Behavior Protocols Verification: Fighting State Explosion

Martin Mach[1], Frantisek Plasil[1,2], Jan Kofron[1]

Charles University, Prague
Academy of Sciences of the Czech Republic

Abstract

A typical problem formal verification faces is the size of the model of a system being verified. Even for a small system, the state space of the model tends to grow exponentially (state explosion). In this paper, we present a new representation of state spaces suitable for implementing operations upon behavior protocols of software components [1]. The proposed representation is linear in length of the source behavior protocol. By trading space for time, it allows handling behavior protocols of "practical size". As a proof of concept, two versions of a verification tool based on the proposed technique are discussed.

Keywords: Formal verification, software components, state explosion, behavior protocols, parse trees.

1. Introduction and motivation

The traditional verification techniques of program correctness are *testing* and *simulation*. However these techniques suffer from two major problems: (i) A working prototype is necessary for the verification, which inherently means belated error discovery within the development cycle. A remedy may require a major change in the program's architecture, which may be very costly in late design stages. (ii) It is usually hardly possible to test all the potential interactions with the program's environment so that some errors may remain undetected during the development, being discovered as late as by an end user.

Formal verification is a well-established method for correctness checking which can be employed during the whole program development cycle. The complete program is described via a mathematical model the properties of which can be verified with the assistance of verification tools.

[1]Faculty of Mathematics and Physics
Department of Software Engineering
Malostranske namesti 25, 118 00 Prague 1,
Czech Republic
{mach, plasil, kofron}@nenya.ms.mff.cuni.cz,
http://nenya.ms.mff.cuni.cz

[2]Academy of Sciences of the Czech Republic
Institute of Computer Science
Pod Vodarenskou vezi 2, 182 07 Prague 8,
Czech Republic
plasil@cs.cas.cz, http://www.cs.cas.cz

However, as forming of the actual model can be quite complicated, these tools are usually not easy to employ. Another important problem is that the representation of the state space associated with the model tends to exhaust all the memory available for a particular verification tool (the "state explosion" problem).

In this paper, we focus on formal models targeting behavior description of software components. In particular, we address the issue of efficient memory representation of the *behavior protocols* [1], which allows behavior compliance checking of cooperating components.

1.1. Components and behavior

Components are modern foundations of building software applications. Frequently understood as a design entity, a component *provides* some services to its environment and *requires* other services from the environment (other components). A service is usually described as an interface (and the methods in this interface). Therefore, in a typical component model, a component features both *provided* and *required* interfaces, like in Darwin [14] and Fractal [15].

In addition to defining interfaces at the syntax level, some of the component models partially capture also the semantics of components by specifying the desired/allowed sequences of method invocations (behavior of components). Such component models include Wright[5], Darwin[14], and SOFA[3]. In this paper, we focus on the behavior specification via behavior protocols [1] employed in SOFA, an open source component model [3].

1.2. Behavior protocols

A behavior protocol is a regular expression-based expression describing behavior at different levels of granularity (interface, interplay of all interfaces of a component, composition of several components). A behavior is a language over symbols that denote either the start or end of a method invocation (*events*). A behavior protocol features additional operators to enhance expressiveness. These additions do not break regularity of the languages described by behavior protocols. We provide only a basic overview of behavior protocols, for further reference we refer the reader to [1] and [4].

Syntax. The symbols denoting *events* are used to describe synchronous and asynchronous method invocations and have the following syntax:

```
(type, interface_name, event_name, flag)
```

where `type` indicates whether `event_name` determines a method invocation accepted on `interface_name` (?), emitted on `interface_name` (!), or it is an internal event taking place within a composed component (τ). Further, `flag` denotes whether the event is a method invocation request (↑) or response (↓). As an example, the acceptance of synchronous call invoking the method b on an interface a is expressed as `?a.b`↑ `; !a.b`↓.

Semantics. In addition to the operators defined for regular expressions, i.e. ; (sequencing), + (alternative), * (repetition), several new operators are added to handle restriction, parallelism, and composition. For the purpose of this paper, it is sufficient to mention the operator | (and-parallel) which produces an arbitrary interleaving of traces generated by its operands.

