abstract situation concerning the relationship between the object- and property models in Figure 16-2 (p.16-7). The reader should also consult the model of descriptions to understand the similarities with our simplified picture

Figure 16-2: Property and object models

Open figure



The *main object model* is the kernel of the effort for V&V. If we are talking of design, the main object model is normally in SDL. We consider it important that it is well established which model is the main model, which tells the whole truth and nothing but the truth.

The *complementary object model* is concerned with aspects which cannot be described in the main object model. Typically an architecture design description supplements the SDL.

Meta properties are properties of the descriptions as such (and not of what the descriptions describe). A meta property could be the size of the description, *General properties* are properties which should hold for a very large variety of systems, such as absence of deadlock.

Prescriptive properties are the properties which have been used to produce the object models. In our methodology the prescriptive properties are most often functional and very often described in MSC.

The *descriptive properties* are those which characterize the object model, but which have not been used to generate it. The non-functional properties are normally descriptive.

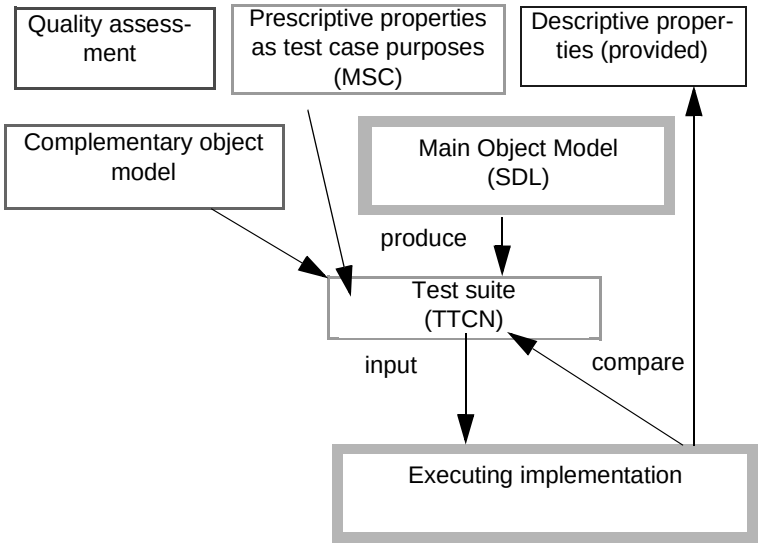
See meta and general descriptions in Inspection (p.16-9). See prescriptive property descriptions on Formal Analysis (p.16-12) and the distinction between required (prescriptive) properties and descriptive (derived) properties in Simulation (p.16-11). See implementation and test suite on Testing (p.16-8).

Testing

Testing is the art of checking the implementation. Conformance testing is to check it against its specification (requirements

Figure 16-3: Testing

Open figure



In Figure 16-3 "Testing" (p.16-8) we have shown a schematic overview of the structure of testing.

Verification through testing

Verification through testing means specifying a set of test cases and then execute the tests. The test cases will be described such that both the required stimuli and the evaluation (the verdict) of the possible outcomes are established in advance.

If the test suite is based on the requirement specification, the testing will establish the degree of consistency between the requirement spec and the SDL description, given that the transformations are correct. If there has been established consistency between the requirements specification and the SDL by a consistency argument, then the testing in this situation establishes the correctness of the transformations (and the implementation design).

The test suite (the set of test cases) can now be made partly automatically on the knowledge of design and requirements specifications. The SDL specification is seen as the main object model. It defines the possible behaviors. The prescriptive properties are described in MSC and the MSC document is seen as the set of test purposes – situations we want to check. The MSCs are simulated in the SDL and a verdict is thereby determined. A test suite (in TTCN [101], [125]) can be produced from these document which when executed will establish whether the execution of the implementation corresponds to the requirements and the design description [69],[142].

Often the test suite is not produced automatically from the earlier descriptions. Rather the test suite is based on the statement of purpose and other very high level documents related to the analysis activity. From these documents a special team of testers will produce the test suite. The advantage with having a special test team is that they have no prejudices which may prevent them from finding pitfalls of the system. The disadvantage is that they do not have the detailed knowledge of the design of the system and thus cannot know where possible pitfalls are hidden.

Validation through testing

When the desired verdict is known in advance, we have a well specified test. Very often the verdict is not specified in advance in testing situations. The idea is that the tester will know a fault when he/she sees it. The tester has experience with similar systems and he/she will perform actions which will disclose favorable and unfavorable properties of the system.

In such a case the test suite is made on the fly and the provided properties found is compared with the experience of the tester. "*Beta-testing*" is a common phrase for testing at the customer where the main purpose is to establish whether the system is doing the right job. The customer will give feedback to ensure that the final version does an even better job. "*Monkey-testing*" is another approach where complete ignorance are put in front of the input devices and encouraged to break the system down. The idea is of course that the system should be robust enough to withstand the attacks of the ignorant "apes".

Inspection

While test orientation relies on the result of executing the implementation, inspection relies on finding errors and problems through human reading and understanding [64].

In other words inspection means assessing quality. We have covered inspection and walkthroughs in greater detail in Walkthrough (p.16-23).

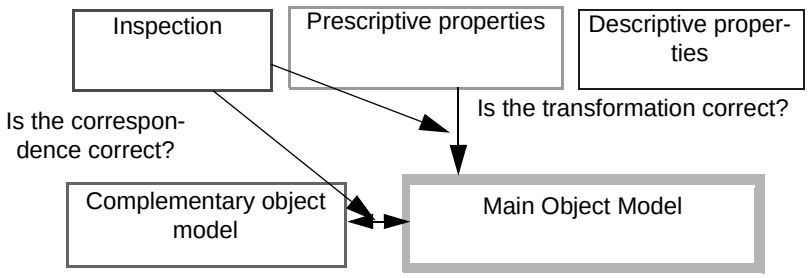
Verification through inspection

While automatic tools for consistency checking are still not able to cope with real system complexity, human walkthroughs may be the only way to establish reasonable confidence in the transformation of requirements into design, or the design into implementation.

A human inspector will use his experience to pinpoint the hazardous places in the code and walkthrough these constructs in great detail.

