

Behavior Protocols: Fighting the “Spearhead” Problem*

J. Kofroň

Charles University Prague, Faculty of Mathematics and Physics, Prague, Czech Republic.

Abstract. The state space explosion problem is the most burning problem of formal verification methods like model checking. In the behavior protocol checker, a tool for model checking of software components, various optimization of state representation were implemented thus speeding up the checking process. In some special cases, however, the optimization introduces a problem referred to as the “spearhead” problem causing some parts of the state space more than once. In this paper, we propose a solution to this problem and demonstrate the efficiency of improved version of behavior protocol checker.

1. Introduction

The automation of various processes tends to infiltrate more and more areas of our lives. Sometimes, a very high level of reliability of a computational system is required; in such cases the system should undergo formal verification to prove that it satisfies certain properties (usually absence of errors). Wide variety of properties can be proven using different verification technique. In case of a component based system, behavior protocols [1] can be used for component behavior specification and verification.

As in model checking [3], using behavior protocols has to deal with several of problems, among them the *state explosion problem* [3] being the most burning one. The state space explosion, i.e. enormous (exponential in the size of input) growth of the system state space, causes high time and memory requirements of these methods and hinders their everyday use as a method for finding bugs in software.

Besides the time requirements, problems lie also in keeping information about what parts of the state space have been already visited. One cannot profit from the use of secondary memory usage, such as disk, because the overall time needed to access this type of memory for storing state information exceeds all acceptable limits. The problem of state space explosion is inherent to model checking and contemporary computational systems are not powerful enough to successfully cope with it.

Sometimes, when trying to speed up the checking process as much as possible, the state space identifier optimizations [7] may introduce a secondary problem – state identifiers of the same state with two different execution histories may be different thus causing the following part of the state space to be explored twice (or even more times).

The goal of this paper is to describe the “spearhead” problem that appeared after new state identifiers were implemented in the behavior protocol checker [7] and to propose a solution for this problem.

1.1. Large State Space Traversal

In order to make the traversal of large state spaces possible, it is necessary to generate the state space *on-the-fly* [3] and not to generate it before the traversal. Using this strategy, the states and transitions interconnecting these states are not pregenerated at the beginning of the checking process, but each time (i.e. in a state) a transition (or another state) needs to be explored. The structure of visited part of the state space can be forgotten thus saving the memory. On the other hand, it is desirable to recognise the visited parts, to avoid visiting them more than once, therefore information about state visits has to be kept at all time during the checking process. Sometimes, the state space to be traversed may be so huge that even holding this kind of information within the memory is not feasible on contemporary systems. There are basically three options to solve this: (i) To “forget” information about visiting a part of the state space when the memory becomes full, (ii) use

* This work was partially supported by the Grant Agency of the Czech Republic project 201/05/H014.

approximate information about states, and (iii) use a symbolic state space representation. Although very fast and memory saving, the second option may cause excluding some parts of the state space from traversal thus lowering the reliability of the checking result [4]. The third option requires further research of applicability of these methods (e.g. OBDDs) in the scope of behavior protocols and currently we are not sure of their contributions in this scope. Thus, we decided to focus on the first option and introduced new state identifiers [7].

1.2. SOFA Behavior Protocols

SOFA behavior protocols [1] are a method for specification of component behavior. They were originally developed for description of SOFA components [5], but can be extended for behavior description in a wide variety of component models, e.g. Fractal/Julia [8]; they describe component behavior on the level of method calls and method call responses.

In SOFA, component behavior is described at the level of component frame – a black-box view that defines provided and required interfaces of the component. The architecture of a composed component describes its structure at the first level of nesting – the connections between subcomponents and the bindings of the component interfaces to the interfaces of the subcomponents.

