Experience with Literate Programming in the Modelling and Validation of Systems

Theo C. Ruys and Ed Brinksma

Faculty of Computer Science, University of Twente.
P.O. Box 217, 7500 AE Enschede, The Netherlands.
{ruys,brinksma}@cs.utwente.nl

Abstract. This paper discusses our experience with literate programming tools in the realm of the modelling and validation of systems. We propose the use of literate programming techniques to structure and control the validation trajectory. The use of literate programming is illustrated by means of a running example using Promela and Spin. The paper can also be read as a tutorial on the application of literate programming to formal methods.

1 Introduction

In the past years, we have been involved in several industrial projects concerning the modelling and validation of (communication) protocols [3, 10, 18]. In these projects we used modelling languages and tools - like Promela, Spin [5, 7] and UPPAAL [13] – to specify and verify the protocols and their properties. During each of these projects we encountered the same practical problems of keeping track of various sorts of information and data, for example:

- many documents, which describe parts of the system;
- many versions of the same document;
- consecutive versions of validation models;
- results of validation runs.

The problems of managing such information and data relate to the maintenance problems found in software engineering [15]. This is not surprising as, in a sense, validation using a model checker involves the analysis of many successive versions of the model of a system.

This paper discusses our experience with *literate programming* techniques to help us tackle these information management problems.

Literate Programming. Literate programming is the act of writing computer programs primarily as documents to be read by human beings, and only secondarily as instructions to be executed by computers.

The term "literate programming" was coined by D.E. Knuth, when he described WEB [11, 12] the tool he created to develop the TEX typesetting software.

In general, literate programs combine source code and documentation in a single file. Literate programming tools then parse the file to produce either

readable documentation or compilable source code. One of advantages of literate programming is that we are not longer forced to comply with the syntactical order of the programming language. Types, variables and constructs can be introduced at alternative locations where they serve the purpose of the literate document better, e.g. where they are best understood by a human reader. The literate programming tool will reassemble the source parts into their formally correct order.

A literate style of programming has proven very helpful, especially for terse and/or complex programs. In the area of functional programming – where, by nature, programs are often quite terse - the art of writing functional scripts has become quite popular (see for instance [8]). It is remarkable that in the domain of formal methods, which deals with the specification of complex and safety-critical systems, the benefits of literate programming techniques have not yet been acknowledged.

In Sect. 2 we will discuss the application of literate programming in the modelling phase. In Sect. 3 we sketch how the same techniques can be used to structure the validation process. The use of literate programming in the validation trajectory is illustrated by means of a running example using Promela and Spin.[1] The paper is concluded in Sect. 4 where some conclusions and future work are discussed.

2 Literate Modelling

In a way, the literate style of specification has already reached the formal methods community. For example, a Z specification [19] can be considered as a literate specification. The "source code" of a Z specification consists of Z constructions like schema's and constraints, whereas the intertwining documentation (in LaTeX) can be regarded as "literate comments". Type checking tools for Z - like fUZZ and ZTC - discard these explaining comments when type checking a Z specification.

The use of literate programming techniques in the modelling phase has several advantages. Several parties are involved in the design of a system, e.g. users, designers, programmers, testers, etc. It is important that they all agree on the same specification of the system. For that reason the specification should be readable and acceptable for all parties. In this way, literate modelling helps to explain formal specifications.

Besides making the documentation more easily accessible, using literate modelling gives us the possibility to annotate the specification or model in a structured way without obscuring the formal specification with too many comments. Common forms of annotations include:

- identifying the *source* of information, which is especially useful when the specification or model is based on different documents from different parties;

[1] This document itself is written as a literate specification. This means that not only the document you are reading now, but also the validation results of the running example have been generated from a single source file.

- discussing *modelling choices*, e.g. the abstractions from the design, or assumptions on the environment.
- identifying points of *attention*, e.g. for a future validation phase, or when documents contain contradictory information.

