The Role of Proof in a Formal Specification of the Speedway Rulebook

Helen Lowe

Department of Computer Studies, Glasgow Caledonian University
Cowcaddens Road, Glasgow G4 0BA

John Lowe

Department of Building & Surveying, Glasgow Caledonian University
Cowcaddens Road, Glasgow G4 0BA

Abstract

Whilst some undergraduate introductions to formal methods play down the role of proof, others have tended to emphasize it as the true payback of using formal methods in the first place. This paper describes how a sports application can be used to illustrate many of these paybacks in a readily understandable way. It illustrates the difficulty of arriving at a formal specification of a complex specification, which is often a collaborative effort between at least two parties, and how this affects the chosen development method. The maintenance of specifications is also considered, in the light of frequent and complex rule changes, in some cases, as a result of "case law" during mid-season.

1 Proof in Formal Methods

Many undergraduates are introduced to formal specification, yet learn little or nothing of proof. The emphasis is on the fact that the formal notation provides a precise and unambiguous specification, unlike a natural language specification, or the original requirements document. Formalizing the specification can help the developer firm out their ideas, resolve ambiguities, and, if suitably translated, provide a basis for discussion and negotiation with the client. These are laudable benefits and undergraduates are expected to practice developing formal specifications in Z, or VDM, or some other notation. Proof, on the other hand, is rarely covered in such depth, if at all. This may be partly because it is difficult, especially as tool support is limited [6].

In teaching VDM to undergraduates on vocational courses, however, the first author has actually emphasized, rather than played down, the role of proof. The undergraduates in question had all worked on industrial placement for a year before taking my modules, many of them in the banking and insurance sector, or for local authorities. If they had any opinion on formal methods at all, it was that they were never used – they were not "for real". Teachers trying to overcome this view find much resistance, as some students find it a very convenient one to hold for a subject which is inherently difficult for many of them, especially if their mathematical knowledge is sketchy. The strategy employed was as follows:

1. Revise the basics needed alongside learning VDM notation.
2. Do plenty of examples from familiar domains.
3. Let them make mistakes. It is particularly easy to forget vital preconditions, for example.
4. Use proof to help detect these mistakes.

The third point is the launching point for motivating formal methods. One thing the students *do* understand from their experience, both academic and industrial, is that people make mistakes. They make mistakes in requirements documents, in informal specifications, in programming, in testing, and in debugging. Mistakes can be embarrassing if undetected and should be uncovered at all costs. Of course, now that they are learning VDM, they make mistakes in that too – so far, just another notation, like a program language, only perhaps without the benefits of being executable and therefore testable. If they can be persuaded that proof will help them discover the errors at an early stage, all that is left for the teacher to do is to convince them that there is then a path from specification to program.

Whereas students can follow textbook examples, such as those given in [3] or [8], and the preconditions seem "obvious", as soon as they start doing examples, they do make mistakes. This is especially true in a tutorial situation,

where they are asked to do examples on the board. A common feature of the resulting attempted proof is that part of the right hand side is completely underivable, when students will say things like "Oh, but of course, we have to make sure that the number of seats is less than capacity before we accept the airline booking" – and realize that there is a missing precondition.

2 A pedagogic example

Students will know that one source of errors is the user, and that validation routines form an important role. An entertaining example, due to Yves Ledru, can be found in the VDM examples repository. This example was used at the Diplôme d'Etudes Supérieures Spécialisées Ingénierie Informatique at the Université Joseph Fourier, and illustrates many issues, including how VDM may be used for a sporting example, user error and detection. It also contrasts two approaches to writing VDM specifications. The example models the part of the referee's rulebook in force during the 1994 World Cup, which stipulates (amongst other things) that no more than one substitute goalkeeper, and two substitute outfield players, may be made during the course of a match. This rule was violated during the Italy-Norway game, when the Italian goalkeeper was sent off. An outfield player was immediately substituted for the substitute goalkeeper, but later on two further outfield substitutions were made. Although it is true to say that this was illegal, this is rather nit-picking, since if the first outfield player had been "designated as the goalkeeper" just before being substituted, no breach would have resulted from the subsequent actions.