The component behavior is defined by a set of possible traces, i.e. sequences of events (method calls and returns from methods) the component is allowed to perform. Since this set might be quite huge or even infinite, explicit enumeration of its members is impossible. As behavior protocols are expressions generating all traces allowed to be executed, they become a suitable specification platform. Besides standard regular-expression operators (e.g. “;” for sequencing, “+” for alternative and “*” for repetition), there are also special operators like and-parallel operator. The and-parallel operator generates an arbitrary interleaving of all events within its operands (e.g. “(a ; b) | (c ; d)” protocol generates six traces starting with ‘a’ or ‘c’ where ‘a’ precedes ‘b’ and ‘c’ precedes ‘d’).

As an example consider the following protocol:

```
(open ; (read + write)* ; close) | status*
```

This behavior protocol generates a set of traces starting with the `open` method and ending with the `close` method with an arbitrary sequence of repeating the `read` and `write` methods between them. The traces may be interleaved with an arbitrary number of the `status` method calls.

Behavior protocols allow for testing protocol compliance and absence of composition errors. Since the compliance (i.e. conformance of a component frame protocol with its architecture protocol) is realized as a composition test of the architecture protocol and the inverted frame protocol [6], only the checking for composition errors will be described in more detail.

The composition test allows for detection of three types of errors: *bad activity*, *no activity*, and *divergence* (sometimes also referred to as *infinite activity*). Bad activity denotes the situation when a call is emitted on a requires interface, but the component bound via its (provides) interface to this required interface is currently (according to its frame protocol) not able to accept that call. No activity denotes deadlock, i.e. the situation when (i) none of the components is able to perform any action and (ii) at least one component is not in one of its final states. Situation where the components input a loop without a possibility to reach a final state (i.e. there is no trace in the state space to a final state from any loop state) is called divergence.

Size of the state space generated by protocols describing behavior of a composed component tends to grow exponentially when using parallel composition, i.e. the and-parallel and consent operators.

Evaluating both compliance and composition relations requires exhaustive traversal of the state space. Coping with the state space explosion problem is therefore inevitable.

2. Behavior Protocol Checker

The behavior protocol checker is a tool performing compliance and composition tests of communicating components’ behavior described by behavior protocols. It uses *Parse Tree Automata* (PTA) [2] for the state space generation; PTAs are subject to various optimizations which reduce the size of the state space they generate. These optimizations involve *Multinodes* and *Explicit Subtree*

Automata [2]. Note that *forward cutting* [2] is not used any more since the consent operator is incompatible with this optimization.

The *multinodes* optimization is applied to the parse tree during its construction (when parsing the input protocol). It replaces repeating occurrences of the same protocol operator by a single occurrence of multi-version of the same operator, i.e. the sequence of $n-1$ occurrences of an operator is replaced by one n -ary version of the same operator.

The last optimization applied to the parse tree is the *explicit subtrees optimization*. This optimization converts some subtrees of the parse tree into an explicit automaton. This means that for this parse subtree the states are not generated on-the-fly, instead, they are generated before the state space is traversed. A subtree converted into an explicit automaton has to have a reasonable size to fit in the memory and its states have to be used more than once (e.g. in subtrees of an *and-parallel* operator) during the composition test, as it is necessary to traverse the whole state space generated by the subtree to construct the explicit automaton.

It is almost obvious that none of the optimization described above affect the language (the set of traces) generated by the PTA. Although the multinode and explicit tree optimizations do not reduce the state space, they increase the speed of checking in a significant way in the sense of higher transition generation speed.

2.1. The “Spearhead” Problem

As described in [7], the new state identifiers significantly speeded up the traversing of the state space. As an example, consider PTA on Figure 1 which corresponds to the example protocol from Section 1.2. The state identifier representing the initial state of the behavior protocol corresponding to this PTA has the form 001100000. The two leading zeros represent initial states of the primitive automata for $open^{\wedge}$ and $open^{\$}$, two following ones express that none of the branches of the alternative ($+$) operator has been selected so far, the following zero (redundant) represents the state of the alternative subtree (to simplify and speed up the identifier manipulation) and the last four zeros represent initial states of primitive automata for $close^{\wedge}$, $close^{\$}$, $status^{\wedge}$ and $status^{\$}$. Having the PTA, the information about type of inner nodes (except for the alternative operator) does not need to be stored inside an identifier. Similarly, the identifier of the final state of this protocol has the form 111101111 (neither the read nor write, unlike the status operation, were executed in the trace corresponding to this final state, thus the automaton for alternative subtree is still in its initial state – 110).