In our modelling work we used the literate programming tool `noweb` [9, 16, 17] developed by Norman Ramsey. `noweb` is a literate programming tool like Knuth's `WEB`, only simpler. Unlike `WEB`, `noweb` is independent of the programming language to be literated. `noweb` has been used in combination with Pascal, C, C++, Modula-2, Icon, ML, Promela and others. A recent book-length example of literate programming using `noweb` is Fraser and Hanson's book on `lcc` [4].

2.1 Example – Modelling

To illustrate the use and benefits of literate techniques in the modelling phase, we present an example using `noweb`. First we give a specification of the specification problem in natural language. Next, we present a literate specification of the model in Promela. In the next section we show how to include validation results into the literate document. Consider the following scheduling problem.

Problem:

> Four soldiers who are heavily injured, try to flee to their home land. The enemy is chasing them and in the middle of the night they arrive at a bridge that spans a river which is the border between the two countries at war. The bridge has been damaged and can only carry two soldiers at a time. Furthermore, several landmines have been placed on the bridge and a torch is needed to sidestep all the mines. The enemy is on their tail, so the soldiers know that they have only 60 minutes to cross the bridge. The soldiers only have a single torch and they are not equally injured. The following table lists the crossing times (one-way!) for each of the soldiers:
>
> | | |
> |---|---|
> | soldier S_0 | 5 minutes |
> | soldier S_1 | 10 minutes |
> | soldier S_2 | 20 minutes |
> | soldier S_3 | 25 minutes |
>
> Does a schedule exist which gets all four soldiers to the safe side within 60 minutes?
>
> *Hint:* Before proceeding it may be instructive to try to solve the "soldiers" problem on paper.

During the modelling phase we are not interested in the 60 minutes constraint, but we only want to model how we can get the four soldiers from the unsafe to the safe side of the bridge.

Please note that the specification of the problem is straightforward, and consequently, the `noweb` specification may be trivial and even overly simple in some places. This is deliberately done to give a better overview of a *complete*

literate validation trajectory with noweb. A less trivial example of such trajectory can be found in [2].

1 ⟨*soldiers.pr* 1⟩≡ (20)
⟨*prelude* 2⟩
⟨*proctypes* 8⟩
⟨*init* 27⟩

Our Promela specification `soldiers.pr` consists of a ⟨*prelude*⟩, some process definitions ⟨*proctypes*⟩ and an initialization process ⟨*init*⟩.

In the left margin, noweb identifies the number of the *code chunk.* [2] In the right margin of the definition of a code chunk, between parenthesis, the number of the code chunk is listed, which *uses* this particular code chunk. All code chunks with the same name will be collected by noweb and put in the place where the chunk is used.

2 ⟨*prelude* 2⟩≡ (1)
⟨*constants* 4⟩
⟨*macros* 13⟩
⟨*types* 5⟩
⟨*channels* 3⟩
⟨*globals* 6⟩

The Promela ⟨*prelude*⟩ of the specification consists of constants, macros, types, channels and global variables. These parts will be defined when they are needed in the process definitions of the specification. We start our model of the "soldiers" problem by modelling the bridge.

3 ⟨*channels* 3⟩≡ (2)
```
chan unsafe_to_safe = [0] of {soldier, soldier} ;
chan safe_to_unsafe = [0] of {soldier} ;
```

The bridge is modelled with two channels. The channel `unsafe_to_safe` models the *unsafe* → *safe* direction, whereas `safe_to_unsafe` models the *safe* → *unsafe* direction.

Every time two soldiers have made it to the safe side, one of the men on the safe side has to go back with the torch. It is clear that it is not very helpful to go back with two men. For that reason, the channel `safe_to_unsafe` only passes a single `soldier`.

The four soldiers are identified by integers in the range `0..N-1`, where `N` is defined as follows:

4 ⟨*constants* 4⟩≡ (2)
```
#define N           4
```

This means that a `soldier` can be represented by a `byte`:

5 ⟨*types* 5⟩≡ (2)
```
#define soldier     byte
```

[2] In this paper the WEB style of chunk numbering is used. Another popular way of chunk identification is a tag *page.n*, where *n* indicates the *n*-th chunk on page *page*.