We mentioned earlier that we need to show students how to find a path from specification to code. Ledru sketches two alternative approaches. The first approach used the KIDS/VDM environment [4,5] including semi-automatic program synthesis [7]. The initial specification is in an implicit style. The program synthesized is written in REFINE, an ML-like functional language. This program is then run on the actions performed during the match: namely, the sending-off, and the three substitutions. The program is run with and without an outfield player being designated as the goalkeeper. However, the program specified is very simple, and does not contain any validation routines. As the result, the program does no checking of user input, and the user is left to discover the mistake by examining the state after each execution.

The second approach uses the IFAD toolbox [2]. This tool support for writing VDM specifications is proving very popular, but, as yet, the commercially available version does not support proof. Users are therefore encouraged to use its very sophisticated testing support instead. However, this involves the user in writing an executable specification. In the simple example above, this meant only minor changes, for example

Implicit style

```
RED-CARD(p : player)
ext wr on-field-players : player-set
    wr potential-substitutes: player-set
pre p ∈ on-field-players ∨
    p ∈ potential-substitutes
post
    on _ field _ players = on _ field _ players \ {p} ∧
    potential _ substitutes = potential _ substitutes \ {p}
```

Explicit style

```
RED-CARD : player → ( )
RED-CARD(p) Δ
    on-field-players : = on-field-players \ {p};
    potential-substitutes: = potential-substitutes \ {p}
pre p ∈ on-field-players ∨ p ∈ potential-substitutes
```

This time, when the user tests the specification itself on the faulty example, the third illegal substitution cannot be made. The toolbox responds with:

```
Run-time error 58:
The pre-condition evaluated to false
At line: 141 column: 5
```

The user may inspect the specification to examine the precondition which has been violated. Again, in real life, this would alert the software developer of the need to provide proper validation routines if the program is to be robust.

3 Speedway

Whilst engaging, the example above is small, and once this particular rule has been formalized there is little else of interest, other than in writing the validation routines. The speedway rulebook, on the other hand, is complex and

constantly changing, as each season brings a new crop of rule changes, designed to make the sport fairer or more entertaining; but seemingly destined to lead to confusion and last minute patches, in some cases by "case law" after the season has started.

The basic format of a premier league speedway match is fairly simple. Each team is comprised of (normally) seven riders. Riders are ranked according to how well they have ridden previous matches (or, at the start of the season, by how well they performed in the last season). The rules lay down a maximum total for each team: currently, it is forty-one points. It is up to the team manager when he picks his team at the start of the season how he stays within the limit. Some teams will be "top-heavy": in other words, have two or three extremely good riders and the rest fairly weak; others will have the talent more evenly spread, with no particularly outstanding riders. When the riders are arranged in order of their averages, the three best riders are known as the *heat leaders*, and take numbers 1, 3, and 5. The two next best riders are known as *second strings*, and take numbers 2 and 4. The remaining two riders, 6 and 7, are known as the *reserves*, although they are not reserves in the football sense, as they take a full part in the meeting from the beginning. There are fifteen heats. In each heat, each of the two teams fields two riders. For the first fourteen heats, these will normally be the two riders specified in the programme, unless the team manager makes a substitution. The riders race around the track for four laps. At the end, assuming that there are at least three finishers, the rider coming first receives three points, second receives two points, and third receives one point. At the end of the match, the team with the most points wins the match and receives two points in the league. If there is a draw, each team receives one point. After the teams have raced each other home and away, the team winning overall, on aggregate score over the two legs, receives a bonus point in the league.

The main complications come with the rules for substitutes. An excerpt from the rule book is shown in Figure 1. The riders for the first fourteen heats are stipulated in the programme¹. For example, in heat one, each team sends out the riders numbered 1 and 2. In heat two, they both send out riders 6 and 7. In heat five, the home team sends out riders 3 and 4 whereas the away team sends out riders 1 and 2. Put another way, rider 1 normally takes heats 1, 6, 10, and 13; rider 2 normally takes heats 1, 6, 8, and 10; and so on.

Riders Numbers 1 to 5 have four programmed rides but may take a maximum of seven rides.

Reserves may have a maximum of seven rides that can be taken at any time. All exclusions for starting infringements or exceeding the two minute time allowance count as rides once a reserve has completed three races. A reserve that has been replaced in a race is not eligible to replace a rider subsequently excluded from the same race.