Example. Consider a component representing a file. It provides one interface that contains five methods to manipulate the file: `open`, `read`, `write`, `close`, and `status`. The supported behavior either (i) starts with calling `open`, then an arbitrary interleaving of `read` and `write` follows and finally `close` has to be called; or (ii) allows `status` to be called at anytime (in parallel with (i)). The corresponding behavior protocol takes the form (for simplicity we use shortcut `method_name` for `?method_name`↑`; !method_name`↓):

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

Compliance. Behavior protocols allow static testing of behavior compliance of tied components. This way questions like "Is it possible to safely replace a component by another one if we know their interfaces and behavior?" or "Is it possible to interconnect these two components if we know the behavior interplay on the provided and required interfaces of each of them?" can be answered. Basically, the components are compliant if they fulfill two conditions based on subset relations. The publication [1] describes the compliance concept thoroughly and also provides an algorithm of compliance verification.

State explosion. Basically, the state space associated with a behavior protocol is the state space of the finite automaton accepting the regular language generated the behavior protocol.

Above, we mentioned that formal verification has typically to cope with the state explosion problem. Also behavior protocols suffer from this problem, because the compliance is tested via the corresponding automata determined by the behavior protocols in question, since any parallel activity causes exponential growth of the state space. For example in the original SOFA verifier [3], the state space corresponding to an expression involving more than 13 parallel operators does not practically fit into the memory available for the verifier even on a decent PC.

1.3. Goals and structure of the paper

To target the problem mentioned above, we designed a novel automata representation, which significantly improves the efficiency of the compliance verifier. In the inherent space versus time tradeoff, it shifts the complexity towards time in such a way that it allows to solve practical problems at least twice as big as the original verifier could handle. The main goal of this paper is to present the basic idea of this novel representation and share with the reader the lessons we learned during experiments with the new verifiers.

The structure of the paper is following. In Section 2, we discuss the flaws of classical automata representations (Section 2.2), while the Sections 2.3 and 2.4 bring the core of the paper by introducing *parse tree automata* and their optimizations. In Section 3, we describe an experimental behavior protocol verifier based on parse tree automata and Section 4 describes an enhanced Java version of the verifier. In Section 5, we evaluate the proposed representation and compare it with other techniques addressing state explosion. Section 6 concludes the paper.

2. Behavior protocol representation

2.1. Representation and efficiency

Different representations of a state space corresponding to a behavior protocol (*expression* for short) have specific benefits and drawbacks. Such a situation makes any reasoning on the representation efficiency a complicated task.

To show the properties of different finite automata representations (*representation* for short), we have identified four criteria proved to be important for a successful choice of a particular representation. The chosen criteria are:

- *Size of representation* is the amount of the memory required to store a (state space) representation. This is determined by all the data structures involved.
- *Building time* is the time required to create the representation from an expression.
- *Space requirement of composed state identifiers* is the amount of memory required to identify the states in a state space.
- *Access time* is the average time needed to determine the list of transitions associated with a state.

2.2. Basic representation techniques

To illustrate how the evaluation criteria help (i) characterize different representation techniques and (ii) show trade-off between time and space complexity, we present an overview of two classical finite automata representation techniques.

Explicit representation is the most simple and straightforward technique to represent an automaton. All necessary information is *explicitly* held in memory – lists of states, transitions, and accepting states (as lists, hash tables, matrices, ...).

As to size of such representation, state explosion is very likely. Also building time is fairly low as the construction of a state space is usually done recursively by composing the state spaces of sub-expressions and as the whole state space has to be traversed during this construction.

On the other hand, explicit representations shine in access time and size of identifiers. Hardly anything can beat the usage of pointers in states identification and retrieving a list of transition from memory.

Size of a representation is the major drawback of explicit representation causing that verification tools avoid using it. As explained in [2], the original SOFA behavior protocol verifier uses this type of representation. States are implemented as Java objects holding lists of labeled references to other states.

Symbolic representation is a group of techniques that use a different approach. The required state space is not generated in advance as in explicit representations but it is rather computed on-the-fly. This approach brings two benefits in terms of fighting state explosion: (i) In most cases, very large numbers of states can be handled, and (ii) the unvisited portions of the space are not generated at all. However access time is slower than in explicit representation because several computations are needed to obtain a list of transitions. Also a state identifier is usually implemented via a composed data structure, hence consuming more memory than a state identifier in the explicit representation technique.