Figure 16-4: Verification through inspection

Open figure



Validation through inspection

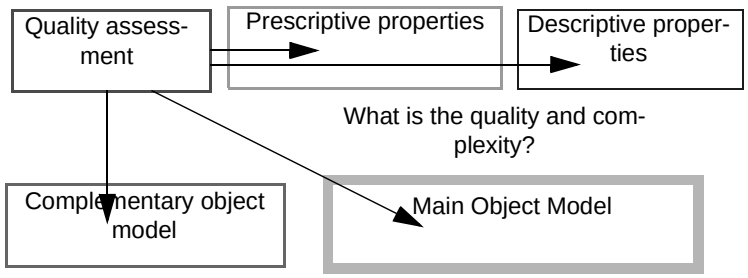
Very often inspection is used not only to establish correctness of transformations and correspondence of independent descriptions. Human inspection is important to establish the worth (validity) of the system description.

Sometimes it is possible to predict the reliability and security of a system from its complexity measures. Complexity measures can be calculated from the main object model and complementary descriptions. Maintainability and reusability are other interesting characteristics which may be established from evaluating the descriptions.

We depict this situation in Figure 16-5 "Quality assessment" (p.16-10)

Figure 16-5: Quality assessment

Open figure



Sometimes the quality assessments must be determined from testing or from empirical data from (beta) customers.

Quality assessment uses a number of different techniques on very different descriptions. Automatic means can be used where the descriptions are in a formal language. This holds for the MSC and SDL descriptions as well as the program code of the implementation. On more informal descriptions as is often the case with some prescriptive property models, the use of walkthroughs are commonplace.

Simulation

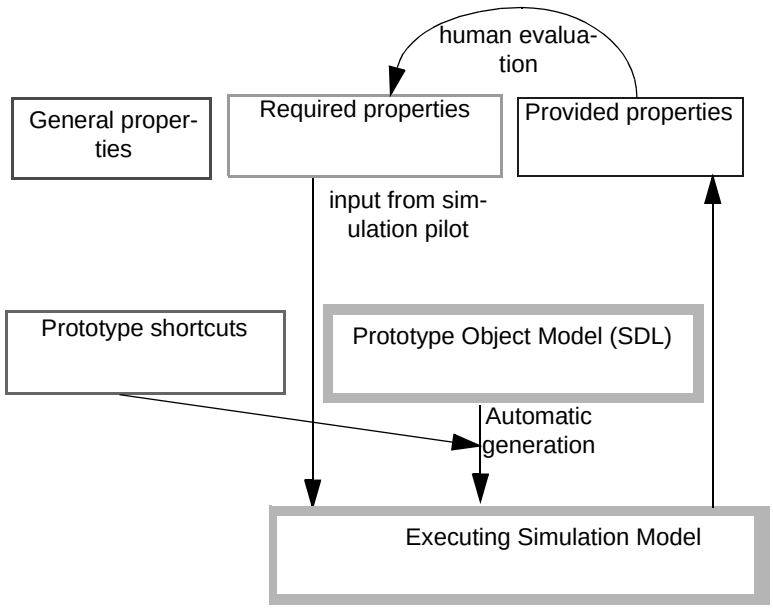
Simulation means to execute a system which is not the final target system under circumstances which are not quite like the situation will be for the implemented system. Simulation normally involves abstracting from certain details to focus on functionality [184].

How far the simulated system is from the real one may vary. Sometimes the system is running on hardware with more capacity, sometimes the simulated system is incomplete. Sometimes the simulation model is a model on a higher abstraction level than the final system. This is the case when the SDL description is the base for a simulation model.

Simulation orientation can be seen as a mixture of test orientation and inspection orientation. The testing is done on a model which is not the final implementation. Thus simulation may be performed much earlier in the development process. Simulation is similar to inspection orientation by the fact that a human engineer will perform the simulation and its success will be dependent upon the simulation pilot.

Figure 16-6: Simulation

Open figure



In Figure 16-6 "Simulation" (p.16-11) we sketch how simulation is used in system development. The simulation pilot takes as his starting point the required properties and runs the simulation model. The provided properties are then compared with the required properties. We have shown the decisions concerning how the simulation model (the prototype) differs from the real system by a supplementary object model.

Verification through simulation

As with test orientation when the desired results of the simulation have been stated in advance, we may say that the simulation is used for verification purposes. However, simulation is more often used for validation purposes.

Validation through simulation

Simulation used for validation is “an open-ended consistency check”. The simulation is open-ended because the simulator has not necessarily specified formally what results are expected. The idea of informal simulation is that the (human) observer of the simulation will recognize an error or undesired situation when he sees it.

Simulation executions are often supplied with complementary feedback such that it is possible to follow the execution more closely than when a real system executes. Thereby also non-professionals may take advantage of simulation. The market department may get a glimpse of what is coming and customers may validate their functional requirements.

Formal Analysis

While simulation is concerned with singular simulation cases, (formal) analysis is concerned with all possible cases. In some cases it is possible to traverse all possible cases in a systematic way, but often the number of different cases prevents such a strategy. Then we need to cover more than one case in one reasoning unit. The solution to this is to work with symbols rather than values. By this we mean that instead of testing for all values of a variable X (which may range over all integers), we simply use X as a symbol. Since the specification will specify choices based on values, we must be prepared to follow all branches of decisions and other kinds of choices. Still this may amount to less variants than going through all possible values of X.

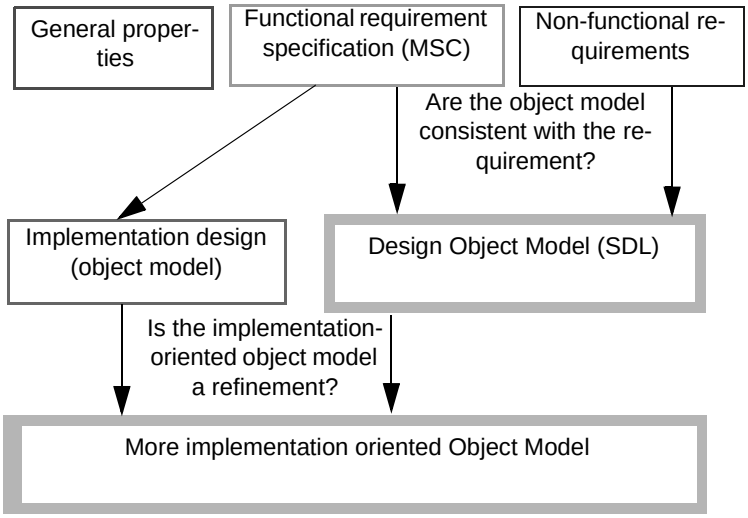
We should, however, already here urge to preach modesty. Formal analysis is very valuable when applicable, but there are situations where formal analysis is not applicable either because the situation cannot be properly formalized, or because the complexity of the situation is beyond the capacities of the formal method which theoretically could be applied. That a situation is beyond the capacity of the formal method may be due to fact that executing one case during simulation may take a matter of microseconds, symbolic execution may take seconds or if manually performed minutes or hours.