All riders must have at least three rides prior to Heat 15 unless declared injured by the track medical officer. If a rider is unable to take the minimum number of rides, their place in a heat may not be taken by any other rider until this requirement is met. Exclusions for starting infringements or exceeding the two minute time allowance do not count as rides for the purposes of this regulation.

After Heat 4, a team that is behind by eight or more points may substitute a rider in the next heat with another rider from the team. Rider Numbers 1 to 5 however, may only be used as a tactical substitute once each.

· · · · ·

Replacement facilities are available to BPL teams as follows:

- *One heat leader absent - Guest or Rider Replacement.*
- *Two heat leaders absent - Guest for the rider with the highest average, Rider Replacement for the other.*
- *Three heat leaders absent - Guests for the riders with the highest and lowest averages, Rider Replacement for the other.*
- *Other top five rider absent - Rider Replacement.*

Figure 1: Two Rule Book Excerpts

Because matches where one team is running away with the win are very boring, a rule was introduced to help a team which is falling away. If a team is more than a stipulated number of points behind (eight points last season; six points for the present season), they may substitute one rider for another. This may be any of the other six riders, and is known as a *tactical substitution*. It differs from a *reserve substitution*, where rider 6 or 7 may be substituted for any other rider at any time. All these substitutions are subject to the constraint that every rider must have at least three, and no more than seven, rides, by the end of heat fourteen.

¹ For heat fifteen, the teams must select two out of their best three riders on the night, i.e. the riders scoring highest in heats one to fourteen.

A situation arose last season where a potentially illegal substitution was made. Rider number 2 had taken two of his four designated heats and scored zero on both. At this point, he had two rides left on the programme. On the next ride, his team being nine points behind, he was substituted by a heat leader, a move which paid off, as the substitute gained two points. This was perfectly legitimate, as he still had one ride left in the programme, in heat eleven. However, when it came to this heat, his team were thirteen points behind. He was substituted again, which was still legitimate, as even had his team won 5-0, the maximum win possible, his team would still have been eight points behind, and he could have been tactically substituted for another rider to make up his third ride. In the event, his team won the heat 5-1 and the scores lay at 37-28. It was at this point that the illegality occurred. He was *not* given the next heat. The subsequent events were:

Heat	Heat score	Overall score	Comment
11	1-5	37-28	They should make the tactical substitution now for heat 12 ...
12	3-3	40-31	They didn't, but got away with it. They should make it now ...
13	4-2	42-33	They didn't, but got away with it. They make it for heat 14...
14	4-2	48-35	... the last possible. (He got 0 again!)

If the situation had been:

Heat	Heat score	Overall score	Comment
11	1-5	37-28	They should make the tactical substitution now for heat 12 ...
12	1-5	38-33	Now they can't make a tactical substitution ...
13			... and may never be able to make one.
14			How can he get his third ride?

As he was not a reserve, the only opportunities to substitute him back into the meeting would occur when his team was behind by eight points or more. The correct interpretation of the rules would have been to take a heat up to an including the last guaranteed such opportunity – in this case, heat twelve.

We do not know what the referee would have done at that point, other than abandon the meeting, or rerun all the heats from heat twelve.

4 Initial specification

The speedway rule book is far too complex to attempt an explicit-style specification as an initial formulation, with or without tool support. The first author of this paper can write formal specifications but is not always up to date and clear about the minutiae of speedway rules. The second author is not a software engineer but is more reliable on speedway matters! Thus it was going to be a collaborative effort, with possibly many attempts to capture the full picture.

We decided to start with an abstract, implicit, unbiased specification. The speedway rule book is complex, and contains ambiguities. There are even missing cases, as in the second fragment in Figure 1. The function described there is partial, giving mappings only from {1}, {3}, {5}, {1,3}, {1.5}, {3,5}, {1,3,5}, {2}, and {4}; whereas it should be total over the power set of riders 0..5.

We do not think there are any actual inconsistencies in the written rule book, although these could occur in the "virtual rule book" which exists as rules are disambiguated, possibly by different referees at different matches on the same night. This is the "case law" situation we referred to earlier. Formalization helps us to predict what these are likely to be as the season gets under way.