6 ⟨*globals* 6⟩≡ (2)

```
byte    time ;
byte    val[N] ;
```

The variable `time` is the number of minutes that have elapsed since the soldiers have started to move. Every time a soldier reaches the other side of the bridge, the variable `time` is updated.

The array `val` holds the times it takes the soldiers to cross the bridge. Because Promela doesn't support constant arrays, the array `val` has to be initialized in the `init` process:

7 ⟨*init val* 7⟩≡ (27)

```
val[0] =  5 ; val[1] = 10 ; val[2] = 20 ; val[3] = 25 ;
```

As specified in the problem description, the times for the soldiers to cross the bridge are 5, 10, 20 and 25 minutes, respectively.

8 ⟨*proctypes* 8⟩≡ (1)

⟨*proctype Unsafe* 9⟩
⟨*proctype Safe* 25⟩

Crossing the bridge is modelled by `soldiers` that are passed between two processes: `Unsafe` and `Safe`. In the beginning, all soldiers are at the `Unsafe` side, and the goal is to get all soldiers to the `Safe` side.

In the remainder of this section we only define the chunk ⟨*proctype Unsafe*⟩. For completeness, we have included the chunks ⟨*proctype Safe*⟩ and ⟨*init*⟩ in the Appendix.

9 ⟨*proctype Unsafe* 9⟩≡ (8)

```
proctype Unsafe()
{
  ⟨Unsafe: locals 10⟩
  ⟨Unsafe: body 11⟩
}
```

We model the location of the soldiers by the `bit`-array `here`. If `here[i]` is `1`, then soldier `i` is at the unsafe side of the bridge.

10 ⟨*Unsafe: locals* 10⟩≡ (9) 15▷

```
bit  here[N] ;
```

Initially, all soldiers are on the unsafe side of the bridge, so the body of `Unsafe` starts by initializing the array `here`:[3]

11 ⟨*Unsafe: body* 11⟩≡ (9) 12▷

```
here[0] = 1 ; here[1] = 1 ; here[2] = 1 ; here[3] = 1 ;
```

[3] Note that `noweb` has added new information in the right margin. The ▷ indicates the next chunk with the same name. Similarly, a ◁ symbol indicates the previous chunk with the same name.

The rest of Unsafe's body is responsible for crossing the bridge.

12 ⟨*Unsafe: body* 11⟩+≡ (9) ◁11

```
do
:: ⟨Unsafe: send two soldiers 14⟩
   ⟨Unsafe: one soldier arrives back 18⟩
od
```

In every iteration two soldiers are sent to the other side and one soldier is expected back with the torch. In the ⟨*Unsafe: send two soldiers*⟩ part, we need to randomly choose a soldier that is still at the unsafe side. For this purpose we introduce the macro `select_soldier(x)`:

13 ⟨*macros* 13⟩≡ (2) 17▷

```
#define select_soldier(x) \
if                        \
:: here[0] -> x=0         \
:: here[1] -> x=1         \
:: here[2] -> x=2         \
:: here[3] -> x=3         \
fi ;                      \
here[x] = 0
```

Only the guards for which `here[i]` is 1 are executable. One of these executable guards is randomly chosen and the variable `x` gets the number of this soldier. Now we can define the ⟨*Unsafe: send two soldiers*⟩ chunk:

14 ⟨*Unsafe: send two soldiers* 14⟩≡ (12) 16▷

```
select_soldier(s1) ;
select_soldier(s2) ;
unsafe_to_safe ! s1, s2 ;
```

where `s1` and `s2` are soldiers:

15 ⟨*Unsafe: locals* 10⟩+≡ (9) ◁10

```
soldier s1, s2 ;
```

16 ⟨*Unsafe: send two soldiers* 14⟩+≡ (12) ◁14

```
IF all_gone -> break FI ;
```

If there are no soldiers left at the unsafe side, the do-loop of the `Unsafe` process should be terminated. This `break` is really needed here, because otherwise the `Unsafe` process will be blocked (i.e. an invalid endstate in Spin) by `select_soldier(s2)` if there is only one soldier at the unsafe side.
This last construction uses the following macro definitions:

17 ⟨*macros* 13⟩+≡ (2) ◁13 26▷

```
#define IF          if ::
#define FI          :: else fi
#define all_gone    (!here[0] && !here[1] && !here[2] && !here[3])
```

The IF-FI combination implements a single IF clause and the all_gone predicate is *true* when all values in the here array are 0.