Verification based on formal analysis

We want to establish the formal correspondence between the requirement specification and the SDL description and between the SDL description, the implementation design and the implementation.

Figure 16-7: Consistency and Refinement

Open figure



Formally we would prefer that there is a refinement relation [1] between the three main descriptions such that the design is a refinement of the domain descriptions and the implementation a refinement of the design. To be a refinement means that the refined has all the properties of the starting point, but not necessarily the opposite [32].

This notion of refinement seems to be in accordance with our approach as long as properties may be seen as entities which one can collect in a “basket”. During design one collects more properties and during implementation there are even more properties to collect. However, this is not quite the way properties work. They are not entities like peas in a jar, they are rather predicates on the behavior of the system. This means that whatever is true in design should be true in the domain description. Then it is simple to see that adding properties could easily violate properties of the starting point.

Still informally we shall keep the notion that whatever behavior is legal for the refinement should be legal also for the refined (the more abstract system).

In general the consistency is not equivalence, but implications. In Figure 16-7 "Consistency and Refinement" (p.16-13) the implementations must imply the SDL and the SDL must imply the requirement specification.

When we check requirement specification against the SDL description, we talk about *model checking* meaning that we check in the SDL model whether the requirements may be fulfilled. Modern case tools perform such model checking for moderately sized systems, see [38], [177], [55] or [179].

The relation between the design object model and the implementation oriented model is more intricate as both models in principle are complete and describe an infinite set of behaviors. Exhaustive examination is out of the question. In order to convince ourselves of the refinement relation, compositionality is of great importance. Here it means that it may be possible to relate smaller pieces of the design to smaller pieces of the implementation. Compositionality then assures that the composition of refined pieces is a refinement of the composition of the starting points.

A compiler (or code generator) is a program which relates small pieces in a consistent way. There has been several projects which aim at validating a compiler. If the compiler is proved correct, there is no need to perform any other analysis on the refinement step between the two levels of object descriptions. For modern complicated code generators, the code generator has not been proven correct and an independent consistency check is reasonable. Still we should bear in mind that what we want to do is to prove the code generator correctness since most often we can rely on the assumption that it does the same every time it encounters the same situation.

Validation based on formal analysis

General properties

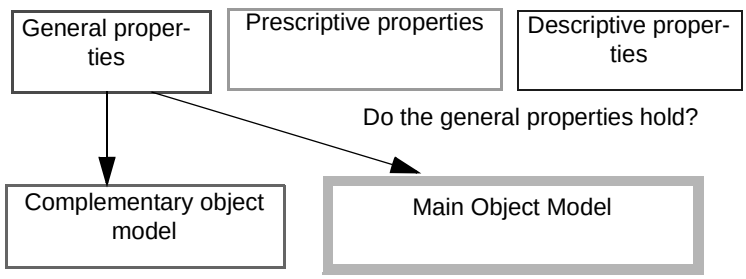
There are certain general statements which we always want our system to fulfill. Examples of such general predicates are:

- absence of deadlock;
- absence of livelock;
- all control states in a process should be reachable;
- the input port of a process should not increase beyond any bound.

There are a number of such general statements. We shall try to indicate in the succeeding sections what statements the different methods and tools can handle. We depict the situation graphically in Figure 16-8 "General properties" (p.16-14).

Figure 16-8: General properties

Open figure



It is important to note that for such general statements it suffices to analyze the object model by itself. We try to establish the absence of different kinds of internal inconsistencies.

Reachability analysis

In systems of communicating finite state machines, reachability analysis has become a popular technique for establishing general properties or the validity of temporal logic formulas [94]. Reachability analysis means simply to execute all (or a selected set of) possible behaviors and check each system state reached during such a multi-execution. Reachability analysis is seldom practical without computer support due to the large number of states which needs to be considered. The theoretically best approach is obviously an exhaustive search which means to execute all possible behaviors. This may be impossible or in practice unmanageable, and a set of behaviors must be selected. Simulation is actually when this set is manually chosen. The commercial SDL tools offer implementations of Supertrace an algorithm with random selection of the set of behaviors invented by Gerard Holzmann [94]. Still another approach to the selection of behaviors to test is when an MSC document defines the set. Then the reachability technique is used to establish the consistency between the MSCs and the SDL [55].

Interface projections

In this section we shall point out techniques which will help in the construction of the (object-) models. The techniques give the designer the ability to evaluate his/her descriptions incrementally when they are being developed.

A technique which we use to give indications of error risk areas, is based on interface projections [126]. For each interface (channel, signal route) which we consider interesting, we simplify the processes on both sides of the channel and perform reachability analysis on these simplified processes.

The simplification follows a straightforward strategy:

1. Let processes A and B communicate over channel C. Take A and B separately.
2. All transitions of A (B) which are triggered by input from other channels than C is replaced by a spontaneous transition.
3. All decisions of A (B) are replaced by **any** decisions and all tasks (assignments) are removed.
4. States which are connected by spontaneous transitions are joined into a *multistate*. Such a multistate may have transitions which are not uniquely determined by the input signal. How faithful to the original we can say this merger is, determines the risk of error in this area. (cf. Risk index (p.16-32))
5. Equal transitions can be unified (and description thus simplified), while different (non-deterministic) transitions must all be followed in a reachability analysis.

The introduction of spontaneous transitions stresses that from the interface we cannot know what happens on other interfaces, they are non-deterministic from our point of view. This obviously increases the set of possible behaviors the process may engage in.

The same applies to the introduction of **any** decisions and elimination of the tasks. This eliminates data, and we are back to the simpler finite state machine. Still the set of behavior increases.

The merger of states is the important simplification step, but we have to evaluate how “risky” the merger is, meaning how different one would expect the simplified process to execute compared with the original.