The most recognized member of the symbolic representation technique category is the Ordered Binary Decision Diagram (OBDD) [6] technique. An OBDD is an acyclic directed graph representing a Boolean function $f(x_1,\ldots,x_n) \rightarrow \{0, 1\}$. In this graph, the internal nodes correspond to functional arguments and the two possible terminal nodes correspond to the output of the function. The arguments appear in the same order on the path from the root to leaves (Fig. 1). However the size of an OBDD graph strongly depends on the order of the function arguments.

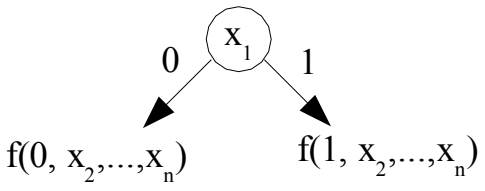

Fig. 1. Root of the decision diagram determining the function $f(x_1,\ldots,x_n)$

There are functions that are described by a graph of linear size for a specific argument ordering and of exponential size for a different ordering. And, unfortunately, deciding on an optimal ordering is an NP-complete problem [6].

To our knowledge, a precise evaluation of using OBDDs for representation of regular expressions has not been provided so far.

2.3. Parse trees and parse tree automata

To tackle the state explosion problem in representation of behavior protocols, we suggest and describe bellow *parse tree automata*, a novel symbolic representation technique.

Parse trees (also *syntax* or *expression trees*) are a common way to represent expressions in memory. They are mainly used to represent mathematic formulas and program source codes in compilers. Obviously, they are also capable to represent behavior protocols (Fig. 2).

A parse tree is a tree structure that describes a given expression unambiguously. When representing behavior protocols, the parse tree features the following important properties:

- Event symbols featuring in an expression appear only in the leaf nodes and operators in inner nodes of the corresponding parse tree.
- The operator nodes representing the repetition and restriction operators are unary; all others are binary.
- Every subtree describes an expression (valid behavior protocol).

The main advantage of parse trees is the size of representation, linearly dependent on the expression length and having no direct relation to the number of states. Also the building time is linear in the length of expression. Evaluation of access time and state identifiers' space requirement will be discussed later after we present parse tree-based representation technique (parse tree automata).

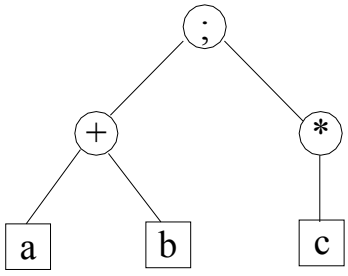

Fig. 2. A parse tree representing (a+b) ; c*

Parse tree automata (PTA). Construction of a PTA follows the idea of recursive state space creation in the explicit representation technique. As PTA is a symbolic technique, the actual full state space of PTA is never represented as a single complex data structure. On the contrary, the key idea is to (i) directly represent only the parse tree (PT) of the expression and the *primitive automata* which accept the event symbols in the leaves of the parse tree, (ii) introduce composed state identifiers allowing to detect the current state and avoid unnecessary multiple traversals of PTA states, and (iii) define the transition function of PTA via recursive rules determining the (direct) transitions from a state, given its composed identifier. An example of PTA and its correspondence to a parse tree is illustrated on Fig. 3.

We will demonstrate the idea on three simple examples: (1) representation of a primitive automaton, (2) implementation of automata composition driven by the sequence operator, and (3) implementation of automata composition driven by the parallel operator. Automata compositions driven by the other operators are implemented in a similar manner (a detailed description is in [2]).

A primitive automaton has two states (initial and accepting) and a single transition between them. The transition label is an event symbol.

The sequencing operator expresses concatenation of the languages accepted by the left- and right - hand automata PTA_L and PTA_R. To create the respective composed automaton $PTA_;$, it is sufficient to establish implicit transitions (λ) from the accepting states of PTA_L to the initial state of PTA_R (Fig. 4b). The resulting set of accepting states in $PTA_;$ consists of the accepting states of PTA_R . The accepting states of PTA_L are added only if the initial state of PTA_R is accepting. Obviously, modifications of PTA_L and PTA_R are not necessary, since the implicit transitions λ are added in the implementation of the sequencing operator in $PTA_;$.

The parallel operator expresses arbitrary interleaving of all the words of the languages accepted by the left- and right hand automata PTA_L and PTA_R. In order to create the respective product automaton, it is sufficient to establish a state space "grid" and corresponding transitions as illustrated in Fig. 4c.

