Enhanced Property Specification and Verification in BLAST*

Ondřej Šerý

Charles University in Prague
Malostranské náměstí 25
118 00 Prague 1
Czech Republic
ondrej.sery@dsrg.mff.cuni.cz
http://dsrg.mff.cuni.cz

Abstract. Model checking tools based on the iterative refinement of predicate abstraction (e.g., SLAM and BLAST) often feature a specification language for expressing complex behavior rules. The source code under verification is instrumented by artificial variables and statements in order to transform the problem of checking such a rule into the problem of program location reachability. This way, the source code get bloated and additional predicates have to be discovered and tracked during the verification. We suggest that a significant performance improvement can be achieved by tracking state of the behavior rules aside from the source code instead of instrumenting them. We have implemented an extension to BLAST, which accepts a specification language (a simplified version of *behavior protocols*), and checks its validity without modifying the input source code. An experiment with two Linux kernel drivers confirms the performance gain using the extension.

1 Introduction

For the last few years, the explicit state and predicate abstraction based model checking techniques have developed rather independently to each other. Success stories of the predicate abstraction based tools, like SLAM [2] and BLAST [14], earned a lot of both research and industry attention. Basically, these tools create a very coarse existential abstraction (over-approximation) of a system, try to find an error trace (if there is none, the system is safe), decide whether the error trace is a real one (i.e., the system is erroneous) or not, in which case the existential abstraction is refined and the cycle repeats. The technique has very good performance on single-threaded programs even those containing high level of data nondeterminism. On the other hand, these tools typically feature a very limited support for multi-threading, complex data types (e.g., floats and arrays), reasoning about heap objects, and perform poorly on certain inputs (e.g., containing `for` cycles).

* This work was partially supported by the Grant Agency of the Czech Republic project 201/08/0266.

M. Chechik and M. Wirsing (Eds.): FASE 2009, LNCS 5503, pp. 456–469, 2009.
© Springer-Verlag Berlin Heidelberg 2009

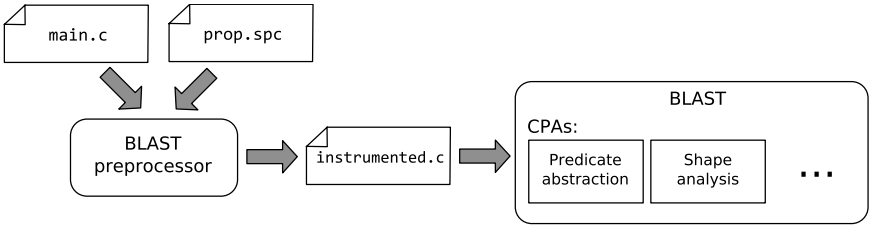

Fig. 1. Architecture of the BLAST model checker

In contrast, the explicit state model checkers, like Java PathFinder [20] and SPIN [15], which are based on explicit representation and exploration of the state space, perform complementary in many cases. They are typically equipped with optimizations for dealing with multi-threaded programs (e.g., partial order reduction, transactions), and complex data types as well as heap objects are represented explicitly without much trouble. A major obstacle, however, is data nondeterminism, for which the predicate abstraction based tools excel.

In general, this suggests not only that for some inputs one or the other technique is preferred but also that relying on one technique only might not be enough; that mixing these two techniques in a single tool is a promising idea. Quite recently, approaches to mixing explicit state and abstraction based model checking have been published [13,17,8].

In this paper, we apply the idea on the BLAST model checker. We propose an extension for tracking the state of a behavior specification during verification explicitly rather then encoding it into the source program and then using the purely abstraction based verification (as done in BLAST). The original process is depicted on Fig. 1. Before the actual verification, the input source code is instrumented so that the problem of checking a rule is converted into the problem of program location reachability. The resulting program is bloated by a code whose only purpose is to identify the error states. To analyze the instrumented code, additional costly theorem prover calls are necessary. In contrast, our extension tracks the state of the behavior specification explicitly without any modification of the source program and with no unnecessary theorem proving overhead.