The first risk criterion is if the spontaneous transitions include output. This means that seen from the interface, output may occur spontaneously even though the state of the process will not change. Since the process is in a state where either an input or an output may be the next event, we call it a state of *conflicting initiatives*. Such states are often risk areas unless their corresponding transitions (same input signal) show strong similarities.

The next risk criterion is whether the original states of a joined state are reached from the same states. The merger of states into a joined state is based on the assumption that whenever the execution reaches such a state the possible continuations are defined by the transitions of the joined state. This is not the case if the original states are reached from different states. Then the set of possible behaviors is restricted in the original relative to the simplified process.

The analysis based on interface projections may give the following insight into the original system:

1. Firstly there are the risk indicators of the simplification process:
   - The degree of non-determinism in the joined states indicates how independent the interface used for projection is of other inputs of the process which is being simplified. A low degree of independence means that we cannot expect the simplified execution to be very faithful to the original.
   - A state of conflicting initiatives represents possible risk.
   - When original states of a joined state is reached for (many) different states, this is also a possible risk.
   - The more of the risk indicators, the more likely that the state represents an error risk area.
2. Secondly there is the execution of the simplified processes as a system around the interface on which they were projected. Errors and complexities found during the reachability analysis of the simplified system indicates errors and complexities of the original. However, we cannot normally conclude that the problem exists also in the original, but when error risk areas have been spotted, to check them out in the original requires less effort.

The interface projection technique makes it possible to run reachability analysis on parts of systems in a systematic way. Often the tools may even run such projected systems exhaustively. Then the set of behaviors can be deduced from the found error risk areas, and fed manually into simulations of the original system for a final analysis.

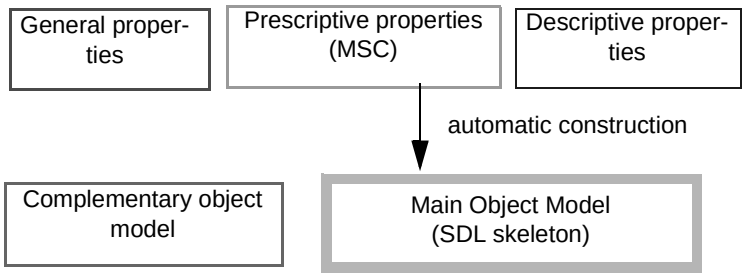
Constructively the projection techniques can be used manually to determine how independent a certain interface is in a process. We cannot expect complete independence which would mean that the process could be split into one service for each interface. On the other hand, very strong dependence between presumably independent interfaces is a strong indication of problems.

Synthesis

We highlight strategies which take advantage of automatic or semi-automatic transformations from property descriptions (requirements) to object descriptions (design model).

Figure 16-9: Synthesis

Open figure



Verification through synthesis

The idea behind synthesis is that a very high level description should lead to the implementation through steps which are very much mechanized. While early computer history saw absolute object code and assembly programming, our methodology focuses on requirements specification (in e.g. MSC) and design descriptions (in SDL). Common programming practices focuses on third generation programming languages such as Java and C++.

As long as the automatic transformers (compilers) are correctly implemented, the transformations will be correct all the time. There should be no reason to verify that the object code corresponds to the high level code. In practice, however, compilers are not correct, and that holds also for other transformers. Therefore verification based on other techniques are still important.

In cases where the source of a transformation is not as complete as the final implementation, the transformation will only yield a partial result. When MSCs are transformed to an SDL description, the best we can get is a skeleton which we have to supplement by data statements. The MSCs may not have covered all possible situations either.

Still as long as the additions on lower level does not violate the skeleton in which it is put, the refinement is still valid. What additions then might violate the enclosing skeleton? If we keep to the transformation of MSCs into SDL skeletons, violation occurs if introduced data constructs (e.g. a decision) makes the old possibilities impossible. The decision may not have any chance to choose the option specified by the original MSC. It is also possible to change the behavior such that certain parts of the process specification will never be reached and thus the original MSC may be impossible.

For a closer look at MSC to SDL transformation consult the Property theme.

It is also possible to generate SDL skeletons from UML domain object models. If the domain model contains UML models as object model and MSC models as property models and they both are used to generate SDL skeletons, there is the challenge to unify these skeletons.

Validation through synthesis

We have to start somewhere! In some way it is necessary to express what we want before any support system can support us. At this point in time it is still impossible to put a machine onto the customers head and have it extract his requirements for the new system.

What we have which approaches this very user-friendly development are generic systems and GUI-tools (Graphic User Interface).

Generic systems

For areas which we know well it is possible to make generic systems which have a large variety of different possibilities from which the customer may choose. An advanced configuration before simulation (see Simulation (p.16-11)) makes it possible to prototype a wide class of systems. The advanced “configure and simulate” session then results in a specification of the desired system.

GUI tools

An even more versatile simulation mechanism is found with GUI tools. The dialog features are constructed by GUI editors, but the actual service performance has to be done by the customer himself. The desired user interface, however, indicates clearly what the customer wants. The GUI tool provides service descriptions which resembles pre- and post-conditions since the user interface is described for these points.

Philosophy of V&V

In this section we want to make some points about which results it is reasonable to expect from verification and validation efforts. Can V&V be totally automated? If so, what does that actually mean? If not so, what can be automated? Can manual techniques be of any interest at all in complex real time systems comprising up to a million lines of (erroneous) code?

Validation is impossible

Our definition of validation emphasizes that we are concerned with what the system is worth. Either we have specified (in a formal way) what “worth” means, or we leave that to human evaluation. In the first case, we are back to verification since the validity has been defined as a property description.

As long as we keep to validation having to do with human (customer) evaluation, we cannot fully believe that formal analysis can tell us whether our system is the right one for our customer.

What we may achieve by formal means is that the customer and we understand the situation in the same way, that we are actually making the system that the customer believes we are making.

Verification is impossible

If we have been able to formalize all our requirements and all our design, it is still not possible to prove everything we would have liked to prove. In the general case it is impossible to prove the termination of a computer program. For an SDL system (a system of communicating finite state machines) there is no general algorithm which makes it possible to find out whether a given system state will ever be reached.

Theoretically these results are worrying, but in practice they are not the most frightening. Our systems are not necessarily communicating finite state machines in their generality. The algorithms involved may not be extremely difficult to assess terminate. Timers may be used to cut the Gordian knot of eternal loops.