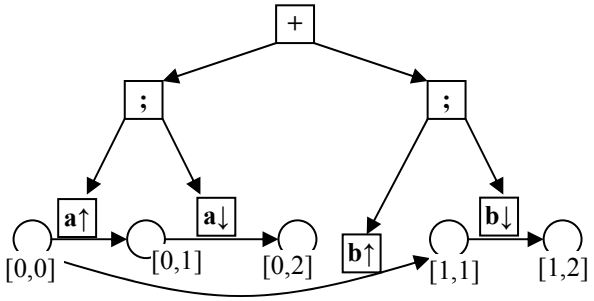

Fig. 3. Generating states and transitions of PTA. Circles represent states. Squares represent nodes of PT.

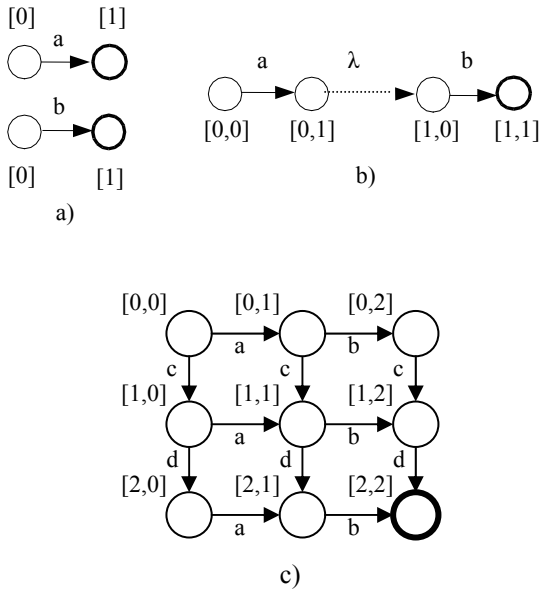

Fig. 4. a) Primitive automata for the "a" and "b" event symbols. b) PTA for "a;b". c) PTA for (a;b) | (c;d). Legend: A dotted arrow represents an implicit transition λ. State identifiers are in brackets (simplified).

Composed state identifiers in PTA. To address the idea (ii) above, a state identifier must reflect the structure of the subtree of PT it is associated with and capture the state of the primitive automata within the subtree. For a specific PT, all the top-level identifiers will be of the same size (linear in the size of PT). As a technicality, memory allocation for state identifiers can cause substantial memory overhead. It is recommended to use an allocator that is optimized for allocating small memory chunks of the same size.

Time requirements for generating PTA transitions. The average time required is influenced by the number of PT nodes that have to be visited to calculate the list of transitions associated with a particular state. In each of these nodes some computation is necessary, as the potential transitions are determined on the fly. For each transition,

also the state identifier of the target state has to be evaluated for keeping track of the states visited.

The number of visited PT nodes is greatly influenced by the actual operators encountered in PT. For example, for the standard regular expression operators only one subtree has to be visited. On the contrary, encountering a parallel operator means visiting both subtrees.

2.4. PTA optimizations

As discussed in Section 2.3, performance of PTA depends on the number of nodes in PT. If the number of PT nodes were reduced, performance would greatly improve. Therefore we experimented with several optimizations in PTA representation.

Multinodes. The idea of multinodes is to collapse the nodes of PT featuring the same operator into a single node. For example, in Fig. 5 collapsing means representing only a single node for the sequence operator '**;**' (associated with a list of PT subtrees a, b, c, d).

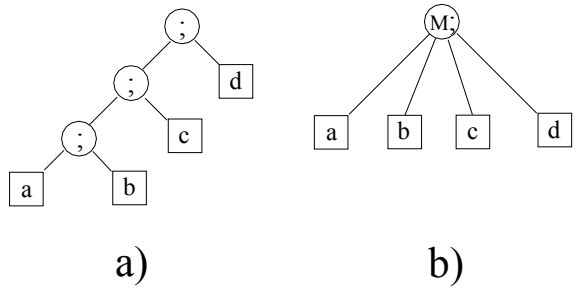

Fig. 5. a) Original parse tree. b) Parse tree with multinodes for the protocol a;b;c;d

This way, access time is greatly improved since less computation is required.