18 ⟨*Unsafe: one soldier arrives back* 18⟩≡ (12)

```
safe_to_unsafe ? s1 ;
here[s1] = 1 ;
time = time + val[s1] ;
```

The soldier s1 is the soldier that gets back with the torch. The time is updated accordingly to the time it took soldier s1 to cross the bridge.

The process Safe just mirrors the operations of the Unsafe process. The body of Safe - together with init - can be found in the Appendix.

This concludes our literate model of the "soldiers" example. The observant reader will have noticed that the literate style of specification allows us to introduce types, variables, etc. at the location where they are needed, and *not* where the Promela grammar would have forced us to do so.

noweb provides index and cross-reference features for code chunks and identifiers. The "soldiers" example only uses cross-references to code chunks. Larger programs or models are easier to understand if identifiers are also cross-referenced.

3 Literate Validation

Although the advantages of literate specification techniques in the modelling phase already proved quite useful in our projects, we have also tried to use literate programming in the validation trajectory.

As mentioned in the introduction, one of the difficulties of using model checkers is the management of all (generated) data during the validation trajectory. It is important that the validation results obtained using the model checker should always be reproducable [6]. Without tool support for the validation phase, one has to resort to general engineering practices and record all validation activities into a log-book.

Recording all this information requires rigorous discipline. The quality of the validation depends on the logging discipline of the validation engineer. Moreover, there remains the problem that after the validation phase one has to compose a coherent validation report from this huge collection of validation data. Experience has shown that this is not easy. In our validation projects, we have profited from literate techniques to help us record the collection of data involved in the validation trajectory:

- *validation models.* As discussed in Sect. 2, literate programming can be used to explicitly specify and annotate a model. Moreover, the *differences* between several validation models can be elegantly presented in a report containing several versions of the model. An example of this can be found in Sect. 3.2 (viz. ⟨*soldiers-60min.pr*⟩) and in [2].
- *validation results.* The results of validation runs, e.g. simulation traces, counter-examples can be included into the literate validation report.

- *directives for the validation runs.* The - often cryptic - directives and options to control validation tools usually end up in a `Makefile`. The rationale behind such directives, however, is usually not recorded. In a literate specification these directives can be annotated together with them. See for instance the code chunks ⟨*directives.dat*⟩ in Sect. 3.2.

Recall that the information above is usually scattered over several files, simulation traces, entries in log-books, etc. When using a literate style of validation, all this information can be collected into a single, literate document. Thus, not only do literate techniques solve the management of validation data, it also releases much of the burden of writing a validation report.

A validation report is especially needed when no 'serious' errors have been found during the validation of a system. A simple "no errors found" doesn't suffice. In such cases, the validation report should describe all the succesful scenarios to identify exactly those parts of the model which have been validated thoroughly. An example of a report of a validation trajectory of a "correct" model can be found in [2].

3.1 Validation Approaches

Before we continue with our (running) example, we discuss the two - extreme - validation approaches that can be followed when using a model checker: the verification approach and the falsification approach.

The purpose of the verification approach is to come up with a correct model on a certain level of abstraction. The verification approach is characterized by the following:

- During the validation phase the model of the system is fixed at a certain level of abstraction.
- All aspects of the model are systematically validated.
- During the validation of a certain aspect of the model, abstractions have to be made of other parts of the model.

The falsification approach aims at finding errors and weaknesses in the (initial) design of a system. The falsification approach focuses its attention on those parts of the system where flaws are most likely to occur. This approach is characterized by the following:

- The validation phase is started with a model on a high level of abstraction.
- During the validation phase, one zooms in at certain aspects of the model using local refinement techniques.
- Only a limited part of the system is validated and no information is obtained about the non-validated components.