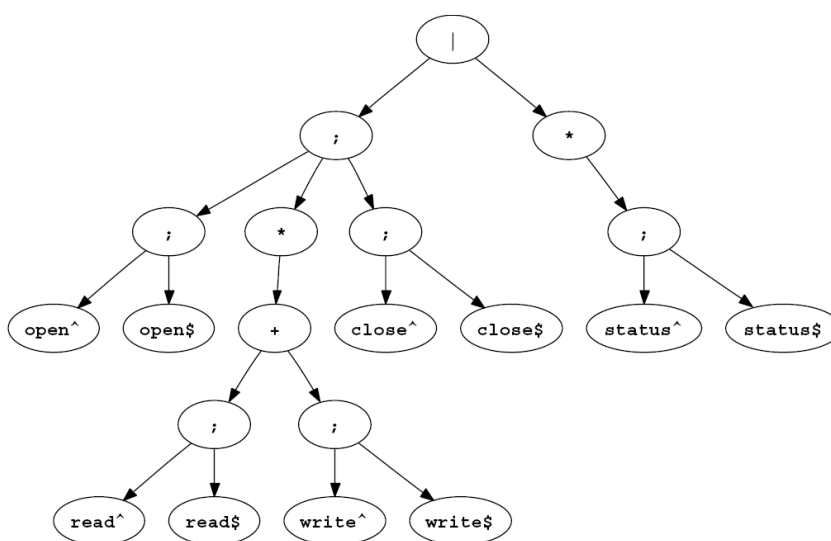


Figure 1: Protocol parse tree

Using this kind of state identifiers has introduced a problem we have not anticipated at the beginning – the “spearhead” problem. It stems from the fact that the identifiers are sometimes “aware” of the execution history, i.e. a state may be identified by two (or even more) different state identifiers depending on the actual path through which it has been reached. To make it clearer, consider the following example:

```
(open ; (read + write) ; close)
```

Here, the state after the execution of the sequence `open ; read` would be described by an identifier `11001100`, while the state after the execution of `open ; write` will be denoted by `11011100` (see Figure 2). Although identifying the same state, i.e. state before execution of the `close` method, these two identifiers are different and therefore considered to describe two different states. The state space, as seen by the behavior protocol checker, is depicted in Figure 2. As a consequence, the part of the state space modelling the execution of the `close` method is traversed two times although a single traversal would be sufficient.

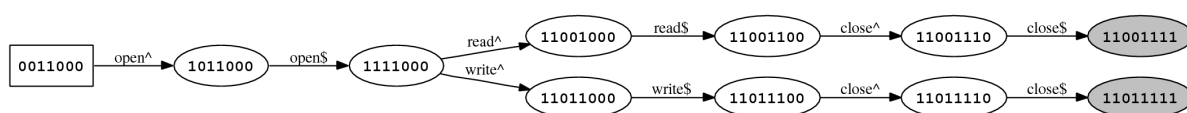


Figure 2: State space featuring the “spearhead” problem.

Although it may be not obvious, the “spearhead” problem may cause (and typically does) cause a significant growth of the state space– note that components are combined using a parallel composition operator (e.g. `consent`), thus the number of states generated by a protocol is multiplied by the number of states of the rest of the system. The resulting state space may be therefore even an order of magnitude larger than it is necessary to be.

2.2. Solution of the “Spearhead” Problem

The solution of the “spearhead” problem lies in the detection of the states that represent an “end” (there may be more than one final state) of an alternative branch and replace the “implicit” state identifier with a special one. Thus, all the final states of an alternative branch will merge into a single state. In the aforementioned example, the alternative final state would be the state before execution of the `close` method. The resulting state space is depicted in Figure 3 (here, without the `open` part, the state space would resemble the shape of a spear and the problem would lie in its left part, therefore we referred to this problem as to the “spearhead” problem).

