Addressing State Explosion in Behavior Protocol Verification

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

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

[2]Academy of Sciences of the Czech Republic
Institute of Computer Science
plasil@cs.cas.cz, http://www.cs.cas.cz

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, a verification tool for behavior protocols is 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. 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 size of the state space associated with the model tends to exhaust all the memory available for a particular verification tool ("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 *provides* and *requires* 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 start (↑) or end (↓).

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 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 provides and requires interfaces of each of them?" can be answered. 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 on a well loader 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 verifier. Another goal is to compare the proposed representation with other frequently recommended automata representation techniques such as OBDDs [6].

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. In Section 4, we evaluate the proposed representation and compare it with other techniques addressing state explosion. Section 5 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 automata representation techniques.

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

As to size of representation, state explosion is very likely. Also building time is fairly low as the construction of a state space is usually done recursively, by joining the state spaces of sub-expressions.

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 we identified 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: (i) In most cases, it helps avoid state space explosion 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 determining a Boolean function $f(x_1,\ldots,x_n) \rightarrow \{0, 1\}$.

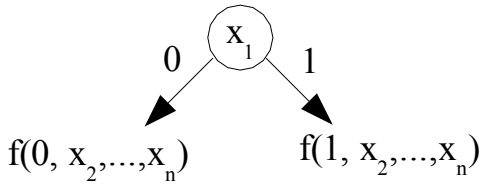


Figure 1: Root of the decision diagram determining the function $f(x_1,...,x_n)$

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 (Figure 1). However the size of an OBDD graph strongly depends on the order of the function arguments. 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 to represent program source codes in compilers. Obviously, they are also capable to represent behavior protocols (Figure 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 leaf nodes and operators appear only as internal 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).

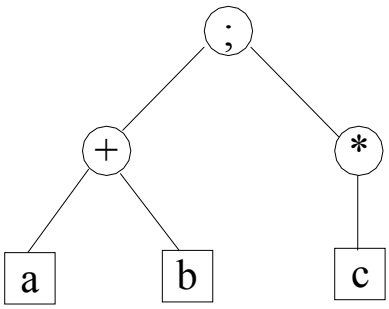


Figure 2: A parse tree representing (a+b);c*

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 to reflect hierarchical composition of the primitive automata (driven by PT), and (iii) form recursive rules determining the (direct) transitions from a state, given its composed identifier, PT, and primitive automata.

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 (Figure 3b). 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 Figure 3c.

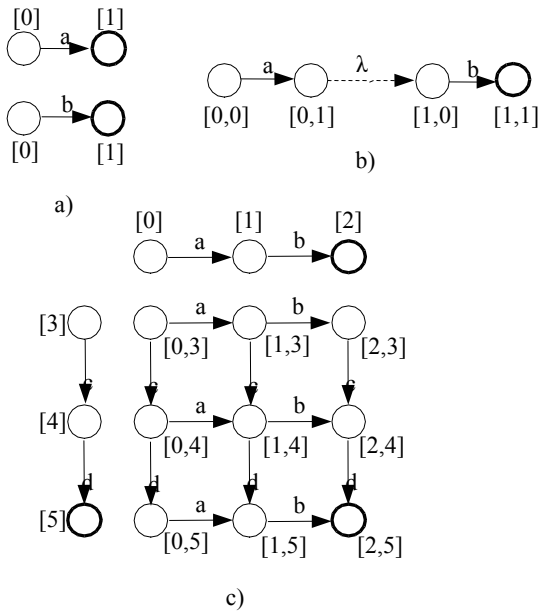


Figure 3: a) Primitive automata for the "a" and "b" event symbols. b) PTA for "a;b". c) PTA for a;b|c;d
Legend: **A dotted arrow represents an implicit transition λ. State identifiers are in brackets (simplified) .**

Space requirement of composed state identifiers in PTA. We used a pair of references for each binary operator representation (a single one for a unary operator); to identify a state in a hierarchical state space, we employed a function (operating upon the hierarchy of objects representing the states) which returns a "string" of the size proportional to the position of the state in the hierarchy. It should be emphasized that the memory allocator employed in a particular run time system can cause substantial memory overhead. It is recommended to use an allocator that is optimized for allocating small memory chunks of the same size.

Access time in PTA. The average access time 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.

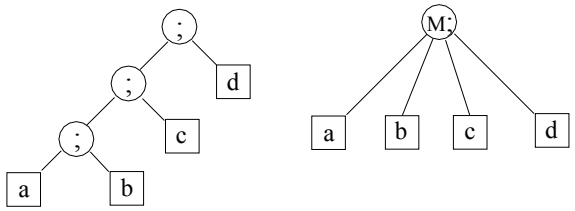
The number of visited PT nodes is greatly influenced by the actual operators encountered in PT. For example, for the standard regular expression operators only one subtree has to be visited. On the contrary, encountering a parallel operator means visiting both subtrees.

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 Figure 4 collapsing means representing only a single node for ; (associated with a list of PT subtrees a, b, c, d).



a) b)

Figure 4: 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. State identifiers' space remain unmodified for the regular expression operators, while for parallel operators, a list of integers is used. (The length of the list is determined by the number of ancestors in PT.)

Forward cutting of primitive automata. Removal of the transitions from the state space which are determined 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 PT subtrees that imply a small state space (e.g. not featuring "many" parallel operators) and represent them via explicit automata embedded in PTA.

