Int J Softw Tools Technol Transfer (2003) 4: 246–259 / Digital Object Identifier (DOI) 10.1007/s10009-002-0078-1

Managing the verification trajectory

Theo C. Ruys, Ed Brinksma

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

Published online: 18 June 2002 – © Springer-Verlag 2002

Abstract. In this paper we take a closer look at the automated analysis of designs, in particular of *verification* by *model checking*. Model checking tools are increasingly being used for the verification of real-life systems in an industrial context. In addition to ongoing research aimed at curbing the complexity of dealing with the inherent state space explosion problem – which allows us to apply these techniques to ever larger systems – attention must now also be paid to the methodology of model checking, to decide how to use these techniques to their best advantage. Model checking "in the large" causes a substantial proliferation of interrelated models and model checking sessions that must be carefully managed in order to control the overall verification process. We show that in order to do this well both notational and tool support are required. We discuss the use of software configuration management techniques and tools to manage and control the verification trajectory. We present Xspin/Project, an extension to Xspin, which automatically controls and manages the validation trajectory when using the model checker Spin.

Keywords: Model checking – Computer-aided verification – Software configuration management

1 Introduction

The research and development of formal methods and related tool environments for supporting the design, analysis, and implementation of software systems now spans more than two decades. Although researchers have pointed out the potential benefit of the tool-supported use of formal notations, models and theories to improve the quality of complicated systems – especially concurrent, reactive systems – the actual application of such methods and tools to real-life examples is a relatively recent phenomenon.

There are several reasons for this. First, the field of research has become mature in the sense that several powerful theoretical tools have been developed to form the basis for such applications. In particular, this involves models for dealing with real-time, stochastic, and hybrid systems that are relevant for the description and analysis of so-called *embedded systems*. These software systems usually control and extend the functionality of the physical systems that they are part of (cars, airplanes, dishwashers, CD players, etc.). Due to the safety critical nature of the overall system, the implementation in hardware and/or the high replication of such products, the correctness requirements of these systems are relatively high, which warrants an investment in the use of formal methods and tools.

Second, the performance of the tools has improved dramatically over the past few years, bringing systems of realistic size within reach of a more formal treatment. In part this has to do with general progress such as the availability of ever-cheaper memory and ever-faster hardware. This is also due to more specific developments, some of which we discuss below.

In this paper we take a closer look at the analysis of designs, in particular of *verification* by *model checking*. Some of what we have to say, however, will, mutatis mutandis, also apply to other analytic techniques, such as proof checking, simulation, and testing. The first of these is also a verification technique, like model checking, i.e., it can be used to show formally that a design fulfils a given property. The latter two techniques can generally be used only to increase our confidence in the correctness of designs, but in some situations they have advantages over strict verification, such as application to very large designs, or application to physical systems.

Although there is substantial progress in the development of both proof and model checking tools, the latter are currently more successful in terms of industrial applications. One reason is that for proof checking the verifier must first suggest a proof outline of the desired verification property to the tool, which is subsequently completed and checked by the tool in an extended interaction with the tool user. In the case of model checking, less user activity seems to be required, although more than is generally assumed, as we will point out in the next section. When provided with a model and a property to be verified, in principle a model checker comes up with a result fully automatically. The reduced level of user interaction is seen as an advantage for industrial applications, as it has better chances of being used successfully by non-experts.

A second reason for the relative success of model checking is that here there has been substantial progress in dealing with the *state space explosion problem*, which affects all tools in which concurrent systems are basically represented in terms of their states and the transitions between them. Over the past years there has been a remarkable improvement in dealing with larger state spaces via the use of new techniques (bitstate hashing [24], BDDs [7–9], on-the-fly [12], compositional [1, 34, 47] and partial-order techniques [19, 25, 40, 52]) that reduce, sometimes dramatically, the memory requirements. It is this development that has taken model checking applications from the stage of toy examples to real-life applications.

This paper is concerned with what could be called the *methodology* of model checking, and has grown out of our experience [13, 28, 46] with this technique in larger applications. Although formal methods is by now an accepted name for the field of computer science that concerns itself with the formal treatment of the problems mentioned above, it does not yet generally offer what its name seems to suggests, viz. *methods* for the application of formal techniques. Research has been concentrated mainly on the techniques themselves, but now that real applications have come within reach the question of how to apply them best can no longer be ignored.

The main observation that underlies this work is the fact that model checking "in the large" causes a substantial proliferation of interrelated models and model checking sessions that must be carefully managed in order to control the overall verification process. To do this well both notational and tool support are required, as we will show. In [44, 45] we proposed 'literate techniques' [30, 42] to structure the modelling and verification trajectory. We concluded that these techniques are useful to structure the modelling phase, but do not scale up to manage the verification phase of validation projects. This paper is concerned with structural and systematic ways to control the verification phase.

The structure of the rest of this paper is as follows: Section 2 contains an analysis of the structure of larger model checking applications, giving rise to what we have termed the *systematic verification model* (SVM). In Sect. 3 we discuss the desired tool support for *verification management*. In Sect. 4 we present XSPIN/PROJECT, a prototype tool to manage and control the validation trajectory when using the model checker SPIN. In Sect. 5 we draw our conclusions and discuss future work.

2 Systematic verification model

Most literature on verification by model checking deals with the verification of a small set of properties with respect to a single model. The contribution of such publications is often found in suggestions of clever ways in which the complexity of the model and/or property may be reduced, so that the model checking procedure requires less time and/or memory, or can be applied to larger cases. Although recent progress in this area has been most impressive, as already discussed in the previous section, the systematic verification of real-life systems usually cannot be dealt with in this restricted framework. The state spaces of such systems are generally many orders of magnitude greater than what can be handled by any existing tool, and there are good reasons to believe that this is a structural problem. A virtually autonomous process leads to the creation of ever more involved systems, where the growth in complexity easily outweighs the increased analytical capacities. Although there have also been reports of verification results for extremely large state spaces using BDD [7–9] and compositional model checking [1, 2, 34, 47] techniques, these approaches do not give uniformly good results. They exploit implicit regularities in the structure of the verification problems that are not at all well understood, and their performance may be considered 'chaotic', in the sense that small changes in the problem statement may lead to arbitrarily large differences in performance.

A second practical problem is that of the *validity* of the formal model (or the properties) that is used for verification. In the literature one is often confronted with benchmark problems (e.g., dining philosophers [14], alternating bit protocol [4], railway crossing [33], bounded retransmission protocol [20], etc.). In practical applications the question whether the formalised problem statement is an adequate reflection of the actual problem can be a significant source of problems, where both the complexity of the involved system and the (lack of) precision of the informal specification of the system's functionality and requirements play a role.

Both of the above problems give rise to a more involved verification method in which (many) different formal models are used in the course of the verification procedure. To deal with large state spaces abstractions of the complete system model are used. These abstractions should preserve the (non)validity of the requirements that must be verified. Formally, the adequacy of an abstraction with respect to a given property should be demonstrated; in practice this is often simply assumed. When more than one formal property must be verified it is likely that abstractions that are sufficiently small can be obtained only with respect to the single properties, and not for their conjunction. This implies that a set of related, but different abstractions must be maintained.

Another way of dealing with too large state spaces is to give up on the precision of the verification result. Some model checkers offer facilities to explore the state space with less than complete coverage (e.g., using the supertrace/bitstate algorithm of Spin [24]). To ensure the reproducibility and interpretability of such verification results information concerning the approximations used should be maintained.

Where the maintenance of relevant sets of models and related information is complicated by the necessity to keep the state space problem in check, the issue is further complicated by the validity problem. In practice, its presence means that whenever a property is *falsified*, the negative result may have two subtly but importantly different causes. It may be a *modelling error*, i.e., upon studying the error it is discovered that the model (in a given case, the relevant abstraction) does not reflect the design of the system. It implies that the model must be corrected, and verification must be restarted with respect to the improved model. If the analysis of the falsification shows that there is no undue discrepancy between the design and the model, then a *design error* has been exposed, and the verification procedure is concluded with a negative result. The design is said to be verified if and only if all properties have been verified with respect to a valid model.