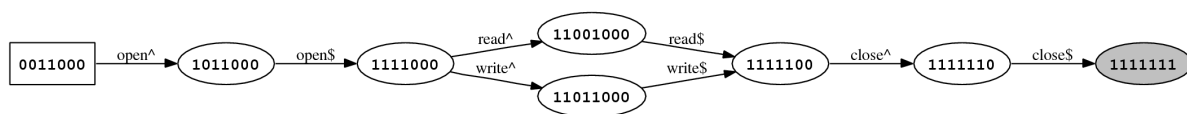


Figure 3: State space with the “spearhead” problem solved.

3. Evaluation

Solving the “spearhead” problem has significantly decreased the size of the state space typically traversed during static check for compliance and composition errors. Should the static test take about one hour before the algorithm addressing the spearhead problem was incorporated in the checker, after re-implementing the state identifiers the time decreased to less than ten minutes. Even though the problem of compliance and composition error checking still requires time exponential in the size of the input protocols, lowering the time requirements to one tenth of the time required by the previous version is a valuable result as it allows using the compliance test in real component design without a significant loss of time.

We have successfully applied the behavior protocol checker with the “spearhead” problem fixed to a real-life application [9] consisting of about twenty components. The duration of the static verification was less than four hours, which we consider reasonable time for a formal behavior verification of a real-life application.

4. Related Work

The “spearhead” problem appears also when verifying code (e.g. in Java) against a behavior protocol. When applying the approach of active code analyzer and passive code protocol checker [10], i.e., the protocol checker is only notified about the transitions present in the code checked, on the side of the code analyzer (e.g. Java PathFinder) it is necessary to record the call stack to be able to distinguish between two different traces, else some compatibility errors might not be discovered. This forces the model checker traverse each trace to the end, thus prolonging the checking process time. Here, the solution described above cannot be applied on the side of code analyzer as the history of calls is necessary for successful evaluation of the compliance relation.

5. Conclusion

In this paper, we have presented a solution to the “spearhead” problem causing unnecessary growth of the state space. After re-implementation of the state identifiers that cause the “spearhead” problem, the checking time requirements have been reduced to 10% – 20% of the original time, which we consider as significant improvement. With current version of the behavior protocol checker, we successfully checked the entire architecture of a real-life application [9] in less than four hours which we consider as a reasonable time for an application of such complexity.

References

1. Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002
2. Mach, M., Plasil, F., Kofron, J.: Behavior Protocol Verification: Fighting State Explosion, published in International Journal of Computer and Information Science, Vol.6, Number 1, ACIS, ISSN 1525-9293, pp. 22-30, Mar 2005
3. Edmund M. Clarke, Jr., Orna Grumberg, Doron A. Peled: Model checking, The MIT Press, 1999
4. Gerard J. Holzmann: The Spin model checker: Primer and Reference Manual, Addison-Wesley, 2003
5. SOFA Component Model, <http://sofa.objectweb.org>
6. Adamek, J., Plasil, F.: Component Composition Errors and Update Atomicity: Static Analysis, accepted for publication in the Journal of Software Maintenance and Evolution: Research and Practice, 2005
7. Kofron, J.: Performance Improvements of Behavior Protocol Checking, in Proceedings of WDS'05, Edited by Jana Safrankova, MATFYZPRESS, Prague, 2005, pp. 25-30, ISBN 80-86732-59-2, Czech Republic, Jun 2005
8. JULIA framework (fractal implementation). <http://fractal.objectweb.org>
9. Component Reliability Extensions for Fractal Component Model. http://kraken.cs.cas.cz/ft/public/public_index.phtml
10. Parizek, P., Plasil, F., Kofron, J.: Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker, Accepted for publication in Proceedings of 30th IEEE/NASA Software Engineering Workshop (SEW-30), Columbia, MD, USA, April 25-27, 2006, IEEE Computer Society Press, Apr 2006