Summarizing, the verification approach tries to ascertain the correctness of a detailed model, whereas the falsification approach tries to find errors in a model.

```
⟨Verification approach⟩≡
  ⟨Start with detailed model⟩
  ⟨Simulate⟩
  while not ⟨convinced by results⟩
  do
      ⟨Focus on particular aspects of the model⟩
      ⟨Make abstractions of the other parts as needed⟩
    [ ⟨Introduce errors into the environment⟩ ]
      ⟨Simulate and model check⟩
  od
```

Fig. 1. Pseudo-algorithm for the verification approach.

In other words, the verification approach is *specification-driven*, whereas the falsification approach is *error-driven*. Please note that both approaches prescribe extreme methods for validation. In practice, one usually adopts a combination of both approaches. We have used both approaches in our validation work.

Figure 1 presents a pseudo-algorithm for the verification approach whereas Fig. 2 presents a pseudo-algorithm for the falsification approach.

The verification approach starts with a detailed model of the system. Before starting the actual validation loop in the verification approach, the detailed model is simulated to obtain an initial degree of correctness. In general, the state space of such a detailed model will be too large for an exhaustive search by a model checker [5]. In the validation loop of the verification approach, one makes abstractions of parts of the complete model to zoom in on certain aspects of the model. These abstractions are needed to allow an exhaustive search by a model checker. The validation phase is ended when all crucial aspects of the model have been verified.

The falsification approach starts with an abstract model of the system. In the validation loop, the falsification approach tries to find errors by adding details to the model or by introducing errors into the environment. The validation phase of the falsification approach is ended when (enough) errors have been exposed or when resources (e.g. time, money) have run out.

In the ⟨*Introduce errors into the environment*⟩ step of both approaches, exceptional behaviour of the environment is introduced to validate the robustness of the system. When errors are found in the ⟨*Simulate and model check*⟩ step, these errors should be corrected, and the simulation and validation step should be started again.

Not surprisingly, the usage of literate programming techniques is different for both approaches. Using the verification approach, the initial model benefits from all annotation facilities of literate programming. Along the validation path, subsequent validation models will be built by making abstractions from the initial model. Using the falsification approach, the initial literate model may only contain annotations identifying the parts (i.e. abstractions) of the system that are *missing*. During the validation trajectory, consecutive validation models

```
⟨Falsification approach⟩≡
  ⟨Start with abstract model⟩
  ⟨Simulate and model check⟩
  while not ⟨errors found⟩ and ⟨resources available⟩
  do
    (  ⟨Zoom in on certain aspects of the model⟩
    or ⟨Introduce errors into the environment⟩ )
    ⟨Simulate and model check⟩
  od
```

Fig. 2. Pseudo-algorithm for the falsification approach.

will be constructed by adding details to the initial model. The annotations of the initial model should guide the details to be added.

When errors are found in the model they should be corrected. The literate document (together with dependency rules in a `Makefile`) will make sure that all previous results will be re-validated. For the verification approach this means that the initial model will be modified, whereas in the falsification approach it does not necessarily mean that the initial abstract model needs to be corrected. For example, an error may be detected in a particular refinement of the previous validation model. Moreover, the purpose of applying the falsification approach is to expose errors, not to come up with a correct model.

3.2 Example – Validation

To illustrate the process of literate validation, we continue our example and try to find a solution which brings the soldiers to the safe side within 60 minutes. The validation of the "soldiers" problem uses the verification approach.

First, we assure ourselves that our original specification `soldiers.pr` does not contain any errors. This means that we have to check for possible deadlocks (i.e. invalid endstates) in our specification.

The validation results themselves are meaningless if we cannot reproduce them. In the realm of Spin this means that we also have to record:

- the *directives* for the C compiler to build the pan analyser[4] ; and
- the run-time *options* for the pan analyser.

For the verification runs with Spin we use a data file (i.e. `directives.dat`), which contains for each Promela validation model these directives and options. A simple script is used to translate this data file into a `Makefile` that drives the complete validation process. For the Promela specification `soldiers.pr` the directives and options are the following:

[4] The pan analyser is the validation program which is *generated* by Spin [5]. It is the program that performs the validation of a system.