The above entails that for systematic verification we must not only maintain different model abstractions, but also, in general, different versions of the (abstractions of) the verification model. It is clear that it can be quite costly and time consuming to have to restart an entire verification procedure at each model revision. There are essentially two ways to reduce such efforts. One is the improve the quality of the model(s) before verification by simulation. Although generally less thorough than model checking, simulation can be used effectively to get rid of the simpler category of modelling errors.

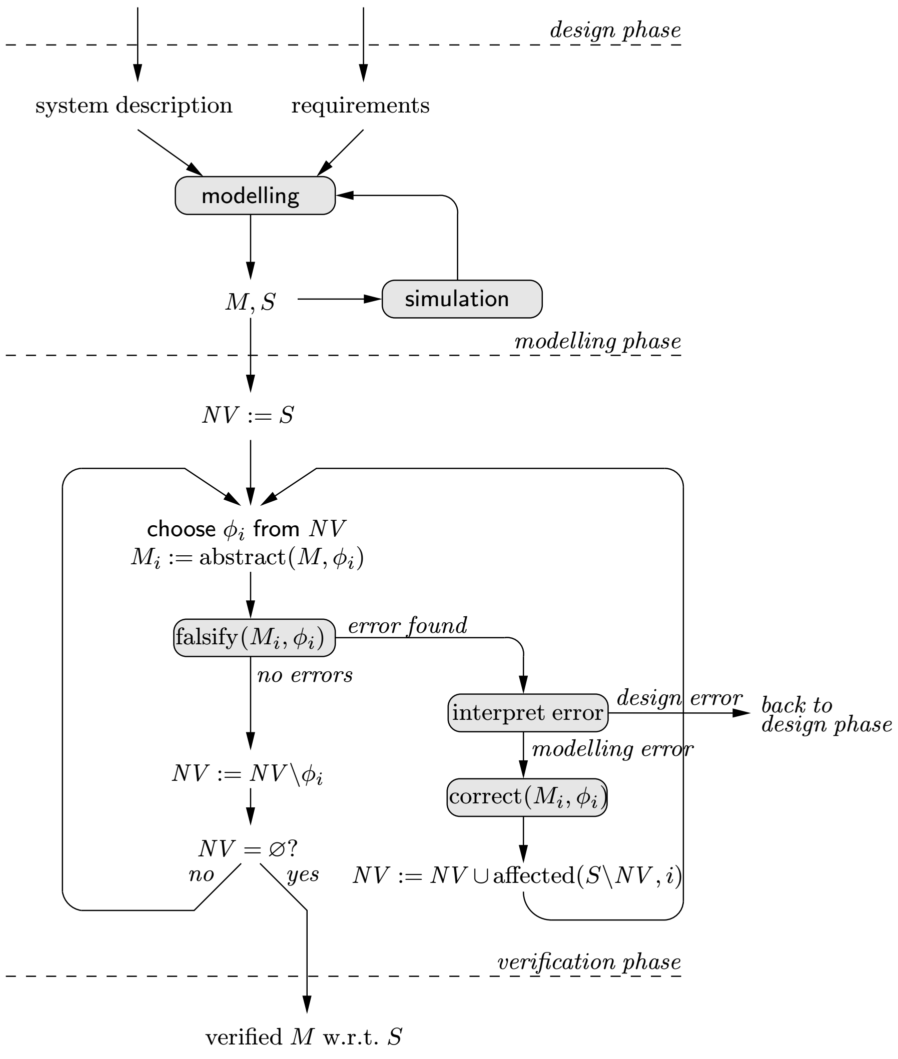


Fig. 1. Verification trajectory

The second improvement keeps track of the interdependencies of the different abstractions and their versions. If, for example, a model abstraction for one verified property abstracts away completely from the model revisions that are needed to repair a modelling error, then obviously the existing verification of such a property retains its validity. Using such information one needs to reverify only those properties whose corresponding models have been invalidated by a model revision.

It is hard to produce a complete characterisation of such interdependencies. For a good number of properties independence may be established on grounds of locality, e.g., they depend only on different sets of concurrent components. Compositional model checking techniques [1, 2] can be used to establish such minimal dependency sets.

It is clear that the many abstractions and versions of the verification model that may result from the processes and considerations described above need to be documented and maintained very carefully to manage a practical verification problem, which requires notational and tool support considerably beyond the core functionality of current model checking tools.

In Fig. 1 we have indicated the algorithmic structure of the problem in the style of a flow-chart. Here, we assume a given design phase that provides us with a (formal or informal) system description and a list of requirements. Both are subsequently formalized in a modelling phase, yielding an operational model M and a set of formal requirements S. A feedback loop based on simulation of M (guided by S) can be used to improve the formalisation in an approximative manner. When the quality of M and S is felt to be acceptable the verification phase is entered.

In this phase a set NV of non-verified requirements is maintained, initialised with S. Iteratively, for each property ϕ_i in NV an adequate abstraction M_i of M with respect to ϕ_i is produced and subsequently verified for ϕ_i. If this succeeds for all ϕ_i in NV then the verification phase is exited with a positive result. If, however, the verification of M with respect to ϕ_i fails then the error information is analysed. If the error is found to be a modelling error, then M_i (and M) is corrected and NV is updated to the union of the unverified requirements and all requirements whose verification is invalidated by the model correction. The verification loop is then restarted on the basis of this new set NV. If the failure of a verification of M with respect to ϕ_i is due to a design error, then the verification phase is exited with a negative result. In Fig. 1 we assume that the properties ϕ_i are correct; in Sect. 3.2 we will explain how to deal with erroneous properties ϕ_i.

It is easy to imagine how the verification trajectory could be incorporated in an adaptive design strategy, where negative verification results result in design modifications, after which new modelling and verification phases are begun. In this paper, however, we want to concentrate on the already sufficiently complicated management of the a posteriori verification of designs. In the next sections we will propose notational and structural means for the version management of the models M and their abstractions, and maintenance of the set NV of non-verified requirements.

3 Verification management

In this section we discuss the activities of the verification phase of Fig. 1 in greater detail. As already mentioned in Sect. 2, one of the difficulties of using model checkers "in the large" is the management of all (generated) data during the verification trajectory. It is indisputable that the verification results obtained using a verification tool should always be reproducible [22]. Without tool support, the verification engineer has to resort to general engineering practices and record all verification activities into a log-book. Consequently, the quality of the verification process depends on the accuracy of the validation engineer. This is clearly undesirable. The careful recording of information on the different models during the verification phase becomes even more indispensable when errors are found in one of the verification models. Apart from the fact that the erroneous models have to be corrected and reverified, all models that have been verified previously and which are affected by the error should be reverified as well. So-called *Software Configuration Management* (SCM) systems are needed to control these problems concerning the *versioned product space* of the verification phase.

In Sect. 3.1 the reader is introduced to the concepts of SCM. After this introduction, all *objects* that are significant during the verification phase are defined in Sect. 3.2. In Sect. 3.3, a reverification procedure is proposed which ensures that previous versions of the model get reverified when an error in the model is found and fixed. Section 3.4 discusses how the verification activities could be controlled using an SCM system.

3.1 Software configuration management

Software configuration management [3, 27] is the software engineering discipline of managing the evolution of large and complex software systems [51]. SCM is the process of identifying and defining the items in a system, controlling the release and change of these items throughout the life-cycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items [26]. The importance of SCM has widely been recognized [41] as reflected in particular in the capability maturity model (CMM) developed by the Software Engineering Institute [39]. Conradi and Westfechtel [11] give an extensive overview of the current state of art of SCM and SCM systems.

A SCM system should be supported by automated tools. Tools for version control and build-management (i.e., the process of automatically building software components, e.g., the tool `make` [17]) are essential. Furthermore, SCM tools should provide the developer with a "sandbox" environment [32]: a consistent, flexible, and reproducible environment to compile, edit, and debug software. SCM tools have evolved significantly over the last 20 years. Tools have gone from file-oriented versioning utilities to full-blown repository-based systems that manage projects and support team development environments, even across geographic locations. Leblang and Levine [32] present a practical view of SCM and offer some guidelines for selecting an SCM tool. In this paper, a discussion on particular SCM tools is clearly beyond our scope. The "Configuration Management Yellow Pages" on Internet [53] give a detailed overview of the available SCM tools.

