# 14 Model Validation and Verification

## 14.1 Introduction

Whatever modelling paradigm or solution technique is being used, the performance measures extracted from a model will only have some bearing on the real system represented if the model is a *good* representation of the system. Of course, what constitutes a *good* model is subjective, but from a performance modelling point of view our criteria for judging the goodness of models will be based on how accurately measures extracted from the model correspond to the measures which would be obtained from the represented system.

By its nature a model is more abstract than the system it represents (even a simulation model). Viewed in one way, abstraction, and assumptions we make to achieve it, eliminate unnecessary detail and allow us to focus on the elements within the system which are important from a performance point of view; viewed in another way, this abstraction process introduces inaccuracy. Some degree of inaccuracy may be necessary, desirable even, to make the model solution tractable and/or efficient. Inevitably some assumptions must be made about the system in order to construct the model. However, having made such assumptions we must expect to put some effort into answering questions about the goodness of our model. There are two steps to judging how good a model is with respect to the system. We must ascertain whether the model implements the assumptions correctly (model verification) and whether the assumptions which have been made are reasonable with respect to the real system (model validation).

We have already seen examples of both model verification and model validation in models which we have considered earlier in the course. For example, in the Markov process model of the PC-LAN we carried out a walk-through of the simplified model with just two nodes in the network to check the characterisation of state which we were using. This was model verification. The level of abstraction which we had first chosen did not allow us to represent the behaviour of the system and make the assumptions about memoryless behaviour. The walk-through allowed us to detect that we needed to distinguish states more finely, leading to a modification of the model. The SPNP modelling package allows us to include assertions within the program representing a model. These assertions are used to encode invariants about the behaviour of the system which we know should be true at all times. For example, in the reader-writer system we were able to assert that it should never be the case that a reader and a writer had access to the database at the same time. This can be used for both model verification and model validation.

It is important to remember that validation does not imply verification, nor verification imply validation. However, in practice, validation is often blended with verification, especially when measurement data is available for the system being modelled. If a comparison of system measurements and model results suggests that the results produced by the model are close to those obtained from the system, then the implemented model is assumed to be both a verified implementation of the assumptions and a valid representation of the system.

## 14.2 Model Verification

Verification is like debugging—it is intended to ensure that the model does what it is intended to do. Models, especially simulation models, are often large computer programs. Therefore all techniques that can help develop, debug or maintain large computer programs are also useful for models. For example, many authors advocate modularity and top-down design. Since these are general software engineering techniques we will not discuss them in detail here; modifications of such techniques to make them suitable for modelling, and modelling-specific techniques are discussed below.

**Antibugging** Antibugging consists of including additional checks and outputs in a model that may be used to capture bugs if they exist. These are features of the model which do not have a role in representing the system, or even necessarily in calculating performance measures. Their only role is to check the behaviour of the model.

A common form of antibugging is to maintain counters within a simulation model which keep track of the number of entities which are generated and terminated during the evolution of the model. For example, in a communication network simulation we might keep track of the number of messages entering the network, the number successfully delivered and the number lost. Then, at any time, the number of messages which have entered should be equal to the number which have been delivered plus the number that have been lost plus the number which are currently in transit.

In Markovian models it is not possible to maintain counters in this explicit way as it would break the irreducibility assumption about the state space. However, it is still sometimes possible to include antibugging structures within a Markovian model. For example, an extra place could be included in a GSPN model into which a token was inserted for each entity created in the model and a token was deleted for each entity deleted. Care should be exercised in applying such techniques to ensure that you are not creating *state space explosion*. The SPNP assertions can also be used for antibugging since they allow a check to be made in each state of the model, as the state space is constructed.

**Structured walk-through/one-step analysis** Explaining the model to another person, or group of people, can make the modeller focus on different aspects of the model and therefore discover problems with its current implementation. Even if the listeners do not understand the details of the model, or the system, the developer may become aware of bugs simply by studying the model carefully and trying to explain how it works. Preparing documentation for a model can have a similar effect by making the modeller look at the model from a different perspective.

In the absence of a willing audience the model developer should try to carry out the same sort of step-by-step analysis of the model to convince himself or herself that it behaves correctly. For a GSPN model this would amount to playing the token game; in a queueing network, stepping through the possible customer transitions. Some modelling packages provide support for doing this.

**Simplified models** It is sometimes possible to reduce the model to its minimal possible behaviour. We have already seen examples of this when we considered the "multiprocessor" example with only one processor, to make sure that the interaction between the processor and the common memory was correct, and when we considered the PC-LAN with just two PCs. Similarly in a closed queueing network model we might consider the model with only a single customer, or in a simulation model we might only instantiate one entity of each type. Since one-step analysis can be extremely time consuming it is often applied to a simplified model.

Of course, a model that works for simple cases is not guaranteed to work for more complex cases; on the other hand, a model which does not work for simple cases will certainly not work for more complex ones.