19 ⟨*directives.dat* 19⟩≡ 24▷

```
soldiers
-D_POSIX_SOURCE -DSAFETY -DNOFAIR -DMEMCNT=22
-c1 -w15 -m1000 -n
```

For this paper, the meaning of the Spin directives is not important. In a validation report, however, an explanation and reasoning behind these directives may be needed. Running the pan analyser produces the following (stripped) output:

```
(Spin Version 3.0.5 -- 5 November 1997)  [run on 07-January-98 17:36:18]

State-vector 52 byte, depth reached 66, errors: 0
    5072 states, stored
     438 states, matched
    5510 transitions (= stored+matched)
       1 atomic steps
hash conflicts: 1432 (resolved)
(max size 2^15 states)

Stats on memory usage (in Megabytes):
0.304   equivalent memory usage for states (stored*(State-vector + overhead))
0.204   actual memory usage for states (compression: 67.14%)
        State-vector as stored = 32 byte + 8 byte overhead
0.131   memory used for hash-table (-w15)
0.024   memory used for DFS stack (-m1000)
0.438   total actual memory usage

        Command being timed: "./soldiers.pan -c1 -w15 -m1000 -n"
        User time (seconds): 0.14
        System time (seconds): 0.03
```

Within 60 minutes. Now we try to find the schedule that get all soldiers to the safe side within 60 minutes. Our idea is to try to verify that "eventually, the `time` will be be greater than 60". This property can easily be formulated as a Linear Time Logic (LTL) formula: $\diamond(time > 60)$. The LTL property is violated if all soldiers are at the safe side and the time elapsed is less then or equal to 60. To let Spin find a counterexample which violates the property, the LTL property is translated to a Promela never claim which is simply added to our original Promela specification:

20 ⟨*soldiers-60min.pr* 20⟩≡

```
⟨soldiers.pr 1⟩
⟨never 22⟩
```

We use Spin's `-F` option to translate the LTL property to a never claim. The claim is the following:

21 ⟨*60min.claim* 21⟩≡

```
! (<> p)
```

where `p` is defined as follows:

22 ⟨*never* 22⟩≡ (20) 23▷

```
#define p (time > 60)
```

With ⟨*60min.claim*⟩ as input, "Spin -F" generates the following 'never claim':

23 ⟨*never* 22⟩+≡ (20) ◁22

```
never {     /* ! (<> p)  */
accept_init:
T0_init:    if
            :: (! ((p))) -> goto accept_S1
            fi;
accept_S1:
T0_S1:      if
            :: (! ((p))) -> goto accept_S1
            fi;
accept_all: skip
}
```

The verification of `soldiers-60min.pr` involves the following directives and options:

24 ⟨*directives.dat* 19⟩+≡ ◁19

```
soldiers-60min
-D_POSIX_SOURCE -DNOFAIR -DMEMCNT=22
-a -c1 -w15 -m1000 -n
```

The pan analyser will try to prove that $\diamond(time > 60)$ holds for all possible executions of the model. Running the pan analyser on `soldiers-60min.pr` produces the following (stripped) output:

```
(Spin Version 3.0.5 -- 5 November 1997)  [run on 07-January-98 17:36:39]

State-vector 56 byte, depth reached 127, errors: 1
     291 states, stored (522 visited)
     230 states, matched
     752 transitions (= visited+matched)
       1 atomic steps
hash conflicts: 79 (resolved)
(max size 2^15 states)

0.336   memory usage (Mbyte)

        Command being timed: "./soldiers-60min.pan -a -c1 -w15 -m1000 -n"
        User time (seconds): 0.04
        System time (seconds): 0.02
```

And we see that Spin has found an error in one of possible executions of the model. We let Spin generate a simulation trace leading to this error.

Running "Spin -M -t" on `soldiers-60min.pr` results in the Message Sequence Chart (MSC) of Fig. 3. The MSC shows a possible schedule to get all soldiers to the safe side within 60 minutes.