Definitions. Below we define the conceptual framework for the rest of this paper, borrowing terminology from the SCM community, in particular [10, 11, 51].

– *Object.* An object (or item) is any kind of identifiable entity put under SCM control.
– *Version.* A version represents a state of an evolving object.
– *Revision.* A version intended to supersede its predecessor is called a revision (historical versioning).
– *Variant.* Versions intended to coexist are called variants (parallel versioning).
– *Configuration.* A configuration is a consistent and complete version of a composite object, i.e., a set of object versions and their relationships.
– *Product space.* The product space is composed of the objects and their relationships. The product space is organized by relationships between objects, e.g., composition relationships and (build) dependency relationships.
– *Version space.* The version space is composed of the versions (i.e., revisions and variants) and their relationships (e.g., successor, offspring, merge, etc). The version space is often organized into a version graph or version grid.

AND/OR graphs. AND/OR graphs [49] provide a general model for integrating product space and version space. An AND/OR graph is a directed, acyclic graph in which each node is either a leaf, an AND node, or an OR node. Analogously, a distinction is made between AND and OR edges, which originate from AND and OR nodes, respectively.

– *Leaf node.* A leaf node corresponds with a primitive object and represent program modules, documentation data, test data, etc.
– *OR node.* An OR node represents a *versioned object.* An OR node implies a choice. One may choose one of its successors: the versions of the object.
– *AND nodes.* An AND node represents a *configuration.* All successors of an AND node must be combined to form a complete configuration.

AND edges are used to represent both the composition of configurations and dependency relationships [11]. Figure 2 shows an example – taken from [11] – of a product graph for a configuration called `foo`. In this example the product structure is selected first; subsequently, versions of components are selected. Other selection orders (e.g., version first or intertwined) are also quite common [10, 11]. The grey nodes of Fig. 2 define a complete configuration, i.e., a particular version of the product `foo`. In the rest of the paper we will use AND/OR graphs to represent relationships between products.

In the following section we describe the product space and version space of a typical verification trajectory using a model checker.

3.2 Product space

Model checking is the process of checking that a model of a system satisfies a certain property. Consequently, the verification activities are centralized around the *verification objects* 'model' and 'property'. In software development, a software object records the data associated with a development or maintenance activity [11]. In the verification phase, a *verification object* records the data associated with a verification activity.

M – Model. From the modelling phase two products are obtained: a model M and a set of properties S which the

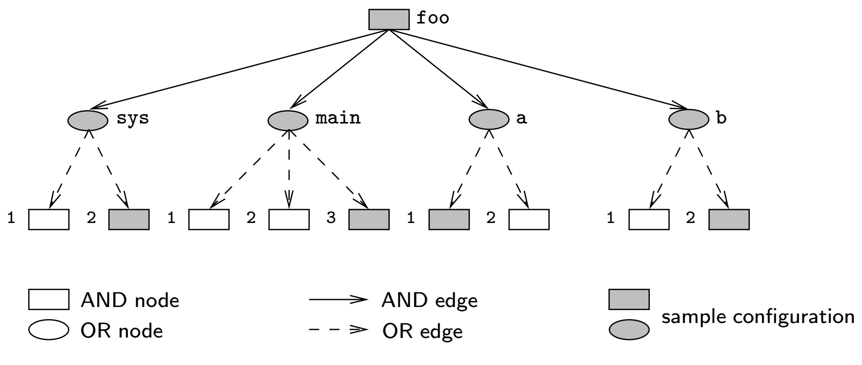


Fig. 2. Example of a product graph using AND and OR nodes [11]

model must satisfy (see Fig. 1). The purpose of the verification phase is to prove that the model M is correct with respect to the set of properties S.

In most verification projects that we know of, the verification has been carried using a single model checker. In such cases, the model M is conveniently represented directly in the specification language of the model checker. Limiting the verification to a single model checker, however, restricts the type of properties that can be checked during the verification phase. This is because – due to the nature of the infamous state space explosion – current model checkers are highly specialized towards the verification of a particular system domain (e.g., hardware vs software systems) or towards a certain type of properties (e.g., functional properties, timing properties). The model checker Spin [21, 23], for example, is optimized towards the functional analysis of communicating processes, i.e., the analysis of protocols. Uppaal [31], on the other hand, is a tool box for the verification and validation of real-time systems.

If the set S contains a diverse set of properties, it will not suffice to use a single verification tool for all properties. Consequently, in such cases, the description language of a single model checker will not be expressive enough to model the system for verification of all properties S. The model M may then be encoded in a hybrid language of the model checkers involved in the verification or in a more general specification language. See [13] for an example of a verification, where two tools were used to verify a simplified version of Philips' Bounded Retransmission Protocol [20]. Spin has been used to verify the functional properties whereas Uppaal has been used to check the timing properties of the protocol.

In the following, we assume that M is a model of the system under verification. The model M is either represented in the description language of the model checker or it is easily translated (i.e., abstracted) into the modelling language of the verification tools.

P^j – *Part.* In general, the detailed model M of a system is not suitable as input for a model checker. The reason for this is that the state space of the model M is generally much too large to be verified exhaustively by any model checker. Although much promising research has been and is being conducted to tackle the state space explosion (e.g., [19, 21, 37]), the verification engineer has to invest substantial effort as well. In order to reduce the state space of the model he or she has to make *abstractions* of the model. To structure the abstraction process and the verification process it is profitable to decompose the model M into a set of *parts*. Decomposing the model M into parts may serve the following goals:

- Controlling the complexity of the model M;
- Guiding the abstraction of the model M with respect to the properties under verification;
- In case of an error found in M, ensuring that only those properties get reverified which are affected by the error.

The last goal may need some explanation. By the nature of the model checking process itself, a single verification run may be both space and time consuming. Therefore, one should strive to reduce the number of model checking runs in the verification trajectory. Using the information contained in the decomposition of the model M and the subsequent abstractions of the model M in the verification phase, it may be possible to reduce the number of reverification runs. This will be discussed in greater detail in Sect. 3.3.
We assume that a model M is decomposed into m parts P^js.

$$M = P^1 \otimes P^2 \otimes ... \otimes P^m \tag{1}$$

The $\otimes$ operator in equation (1) does not have a fixed semantics. It just specifies the decomposition of M into parts $P^1 \dots P^m$. In general, it is not clear which decomposition is best suited for the verification of a model M. The experience of the verification engineer and the properties that have to be checked should guide the decomposition process. Some general guidelines for the decomposition are:

- As a general rule of thumb, one should strive to decompose the model into parts which have the same level of abstraction.
- When the system under verification is modelled as a set of parallel communicating processes, it is best to decompose the model M into processes $P^1 \dots P^m$ and let $\otimes$ indicate the parallelism of the processes.
- If the model M is defined as a literate document [44, 45], the structuring of the model into chunks may guide the decomposition into parts.

In the following, we assume that there exists a fixed decomposition of model M into parts $P^1 \dots P^m$.

ϕ_i – *Property.* A *property* ϕ_i is a property which should hold for the model M under verification. Most properties will be derived (implicitly or explicitly) from the system requirements that have been captured during the initial design phase. Other properties may have been identified during the modelling phase. Although the property itself may be specified in any language – even in natural language – to serve as the input to a model checker, a property should be translated into the property language of the model checker. Most model checkers use a form of modal logic for specification of the correctness property.

Naturally, the abstractions of the model M should be guided by the correctness property ϕ_i that is to be verified.

$$M_i = \text{abstract}(M, \phi_i) \tag{2}$$

In equation (2), M_i represents the abstract version of the model M with respect to the property ϕ_i that is to be verified. The function abstract represents the abstraction activity to arrive at this model M_i. The dependency relation between M, the properties ϕ_i and the corresponding models M_i is depicted in Fig. 3 using an AND/OR graph.