BLAST has a specification language [5] for stating the behavior rules. Although the language is very powerful (almost arbitrary C statements can be used) and well suited for simple rules, we argue that it is not very user-friendly for specifying more complex rules concerning function call sequencing and nesting. In such a case, a user has to manually encode the rule into an additional state variable(s) and ensure proper state transitioning, which is very impractical and error prone.

1.1 Goals and Structure of the Paper

The goal of this paper is twofold, (i) to extend BLAST's algorithm for verification of behavior rules by explicit state representation of the rule without modification of the input source code and any additional theorem proving overhead, and (ii)

to allow for specification of behavior rules restricting sequencing and nesting of function calls in a lightweight easy-to-use formalism.

The rest of the paper is structured in the following way. First, we summarize the BLAST's concept of configurable program analysis (Sect. 2), which we employ in our technique on both the formal and the implementation level. Then (Sect. 3), we present the simplified formalism of behavior protocols to be used for behavior specification (Sect. 3.1) along with necessary extensions to the configurable program analysis concept (Sect. 3.2) and a note about the prototype extension of the BLAST model checker (Sect. 3.3). Experimental evaluation of the proposed technique and discussion in the context of the related work (Sect. 5, 4) are followed by list of directions for future research (Sect. 6) and concluding remarks (Sect. 7).

2 Configurable Program Analysis

For the sake of completeness, we summarize the concept of *Configurable Program Analysis* (CPA) as published by the BLAST authors in [7]. The CPA concept stems from *abstract interpretation* [12] and was originally introduced to support a uniform view on model checking and static analysis. Nevertheless, later in Section 3, we will use CPA with advance as a means for plugging the explicit state space of behavior specification into BLAST.

The basic idea is to have multiple CPAs for tracking different kinds of information (e.g., predicates, heap shape) about the program under analysis. Each CPA tracks the information in either a path sensitive or insensitive way. By combining the different CPAs, various configurations of the resulting analysis can be achieved.

Definition 1 (Configurable Program Analysis). *A* configurable program analysis *is a four-tuple* $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$*, where* D *is an abstract domain,* $\rightsquigarrow$ *is a transfer function,* merge *is an operator for merging states, and* stop *is a termination check.*

Informally, D represents the state space of a CPA. It consists of a set of concrete states C, a semi-lattice (with a preorder $\sqsubseteq$ and a join operator $\sqcup$) of abstract states E, and a concretization function relating the abstract and concrete states. For explicit state CPAs featuring no abstraction, the semi-lattice is the trivial flat lattice over the set of concrete states ($E = C \cup \{\top, \bot\}$). The transfer relation $\rightsquigarrow \subseteq E \times G \times E$ contains transitions among the abstract states of D, where G is a set of labels, which contains statements of the program under analysis.

The two operators merge and stop play their role during the state space traversal. The termination check $\mathsf{stop} : E \times 2^E \rightarrow \mathbb{B}$ is used to decide whether a newly discovered state (first parameter) is covered by the already explored states (second parameter). If so, the new state is not analyzed any further. For purposes of this paper, the operator $\mathsf{merge} : E \times E \rightarrow E$ is not of high importance. An intuitive idea that merge is used to merge information from a newly discovered state (first parameter) to each already visited state (second parameter), i.e., merging

Algorithm: $traverseCPA(\mathbb{D}, e_0)$

Input: a configurable program analysis $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$, an initial abstract state $e_0 \in E$, let E denote the set of abstract states of D
Output: a set of reachable abstract states
Variables: a set *reached* of elements of E, a set *waitlist* of elements of E

```
waitlist := {e_0}
reached := {e_0}
while waitlist ≠ ∅ do
  pop e from waitlist
  for each e' with e ⇝ e' do
    for each e'' ∈ reached do
      // Combine with already visited abstract states.
      e_new := merge(e', e'')
      if e_new ≠ e'' then
        waitlist := (waitlist ∪ {e_new}) \ {e''}
        reached := (reached ∪ {e_new}) \ {e''}
    if ¬stop(e', reached) then
      waitlist := waitlist ∪ {e'}
      reached := reached ∪ {e'}
return reached
```

Fig. 2. Algorithm for CPA state space traversal taken from [7]

information from different execution paths, would suffice. A special case of the operators is $\mathsf{stop}^{sep}(e, R) = (\exists e' \in R : e \sqsubseteq e')$ and $\mathsf{merge}^{sep}(e, e') = e'$. These correspond to a model checking CPA without merging of information from different execution paths.