More important in practice is the fact that even though verification may in principle be possible, the time it takes to realize the proof is beyond the scope of the system development. The proof may also need human intervention in a way which is beyond the abilities of common software engineers. Finally a manual proof has in principle the same pitfalls as a manually made program or specification.

Why try and verify?

Even though we know it may be impossible, we should still try. We should still try even when it turns out to be impossible. Experience from many years of small scale software verification has shown that making software such that it could have been possible to verify, makes it better.

Formalization – the mathematics of software

To make a description formal means that the designer needs to understand his subject down to the smallest detail in a way which is not subject to the interpretation of himself or a small insider group. A formal description may in principle be transformed and evaluated by techniques of mathematics which have been working for other engineering disciplines for hundreds of years.

Defining the formal semantics of artificial languages has helped improve the languages such that their interpretation is unambiguous. This again helps the users of the languages to make more unambiguous descriptions which again improves communication between readers of the specifications.

Proof construction – better than proof reading?

To construct a proof is to understand the specification in great detail and in full completeness. To construct a proof means to check out all the possibilities that no one had thought existed before their existence was disclosed by general mathematical techniques.

Still a proof will normally require strong human intervention and therefore proof reading the proof may be necessary.

A program is 1% innovation and 99% perspiration

A program or any formal system specification is mostly just a matter of writing it down. Few parts are critical and there are large portions which we never would expect any errors from. Therefore effective proof construction means to isolate those parts of the system which are error prone or critical in a way which is obviously faithful to the behavior of the system as a whole.

For the critical parts of a system, considerable effort can be put into validation and verification and still be cost effective.

The power of automation

By the emerge of computers with more memory and higher speed, more intricate logical constructs may be handled automatically. But in a situation where the total number of possibilities exceed the number of atoms in the universe, we cannot expect progress in computer hardware alone to make automatic verification possible [131].

On the other hand, proofs of programs normally have so many alternative variants that the support of computers is a necessity.

Even if we accept that human experience and intuition is necessary in proof construction, the support of computers to lay out the consequences of the chosen proof strategies may be of great importance. Such supporting programs are called *proof assistants*.

Recent research has shown that automation of proving is improving, but human intuition will often speed up the proof construction. Hybrid solutions where the proof constructor cooperates with a proof assistant is the most effective strategy.

The need for formal analysis

Simulation testing of a *delivered* missile firing system disclosed within three hours that there were actions sequences which the designers had not thought. These sequences might lead to launching a missile.

During the years 1985-87 a number of deaths were caused by the malfunctioning of Tharac 25 a machine used for radiation therapy.

Tharac 25 used software of Tharac 20 which had been used for a number of years without accidents caused by high radiation . Tharac 25 had just replaced some hardware with new ones which supposedly were better.

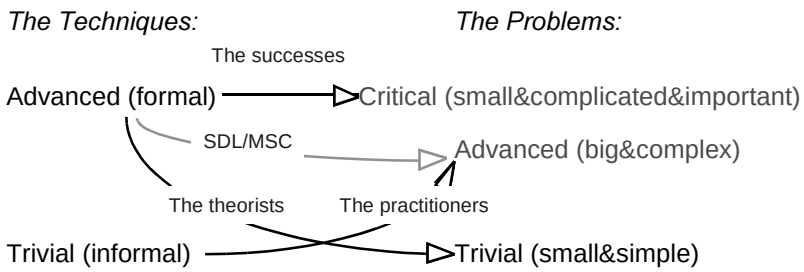
What no one thought about was that the old hardware had functioned as a “fire wall” against too high radiation. This effect was unintended! The new machine had no such effect. The designers thought that Tharac 25 had been tested thoroughly through the use of Tharac 20 [45].

We cannot promise that all such errors can be eliminated through the use of formal analysis techniques, but it is important to realize the limitations of testing and simulation.

Summary

Figure 16-10: The use of formal methods

Open figure



In Figure 16-10 (p.16-21) we show how advanced, formal methods have been used successfully to cope with small, but critical systems. Informal (and inadequate) methods are being used by practitioners to solve advanced problems. We also indicate that a lot of theoretical effort has been put into solving trivial problems by advanced means. With SDL/MSC techniques we have a chance to use advanced techniques to solve advanced problems.

Scaling A formal technique which can work well also in industrial settings must scale linearly. Most techniques now scale exponentially and this will not work for bigger than very small systems.

Transparency A good formal technique must have descriptions which can be understood by others than the theorists. The notation must be similar to notations used for specification and programming in industrial environments.

Tutoring An optimal formal technique must be supported with good tutoring.

Automating Last but not least, a good formal method must to a large extent be automated such that laborious, but not very creative tasks are performed quickly and correctly.

Walkthrough

What is "walkthrough"?

What we call "walkthrough" is a family of activities which is aimed at improving the product quality and the system development process. By scrutinizing documents produced during the development, the idea is that errors will be found and improvements suggested.

In this paper we will let the term "walkthrough" cover a wide range of activities. By "walkthrough" we mean an activity which will make a group of people responsible in solidarity for a document by their joint scrutiny of the document.

Although the definition is very broad, it may not cover all the activities that the companies think is relevant. We have emphasized the aspect of joint responsibility of the product. Others would stress other aspects of the concept. There are a number of characteristics which describe different variants of walkthrough.

- The role of a walkthrough leader
- The role of the producer of the walkthrough material
- The role of forms to fill in
- The role of "game rules"

The leader of a walkthrough may be a peer developer or a leader of the development group or the producer of the material. The producer may walk the participants through the material, or he may presume that the reviewers are well prepared from written documents. The final results of a walkthrough may be documented in specific forms which may or may not be put in a data base. The final results may also appear only as notes in the margin of the producers hardcopy.

The rules of the "game" seem to be important to many organizations. Strict rules help giving the process of walkthrough to achieve a higher internal status. Strict rules also seem to prevent discussions about the agenda or other procedural matters. Unfortunately the strict rules also boost time consumption and emphasis on details. Time consumption is the prime challenge for walkthroughs as company managers find them unproductive in isolation.