Forward cutting (of primitive automata). Removal of the transitions from the state space, which are discarded by a restriction operator, can be easily achieved by removing the affected event symbols nodes from PT.

Again, such optimization can produce PTs with a smaller number of nodes what results in a smaller state identifiers' space and improved access time.

Explicit subtrees. Since performance of explicit representation is very good for state spaces of "reasonable" size, it can be advantageous to combine both the PTA and explicit representations techniques. It is feasible to select those PT subtrees that imply a small state space (typically not featuring "many" parallel operators) and the states of which are generated more than once (e.g. forced by a parallel operator in a higher level of PT) and represent them via explicit automata embedded in PTA.

We implemented two verifiers based on the PTA representation technique ("Python verifier" and "Java verifier"). These implementations provide a flexible framework that allows simple addition of new parsers, optimizations, and verification backend alternatives as explained below.

3. Python implementation of PTA

Architecture platform. The Python verifier consists of three independent parts (parser, optimizer, backend) orchestrated by a simple application. All the parts of the verifier are implemented in Python [7]. However as the original Python provides only interpreted execution, we use the PSYCO [8] optimizing compiler to improve efficiency.

Parser. The goal of the parser is the creation of a PT representation from an expression. Currently only behavior protocols (Section 1.2) are considered as expressions.

Optimizer currently supports forward cutting of events and explicit subtrees optimizations. To choose a subtree that should be converted into an explicit automaton, a simple estimate of the number of states described by the subtree is based on assigning weights: the primitive automata get weight 2; for sequencing and alternative operators we sum the weights of the underlying automata, for parallel operators we multiply the weights. All the subtrees, the weight of which does not exceed a specific value, are addressed via explicit representation.

Backend alternatives. To enhance the application area of behavior protocols, we created three backend alternatives: compliance checking, visualization (using Aisee visualization tool [9]), and model checking (using Caesar/Aldebaran model checker [10]). Technically, compliance is checked by evaluating the subset relations of the compliance conditions (defined in [1]) via inspecting the emptiness of intersection of one set and the complement of the other. Visualization of a state space can ease up protocol perception, especially by highlighting counter examples produced by compliance verifier. When the state space gets too large for visualization, checking of specific properties is easier via a model-checking tool such as the Caesar/Aldebaran toolset. The bottom line is that independent tools are used for visualization and model checking; the verifier prepares only source files for them.

Since all backends use exhaustive traversal of the state space, we implemented a general depth-first-search algorithm that provides hooks for the algorithm specific computations during a state space traversal. The algorithm uses state space caching technique [12] to keep the list of visited states.

Implementation details. For particular operators, operator nodes are implemented as classes derived from a single interface that allows the client to obtain the initial state of the state space, list of transitions for a particular

state, and list of the accepting states. In addition to the behavior protocol operators, we also implemented operators for language complement and automata product. A state identifier is implemented as a tree of Python 2-tuples.

Benchmarks. We used a slightly modified case study from [1] to assess performance of the Python verifier. The case study features a database server composed of two components and the protocol describing the server's behavior is:

```
!dbAcc.Open;
  (?d.Insert
        {(!dadbAcc.Insert; !dbLog.LogEvent)*}
+
  ?d.Delete
        {(!dadbAcc.Delete; !dbLog.LogEvent)*}
+
  ?d.Query
        {(!dadbAcc.Query)*}) *;
!dbAcc.Close
```

Our enhancements to the case study [1] pertain parallelism for accessing the functionality of the database server (replacing the '+' operator by '|') and the addition of two methods, `insert` and `modify`, to the server interface. The new methods are used in a similar way as their siblings. Using parallelism and the addition of new methods significantly increased the size and complexity of the related state space. These modifications are discussed in [2].

We created four benchmarks (1-4): In (1) we tested the compliance of the protocol described in the case study [1] with the composed protocol of nested components. Both state spaces in (1) were very simple and compliance verification was fast. In the subsequent benchmarks, we (2) replaced the alternative operators by parallel operators and (3) added the `insert` and (4) `modify` methods.