In [7], the authors describe a number of CPAs for predicate abstraction, shape analysis, and pointer analysis. They also present a way how to combine more CPAs into a single composite CPA and an algorithm for state space traversal of a CPA. We recapitulate the algorithm in Fig. 2. For further details on CPA, the reader is kindly referred to the original paper.

3 Checking Behavior

Having the CPA concept explained, we first show what kind of behavior specification we are interested in and how it can be encoded using the CPA concept.

3.1 Behavior Specification

The behavior specification used in this paper stems from the formalism of *behavior protocols* [18,1], which comes from the software component world. In a syntax close to regular expressions, a behavior protocol specifies a behavior as a set of finite traces of method calls that are allowed to occur on the component's interfaces. The reason for using behavior protocols (besides our long experience with the formalism) lies in their relative simplicity, which makes them easy to use even for a nonprofessional.

In this paper, a slightly modified definition of behavior protocols, tailored for specification of behavior rules of C code, is used. The basic building block is not a method call on an interface of a component, but a C function call. The simplified syntax and semantics of behavior protocols is as follows.

Definition 2 (Syntax). *A* behavior protocol *over an alphabet Σ is an expression obtained by a finite number of applications of the following rule. Let a and b be behavior protocols, and $func \in \Sigma$, then all expressions of the form: $NULL$, $func\uparrow$, $func\downarrow$, (a), $a*$, $a;b$, $a+b$, $a \mid b$, $func$, $func\{a\}$ are also behavior protocols.*

The semantics of a behavior protocol is then a set of allowed traces of events $func\uparrow$ and $func\downarrow$, where $func\uparrow$ denotes a function call and $func\downarrow$ denotes return from the call. Distinguishing between the two events allows for precise specification of function nesting.

Definition 3 (Semantics). *The* set of traces *specified by a behavior protocol p, and denoted as $\mathbb{L}(p)$, is inductively defined as follows:*

Protocol	***Description***	***Semantics***
$NULL$	*Empty prot.*	$\mathbb{L}(NULL)) = \{\lambda\}$
$func\uparrow$	*Func. call*	$\mathbb{L}(func\uparrow) = \{func\uparrow\}$
$func\downarrow$	*Return*	$\mathbb{L}(func\downarrow) = \{func\downarrow\}$
(a)	*Parentheses*	$\mathbb{L}(\ (a)\ ) = \mathbb{L}(a)$
$a*$	*Repetition*	$\mathbb{L}(a*) = \{u^n \mid u \in \mathbb{L}(a) \wedge n \in \mathbb{N}_0\}$
$a;b$	*Sequence*	$\mathbb{L}(a;b) = \{u.v \mid u \in \mathbb{L}(a) \wedge v \in \mathbb{L}(b)\}$
$a+b$	*Alternative*	$\mathbb{L}(a+b) = \mathbb{L}(a) \cup \mathbb{L}(b)$
$a \mid b$	*Parallelism*	$\mathbb{L}(a \mid b) = \{u \mid u$ *is interleaving of* $v \in \mathbb{L}(a)$, $w \in \mathbb{L}(b)\}$
$func$	*Abbreviation*	$\mathbb{L}(f) = \mathbb{L}(func\uparrow; func\downarrow)$
$func\{a\}$	*Abbreviation*	$\mathbb{L}(f\{a\}) = \mathbb{L}(func\uparrow; a; func\downarrow)$

As an example of a behavior protocol, consider the following usage rule of the SDL graphic library:

```
SDL_Init; (SDL_PushEvent + SDL_WaitEvent)* ; SDL_Quit
```

The rule states that a call to SDL_Init should precede any manipulation with event queues (finite number of calls to $SDL\_PushEvent$ and $SDL\_WaitEvent$) and that the SDL_Quit cleanup function should be called afterwards.