Figure 2: The End State of a Speedway Meeting

The "real world" state we are modelling is best shown as a typical speedway supporter's filled-in program, as shown in Figure 2. The salient features of the state which we have described in this paper are:

1. We need to keep track of individual riders scores, and their total score, to decide who can ride in heat fifteen.
2. We need to see how many races each rider has taken, and how many are still designated, to ensure that the constraints are met.
3. We need to keep track of the running scores of each team, to know when we can make tactical substitutions.
4. Since any rider (apart from reserves) can only be used at most once as a tactical substitution or a rider replacement, we need to keep track of who has been used in this way.
5. We need to record riders who are absent from the meeting (due to injury, illness, or Grand Prix competitions), and of any who are certified injured during the course of the meeting, as they are of course exempt from the requirement to race at least three times.
6. We have chosen to do this specification from the viewpoint of one team only, so the only pertinent value from the opposing team's state is their total score, which we need to compare with our own so that we know whether we are allowed to make tactical substitutions. It would be a simple change to involve both teams, involving replicating most of the state variables and some of the values (such as `home_rider_averages`, `visitors_rider_averages`, etc.)

A fragment of an initial specification is shown in Figure 3 overleaf. The full specification, including changes from the 1998 to the 1999 season, is available from the first author.

5 Decomposition and error discovery

Many mistakes were made in drawing up the initial specification, but these were found easily, either by discussion between the authors, or by proof – chiefly that the data type invariant was preserved. Typical errors in the latter case were of missing, too weak, or too strong preconditions. Proof was invaluable in this respect. The invariants in speedway are relatively simple. Neither author had much trouble translating these from the rulebook. The ramifications involved in substitution operations, in contrast, are much more difficult to get straight in one's mind. Automatically proving (or attempting to prove) that an invariant still holds usually led immediately to the detection of bugs in the specification, if there were any.

We then began to move towards a more concrete specification, imagining a program which would act as an advisor for a referee or team manager. To illustrate this stage, let us look at a slightly simpler case than the situation which arose in §3. Suppose that we wish to make a reserve substitution of rider *sub* for rider *subbed*. The first precondition for this is that the heat about to be raced is the fourteenth or earlier (we cannot make this kind of substitution in heat fifteen). An initial attempt at an abstract specification might be:

```
RESERVE-SUBSTITUTION(sub: reserve, subbed : rider)
  ext rd heats_completed, heats_run(sub), heats_run(subbed)
  ext wr heats_left(sub), heats_left(subbed)
  pre   heats_completed < 14 ∧ ... ∧ ...
        card heats_run(sub) ∪ heats_left(sub) < 7 ∧
        card heats_run(subbed) ∪ heats_left(subbed) > 3
  post
    heats_left(sub) =  $\tilde{heats\_left}(sub) \cup \{heats\_completed + 1\}$  ∧
    heats_left(subbed) =  $\tilde{heats\_left}(subbed) \setminus \{heats\_completed + 1\}$ 
```

This will certainly preserve the invariant:

```
inv mk-match(hc, ts, tos, is, ib, hr, hl, tacs, rr, abs, inj) ==
  ∀ i ∈ rider .
    (hr(i) ∩ hl(i) = { } ∧ card hr(i) ∪ hl(i) ≤ 7 ∧
     i ∉ abs ∪ inj ∪ track_reserve → card hr(i) ∪ hl(i) ≤ 3,
```

values

team_size == 8
 rider_averages == {1 → 9.20, 2 → 4.47, 3 → 8.72, 4 → 6.72, 5 → 8.60, 6 → 3.55, 7 → 3.40, 8 → 3.00}
 heat_leader = {1, 3, 5} - - The eighth rider is the track reserve²
 second_string = {2, 4} - - initialize at start?
 reserve = {6, 7}
 track_reserve = {8}

types

rider = 1 .. team_size
 rider_scores = **map** rider to **N**
 ...
 rider_heats = **map** rider to **set of** 1 .. 15

state match ::

heats_completed : 0 .. 15
 total_score : **N**
 total_opposition_score : **N**
 individual_scores : rider_scores
 ...
 heats_run : rider_heats
 heats_left : rider_heats
 ...
 tac_subs : **set of** rider
 rider_replacements : **set of** rider
 absent_riders : **set of** rider
 injured_riders : **set of** rider

inv mk-match(hc, ts, tos, is, ... , hr, hl, ... , tacs, rr, abs, inj) ==

$\forall i \in \text{rider} .$
 $(hr(i) \cap hl(i) = \{ \} \wedge$
 $\text{card } hr(i) \cup hl(i) \leq 7 \wedge$
 $(i \notin \text{abs} \cup \text{inj} \cup \text{track_reserve} \rightarrow \text{card } hr(i) \cup hl(i) \geq 3))$