For illustration, the protocol in the (4) variant (most demanding as far as the size of state space generated is considered) was:

```
!dbAcc.open; (
  (?dbSrv.insert↑;!trans.begin;
  (!dbAcc.insert;!lg.logEvent)*;
  (!trans.commit+!trans.abort); !dbSrv.insert↓)
|
  (?dbSrv.delete↑;!trans.begin;
  (!dbAcc.delete;!lg.logEvent)*; (!trans.commit +
  !trans.abort); !dbSrv.delete↓)
|
  (?dbSrv.update↑;!trans.begin;
  (!dbAcc.update;!lg.logEvent)*;
  (!trans.commit+!trans.abort); !dbSrv.update↓)
|
  (?dbSrv.modify↑;!trans.begin; (!dbAcc.modify;
  !lg.logEvent)*; (!trans.commit+!trans.abort);
  !dbSrv.modify↓)
|
  (?dbSrv.query↑;!dbAcc.query;!dbSrv.query↓)
)*;
!dbAcc.close.
```

We benchmarked the consumed memory and required time of the original verifier and of the Python verifier with different optimizer settings. The speed without the forward cutting of primitive automata optimization was very poor, being significantly slower when compared to the original verifier; therefore this optimization was applied in all of the following benchmarks.

		(1)	(2)	(3)	(4)
		Simple protocol from [1]	**Protocol with |**	**Protocol with | and `insert`**	**Protocol with |, `insert` and `modify`**
	Original verifier	12.2MB	16.8MB	70.5MB	*Out of memory limit*
Python verifier	**0 explicit states**	5.9MB	6.2MB	12.3MB	72.4MB
	100 explicit states	5.9MB	6.6MB	10.1MB	46.4MB
	10,000 explicit states	0.16s/ 5.7MB	6.7MB	9.9MB	40.2MB
	1,000,000 explicit states	5.7MB	6.4MB	14.0MB	70MB

Table 1. Memory benchmark results of the original verifier and the Python verifier for various sizes of explicit subtrees measured by number of their states.

		(1)	(2)	(3)	(4)
		Simple protocol from [1]	**Protocol with |**	**Protocol with | and `insert`**	**Protocol with |, `insert` and `modify`**
	Original verifier	800.0%	102.9%	197%	*Out of memory limit*
Python verifier	**0 explicit states**	100%	100%	100%	100%
	100 explicit states	123.1%	48.9%	45.8%	44.6%
	10,000 explicit states	123.1%	48.9%	34.0%	32.8%
	1,000,000 explicit states	123.1%	97.1%	45.0%	28.2%

Table 2. Relative time requirements: The original verifier vers. Python verifier for various sizes of explicit subtrees measured by number of their states.

The explicit subtrees optimization was applied to a different numbers of states embedded in explicit

representation: 0 (no optimization), 100, 10,000, and 1,000,000. The results of the benchmarks were a little bit surprising: The Python verifier (based on PTA) with forward cutting of primitive automata outperformed the original SOFA verifier (based on explicit representation). However, there was a major difference in CPU time dedication: The original verifier spent most of the time by creating the explicit representation, while the actual verification was very quick (about two seconds for (3)). The Python verifier spent some time on optimizations and a significant amount of time on verification. The time spent by the optimizer heavily depended on the size of explicit subtrees. For example, in (4) the creation of explicit subtrees with 1,000,000 states took about 135 seconds. The increase of the overall execution time of (2) and (3) (comparing 10,000 and 1,000,000 states) was caused by the time necessary for creation of explicit subtrees.

4. Java implementation of PTA

To fully incorporate a checking tool into the SOFA environment, we decided to reimplement the Python verifier in the Java language.

The Java verifier uses the approach and techniques employed in the former Python verifier, but it introduces new optimizations and backend features. By these optimizations, both time and space requirements decreased and, therefore, the complexity of the protocols that can be checked was pushed a bit further.

Optimizations. Besides the optimizations included in the Python verifier (explicit subtrees and forward cutting), the multinodes optimization (Fig. 5) was implemented and found very beneficial. This optimization is performed during the construction of a parse tree in a straightforward, efficient way.

Backend alternatives. In the Java verifier we implemented only two backends: compliance checking and visualization, since these two had been identified as the most frequently needed.

For visualization, we decided to use the dot tool of the Graphviz package [16], since it is freely distributed and its features greatly suffice for our purposes. The visualization backend is able to provide both protocol parse tree and graph of the PTA state space. Since the dot tool supports, among other types of output, the Virtual Reality Modeling Language (VRML), this format can be advantageously used for complex protocols both to get the whole picture of the automaton and zoom into its specific parts.