Naturally, the set of traces specified by a behavior protocol can be represented by the means of a finite automaton. The transformation follows the standard algorithm of transformation of a regular expression into an automaton [16], straightforwardly extended by the parallel operator |.

Definition 4. *Let p be a behavior protocol over an alphabet Σ, then $A_p = (S_{A_p}, \Sigma_{\uparrow\downarrow}, \rightarrow_{A_p}, initial_{A_p}, F_{A_p})$, where $\Sigma_{\uparrow\downarrow} = \{func\uparrow, func\downarrow \mid func \in \Sigma\}$, denotes the minimal deterministic finite automaton over the alphabet $\Sigma_{\uparrow\downarrow}$ accepting traces specified by p.*

3.2 Behavior CPA

Decoupling the behavior specification from code into a separate CPA requires one change to the concept of CPA. During state space traversal, different CPAs can add different kinds of information to the states being traversed. In principle, this allows CPAs to affect the shape and the size of the state space to be traversed. However, there is no mechanism that would allow CPA to identify erroneous states. Only the states involving a program location, which is labeled as erroneous (i.e., passed as an input to BLAST) are considered erroneous. Information tracked by individual CPAs is not used for error state detection.

This is because BLAST was originally designed to decide reachability of program locations (marked as erroneous). In theory, this is sufficient for deciding any safety property, as the problem can be always transformed into decision of program location reachability on an accordingly modified program. In practice, however, the necessity to transform all properties into reachability of a program location is prohibitive. It seems more natural to allow the individual CPAs to identify error states based on the information a particular CPA tracks rather than requiring modification of the original source code and thus affecting (obfuscating) input shared by all the CPAs.

For example, to check absence of null pointer dereference errors in a program using BLAST, one can insert an if statement asserting that `p != NULL` before dereferencing any pointer `p`. In case of a null pointer, the statement would lead to an error program location. BLAST is then executed to check that the error location is unreachable and thus no null pointer dereference error can occur (this technique was used and documented by others in [6]). In contrast, we argue that such a change of the original program affects all CPAs (mainly the predicate abstraction CPA) and that such an error could be detected by the shape or pointer analysis CPA, which tracks the information concerning heap.

Note that we discuss only "identification" of the possible error states. Once there is a candidate error trace, all the CPAs can contribute to prove the trace infeasible by their means and, if it is a spurious one, get refined to disallow the trace in the future. The point here is that there is no need to bother all the CPAs by the information necessary for identification of the possible errors related to only some of them.

This becomes even more pronounced in the case of behavior specification, whose purpose is only to observe an execution of a program (without altering it in any sense) and to signal any violation. As mentioned above, this problem can also be transformed into the program location reachability by encoding the behavior specification into the program itself (as is done for the BLAST specification language). However, then all CPAs are affected by the additional code, whose purpose lies only in identification of the error states. Namely, the predicate abstraction CPA will be cluttered by artificial predicates. This is costly, because finding and managing additional predicates means additional theorem prover calls.

Therefore, we extended the concept of CPA to allow identification of error states of two kinds. First, a *standard error state* ε_{reach}, is a state which should

never be reached in a correct program; i.e. a program is considered incorrect, if there is a prefix of a concrete path in the program reaching the error state. Second, an *error final state* ε_{final}, is a state which represents an error only for the final state of the model. In other words, a program is considered incorrect, if there is a finite concrete path ending in an error final state. As an example, the null pointer dereference error is a standard error (i.e., it should never happen), while finishing without deallocating all resources is an error final state (i.e., it is alright to have allocated memory during execution but not at its end).

Definition 5 (Configurable program analysis – revisited). *A* configurable program analysis *is a five-tuple* $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{error})$*, where* D *is an abstract domain,* $\rightsquigarrow$ *is a transfer function,* merge *is an operator for merging states,* stop *is a termination check, and* error *is an error identifying relation.*

Given that E is the set of abstract states of D, then $\mathsf{error} \subseteq E \times \{\varepsilon_{reach}, \varepsilon_{final}\}$ annotates the abstract states which are considered erroneous. An abstract state e implies a whole system error state if $\mathsf{error}(e, \varepsilon_{reach})$ holds. If $\mathsf{error}(e, \varepsilon_{final})$ holds, the whole system error state is implied only if the system state is final; i.e., the system can terminate in its current state.