4 Conclusions

In this paper we have discussed our experience with the literate programming tool noweb in validation projects. Using a simple model as a running example,

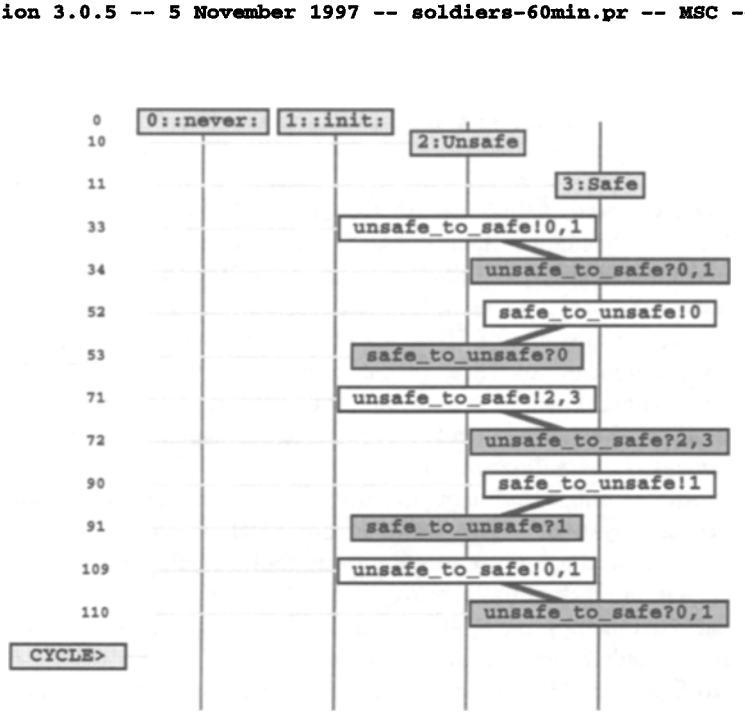

Fig. 3. Schedule to cross the bridge in 60 minutes.

we have introduced the use of literate techniques in the modelling and validation of systems.

The use of literate programming tools in the modelling phase has proven quite valuable. Especially the possibility to annotate the model has proven quite helpful to make the models more accessible and readable for all parties that are involved in the design of a system.

Literate techniques are also useful during the validation trajectory. All details on validation runs can be nicely structured into a validation report. However, essentially, the management of the validation trajectory is nothing more than the management of different versions of the model together with the validation results. For this class of management problems several so-called source-control tools [1] are available. One may argue that the use of literate programming techniques in the validation trajectory implements a source-control system by hand. To a certain degree this is true. When using literate techniques, the validation models are incrementally constructed, which could be automated using source-control tools.

However, a literate style of validation has several advantages over source-control tools alone:

- most source-control systems lack the possibility of annotating and describing modelling and validation choices;
- the management of validation results is more problematic;
- a source-control system typically does not help in composing a validation report.

On the other hand, using literate techniques during the complete validation trajectory may become tedious and time-consuming. Furthermore, it is probably not desirable (and helpful) that all validation results end up in a single document. One may wish to prune the validation tree to only include those results and models that are meaningful. Here, a source-control system could be helpful.

So far, validation activities in our group have been conducted by a single person at a time. As soon as more than one person is working on the validation of the same project, source-control systems are indispensable with respect to the management of the validation process.

The bottom line is that using literate programming techniques alone or using source-control systems alone are both not ideal. A combination of a source-control system and a literate programming tool probably works best to support a structured validation methodology.

In our current approach, all details on validation results of Spin have to be manually included into the literate specification. We are working on enhancements to Spin and XSpin to have these results automatically generated and included into a literate specification. Furthermore, we are trying to combine a source-control system like RCS [20] or PRCS [14] with XSpin.