There are a number of different techniques which we have collected under the term "walkthrough" and which have a number of different terms associated with them. We have seen the following terms:

- walkthrough [65] [187]
- technical review [7]
- author-reader cycles
- inspection [64] [67]

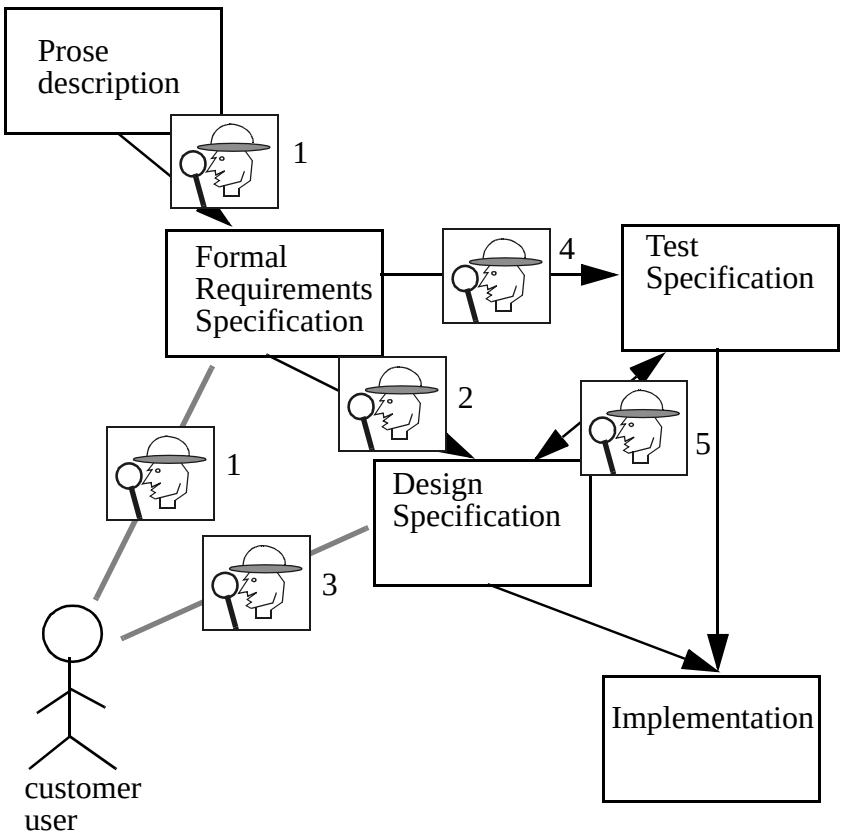
We find that more important than the terms is the fact that companies normally apply more than one variant. They find it necessary to have some formal reviews and some informal walkthroughs and author-reader cycles.

What can walkthroughs be used for?

We may give a sketch of the main documents of a system development in Figure 16-11 "Overview of V&V points" (p.16-24). The numbers in the figure refer to different potential uses of walkthroughs.

Figure 16-11: Overview of V&V points

Open figure



We may distinguish between validation and verification activities. Numbers 1 and 3 are validation activities while numbers 2, 4 and 5 can be labelled verification.

(1) Is the requirements specification good enough?

We consider here the procedures for assuring that the requirements as such are good enough and that they cover the purposes of the system. We also cover here the consideration of whether any formal requirement specification is in accordance with existing contracts and other informal requirements.

Walkthroughs play an important role to unify the project and to assure that contracts are as unambiguous as possible. This is a very typical validation activity where the participants want to ascertain that they are making the “right product” [24].

(2) Does the design cover the requirements?

This is an important phase, where the design is decided to cover the requirements.What procedures exist for this decision? Which people (what positions/roles) are involved in this decision? Is this decision a one-time decision based on some specific walkthrough or is it being taken several times or all the time?

This task is typically a verification task which could have been performed by a tool if the requirement specification and design specification were formally described. Then the tool could in principle have ascertained the consistency between the requirements and the design. Since the world still wants adequate tools we shall have to do with walkthroughs.

The task aims at proving that the project is making the “product right”.

(3) Is the design good enough?

Given that the design covers the requirements, there may still be some questions about whether it is good enough since more information has been put into the model after deciding on the requirements. Is this question considered? Is everything of this kind supposed to be within the requirements specification? Will it be put there when the design is scrutinized?

This validation task should not be mixed with the verification assignment in (2) Does the design cover the requirements? (p.16-25) as this task considers the new information which has been added during the design phase. Such new information may include architectural design or the distribution of the processes or the decision of which processors to use etc.

(4) Do the test cases cover the requirements?

The proof of the pudding is in the eating. Few people accept a product without trying it. On the other hand, one cannot eat all the puddings before buying one. The problem is to select an adequate set of cases which will give proper reason to hold that the whole product is all right.

One cannot test everything. The requirements may use universal quantification such that a proper full coverage cannot be obtained within reasonable time. How is the selection of test cases done? Are there any walkthroughs of the test cases? What is the background material for the ones that produce the test cases (or sanction them)?

Once again one should be aware of the nature of the task. The task is not to have an opinion about whether one likes the pudding, but about whether it fulfills its requirements.

(5) Do the test cases cover the design?

Is this considered to be the same question as whether the test cases cover the requirements? Are the extra information added in the design phase tested properly?

Evaluating the effects of walkthroughs

Some SISU companies evaluated how walkthroughs function in their company. How are walkthroughs helpful in the company and how are they counterproductive.

- What are the major defects of using walkthroughs extensively?
- What are the dangers?
- Can it scale properly?
- Does it work better or worse over time?
- Do some people become “walkthrough experts”?
- Should specific people be trained to do walkthroughs?
- What kinds of problems are normally caught?
- Which kinds of problems are not caught?

The companies seem to have good faith in walkthroughs. State-of-the-art development methods all use walkthroughs in some way, and our guess is that no company dare challenge their effectiveness.

We have found that walkthroughs are highly regarded as effective in early stages of a project. Requirement specifications and design specifications are documents which lend themselves well to walkthroughs.

Walkthroughs are less effective on detailed code. This fact is attributed to the demands reading detailed code put on the reviewer.

The companies make no distinction between validation and verification activities which means that they seldom see the difference between assuring the quality of one single document (or set of documents) and ascertaining the consistency within a set of documents.

How integrated is walkthrough in the development method of the companies?

As one can expect the effects of walkthroughs are better understood and more easily appreciated in larger companies and in larger projects. Smaller companies like Garex where information is readily available on an informal basis find that walkthroughs may cost more than they pay. Cap Computas being a consulting company dedicated to keeping good communications with a number of different customers find that walkthroughs are just a part of life. At NFT-Ericsson they also hold that walkthroughs have to take place, but admit that they are not always performed at the prescribed time.