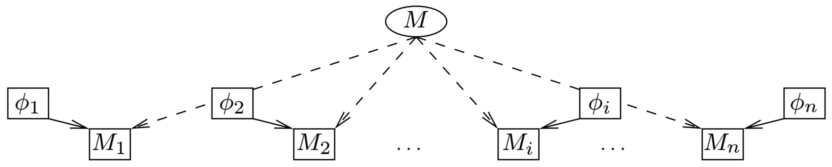


Fig. 3. Dependency tree between the model M, the properties ϕ_i, and the corresponding abstract models M_i

R_i – Verification data. A single verification run of a model checker involves several pieces of information, that all have to be recorded. First of all, the outcome of the run itself has to be logged. If the property ϕ_i holds for the model M_i, this outcome may be nothing more than "Yes, the model M_i satisfies property ϕ_i". There may be cases – when the state space of the model M_i is still too large to be verified exhaustively – that we are satisfied with the answer "No errors found, n% of the state space visited". Furthermore, the proof of a property ϕ may sometimes be given by a trace that leads to the state that we are interested in. The situation where the model checker concludes that a model M_i does not satisfy the property ϕ_i is discussed later in this section. Apart from the outcome of the various runs, other verification data includes:

– *Options.* Current model checkers provide the user with an extensive set of options and directives to optimize and tune the functionality and performance of the verification run. To reproduce a verification run of a model checker all these options should be recorded.
– *Statistics.* Furthermore, the nature of verification tools is that they either need a lot of processing time or need a lot of memory or even both. Statistics on such properties of verification runs are valuable attributes of the verification trajectory.

All information on a particular verification run should be regarded as a *verification object* as well.

$$R_i = \mathrm{run}(\mathrm{mc}_i, M_i, \phi_i) \tag{3}$$

R_i captures all verification data of the verification run of the model checker mc_i on the model M_i against the property ϕ_i. Although the model checker may not be fixed during the complete verification trajectory, we will not make the distinction between different model checkers in the rest of this paper. We consider the particular model checker mc_i to be part of the model M_i, implicitly. In Fig. 4 the verification data R_i is added to the dependency graph of Fig. 3.

Abstraction of parts. In this section we investigate the relation between a particular model M_i and the parts P^j. In Sect. 3.3 this relation will be used to relate the various models M_i's.

In equation (1), we proposed the decomposition of the model M into parts $P^1 \ldots P^m$. This decomposition is orthogonal to the abstraction of the model M into abstract models M_i as specified in equation (2). From equation (1) and equation (2) it follows that

$$M_i = \mathrm{abstract}(\otimes_{k=1}^{n} P^k, \phi_i)\,. \tag{4}$$

In general, the function abstract does not distribute over the parts P^j. In other words, it is not always possible to construct the abstract model M_i by simply making abstractions of all the parts of M. For example, if the model M is defined as a set of communicating processes, a typical abstraction is to combine several processes into a single one.

Even so, to construct a model M_i, one typically makes abstractions of parts P^j. In doing so, one often reuses abstractions that worked for previous models. To control the verification trajectory, the set of all abstractions of parts must be managed. For this purpose, we introduce the notion of *abstraction graph*. An abstraction graph contains all parts of a model M and its abstractions as used in the verification phase.

The bottom half of Fig. 5 shows an example of the abstraction graph of a model M. The abstracted parts P are indexed by a string of Greek letters, called the *abstraction identifier*. A dotted arrow from P^j_ψ to P^j_ω means that P^j_ω is a direct abstraction of P^j_ψ. The abstraction identifier ψ of a father P^j_ψ is always the prefix of the abstraction identifiers of its direct (more abstract) sons. For the first abstraction of P^j_ψ, an α is added to the abstraction identifier ψ, for the second abstraction a β, etc. The bottom half of Fig. 5 also shows that an abstraction can be obtained by merging two different parts: $P^{\{k,l\}}_\alpha$ is an abstraction of P^k and P^l. Note that the model M in the abstraction

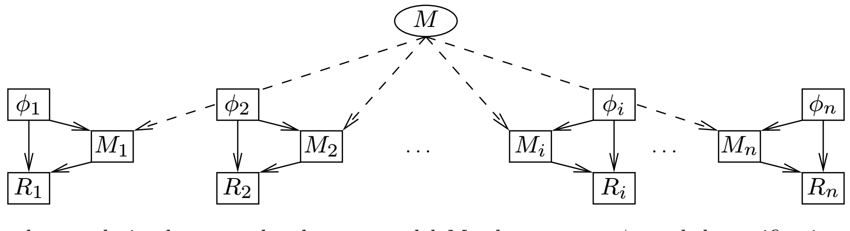


Fig. 4. Dependency relation between the abstract model M_i, the property ϕ_i, and the verification data R_i

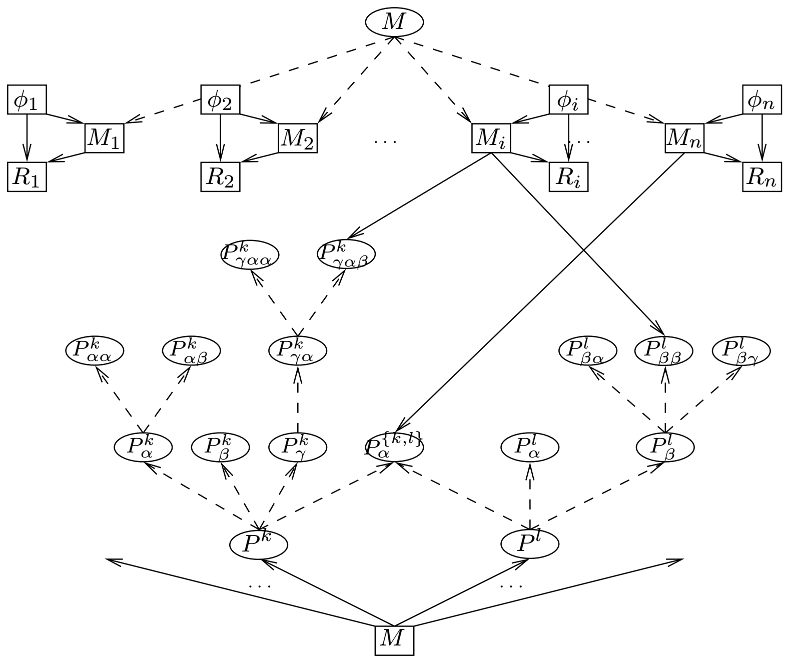


Fig. 5. Part of the product space of a verification process

graph is an AND-node in the sense that it is composed out of all parts $P^1 \dots P^m$. All abstracted parts P^j_ψ are OR-nodes.

Summarizing, we have identified the following verification objects: the model M, a part P^j, a property ϕ_i, the corresponding abstract model M_i and the verification data R_i. Furthermore, we have distinguished the abstracted parts P^j_ω – which are revisions of P^j.

Figure 5 shows an example of a dependency graph containing some verification objects of the product space of a particular verification project. The model M occurs twice in the graph. At the top M is the origin of the abstract models M_i whereas at the bottom it is the parent of all abstracted parts P^j. In the example of Fig. 5, the model M_i contains the parts $P^k_{\gamma\alpha\beta}$ and $P^l_{\beta\beta}$, whereas the model M_n contains the part $P^{\{k,l\}}_\alpha$.

3.3 Reverification

Apart from the verification objects discussed in the previous section, another dimension can be distinguished in the verification process: *versions*. In this section we will see that several versions of the same verification object may exist during the verification phase.

Errors. So far we assumed that during the verification phase no errors are being found: every abstraction M_i is proven correct with respect to its corresponding property ϕ_i. It is, however, common practice that instead of proving the absence of errors, a model checker will detect errors in the model. When an error is found by a model checker, the error can be classified into four types:

– *Property error*: there is not an error in the model or the system itself, but the property ϕ_i is incorrectly stated;
– *Abstraction error*: M_i is not a correct abstraction of M;
– *Modelling error*: M_i is a correct abstraction of M, but M itself is not a correct representation of the system;
– *Design error*: M_i is a correct abstraction of M and M is a correct representation of the design of the system, but the design itself is found to be erroneous.