Implementation details. Because of the differences between Python and Java, we had to cope with a lot of specific problems when rewriting the verifier from Python to Java. A main problem was the state identifiers in Java (handled internally by Python): As implied by the argumentation in Section 2.3, we needed state identifiers that could be computed fast and consume as small amount of memory as possible. We could not use Java references, because of the on-the-fly state generation (potentially repeated for a particular state).

Therefore, each state is represented by a *state tree*, where its leaves represent the states of primitive automata, while inner nodes represent the state of the composed automata corresponding to the nodes' subtree. The state identifier of a primitive automaton indicates its active state (0 or 1) (Fig. 4a). The state identifier of a composed automaton is created as concatenation of its children's identifiers. Thus, the resulting state identifier reflects the structure of PT, uniquely denotes a state within the state space, and its length is linear in the size of PT. Obviously, the state identifier of the main automaton is determined at the root of the state tree. The state identifiers are computed in a lazy way (only when actually needed) and are stored in a cache. Traversal of the state space employs frequent comparison of the identifiers (that is quite fast). Even though the computation of state identifiers was optimized for speed, it is still the most time consuming operation in the checking process (since it is performed for each state visit).

Benchmarks. We employed two types of benchmarks: the first type was focused on the benefits of particular optimizations in the Java verifier and the second one on a comparison of performance of the three verifier versions: the original verifier (written in Java), and Python and Java PTA verifiers. Always we used protocols of various complexity; both real-life and "academic" protocols inducing large state spaces were checked.

The real-life protocols included again a set of database server protocols similar to those used in Section 3. The "academic" protocols involved only the parallel operator (such as `a | b`, `a | b | c`, …), which is one of those causing the exponential growth of the state space, so that using it enabled us to generate really large state spaces and easily compute their sizes.

The optimization benchmarks have shown that disabling the forward cutting optimization results in a very poor performance. This is caused by the complement operator expanding the state space to an enormous size. Hence, as well as in Section 3, forward cutting is used in each of the benchmarks below. The benefits of the other types of optimization depend on the concrete structure of the protocols being checked (Table 3). For example, in the case of "academic" protocols using the parallel operator, the most worthwhile optimization are multinodes; the explicit subtrees optimization cannot be used here, because the states of the automaton represented by the only (multi-) node in the parse tree are used only once. While checking the real-life protocols, the explicit subtrees optimization is most beneficial.

Since the most important parameter of the protocol verifier is the state processing speed, in Table 4 we present the comparison of all verifiers based on checking the

"academic" protocols (the results of checking the real-life protocols are not so interesting). A comparison of memory requirements is not involved since it is clear from Table 1 that a PTA representation requires a smaller amount of memory than a corresponding explicit representation. In all benchmarks considered below all optimizations were applied.

In the case of "academic" protocols, the Java verifier is faster than the Python verifier even if we turn off the multinodes optimization; the state processing is about two times faster in the Java verifier, which is probably caused by the fact that the Java Virtual Machine outperformes the Python Psyco compiler. On the other hand, the construction of the explicit subtrees is much slower in Java because of the evaluation of the state identifiers; the Python verifier is also able to keep more states (and larger explicit subautomata) in memory, because its state identifiers are shorter. In any case, the PTA approach beats the original explicit state representation.

	Forward cutting only	**All optimization**	**No multinodes**	**No explicit subtree**
Academic (parallel)	100%	76.2%	100.8%	75.7%
Real-life	100%	50.5%	67.7%	81.4%

Table 3. Average relative time the Java verifier spent by checking with various optimizations enabled.

Number of parallel operators used	**Original verifier**	**Python verifier**	**Java verifier**
6	100%	38.3%	22.3%
7	100%	16.5%	7.7%
8	100%	6.9%	2.3%
9	100%	2.7%	0.7%

Table 4. Relative time spent by checking the "academic" (parallel) protocols by all verifiers.

5. Evaluation and related work

Evaluation. The idea of using parse trees for symbolic representation has proven to be useful for the verification of behavior protocols. The newly implemented verifiers outperformed the original SOFA one, both in time and space complexity. The results provide a solid base for the hypothesis that the symbolic PT representation supported by the forward cutting of primitive automata optimization outperforms the explicit representation. Nevertheless, this hypothesis is still to be justified by a more thorough benchmarking.