References

1. Don Bolinger and Tan Bronson. *Applying RCS and SCCS.* O'Reilly & Associates, Inc., Sebastopol, 1995.
2. Pedro R. D'Argenio, Joost-Pieter Katoen, Theo C. Ruys, and G. Jan Tretmans. The Bounded Retransmission Protocol must be on time! (Full Version). CTIT Technical Report Series 97-03, Centre for Telematics and Information Technology, University of Twente, Enschede, The Netherlands, 1997. Also available from URL: `http://wwwtios.cs.utwente.nl/~dargenio/brp/`.
3. Pedro R. D'Argenio, Joost-Pieter Katoen, Theo C. Ruys, and G. Jan Tretmans. The Bounded Retransmission Protocol must be on time! In Ed Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, number 1217 in Lecture Notes in Computer Science (LNCS), pages 416–431, University of Twente, Enschede, The Netherlands, April 1997. Springer Verlag, Berlin.
4. Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley, 1995.
5. Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

6. Gerard J. Holzmann. The Theory and Practice of a Formal Method: NewCore. In *Proceedings of the IFIP World Congress*, Hamburg, Germany, August 1994. Also available from URL:
`http://cm.bell-labs.com/cm/cs/doc/94/index.html`.
7. Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. See also URL:
`http://netlib.bell-labs.com/netlib/spin/whatispin.html`.
8. Johan Jeuring and Erik Meijer, editors. *Advanced Functional Programming – Tutorial Text of the First International Spring School on Advanced Functional Programming Techniques*, number 925 in Lecture Notes in Computer Science (LNCS), Båstad, Sweden, May 1995. Springer Verlag, Berlin.
9. Andrew L. Johnson and Brad C. Johnson. Literate Programming using `noweb`. *Linux Journal*, pages 64–69, October 1997.
10. Pim Kars. The Application of PROMELA and SPIN in the BOS Project. In Jean-Charles Grégoire, Gerard J. Holzmann, and Doron A. Peled, editors, *Proceedings of SPIN96, the Second International Workshop on SPIN (published as "The Spin Verification System)*, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Rutgers University, New Jersey, August 1996. American Mathematical Society. Also available from URL:
`http://netlib.bell-labs.com/netlib/spin/ws96/Ka.ps.Z`.
11. Donald E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, May 1984.
12. Donald E. Knuth. *Literate Programming.* Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information (CSLI), Stanford University, California, 1992.
13. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Springer International Journal of Software Tools for Technology Transfer (STTT)*, 1(1/2), October 1997.
14. Josh MacDonald. PRCS – Project Revision Control System. Available from URL:
`http://www.xcf.berkeley.edu/~jmacd/prcs.html`.
15. Roger S. Pressman. *Software Engineering – A Practioner's Approach.* McGraw-Hill, New York, third edition, 1992.
16. Norman Ramsey. `noweb` – homepage. Available from URL:
`http://www.cs.virginia.edu/~nr/noweb/`.
17. Norman Ramsey. Literate Programming Simplified. *IEEE Software*, 11(5):97–105, September 1994.
18. Theo C. Ruys and Rom Langerak. Validation of Bosch' Mobile Communication-Network Architecture with SPIN. In *Proceedings of SPIN97, the Third International Workshop on SPIN*, University of Twente, Enschede, The Netherlands, April 1997. Also available from URL:
`http://netlib.bell-labs.com/netlib/spin/ws97/ruys.ps.Z`.
19. J.M. Spivey. *The Z Notation – A Reference Manual.* Prentice Hall, New York, second edition, 1992.
20. Walter F. Tichy. RCS – A System for Version Control. *Software, Practice & Experience*, 15(7):637–654, July 1985.

Appendix

The **Safe** process is defined as follows:

25 ⟨*proctype Safe* 25⟩≡ (8)

```
proctype Safe()
{
  bit     here[N] ;
  soldier s1, s2 ;
  do
  :: unsafe_to_safe ? s1, s2 ;
     here[s1] = 1 ;
     here[s2] = 1 ;
     time = time + max(val[s1], val[s2]) ;
     IF all_here -> break FI ;
     select_soldier(s1) ;
     safe_to_unsafe ! s1
  od
}
```

where the macro **max** and **all_here** are defined as:

26 ⟨*macros* 13⟩+≡ (2) ◁17

```
#define max(x,y)        ((x>y) -> x : y)
#define all_here        (here[0] && here[1] && here[2] && here[3])
```

The **init** process initializes the array **val** and starts the processes **Unsafe** and **Safe**.

27 ⟨*init* 27⟩≡ (1)

```
init {
  ⟨init val 7⟩
  atomic { run Unsafe() ; run Safe() ; }
}
```