With the CPA definition extended by error state signalization, we can define a CPA which tracks a single behavior protocol and signal its violation as an error state to the model checker. In turn, the model checker can attempt to avoid the error state by refining abstractions captured by all the CPAs.

Definition 6 (Behavior CPA). *Let* p *be a behavior protocol over an alphabet* Σ*, then the* behavior CPA *with respect to* p *is denoted as* $\mathbb{B}(p) = (D_{\mathbb{B}(p)}, \rightsquigarrow_{\mathbb{B}(p)}, \mathsf{merge}_{\mathbb{B}(p)}, \mathsf{stop}_{\mathbb{B}(p)}, \mathsf{error}_{\mathbb{B}(p)})$*.* $D_{\mathbb{B}(p)}$ *is based on the flat lattice over states of the automaton* A_p *(i.e.,* $S_{A_p} \cup \{\top, \bot\}$*). The transfer relation* $\rightsquigarrow_{\mathbb{B}(p)}$ *follows the transition function of* A_p*, extended by a self-transition* ($s \overset{g}{\rightsquigarrow} s$) *for every state* s *and a control-flow edge* g *which does not represent any event tracked by the protocol* p*. More precisely:* $s \overset{g}{\rightsquigarrow} s'$ *iff any of the following holds:*

(i) $s \xrightarrow{g}_{A_p} s'$
(ii) $s = s'$ *and* $g \notin \Sigma_{\uparrow\downarrow}$
(iii) $s' = \bot$ *and* $g \in \Sigma_{\uparrow\downarrow}$ *and* $\neg\exists s'' \in S_{A_p} : s \xrightarrow{g}_{A_p} s''$
(iv) $s = s' = \bot$

The operators $\mathsf{merge}_{\mathbb{B}(p)}$ *and* $\mathsf{stop}_{\mathbb{B}(p)}$ *are chosen to be the simple model checking variants* $\mathsf{merge}_{\mathbb{B}(p)} = \mathsf{merge}^{sep}$ *and* $\mathsf{stop}_{\mathbb{B}(p)} = \mathsf{stop}^{sep}$*. Last, the error identifying relation is chosen so that* $\mathsf{error}_{\mathbb{B}(p)}(s, \varepsilon_{final})$ *iff* $s \notin F_{A_p}$ *and* $\mathsf{error}_{\mathbb{B}(p)}(s, \varepsilon_{reach})$ *iff* $s = \bot$*.*

Behavior CPA is straightforwardly derived from the deterministic automaton A_p representing the given behavior protocol p. Those states that do not correspond to a final state of the automaton A_p are identified as error final states. In such states, the behavior protocol expects further activity and does not allow termination of the program yet. Whenever there is an activity not allowed by the protocol (see (iii) of Def. 6), the next state is chosen to be $\bot$, for which $\mathsf{error}_{\mathbb{B}(p)}(\bot, \varepsilon_{reach})$ holds and is therefore a standard error state.

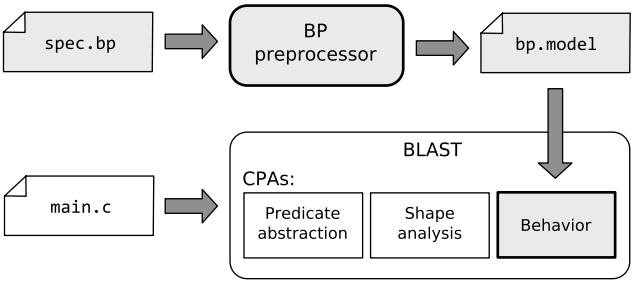

Fig. 3. Architecture of the BLAST extension

3.3 Tool Support

The concept of behavior CPA was implemented as a prototype extension of the BLAST 2.4 release[1]. The resulting architecture is depicted in Fig. 3.