**Deterministic models (simulation only)** For simulation models the presence of random variables can make it hard for the modeller to reason about the behaviour of a model and check that it is as expected or required. Replacing random variables which govern delays or scheduling with deterministic values may help the modeller to see whether the model is behaving correctly. Only when we are satisfied that the behavioural representation of the entities is indeed correct should we introduce random variables to represent inter-event times using continuous time distributions.

Note that this technique is only appropriate for simulation models—Markovian models can only be solved with exponential distributions.

**Tracing (simulation only)** Trace outputs can be extremely useful in isolating incorrect behaviour in a model, although in general other techniques will be used to identify the presence of a bug in the first place. Since tracing causes considerable additional processing overhead it should be used sparingly in all except the simplest models.

SimJava2 allows you to trace models in terms of specific events or entities, as well as a default trace which will can be produced automatically. Default tracing can be switched on and off using the method `Sim_system.set_auto_trace()` which takes a boolean parameter. The entity trace is produced when an entity includes `sim_trace` commands. The first parameter sets a notional level for the trace message and the level of tracing for any particular run is set in the model body using the method `Sim_system.set_trace_level(l)` where `l` is an integer used as a bit mask with respect to entity trace messages.

Event traces are produced using the method `Sim_system.track_event(s)()`. If a single integer is passed as an argument events with that tag value are traced; if an array of integers is passed (`track_events`) events of all types listed will be traced.

The method `Sim_system.set_trace_detail(b1, b2, b3)` which takes 3 boolean parameters, lets the modeller choose what form of tracing to have in a particular run. For example `Sim_system.set_trace_detail(false, true, false)` switches off the default trace, switches on entity tracing and switches off event tracing. Note that tracing is very expensive in terms of simulation time and the default is to have no tracing if none of the tracing specific methods of `Sim_system` are called. The model fragments in Figure 31 show a variant on the M/M/1 model with user defined tracing.

**Animation (simulation only)** From a verification perspective, animation is similar to tracing but provides the information about the internal behaviour of the model in a graphical form. Some modelling packages with graphical interfaces, like SimJava, provide a dynamic display of model behaviour whilst the model is executing. In some systems the display will represent high level information about the current value of the performance measures of interest shown as dials or meters which change as the values change. The SimJava animation view is rather low level, showing events moving around the system capturing the interactions between entities. This low level view is particularly suited to verification. Animation can take the form of automated one-step analysis, if the animation facilities allow the view of the model to advance one event at a time. Graphical stochastic Petri net and queueing network tools often provide a animated form of one-step analysis in which tokens or customers can be seen moving around the network.

Handling the display as well as the evolution of the model slows down the simulation considerably. As with tracing, animation is most useful for isolating an error once its presence has been established.

**Seed independence (simulation only)** The seeds used for random number generation in a simulation model should not significantly affect the final conclusion drawn from a model, although there will be variation in sample points as seeds vary. If a model produces widely varying results for different seed values it indicates that there is something wrong within the model. Seed independence can be verified by running the simulation with different seed values, something which is probably necessary in any case. In SimJava2 the root seed value to be used in a run can be set explicitly; for example calling the method `Sim_system.set_seed(23533)` sets the root seed to 23533. `Sim_system` will still give each separate distribution used in the simulation a distinct well-spaced seed.

**Continuity testing** At an abstract level all systems and models can be thought of as generating a function from input values to output values, and in most cases we expect that function to be continuous. In other words, in most cases, we do not anticipate that a slight change in an input value will result in very large changes in the corresponding output value.

Continuity testing consists of running a simulation model, or solving a Markovian model, several times for slightly different values of input parameters. For any one parameter, a slight change in input should generally produce only a slight change in the output. Any sudden changes in the output are taken to be an indication of a possible error which should be investigated unless this is known behaviour of the system.

**Degeneracy testing** Degenerate cases for a model are those values of input parameters which are at the extremes of the model's intended range of representation. Degeneracy testing consists of checking that the model works for the extreme values of system and workload (input) parameters. Although extreme cases may not represent typical cases, degeneracy testing can help the modeller to find bugs that would not otherwise have been discovered.

**Consistency testing** For most models and systems it is reasonable to assume that similarly loaded systems will exhibit similar characteristics, even if the arrangement of the workload varies. Consistency tests are used to check that a model produces similar results for input parameter values that have similar effects. For example, in a communication network, two sources with an arrival rate of 100 packets per second each should cause approximately the same level of traffic in the network as four sources with arrival rate of 50 packets per second each. If the model output shows a significant difference, either it should be possible to explain the difference from more detailed knowledge of the system, or the possibility of a modelling error should be investigated.