In all our three companies it is well understood that one developer cannot take the full responsibility of critical software. Therefore they all have procedures which to a certain extent transfers the responsibility to the team or to the development/project leader. Still the single developer´s position is very strong in these companies (and in the SISU companies in general, we believe). Egoless programming is not a reality today, but all the companies perform informal hearings (author-reader cycles) to take some pressure off the shoulders of the single programmer.

How can walkthroughs most effectively supplement automatic verification tools?

Walkthroughs are used more effectively for validation purposes than for verification purposes. By this we mean that walkthroughs are mainly used to ascertain the general quality of a document and whether the document fulfills its purpose. In less situations walkthroughs are used to control the consistency between documents.

It is true, however, that walkthroughs bring together authors and designers of different modules of the product and thus implicitly ensures the consistency of the various parts.

At Cap Computas, walkthroughs are used in connection with tracing of requirements throughout the development. The demanded presence of the traces are checked and the general consistency of the traces is also controlled. This is an example of walkthroughs used for verification.

The other companies emphasize walkthroughs as well suited for finding problem areas and shortcomings of early phases. The role of the walkthroughs to harmonize interpretations is underlined. In general NFT-Ericsson and Garex seem to appreciate the information distribution effects of the walkthroughs. These are all aspects which can be labelled validation.

None of the companies comment upon the interplay between automatic tools and walkthroughs. NFT-Ericsson suggests an advanced computer aided walkthrough session which definitely would increase the efficiency of the documentation of the walkthrough, but which indirectly can be seen as supporting the development of the system as such.

How can walkthroughs be improved?

Experience improves the effect of walkthroughs. The companies have not noticed any negative effects of developers becoming walkthrough experts.

Some of the companies have problems performing the walkthroughs “according to the book”. This means either that the reviewers are not properly prepared, or that they cannot be summoned at the right time, or that the walkthrough is performed too late. The effects of the walkthroughs in these situations can be improved by finding the optimal sizes and times for the walkthroughs and assigning the right priorities to the activity.

The companies have not commented upon the rewarding mechanisms associated with walkthroughs. We assume there are no specific rewards for participating or conducting a walkthrough. This involves a definite risk, since activities without prospects of some kind of reward are often given less priority than activities associated with a reward. Since the quality of the walkthroughs themselves are seldom reviewed, they become side activities which just has to be performed in the quickest possible way without being obviously neglected. The obvious way to motivate good walkthroughs is to compare them with successive test results.

The companies make little distinction between validation and verification activities. Improvements may be reaped by making this distinction explicit. We are puzzled by the apparent lack of proper walkthrough procedures to assure the consistency between the

requirements specification and the test specification. The companies seem to rely on the general knowledge and competence of the people who made the requirements specification.

Walkthroughs fit better into a waterfall development model than a model which includes iterations. The general problem of describing an increment also applies to walkthroughs. When there has been a change in some part of the product, and it should be validated again by a walkthrough, how can this walkthrough be made more effective by the knowledge of its incremental nature? Some kind of delta-marking of documents and walkthroughs will be helpful.

Pros and cons of walkthroughs

Advantages of walkthroughs

- Human analysis takes into account a wider range of assumptions than automatic tools.
- Walkthroughs help communicate common design decisions such that the product as a whole becomes more unified.
- Walkthroughs are motivating for the producer. He will present a product which he thinks is readable and without errors. The producer is less timid about presenting a rotten product to a compiler or an automatic verification tool.
- Walkthroughs can be used to detect “problem areas” by the different reviewers professional instinct. It also helps solve interpretation conflicts.
- Walkthroughs can be used on informal documents as well as formal ones.

Disadvantages or hazards of walkthroughs

- Walkthroughs are time-consuming and it is not obvious that the benefits outweigh the costs.
- It is not well understood what the walkthroughs can give and what they cannot give. This may mean that some kinds of shortcomings are systematically neglected.
- Walkthroughs are affected by personal relations. One assumes the correctness of a product due to knowledge about the producer.
- The effect of walkthroughs may decrease over time as the novelty of the approach disappears. The motivation for performing a good walkthrough diminishes accordingly.
- Lack of reward/punishment mechanisms associated with walkthrough may decrease the motivation for performing walkthroughs properly.

Visions for the future

Humans should be used for what they are good at, while machines should be used for tasks that are repetitive or which requires large amounts of formal information.

Today the automatic tools are still not good enough to perform verification tasks which in principle are soluble by automatic means. The systems must be simplified in order to be manageable by the automatic tools. The simplification can be aided by tools, but must often be performed by the developers.

Below we have sketched a pathway to improved integration of walkthroughs with upcoming automatic tools.

1. Determine what automatic tools can do in your project (with your product). Determine why your system cannot be completely handled.
2. Determine whether simplification techniques can be applied to make your system more edible for the automatic tools. Simplification techniques may include projections by static elimination of communication (see Interface projections (p.16-15)) or parts of the system. Simplification may also mean semantic reduction through dynamic analysis[1]. Exceptional cases may be eliminated or highlighted.
3. Perform simplification and perform walkthroughs to certify that the simplifications cover essential aspects of the complete system.
4. Perform walkthroughs on the simplification process as the simplification may have disclosed problematic areas or complexities which were unsuspected.
5. Use automatic tools to verify the simplifications. Such verification may include discovering general shortcomings like deadlock and livelock or inconsistency between requirements and design / coding.
6. Use the simplifications to produce test cases. Supposedly this technique will result in a more adequate set of tests than a more random strategy.

The above strategy can also be used even when automatic tools are not available. The automatic tool activities will then be substituted by reviews or manually performed techniques. Strict walkthrough techniques can simulate an automatic tool.

1. cf. [169] on the rudimentary verifier.

Strategies for Verification and Validation

Our strategies will follow our classification of V&V efforts: testing, inspection, animation, analysis and synthesis. The reader should recall that focus on higher levels of these do not make lower levels obsolete such that the strategies are relevant for companies and projects on all levels.

Strategies for testing

1. Make explicit the purposes of the testing effort. Categories may be:
   - Evaluation of product (validation) for the purpose of buying or using;
   - Evaluation of new release (validation) for the purpose of moving to the new release;
   - Detect break down of system (validation) for the purpose of increasing robustness of system developed;
   - Determine correspondence between requirements and implementation (verification) by trying to test the requirements;
2. Record the tests such that:
   - the tests can be run again (automatically);
   - the earlier results can be retrieved;
   - the tests can be used also on higher levels (simulation, analysis).
3. Formalize tests through TTCN, MSC or other appropriate languages. Make explicit the desired results of the tests (verdict).