3 Case study – implementation of PTA

We implemented a verifier based on the PTA representation technique. The implementation provides a flexible framework that allows simple addition of new parsers, optimizations, and verification backend alternatives (explained below).

Architecture platform. The verifier itself consists of three independent parts (parser, optimizer, backend) orchestrated by a simple application. All the parts are implemented in Python [7]. However as 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 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 options 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]). Visualization of a state space can ease protocol perception, especially by highlighting counter examples produced by compliance checker. When the state space gets too large for visualization, checking of specific properties is easier via a model checking tool such as the Caesar/Alderbaran 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[1]. We used a slightly modified case study from [1] to assess performance of the new verifier. The case study features a database server composed of two

[1] All tests were done on a HP Omnibook 6100 laptop (Pentium III 1GHz, 256MB RAM and operating system Linux Mandrake 9.1). For the SOFA checker, Java 1.4.0/SUN Microsystems and Python 2.3/ Psyco 1.0 were used.

components. Our enhancements to the case study [1] pertain parallelism for accessing the functionality of the database server (replacing the original alternative operator) 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 bechmarks (1-4): In (1) we tested the compliance of the protocol described in the original case study with the combined protocol of nested components. Both state spaces in (1) were very simple and complaince verification was fast. In the subsequent benchmarks, we replaced the alternative operators by parallel operators (2) and added the `insert` (3) and `modify` (4) methods.

For illustration, the resulting, most complicated protocol used in (4) 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.
```

In benchmarks, we measured the consumed memory and required time of the orginal SOFA verifier and of our new implementation with different optimizer settings. The speed without the forward cutting of primitive automata optimization was very poor, significantly slower than of the original verifier, so that this optimization was applied in every benchmark. The explicit subtrees optimization was applied to a different numbers of states embeded in explict representation: 0 (no optimization), 100, 10 000, and 1 000 000.

	(1) Orginal protocol	**(2) Parallel protocol**	**(3) Parallel protocol with `insert`**	**(4) Parallel protocol with `insert` and `modify`**
SOFA verifer	*1.04s/ 12.2MB*	*3.6s/ 16.8MB*	*139.7s/ 70.5MB*	*Out of memory limit*
0 explicit states	0.13s/ 5.9MB	3.5s/ 6.2MB	70.9s/ 12.3MB	1374s/ 72.4MB
100 explicit states	0.16s/ 5.9MB	1.7s/ 6.6MB	32.5s/ 10.1MB	613s/ 46.4MB
10 000 explicit states	0.16s/ 5.7MB	1.7s/ 6.7MB	24.1s/ 9.9MB	451s/ 40.2MB
1 000 000 explicit states	0.16s/ 5.7MB	3.4s/ 6.4MB	31.9s/ 14.0MB	387s/ 70MB

Table 1: Benchmark results of the *original* and new verifier

The results of the benchmarks were little bit surprising: The new 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 SOFA verifier spent most of the time by creating the explicit representation, while the actual verification was very quick (about two seconds in (3)). The new verifier spent some time on optimizations and a significant 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 time necessary for explicit subtrees creation.

4 Evaluation and related work

Evaluation. The idea of using parse trees for symbolic representation proved to be useful for the verification of behavior protocols. The new implemented verifier 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 to be justified by a more thorough benchmarking.

While experimenting with the new verifier, 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 drop by degree of two, because the Python allocator uses additional 8B to every allocated structure (and our structures are of 8B size).

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 can not 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, 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.

5 Conclusions and future intentions

In this paper, we presented a new representation called Parse Tree Automata that handles the state explosion problem encountered in behavior protocol handling. PTA avoids this problem successfully for behavior protocols of "practical size". The verifier based on this representation managed to outperform the original verifier implemented within the SOFA project not only in memory requirements but the speed of verification as well.

In the future, we intend to focus on handling the implementation issues described in Section 4. In particular we would like to implement the multinode optimization and create an adaptive version of explicit subtrees optimization.

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 Jan Kofron for valuable comments.

References

[1] Plasil F., Visnovsky S.: Behavior protocols for software components. IEEE Transactions on SW Engineering, 28 (9), Sep 2002.
[2] Mach M.: 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] Plasil F., Adamek J.: Behavior Protocols Capturing Errors and Updates. Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE 2003), ETAPS, University of Warsaw, 2003.
[5] Allen R.J., Garlan D.: A Formal Basis For Architectural Connection. ACM Transactions on Software Engineering and Methodology, Jul 1997.
[6] Meinel C., Theobald T.: "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] Srinivasan A., Kam T., Malik S., Brayton R.: Algorithms for discrete function manipulation. Int'l Conf. on CAD, 1990.
[12] Godefroid P., Holzmann G., Pirottin D.: State-Space Caching Revisited. Formal Methods in System Design: An International Journal, 1995.
[13] M. Fujita, P. C. McGeer, and J. C.-Y. 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] Giannakopoulou D., Kramer J., Cheung S.C.: Analysing the Behaviour of Distributed Systems using Tracta. Journal of Automated Software Engineering, special issue on Automated nalysis of Software, vol. 6(1), Jan 1999.
[15] Bruneton E., Coupaye T., Stefani J.B.: The Fractal Composition Framework. Proposed Final Draft of Interface Specification version 0.9, The ObjectWeb Consortium, Jun 2002.