The gray emphasized boxes represent the parts of the tool chain newly added to BLAST as a part of this effort. *BP preprocessor* is a simple command line tool for preprocessing the behavior specification. It parses the specification and transforms it into the minimal deterministic automaton. This tool was based on the core library of *dchecker*, the distributed model checker for behavior protocols [19], and is written in Java. As well as the rest of BLAST, the implementation of behavior CPA is written in OCaml and uses the CPA interface as an extension point. The CPA interface itself was modified to allow identification of error states by individual CPAs.

Unfortunately, libraries, which BLAST uses for theorem proving and constraint solving, are available only as Linux binaries. Even though the rest of the implementation is platform independent, the prototype implementation runs also only under Linux, due to these dependencies.

4 Evaluation

Easier usage of behavior protocols for rule specification is, of course, a subjective matter. However, we show the performance improvements in the following experiment. BLAST was used to analyze two Linux 2.6.24 kernel driver files `drivers/char/esp.c` and `drivers/net/znet.c` with and without our extension. There were two behavior rules (i) correct spinlock locking and unlocking, and (ii) correct sequencing of DMA manipulating function calls used in these files (namely the functions `claim_dma_lock`, `release_dma_lock`, `enable_dma`, `disable_dma`, `clear_dma_ff`, `set_dma_mode`, `set_dma_addr`, `set_dma_count`, and `get_dma_residue`). The rules (i) and (ii) specified using both BLAST specification language and behavior protocols are available in the Appendix.

[1] Source code of the prototype implementation along with test files from Sect. 4 are available for download at `http://dsrg.mff.cuni.cz/~sery/blast/`

Table 1. Verification of the spinlock rule

File	`drivers/char/esp.c`			`drivers/net/znet.c`		
Test	*no hint*	*hinted*	*bp*	*no hint*	*hinted*	*bp*
Preparation	0.17s	0.17s	0.30s	0.17s	0.17s	0.25s
Verification	3.35s	0.77s	0.26s	1.42s	0.29s	0.14s
Sum	3.52s	0.94s	0.56s	1.59s	0.46s	0.39s

Table 2. Verification of the DMA rule

File	`drivers/char/esp.c`			`drivers/net/znet.c`		
Test	*no hint*	*hinted*	*bp*	*no hint*	*hinted*	*bp*
Preparation	0.17s	0.17s	0.30s	0.17s	0.17s	0.30s
Verification	7.07s	1.07s	0.27s	2.55s	0.42s	0.14s
Sum	7.24s	1.24s	0.57s	2.72s	0.59s	0.44s

The verification times[2] are then summarized in Table 1 and 2 for the rules (i) and (ii), respectively. On each source file and for each rule, three different configurations were executed (columns *no hint*, *hinted*, *bp*). Columns *no hint* and *hinted* contain times for the rule specified in the Blast specification language. In the *hinted* column, Blast got the list of predicates suitable for checking the specified rule as a part of the input, while in the *no hint* column, it had to discover the necessary predicates by itself. The *bp* column contains times for the rule specified using behavior protocols and verified using the proposed extension. The difference between *no hint* and *hint* is due to the need for discovery of necessary predicates. However, the predicates can be generated from the specification beforehand. The difference between *hint* and *bp* is due to the need for tracking the additional predicates and coping with the inflated input. The *preparation* row contains times necessary for running the C preprocessor and either code instrumentation or behavior protocol preprocessing. Note also that behavior protocol preprocessing has to be done only once for each rule, while code instrumentation has to be performed after every code modification.

5 Related Work

There is quite a number of software model checkers based on the counter example guided abstraction refinement [11] (CEGAR) and predicate abstraction [3]. Of those, probably the most famous are Slam [2], Blast [14], and SatAbs [10]. All the three tools analyze programs written in the C programing language.