Strategies for inspection

1. Make explicit in project plans and project budgets that inspections will be used. The effect of inspections is smaller when inadequate resources have been available for it.
2. Apply a walkthrough strategy during the development such that the following aims are in focus:
   - The project team achieves a common understanding of the requirements.
   - Major strategic choices have reached a reasonable degree of consensus.
   - No part of the system is totally dependent on only one designer.
3. Associated with important milestones apply inspection techniques which emphasize:
   - The whole project team (or subteam) accepts the responsibility of the deliverables.
   - The whole project team has a reasonable knowledge of the whole deliverable.
   - For each single part of the deliverable a group of more than one person shall have accepted the detailed joint responsibility.

4. For each milestone evaluate the effects of different V&V techniques (including the inspection techniques). Make sure this evaluation has reasonable consensus. Consider changing inspection routines.

Strategies for animation

1. Make explicit how the simulation model differs from the final implementation model. Here are some possible categories:
   - The real time speed is slow because the actual hardware is much faster when it comes.
   - The actual user interface has not been devised.
   - The simulation is on high abstraction level through a generic interface (e.g. SDL signals, MSC events)
2. Apply Strategies for testing (p.16-30) modified to the simulation environment.

Strategies for formal analysis

1. Make explicit what important aspects of the system *cannot* be treated formally. Express how such aspects should be handled.
2. For every piece of requirement, classify how the requirement should be validated. Such classification categories could be:
   - Testing without prior result determination
   - Testing with verdict assessment
   - Informal inspection
   - Formal inspection (“formal” in the sense that minutes are taken and common responsibility is acknowledged)
   - Model checking (e.g. that MSCs are possible in a corresponding SDL)
   - Manual proofs (e.g. through use of some proof system supported by mathematical logic. The proofs are all done manually.)
   - Proofs made with proof assistant
3. Make explicit what results are expected through the use of formal analysis. Such results could be:
   - Determine the absence of deadlock.
   - Find consistency between MSCs and SDL descriptions.
   - Determine the validity of temporal logic formulas.
4. Evaluate each process through interface projections onto its communication channels/signal routes to give indications of error risk areas. Use the rule Risk index (p.16-32).

5. Perform full scale formal analysis on critical parts, or parts found to be error risk areas by the interface projection evaluation.

Risk index

Risk areas of a complicated process can be determined through the use of Interface projections (p.16-15). Here we define a risk index of multistates based on interface projections. The lower the number, the less risk of problems in this multistate and the more faithful the projection is compared with the original.

1. Consider the spontaneous transitions resulting from projection.
   - 1 risk point for each timer set or reset in a spontaneous transition.
   - 2 risk points for each output in spontaneous transitions (conflicting initiatives).
2. Consider ambiguous transitions of a multistate.
   - 0 risk points if all the ambiguous transitions are equal
   - 1 risk point for each difference in tasks (data assignments)
   - 1 risk point for each difference in output signal parameter values
   - 1 risk point for each difference where one transition outputs one signal and the other outputs another signal (with the same parameters)
   - 2 risk points for each difference where one transition outputs a signal while the other does not
   - 2 risk points if one transition fails to set a timer while the others do
   - 2 risk points if one transition fails to reset a timer while the others do
   - 3 risk point for a default transition (i.e. in one of the states of the multistate this transition is not defined)
   - 3 risk points for a transition which is fundamentally different from the others
3. Consider all transitions leading to the multistate (i.e. with nextstate to a state in the multistate). Sort the states in the multistate topologically by considering the spontaneous transitions the ordering criterion. This will identify a subset of the states in the multistate which can be called the *entry states*, the states that need no spontaneous transition to be reached within the multiset, the roots of the multiset.
   - 2 risk points for each incoming transition which has a nextstate different from the entry states.

A risk index higher than or equal to 5 indicates trouble, while a risk index lower than 3 indicates a fair conformance between the projection and the original.

Since the calculation of risk indices is in principle automatic, it can be performed for projections on every interface and thus a sorting of risk values should give a good indication of where to start digging for problems. The sum of all such risk values will give an indication of the overall risk of concurrency errors in the process.

Strategies for synthesis

1. Assuming formal MSCs are available, produce SDL skeletons.

2. Assuming formal UML object diagrams are available, produce SDL skeletons.
3. Align and unify the produced SDL (from MSC) and SDL (from UML).
4. Supplement the unified SDL by data statements etc.
5. Model check the final SDL relative to the requirement MSCs.

List of figures

List of definitions

Automaton

An *automaton* is an abstract machine which can be in a set of *states*. It takes a stream of *input symbols*. The consumed input symbol and the state together determines which actions the automaton takes. After the actions have been performed the automaton enters another state. The passage from one state through the consumption of an input symbol to another state is called a transition.

Finite State Machine (FSM) is an automaton where the state space and input alphabets are finite.

In our methodology, SDL uses FSMs as their theoretical base for describing interaction processes.

Model checking

Given a model which is (typically described by automata), decide whether a given logical statement (typically describe in some temporal logic) is valid.

In our methodology we use model checking to determine the consistency between an SDL model and an MSC temporal specification.

Proof

A proof is a systematic sequence of statements aimed at establishing the truth of some given sentence. A proof is often supported by mathematical notation, and based upon formal inference rules. Proofs may also be performed automatically by a computer program, or semi-automatically by the use of proof assistants.

Refinement

By refinement we mean that the refinement is a system where all behaviors are also behaviors of the refined, but not necessarily conversely.

State

A *state* is a well defined situation which a system or component of a system can *be in*. A state can be defined by a unique name or the values of a set of variables, or through a set of constraints.

A *system state* is normally used for the state of a whole system. A *process state* or *basic state* refers to a state in the finite set of defined, named states in an SDL process (or equivalent). A *complete state* of an SDL process will include values of all local variables and the value of the input port and save queue.

Verification

to establish the truth of correspondence between a software product and its specification (from the Latin *veritas*, “truth”).

Validation

to establish the fitness or worth of a software product for its operational mission (from the Latin *valere*, “to be worth”).

Walkthrough

By “walkthrough” we mean an activity which will make a group of people responsible in solidarity for a document by their joint scrutiny of the document.