init mk-match(hc, ts, tos, is, ... , hr, hl, ... , tacs, rr, abs, inj) ==

(0, 0, 0,
 {x → 0 | x ∈ 0..8},
 ...
 {x → { } | x ∈ 0..8},
 1 → {1,6,10,13}, 2 → {1,6,8,10}, 3 → {3,5,9,12}, 4 → {3,5,9,14}, 5 → {4,7,11,13}, 6 → {2,7,11,14},
 7 → {2,4,8,12}, 8 → { }},
 ...
 { }, { },
 { } - - initialized to a non-empty set if riders injured etc.
 { })

Figure 3: Fragment of the VDM specification

² If rider replacement is being used, a team is allowed an additional member, the track reserve.

bearing in mind that in the output state the substitute's total rides increase by one, and the substituted rider's total decreases by one.

We definitely require the substitute rider to have actually completed no more than seven races. However, is the second precondition too strict? If the reserve has races left (but not completed) on the programme, but we keep on giving him extra rides, then presumably these could be abandoned. The team manager might prefer it if the reserve was potentially substitutable by another rider once he has reached seven races if he still has some left on the programme, but this is not strictly necessary – the team can go out one rider short if it has to. It is up to us whether we use the stronger or the weaker condition here. The weaker precondition would be simply

$$\text{card heats_run(sub)} < 7$$

A more complex precondition, which allowed the substitution to go ahead if there were heats left on the programme but the total came to more than 7, would insist that the reserve could be substituted by another rider in one of the races left, as intimated above. This is similar to the case described below, for the substituted rider, and we shall not discuss it further here.

The third precondition states that the cardinality of the union of the heats run and the heats left on the program shall be strictly greater than three, so when one is lost through the current substitution, the constraint on minimum number of rides is still met.

However, as we have seen in the example above, it is perfectly legitimate for a rider to temporarily break this precondition, provided it can be made good. In formal methods we cannot break preconditions, however temporarily, but we can see that what is required here is an operation to find a heat where the rider can be potentially substituted back, *for all possible states* reachable from the current state. In the example above, there was exactly one heat fulfilling this requirement, heat 12.

Before the specification of this operation becomes unnecessarily complex, we note that the top level operation that we really require is not one to *make* a substitution, but one which *tries* to make a substitution, and returns true or false according to whether it has been successful or not. It is still an operation, rather than a function, because if it returns true, it will also change the state. This operation can be decomposed into a conditional composite

TRY-SUBSTITUTION(sub: reserve, subbed : rider) $\triangle$
if *REQUEST_RESERVE-SUBSTITUTION*(sub: reserve, subbed : rider) **then**
MAKE_RESERVE-SUBSTITUTION(sub: reserve, subbed : rider)

which:

- requests a substitution – this returns a value of true or false, depending on whether a substitution can be made or not and
- if true, makes the substitution.

The specification for the first of these is simple:

REQUEST_RESERVE-SUBSTITUTION(sub: reserve, subbed : rider) b : B
ext rd heats_completed, heats_run(sub), heats_run(subbed)
ext wr heats_left(sub), heats_left(subbed)
pre heats_completed < 14 $\wedge$ card heats_run(sub) < 7
post b = substitutable(subbed)

The second is

MAKE_RESERVE-SUBSTITUTION(sub: reserve, subbed : rider)


```

ext rd heats_completed, heats_run(sub), heats_run(subbed)
ext wr heats_left(sub), heats_left(subbed)
pre    heats_completed < 14  $\wedge$ 
        card heats_run(sub) < 7  $\wedge$ 
        substitutable(subbed)
post   heats_left(sub) = heats_left(sub)  $\cup$  { heats_completed + 1 }  $\wedge$ 
        heats_left(subbed) = heats_left(subbed)  $\setminus$  { heats_completed + 1 }

```

This is valid according to the decomposition laws, the combination of the preconditions and postconditions of the first operation guaranteeing the preconditions of the second.

Again we see the role of proof: the use of hierarchical structuring obviates the need for over-complexity, combining a clear top-level abstract specification with the use of proof to ascertain that decomposition into the more concrete specification is correct.