In most cases, the property error will be straightforward to handle. Only the erroneous property ϕ_i has to be corrected and the verification should be run again. However, as the abstraction M_i of M is guided by ϕ_i, there may be cases where the abstraction M_i should be corrected as well.

The other three types of errors have more implications. In all three cases the model M or its abstraction M_i contained an error. We still have to repair this error and reverify the corrected model against the property ϕ_i. However, the fix to the error may have effects on all abstractions M_k that have previously been verified. This means that all models M_k that are affected by the change to M_i should be reverified against their corresponding property ϕ_k. When a design error is exposed by the model checker, the error should be reported back to the design team. When the error in the design has been corrected, both the modelling and verification phase should be carried out again.

In the following we will discuss the consequences of errors on the various verification objects. We will show that for a controlled and reliable verification process, version management on the objects is essential.

```
1   procedure verify($M, S$)
2     $M$ : model to be verified
3     $S$ : set of properties to be checked
4     $C$ := {}
5     foreach $\phi_i$ in $S$ do
6       $M_{i,1}$ := abstract($M, \phi_i$)
7       $R_{i,1}$ := run($\mathrm{mc}_i, M_{i,1}, \phi_i$)
8       $outcome$ := interpret($R_{i,1}$)
9       while $outcome$ = 'error in property' do
10        correct($\phi_i$)
11        $R_{i,1}$ := run($\mathrm{mc}_i, M_{i,1}, \phi_i$)
12        $outcome$ := interpret($R_{i,1}$)
13      od
14      if $outcome$ = 'no errors'
15        $C$ := $C \cup \{\phi_i\}$
16      else if $outcome$ = 'abstraction error' $\vee$ $outcome$ = 'error in the model'
17        $C$ := $C \cup \{\phi_i\}$
18        $P_{error}$ := errorparts($M_{i,1}, R_{i,1}$)
19        $E$ := affected($C, P_{error}$)
20        reverify($E, C$)
21      else if $outcome$ = 'design error'
22        go back to the design phase to correct the error
23        exit
24      fi
25    od
26  end verify
```

Fig. 6. Global verification pseudo-algorithm

verify. Figure 6 presents a detailed pseudo-algorithm for the verification phase as introduced informally in Fig. 1. The input for verify is the model M to be verified and the set S of properties to be checked. In line 4, the set C is assigned the empty set. The set C is the set of properties that have been verified at least once.[1] In the foreach-loop of lines 5–25, all properties ϕ_i are systematically checked. In line 6, the first version of model M_i is constructed using the abstract function. We will use a ",j" suffix to a subscript of a verification object to identify its jth version. The corresponding verification results are obtained in line 7. If the property ϕ_i turned out to be incorrect, the property ϕ_i is corrected and reverified in the loop of lines 9–13. If the property ϕ_i holds for the model M_i, this ϕ_i is added to C. If the verification reveals an 'abstraction error' or an 'error in the model', the error should be corrected as well. However, the error found may affect the correctness of previous verification runs. In line 18, the parts P_{error} of $M_{i,1}$ that caused the error are isolated. In line 19, the function affected calculates the properties (i.e., the corresponding models M_j) which are affected by the error (i.e., using the set P_{error}). All affected properties E are reverified in line 20. If the verification reveals an 'design error' (line 21), one has to re-enter the design phase and the verification phase is terminated.

reverify. Figure 7 lists the pseudo-algorithm for reverify, the reverification process. The procedure reverify systematically verifies all properties E_0 that are affected by the last error(s). We assume, that all versions of the verification objects are globally available to this procedure. The structure of reverify resembles the structure of verify, but there are a few apparent differences. To reverify a property ϕ_s, in line 5, the latest version t of the model M_s is retrieved. In line 6, the (t+1)th version of model M_s is constructed. This new, corrected version of M_s is subsequently verified in line 8. Again, the outcome of the verification needs interpretation. As long as only errors in the property ϕ_s are found, the model $M_{s,(t+1)}$ should be adjusted and reverified (lines 10–14). If the new version $M_{s,(t+1)}$ is still correct with respect to property ϕ_s (lines 15 and 16), the property ϕ_s can be removed from E_0. If the verification reveals an abstraction error or an error in the model, however, this error may affect all properties which have previously been verified. The erroneous parts P_{error} are isolated in line 18. The set of affected parts E_1 is obtained in line 19. The set of properties left to be verified now consists of $E_0 \cup E_1$ and the procedure reverify is recursively called in line 20 with this augmented set.

affected. The function affected in Fig. 8 returns a set of properties, whose corresponding models might have been affected by the erroneous parts P_{error}. The safe but naive approach is to return the complete set C without looking at the parts P_{error}. The drawback of this approach is that properties will get reverified that have no relation with the parts P_{error}. In affected, we will use the abstraction graph of the model M to only return the properties that are really affected by P_{error}. The body of affected is quite straightforward. For each property ϕ_s

[1] Note that in the flow chart of Fig. 1 a set of properties *left to be verified* (i.e., NV) is maintained, whereas in Fig. 6, the set of properties *already verified* (i.e., C) is used.

```
procedure reverify($E_0, C$)
    $E_0$ : set of properties that are affected by the last error(s)
    $C$ : set of properties that have been verified at least once
    foreach $\phi_s$ in $E_0$ do
        $M_{s,t}$ := retrieve latest version of $M_s$
        $M_{s,(t+1)}$ := construct $(t+1)$-th version of model $M_s$ by correcting
            the affected parts of $M_{s,t}$ w.r.t. to the last errors found
        $R_{s,(t+1)}$ := run($\mathrm{mc}_s, M_{s,(t+1)}, \phi_s$)
        $outcome$ := interpret($R_{s,(t+1)}$)
        while $outcome$ = 'error in property' do
            correct($M_{s,(t+1)}, \phi_s$)
            $R_{s,(t+1)}$ := run($\mathrm{mc}_s, M_{s,(t+1)}, \phi_s$)
            $outcome$ := interpret($R_{s,(t+1)}$)
        od
        if $outcome$ = 'no errors'
            $E_0$ := $E_0 \backslash \phi_s$
        else if $outcome$ = 'abstraction error' $\vee$ $outcome$ = 'error in the model'
            $P_{error}$ := errorparts($M_{s,(t+1)}, R_{s,(t+1)}$)
            $E_1$ := affected($C, P_{error}$)
            reverify($E_0 \cup E_1, C$)
        else if $outcome$ = 'design error'
            go back to the design phase to correct the error
            exit
        fi
    od
end reverify
```

Fig. 7. Reverification pseudo-algorithm

```
function affected($C, P_{error}$) : set of properties
    $C$ : set of properties that have been verified at least once
    $P_{error}$ : set of parts that have been found to be erroneous
    $E$ := {}
    foreach $\phi_s$ in $C$ do
        $M_{s,t}$ := retrieve latest version of $M_s$
        foreach $P^k_\omega$ in $M_{s,t}$ do
            foreach $P^e_\psi$ in $P_{error}$ do
                if related($P^k_\omega, P^e_\psi$)
                    $E$ := $E \cup \{\phi_s\}$
                    break out to outer loop
                fi
            od
        od
    od
    return $E$
end affected
```

Fig. 8. Pseudo-algorithm to compute the set of 'affected' properties

that has been verified before, it checks whether the latest version of its model – $M_{s,t}$ – contains a part P^k_ω that is related to a part P^e_ψ from P_{error}. If there is a relation, the property ϕ_s is added to the set of properties that are affected.

The real work of affected is delegated to the function related, which checks whether two abstract parts are related. The function related uses the abstraction graph of the model M to decide whether two parts are related. Five possible relations can be identified:

– P^e_ψ and P^k_ω are the same, i.e., $e = k \wedge \psi = \omega$.
– P^e_ψ is a direct or indirect abstraction (descendant) of P^k_ω: $(e = k \wedge \omega$ is a prefix of $\psi) \vee (k \in e)$.
– P^k_ω is a direct or indirect abstraction (descendant) of P^e_ψ: $(e = k \wedge \psi$ is a prefix of $\omega) \vee (e \in k)$.
– P^e_ω and P^k_ψ are related via a common ancestor, i.e., $(e = k \vee e \in k \vee k \in e) \wedge$ $(\exists \rho \bullet \rho$ is a prefix of $\omega \wedge \rho$ is a prefix of $\psi)$.
– P^e_ω and P^k_ψ are unrelated; they do not share the same ancestor P^j, i.e., $(e \neq k) \wedge (e \notin k) \wedge (k \notin e)$.