While experimenting with the proposed technique of PTA, we identified the following implementation issues: (i) Access time was significantly influenced by applying the forward cutting of primitive automata optimization. This implies there might be huge method calls overhead during the list of transition computation. (ii) Another access time improvement may be achieved by an adaptive selection of explicit subtrees, since our benchmarks showed that access time depends on the size of the parse trees as well. (iii) State identifiers may involve allocation of small structures what means a significant memory allocation overhead. Using a customized allocator, the amount of consumed memory might be greatly decreased in both Python and Java cases.

Related work. To our knowledge, there is no other work that would focus on evaluation of an optimal representation for regular expressions. Therefore we can provide bellow only a comparison with representation techniques that face state explosion in other transition systems.

Space explosion handling techniques can be divided into two categories: (i) efficient representation of the state space and (ii) structural simplification of the state space. OBDDs (mentioned in Section 2.2) and their derivatives Multiple-value Decision Diagrams (MDDs) [11] and Multiple-terminal BDDs (MTBDDs) [13] are typical representatives of (i). All these representations suffer from the optimal ordering problem [6]. There are heuristics developed, but they cannot guarantee the optimal results. Structural simplification of the state space is usually achieved by employing several level of abstraction in model description [14]. In fact, this technique was implicitly employed in SOFA component model [3], since a behavior protocol is always defined for a particular level of component nesting (as opposed to [14]) and behavior compliance is evaluated separately at the adjacent levels of component hierarchy.

6. Conclusions and future intentions

In this paper, we presented a new representation of a state space called Parse Tree Automata that addresses the state explosion problem encountered in behavior protocol checking. PTA fights this problem successfully for behavior protocols of "practical size". Both verifiers based on this representation outperformed the original verifier implemented within the SOFA project not only in memory requirements but in the speed of verification as well.

In the future, we intend to focus on handling the implementation issues described in Section 5. In particular we would like to implement an adaptive version of explicit subtrees optimization and make experiments with various memory allocators.

Acknowledgement

The work was partially supported by the Grant Agency of the Czech Republic (project number 102/03/0672). We are grateful to our colleagues Vladimir Mencl and Jiri Adamek for valuable comments.

References

[1] F.Plasil and S.Visnovsky: Behavior protocols for software components. IEEE Transactions on SW Engineering, 28 (9), Sep 2002.
[2] M.Mach: Formal verification of behavior protocols. Master thesis, Dept. of SW Engineering, Charles University, Prague, 2003.
[3] SOFA project, http://nenya.ms.mf.cuni.cz/sofa/
[4] F.Plasil and J.Adamek: Behavior Protocols Capturing Errors and Updates. Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE 2003), ETAPS, University of Warsaw, 2003.
[5] R.Allen and D.Garlan: A Formal Basis For Architectural Connection. ACM Transactions on Software Engineering and Methodology, Jul 1997.
[6] C.Meinel and T.Theobald: “Algorithms and Data Structures in VLSI Design: OBDD Foundations and Applications”. Springer Verlag, 1998.
[7] Python, http://www.python.org
[8] PSYCO compiler, http://psyco.sourceforge.net
[9] Aisee visualization tool, http://www.aisee.com
[10] Caesar/Aldebaran model checker, http://www.inrialpes.fr/vasy/cadp/
[11] A.Srinivasan, T.Kam, S.Malik, and R.Brayton: Algorithms for discrete function manipulation. Int’l Conf. on CAD, 1990.
[12] P.Godefroid, G.Holzmann, and D.Pirottin: State-Space Caching Revisited. Formal Methods in System Design: An International Journal, 1995.
[13] M.Fujita, P.McGeer, and J.Yang.: Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. Formal Methods in System Design: An International Journal, 10, April 1997.
[14] D.Giannakopoulou, J.Kramer, and S.Cheung: Analysing the Behaviour of Distributed Systems using Tracta. Journal of Automated Software Engineering, special issue on Automated Analysis of Software, vol. 6(1), Jan 1999.
[15] E.Bruneton, T.Coupaye, and J.Stefani: The Fractal Composition Framework. Proposed Final Draft of Interface Specification version 0.9, The ObjectWeb Consortium, Jun 2002.
[16] Graphviz - open source graph drawing software, http://www.research.att.com/sw/tools/graphviz