REQUEST_RESERVE-SUBSTITUTION is also decomposed to make any necessary substitutions back, to give the extra required subtlety to the substitutable function. More details are available in the full specification online.

6 Further work and conclusions

The first stage is to implement our system from the specification. We would like to extend the remit from testing the validity of actions to advising on strategy, a field in which we have also done some work, and test the validity of our resulting model against real matches. We anticipate the use of Bayesian techniques, with a range of factors involved in calculating the prior probabilities for heat outcomes.

There has been the usual flurry of activity during the closed season, and we can already anticipate problems with two rule changes in particular. Whilst these rules are perfectly valid in most senses of that word, we believe that they will introduce bias into the league, in that they will favour teams with (say) two extremely good heat leaders, and handicap teams which are more even. We can test this hypothesis by building a program based on our specification, and using it as the basis for a match simulator.

Related to this, and of more interest in the formal methods field, we want to test the maintainability of our specification, and the resulting programs, as the new season begins. The new rule changes do not cause problems for our specification method, even if they may cause the more general speedway problems mentioned above, but we shall track any arising "case law" with interest.

One possibility is that rules are simplified, necessitating more drastic changes to abstract specification and decompositions such as the one in §5 alike. An alternative approach to the top-down (with feedback loops) method employed so far might be a more bottom-up, granular approach. We would specify a few low level actions with strong preconditions – for example, a substitution could only be made if the rider has a total of more than three rides completed and on the programme – and then a technique akin to proof planning [1] could be used to compose these into complex operations involving sequencing, case splits, conditionals, and so on.

Overall, this is proving to be an interesting ongoing case study of the use of formal methods to specify a problem where the requirements are not so much complex (compared with other domains) but under-specified, and subject to arbitrary and sometimes whimsical changes. It reinforces our belief that the correspondents collaborating on the development need an abstract, implicit specification (almost) as a lingua franca, to agree upon what is to be represented and implemented in the program. The traditional VDM method of reification and (especially) decomposition have proved useful for developing the initial specification. Over the coming season we shall start to investigate how maintainable such a multi-level specification can be, and whether a more bottom up approach might be called for.

Finally, and perhaps most usefully – we referred earlier to the pedagogic use of examples such as these. In particular, we believe that carrying out proof can help to detect bugs in a specification. However, whether this will turn out to be constructive help for typical students, as opposed to their more proficient instructors, is as yet unproven. Our evidence so far is patchy and anecdotal. We are looking for collaborators to help us carry out both pilot and large-scale evaluation of these methods. In the longer term, a related proof tool is currently under development, which we hope to adapt so that it provides a critique of a specification, based on faulty conjectures discovered whilst attempting to discharge proof obligations.

References

1. A. Bundy. The Use of Explicit Plans to Guide Inductive Proofs *9th Conference on Automated Deduction*, Argonne National Laboratory USA May 1988. In R. Lusk and R. Oberbeek (eds.), *Lecture Notes in Computer Science* **310**, Springer-Verlag, London, pp 111–120, 1988.
2. R. Elmstrom, P.G. Larsen, and P.B. Lassen. The IFAD VDM-SL toolbox: a practical approach to formal specifications. *ACM SIGPLAN Notices*, 29(9), 1994.
3. C.B. Jones. *Systematic Software Development using VDM*. Second Edition, Prentice-Hall, London, 1990.
4. Y. Ledru. Proof-based development of specifications with KIDS/VDM. In M. Naftalin, Tim Denvir, and Miquel Bertran, editors, *FME'94: Industrial Benefits of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 214{232. Springer-Verlag, 1994.
5. Y. Ledru and M.-H. Liegeois. Prototyping VDM specifications with KIDS. In *Proceedings of the 7th Knowledge-Based Software Engineering Conference*. IEEE Computer Society Press, 1992.
6. H. Lowe. *The Use of Theorem Provers in the Teaching and Practice of Formal Methods*. Electronic Workshop Series in Computer Science, Springer-Verlag , 1997.
7. D.R. Smith. KIDS: a semi-automatic program development system. *IEEE Transactions on Software Engineering*. Special Issue on Formal Methods,, 16(9), 1990.
8. J.G. Turner and T.L. McCluskey. *The Construction of Formal Specifications: An Introduction to Model-based and Algebraic Approaches*, McGraw-Hill, 1994.