Without further inspection of P^e_ψ and P^k_ω, only for the last alternative it is certain that P^k_ω is not affected by the error in P^e_ψ.

3.4 Managing the verification phase

In Sect. 3.2 we discussed all verification objects of the verification phase. In Sect. 3.3 we showed the need to control the versions of these objects due to the reverification process. It is clear that the verification process does not only suffer from a state space explosion, but from a *versioned* *product space* explosion as well. To tackle this latter explosion, it is apparent that the verification process should be supported by an SCM system. Current model checking tools do not provide any SCM functionality. On the other hand, the functionality of full-blown state-of-the-art SCM tools seems excessive. When SCM policies and tools are already being used in the software engineering process of the system under verification, these SCM tools could be extended to control the verification activities as well. In either case, verification tools should be capable of interworking with (at least) basic SCM functionality.

For a controlled and reproducible verification phase, version control and build-management are most important. For this purpose, a file-based version control tool like RCS [50], CVS [5, 18] or PRCS [36] in combination with a basic build-management tool like `make` [17] might be sufficient. If the verification efforts are carried out by a team of people, there should also be solid support for teamwork. To ease the composition of a verification report [45] that summarises the verification activities the SCM tool should also include strong reporting facilities.

For a managed and efficient verification phase, it should be easy to navigate through the model of the system, the abstractions of the model, the relations between the models, and the properties and the verification results. Preferably, these navigation features should be integrated within the verification tool. Clearly, the SCM functionality should be a offered as a "sandbox" environment: SCM functionality should not hamper the verifier in the verification of the system.

4 Xspin/Project

In this section we briefly discuss Xspin/Project, an extension of Xspin for the management of validation data. Xspin/ Project has been developed to validate and support the ideas that we presented in the previous sections. Xspin/Project is discussed in greater detail in [43].

Xspin [23] is a graphical front-end to the verification tool Spin. With Xspin the user can edit, simulate, and verify models written in the specification language Promela. Xspin/ Project is an extension of Xspin [23] to *manage* the simulation and verification activities when using Xspin: every validation activity can be saved into a database of versioned verification objects.

Xspin/Project uses PRCS – the Project Revision Control System [36] – as its underlying SCM system. PRCS is a version-control system for collections of files with a simple operational model, a clean user interface and high performance. PRCS is freely available from [35]. The current version of PRCS is implemented using RCS [50] as its back-end storage mechanism. PRCS has some additional features which makes it well suited for integration into Xspin [43]:

– *Conceptually close to validation objects.* PRCS defines a project version as a labeled snapshot of a group of files, and provides operations on project versions as a whole. Thus, a project version naturally relates to a specific validation model and all its validation results.
– *Version naming scheme.* PRCS' version naming scheme corresponds closely to the version concepts from the validation framework: abstraction and revisions of these abstractions.
– *Simple operational model.* In PRCS, each project version is identified by a single distinguished file, the version descriptor; this file contains a description of the files included in that particular version. Adding files (i.e., validation results) only involves adding the filename to this version descriptor file.

Figure 9 shows the architecture of Xspin/Project. The Project-part of Xspin/Project is responsible for collecting the Promela models and simulation and verification results from Xspin and passing them to PRCS. Furthermore, the Project-part integrates a visual front end to PRCS into Xspin. The Project-extensions are written in Tcl/Tk [38, 54].

Each Promela model $M_{i,j}$ can be saved into the PRCS repository. Furthermore, the contents of any message box of Xspin which is the result of some simulation or verification run can be saved into the PRCS repository. Xspin/Project uses a special file, i.e., `description.log` in which it stores additional information – i.e., parts of the verification data $R_{i,j}$ – about the validation files (e.g., validation goals, options, directives, timestamps) into the current version of the project. In a nutshell the current version of Xspin/Project can be characterized as follows:

– Xspin/Project implements a visual front-end to PRCS into Xspin. To the user, Xspin/Project is presented as a conceptual database of Promela models together with their validation results.
– The user of Xspin/Project can save all its simulation and verification activities into the PRCS database.

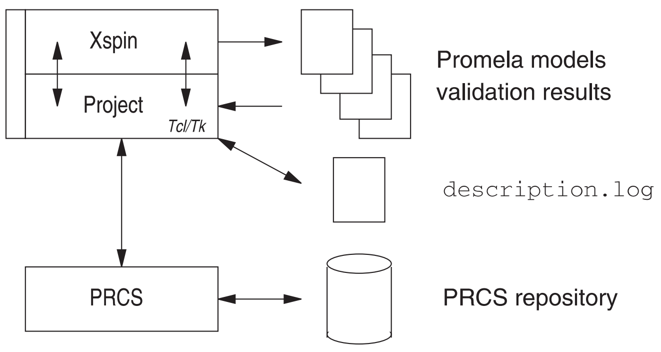


Fig. 9. Architecture of Xspin/Project

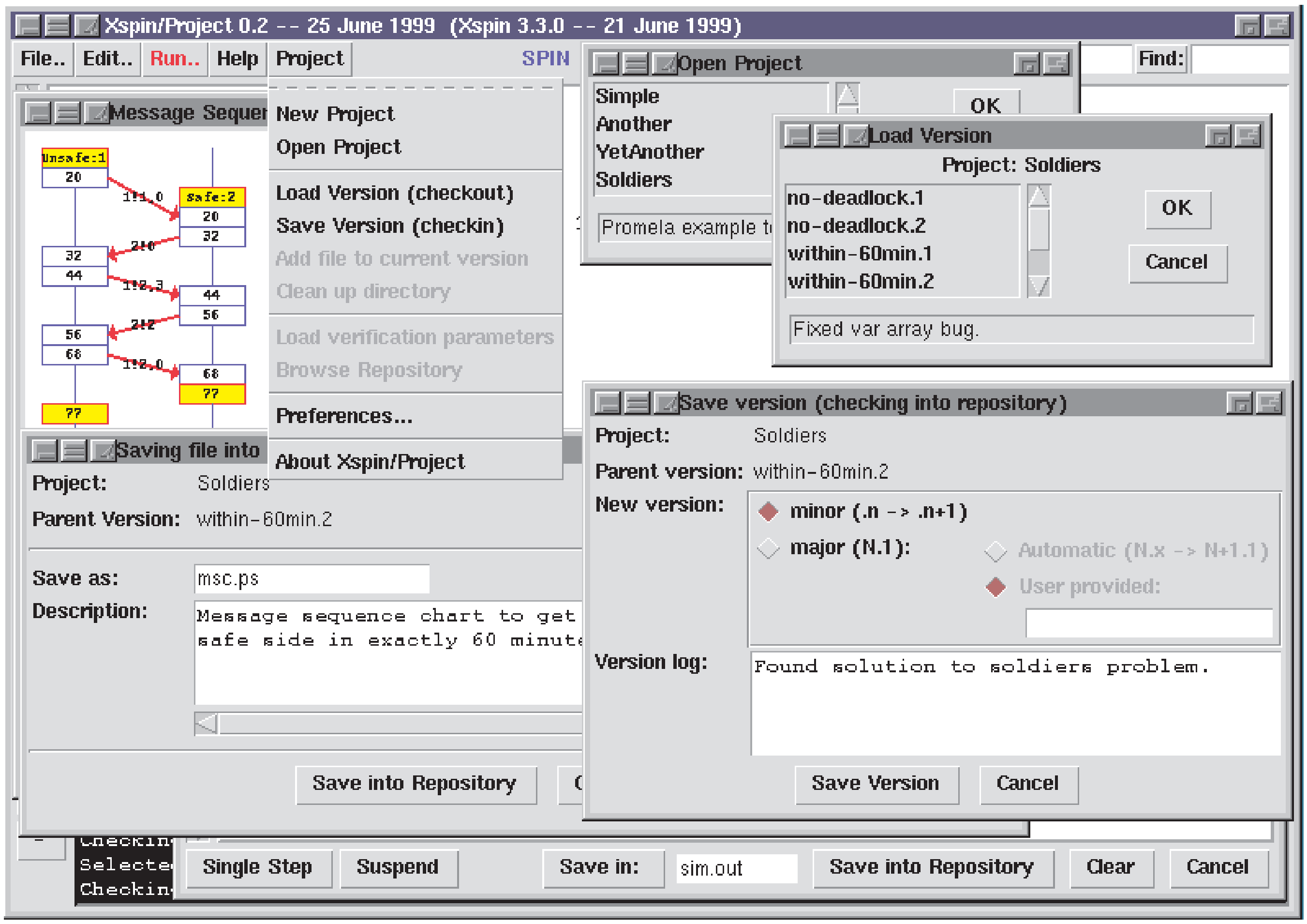
Furthermore, the user is given the possibility to annotate these validation activities.
– All essential verification data such as directives and options to the C compiler and the `pan` verifier are automatically saved into the PRCS repository.
– XSPIN/PROJECT ensures the integrity of the PROMELA models and their validation models.