## 14.3 Model Validation

Validation is the task of demonstrating that the model is a reasonable representation of the actual system: that it reproduces system behaviour with enough fidelity to satisfy analysis objectives. Whereas model verification techniques are general the approach taken to model validation is likely to be much more specific to the model, and system, in question. Indeed, just as model development will be influenced by the objectives of the performance study, so will model validation be. A model is usually developed to analyse a particular problem and may therefore represent different parts of the system at different levels of abstraction. As a result, the model may have different levels of validity for different parts of the system across the full spectrum of system behaviour.

For most models there are three separate aspects which should be considered during model validation:

- assumptions
- input parameter values and distributions
- output values and conclusions.

However, in practice it may be difficult to achieve such a full validation of the model, especially if the system being modelled does not yet exist. In general, initial validation attempts will concentrate on the output of the model, and only if that validation suggests a problem will more detailed validation be undertaken.

Broadly speaking there are three approaches to model validation and any combination of them may be applied as appropriate to the different aspects of a particular model. These approaches are:

- expert intuition
- real system measurements
- theoretical results/analysis.

In addition, as suggested above, ad hoc validation techniques may be established for a particular model and system.

**Expert intuition** Essentially using expert intuition to validate a model is similar to the use of one-step analysis during model verification. Here, however, the examination of the model should ideally be led by someone other than the modeller, an "expert" with respect to the system, rather than with respect to the model. This might be the system designer, service engineers or marketing staff, depending on the stage of the system within its life-cycle.

Careful inspection of the model output, and model behaviour, will be assisted by one-step analysis, tracing and animation, in the case of simulation models, and the full steady state representation of the state space in the case of Markovian models. In either case, a model may be *fully instrumented*, meaning that every possible performance measure is extracted from the model for validation purposes regardless of the objectives of the performance study.

**Real system measurements** Comparison with a real system is the most reliable and preferred way to validate a simulation model. In practice, however, this is often infeasible either because the real system does not exist or because the measurements would be too expensive to carry out. Assumptions, input values, output values, workloads, configurations and system behaviour should all be compared with those observed in the real world. In the case of simulation models, when full measurement data is available it may be possible to use *trace-driven* simulation to observe the model under exactly the same conditions as the real system.

**Theoretical results/analysis** In the case of detailed Markovian models or simulation models it is sometimes possible to use a more abstract representation of the system to provide a crude validation of the model. In particular, if the results of an operational analysis, based on the operational laws coincide with model output it may be taken as evidence that the model behaves correctly.

Another possible use for the operational laws is to check consistency within a set of results extracted from a simulation model. If a model is behaving correctly we would expect the measures extracted during the evolution of a model to obey the operational laws provided the usual assumptions hold. Failure of the operational laws would suggest that further investigation into the detailed behaviour of the model was necessary. For example, the general residence time law can provide us with a simple validation of the model output values obtained for residence times at individual components if we know their respective visit counts, jobs behave homogeneously and we expect the model to be job flow balanced.

At slightly more detail a simulation model may also be validated by comparing its output with a simple queueing network model of the same system, and conversely, (in academic work) Markovian models are often validated by comparing their outcome with that of a more detailed simulation model.

Validation of models against the results or behaviour of other models is a technique which should be used with care as both may be invalid in the sense that they both may not represent the behaviour of the real system accurately.

Another analytic approach is to determine invariants which must hold in every state of the system. For example, these invariants might capture a mutual exclusion condition or a conservation of work condition. Showing that the model always satisfies such an invariant is one way of increasing confidence in the model, and providing support for its validity. The assertions in SPNP do this, and we could imagine including similar checks within an entity in a SimJava model. The disadvantage of such an approach is that it can be computationally expensive to carry out the necessary checks regularly within a model.

```
class Source extends Sim_entity {
  ...........
  public Source(String name) {
   ..........
  }

  public void body() {
    int i = 0;
    while (Sim_system.running()) {
      sim_trace(1, "about to generate arrival " + (i++));
      sim_schedule(enqueue, 0.0, 0);
      sim_pause(src_hold.sample());
}  }  }

class Server extends Sim_entity {
  ...........
  public Server(String name) {
   ..........
  }

  public void body() {
    Sim_event next = new Sim_event();
    int i = 0;
    while (Sim_system.running()) {
      if (sim_waiting() > 0) {
        sim_select(Sim_system.SIM_ANY, next);
      } else {
        sim_wait(next);
      }
      sim_process(svr_hold.sample());
      sim_trace(2, "completed service of arrival " + (i++));
      sim_completed(next);
}  }  }

class Queue_Trace {
  public static void main(String args[]) {
    Sim_system.initialise();
    ...........
    Sim_system.set_auto_trace(true);
    Sim_system.set_trace_level(7);
    Sim_system.track_event(0);
    Sim_system.set_trace_detail(false, true, true);
    Sim_system.run();
  }
}
```

Figure 31: SimJava model demonstrating user-defined tracing (`Queue_Trace.java`)