XSPIN/PROJECT uses plain PRCS as its underlying configuration management tool. This means that all additional powerful features (such as `diff` and `merge`) of PRCS are also available to the user. However, these advanced features of PRCS are not (yet) available from within XSPIN/PROJECT. To exploit these features, one should use PRCS' command-line options.

In Fig. 10 a screenshot of a validation session with XSPIN/PROJECT has been captured. The added functionality of XSPIN/PROJECT provides the user with a consistent, flexible and reproducible environment to edit and validate PROMELA models. The user of XSPIN should not be unnecessarily hampered during the validation trajectory. Below, we discuss the user awareness with respect to the features added by XSPIN/PROJECT on top of the original XSPIN (see Fig. 10):

– *Accessing* PRCS. An extra top-level menu has been added to XSPIN: "Project". This menu is used to access most XSPIN/PROJECT functions, such as:
  – Starting a new project.
  – Opening an existing project.
  – Loading (checking out) a particular PROMELA model (i.e., an explicit version of the project).
  – Saving (checking in) a particular PROMELA model and all its recorded validation results.
  – Adding files explicitly to the current version. This may be useful when non-XSPIN files are relevant to a validation run or when one has forgotten to save a XSPIN file into the repository.
  – Cleaning up the directory. Using this function all files that have been saved previously in the repository are removed from the current directory.

– *Saving validation results.* To every dialog box containing validation output (e.g., simulation traces, message sequence charts) an extra button has been added: "Save into Repository". When pressing this button, XSPIN/PROJECT will show a dialog box where the user can annotate the particular file with some notes on the particular validation run. The



Promela example t

Load Version — Project: Soldiers
no-deadlock.1
no-deadlock.2
within-60min.1
within-60min.2
Fixed var array bug.

Saving file into
Project: Soldiers
Parent Version: within-60min.2
Save as: msc.ps
Description: Message sequence chart to get safe side in exactly 60 minute

Save version (checking into repository)
Project: Soldiers
Parent version: within-60min.2
New version: [✓ minor (.n -> .n+1)] major (N.1): Automatic (N.x -> N+1.1) User provided:
Version log: Found solution to soldiers problem.

Save in: sim.out

Fig. 10. Screen capture of a validation session with XSPIN/PROJECT

file and the (optional) notes are subsequently saved into the repository. Furthermore, for verification runs, XSPIN/PROJECT saves all options that are needed to build and run the `pan` verifier into the `description.log` file.
– *Forcing version integrity.* Modification of a PROMELA model is prohibited until all validation results on the model have been saved into the version repository.

XSPIN/PROJECT is a prototype tool to aid the validation engineer who uses XSPIN. With respect to the discussion on verification management in the first part of this paper, several features are not yet implemented. XSPIN/PROJECT does not yet support automatic reverification of PROMELA models, for instance. Even so, the current version of XSPIN/PROJECT already promises to be a great help in managing the version space explosion when using XSPIN.

5 Conclusions

In current research on automated verification, much effort is put into verification algorithms, whereas control and management issues have – at best – limited support. Currently, model checkers are being used as efficient debugging tools. As long as nasty errors are being exposed, this may be satisfactory enough. However, when one is aiming at the systematic verification of a system, one needs more than just a smart debugging tool.

This paper has discussed the practical problems of the verifier who uses a model checker to verify a (design of a) system. Apart from the inherent state space explosion of the model of the system, the verifier has to deal with the data explosion of the modelling phase and the versioned product space explosion of the verification phase. In Sect. 3 all objects of the verification phase and their relations were enumerated. A reverification pseudo-algorithm was presented to make sure that errors found in the model do not invalidate previous verification runs.

Verification tools should be supported by basic SCM functionality. A verification engineer should be able to navigate efficiently through the versioned product space to understand and manipulate the relations between the various objects. We have shown that limited SCM support can readily be added to (visual) model checking tools. The current version of XSPIN/ PROJECT presents the user with a conceptual database for PROMELA models and their validation results. Future additions to XSPIN/ PROJECT might include: (i) reporting facilities; (ii) automatic reverification; (iii) reuse of options and directives; and (iv) comparison of different models.

Future work to achieve a complete "systematic verification methodology" should include the definition of methods and policies to guide the modelling and verification phase. With respect to this at least the following lines of research seem interesting:

– *Modularisation and abstraction.* Kesten and Pnueli [29] describe the main tools of compositionality and abstraction in the framework of linear temporal logic. In Sect. 3 we have seen that abstraction is the key process of the verifier in the verification phase.
– *Compositional model checking.* The function related as described in Sect. 3.3 might be partly automated by using techniques from the area of compositional model checking [1, 34, 47].
– *Property patterns.* Dwyer et al. [15, 16] propose a pattern-based approach to the presentation, codification, and reuse of property specifications for finite-state verification. Their results should guide the verification engineer during the modelling phase in constructing the set of properties S which have to be verified.

References

1. Andersen, H.R.: Partial model checking (extended abstract). In: Proc. 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95), La Jolla, San Diego, Calif., USA, July 1995, IEEE Computer Society, pp. 398–407
2. Andersen, H.R., Staunstrup, J., Maretti, N.: Partial model checking with ROBDDs. In: Brinksma [6], pp. 35–49
3. Babich, W.A.: Software configuration management: coordination for team productivity. Addison-Wesley, Reading, Mass., USA, 1986
4. Bartlett, K., Scantlebury, R., Wilkinson, P.: A note on reliable full-duplex transmission over half-duplex lines. Comm ACM 12(5):260–265, 1969
5. Berliner, B.: CVS II: Parallelizing software development. In: Proc. Winter 1990 USENIX Conference, January 22–26, 1990, Washington D.C., USA, (Berkeley, Calif., USA, Jan. 1990), USENIX, pp. 341–352
6. Brinksma, E.: Proc. 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), University of Twente, Enschede, The Netherlands, April 1997, Lecture Notes in Computer Science, vol. 1217. Springer, Berlin Heidelberg New York, 1997
7. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans Comput 35(8):677–691, 1986
8. Bryant, R.E.: Symbolic Boolean manipulation with ordered Binary-Decision Diagrams. ACM Comput Surv 24(3):293–318, 1992
9. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. Inf Comput 98(2):142–170, 1992
10. Conradi, R., Westfechtel, B.: Configuring versioned software products. In: Sommerville, I., (ed.) Proc. ICSE'96 SCM-6 Workshop on Software Configuration Management (SCM'96), Berlin, Germany, March 1996, Lecture Notes in Computer Science, vol. 1167. Springer, Berlin Heidelberg New York, 1996, pp. 88–109
11. Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Comput Surv 30(2):232–282, 1998
12. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory efficient algorithms for the verification of temporal properties. Formal Methods Syst Design 1:275–288, 1992
13. D'Argenio, P.R., Katoen, J.-P., Ruys, T.C., Tretmans, G.J.: The Bounded Retransmission Protocol must be on time! In: Brinksma [6], pp. 416–431
14. Dijkstra, E.W.: Hierarchial ordering of sequential processes. Acta Inf 1(2):115–138, 1971
15. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Ardis, M., (ed.), Proc. 2nd Workshop on Formal Methods in Software Practice, Clearwater Beach, Fla., USA, March 1998, pp. 7–15

16. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proc. 1999 International Conference on Software Engineering (ICSE'99), Los Angeles, Calif., USA, May 1999, ACM, New York, pp. 411–420
17. Feldman, S.I.: Make – a program for maintaining computer programs. Software Pract Exper 9(3):255–265, 1979
18. Fogel, K.F.: Open source development with CVS. Coriolis, Scottsdale, Ariz., USA, 1999
19. Godefroid, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. Lecture Notes in Computer Science, vol. 1032. Springer, Berlin Heidelberg New York, 1996
20. Groote, J.F., van de Pol, J.: A bounded retransmission protocol for large data packets. In: Wirsing, M., Nivat, M., (eds.) Proc. 5th International Conference on Algebraic Methodology and Software Technology (AMAST'96), Munich, Germany, July 1996, Lecture Notes in Computer Science, vol. 1101. Springer, Berlin Heidelberg New York, 1996, pp. 536–550
21. Holzmann, G.J.: Design and validation of computer protocols. Prentice-Hall, Englewood Cliffs, N.J., USA, 1991
22. Holzmann, G.J.: The theory and practice of a formal method: NewCore. In: Proc. IFIP World Congress, Hamburg, Germany, August 1994. Also available from URL: http://cm.bell-labs.com/cm/cs/doc/94/index.html
23. Holzmann, G.J.: The model checker SPIN. IEEE Trans Software Eng 23(5):279–295, 1997. See also URL: http://netlib.bell-labs.com/netlib/spin/whatispin.html
24. Holzmann, G.J.: An analysis of bitstate hashing. Formal Methods Syst Design 13:289–307, 1998
25. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: Hogrefe, D., Leue, S., (eds.) Proc. 7th International Conference on Formal Description Techniques (FORTE'94), Bern, Switzerland, 1995, Chapman & Hill, London, pp. 197–209
26. IEEE.: IEEE Standard glossary of software engineering terminology: ANSI/IEEE Standard 729-1983. IEEE, New York, 1983
27. IEEE.: IEEE Guide to software configuration management: ANSI/IEEE Std 1042-1987. IEEE, New York, 1987
28. Kars, P.: The application of PROMELA and SPIN in the BOS project. In: Grégoire, J.-C., Holzmann, G.J., Peled, D.A., (eds.) Proc. SPIN96, 2nd International Workshop on SPIN (published as The Spin Verification System), Rutgers University, N.J., USA, August 1996, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 32. American Mathematical Society. Also available from URL: http://netlib.bell-labs.com/netlib/spin/ws96/Ka.ps.Z
29. Kesten, Y., Pnueli, A.: Modularization and abstraction: the keys to practical formal verification. In: Brim, L., Gruska, J., Zlatuska, J. (eds.) Proc. 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS'98), Brno, Czech Republic, August 1998, Lecture Notes in Computer Science, vol. 1450. Springer, Berlin Heidelberg New York, 1998, pp. 54–71
30. Knuth, D.E.: Literate programming. Comput J 27(2):97–111, 1984
31. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Int J Software Tools Technol Transfer 1(1/2):134–152, 1997
32. Leblang, D.B., Levine, P.H.: Software configuration management: why is it needed and what should it do? In: Estublier, J., (ed.) ICSE SCM-4 and SCM-5 Workshops – Selected Papers, Lecture Notes in Computer Science, vol. 1005. Springer, Berlin Heidelberg New York, 1998, pp. 53–60
33. Leveson, N.G., Stolzy, J.L.: Safety analysis using Petri nets. IEEE Trans Software Eng 13(3):384–397, 1987
34. Lind-Nielsen, J., Andersen, H.R., Behrmann, G., Hulgaard, H., Kristoffersen, K.J.: Verification of large state/event systems using compositionality and dependency analysis. In: Steffen [48], pp. 201–216
35. MacDonald, J.: PRCS – Project Revision Control System. Available from URL: http://www.xcf.berkeley.edu/~jmacd/prcs.html
36. MacDonald, J., Hilfinger, P.N., Semenzato, L.: PRCS: The project revision control system. In: Magnusson, B., (ed.) Proc. ECOOP'98 SCM-8 Symposium on Software Configuration Management (SCM'98), Brussels, Belgium, July 1998, Lecture Notes in Computer Science, vol. 1453. Springer, Berlin Heidelberg New York, 1998, pp. 33–45
37. McMillan, K.L.: Symbolic model checking: an approach to the state explosion problem. Kluwer, Boston, Mass., USA, 1993
38. Ousterhout, J.K.: Tcl and the Tk toolkit. Addison-Wesley, Reading, Mass., USA, 1994
39. Paulk, M.C., Weber, C.V., Curtis, B., Chrissis, M.B.: The capability maturity model: guidelines for improving the software process. Addison-Wesley, Reading, Mass., USA, 1995
40. Peled, D.: Combining partial order reductions with on-the-fly model checking. Formal Methods Syst Design 8:39–64, 1996
41. Pressman, R.S.: Software engineering – a practioner's approach, 4th edn. McGraw-Hill, New York, 1996
42. Ramsey, N.: Literate programming simplified. IEEE Software 11(5):97–105, 1994
43. Ruys, T.C.: XSPIN/PROJECT – integrated validation management for XSPIN. In: Dams, D.R., Gerth, R., Leue, S., Massink, M., (eds.) Theoretical and Practical Aspects of SPIN Model Checking. Proc. 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999 (5th) and Toulouse, France, September 21 and 24, 1999 (6th), Lecture Notes in Computer Science, vol. 1680. Springer, Berlin Heidelberg New York, 1999, pp. 108–119
44. Ruys, T.C.: Towards effective model checking. PhD thesis, University of Twente, Enschede, The Netherlands, March 2001
45. Ruys, T.C., Brinksma, E.: Experience with literate programming in the modelling and validation of systems. In: Steffen [48], pp. 393–408
46. Ruys, T.C., Langerak, R.: Validation of bosch' mobile communication network architecture with SPIN. In: Proc. of SPIN97, 3rd Int. 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
47. Staunstrup, J., Andersen, H.R., Hulgaard, H., Lind-Nielsen, J., Larsen, K.G., Behrmann, G., Kristoffersen, K., Skou, A., Leerberg, H., Theilgaard, N.B.: Practical verification of embedded software. IEEE Comput 3(5):68–75, 2000
48. Steffen, B.: Proc. 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), Lisbon, Portugal, April 1998, Lecture Notes in Computer Science, vol. 1384. Springer, Berlin Heidelberg New York, 1998
49. Tichy, W.F.: A data model for programming support environments and its applications. In: Schneider, H.-J., Wasserman, A. I., (eds.) Automated Tools for Information Systems Design – Proc. IFIP WG 8.1 Working Conference on Automated Tools for Information Systems Design and Development, New Orleans, La., USA, January 1982, North-Holland, Amsterdam, pp. 31–48
50. Tichy, W.F.: RCS – A system for version control. Software Pract Exper 15(7):637–654, 1985
51. Tichy, W.F.: Tools for software configuration management. In: Winkler, J., (ed.) Proc. International Workshop on Software Version and Configuration Control, Grassau, Germany, January 1988, Teubner, pp. 1–20
52. Valmari, A.: A stubborn attack on state explosion. Formal Methods Syst Design 1:297–322, 1992
53. van der Hoek, A.: Configuration management yellow pages. Available from: http://www.cs.colorado.edu in the file /users/andre/configuration_management.html, 1999
54. Welch, B.B.: Practical programming in Tcl and Tk, 2nd edn. Prentice-Hall, Englewood Cliffs, N.J., USA, 1997