Slam is the oldest of the three tools. It features a straightforward implementation of the abstraction refinement loop. In every iteration, a uniform program abstraction is constructed from scratch, which is costly. Blast enhances the basic idea by constructing the abstraction in memory and refining only the nec-

[2] All the test were run on a Linux 2.6.24, Pentium 4, 3.0GHz machine.

essary portions of it (this is referred to as *lazy abstraction* [14]). This feature significantly improves performance. Like SLAM, the SATABS tool is a straightforward implementation of the abstraction refinement loop. Unlike SLAM, SATABS uses a SAT solver instead of a theorem prover. This allows for precise reasoning about integers as bit-vectors, including arithmetic overflows.

Although the tools can only decide reachability of a program location, other interesting properties can be transformed into this problem by instrumentation of the input program's source code. For this purpose, both SLAM and BLAST use a special purpose specification language (SLIC [4] and the BLAST specification language [5], respectively). However, as already discussed in Sect. 3.2, the instrumentation results in artificial predicates to be discovered and managed, which implies unnecessary theorem proving overhead.

In contrast, our solution exploits the specific nature of the behavior specification and tracks it explicitly in a separate CPA domain without necessity to alter the input source code. This way, no additional theorem prover calls are necessary. Moreover, we argue that using behavior protocols (which are close to regular expressions known to a majority of software developers) for specifying rules restraining method sequencing and nesting is more convenient than using SLIC or BLAST specification language, where such a rule has to be encoded by hand. For more complex rules, this effectively means transformation into a corresponding automaton and its representation using special state variable.

As both SLIC and BLAST specification language permit using almost arbitrary C code, the expressive power is stronger than the expressive power of behavior protocols used in our work. Therefore, we intend to extend the formalism to cover a bigger set of the real-life rules (see Sect. 6).

There are other works that combine explicit state and abstraction based techniques. In [13], the authors propose an algorithm, SYNERGY, which uses concrete execution in cooperation with predicate abstraction. An abstract counter example is used to guide the concrete execution, while the concrete execution traces are used when refining the abstraction. Another technique is presented in [17], where the explicit state space is traversed but abstraction is employed when deciding whether a current state has been already visited. The resulting under-approximation is then iteratively refined. In [8], a technique using explicit representation of some program variables, while predicate abstraction for other, with possibility of precision adjustment, is proposed and an implementation is done in BLAST.

6 Future Work

One of potential directions for future research is extending the power of the formalism used for specification of behavior rules. Regular language is sufficient for conveniently expressing rules concerning correct function call sequencing and nesting. However, other real-life rules the developers are interested in are related to dynamically created and destroyed program entities (e.g., files and locks). In

other words, developers are often interested in correct sequencing and nesting of function calls that refer to the same instances of these entities.

The idea is similar to *tracematches* [9], which are used to specify incorrect behavior patterns that may relate to individual entities. In contrast, behavior protocols are used for positive specification (e.g., specification of expected behavior) not negative (e.g., specification of forbidden behavior), as we believe that the positive specification is less prone to omissions. Signaling a false error is safer then missing a real one. In order to implement checking capability for such *entity protocols* into the Blast model checker, the Behavior CPA would have to track the explicit state of an entity protocol separately for every instance of an entity.

7 Conclusion

We believe that combining abstraction and explicit state based model checking is a promising direction for further work in software verification. We have made another step in this direction by extending a predicate abstraction tool Blast by an explicit state representation of behavior rules specified in a simplified version of the behavior protocols formalism. Thanks to the extension, behavior rules can be verified more efficiently as was shown on an experiment.

A less significant but noteworthy contribution of this paper is the presentation of a novel use case for configurable program analysis, which was originally unanticipated by its authors. We have also proposed changes to this concept in order to allow individual CPAs to identify erroneous states during the verification.

References

1. Adamek, J., Plasil, F.: Component composition errors and update atomicity: static analysis. Journal of Software Maintenance and Evolution 17(5), 363–377 (2005)
2. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. SIGOPS Oper. Syst. Rev. 40(4), 73–85 (2006)
3. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of c programs. SIGPLAN Not. 36(5), 203–213 (2001)
4. Ball, T., Rajamani, S.K.: Slic: A specification language for interface checking. Technical Report MSR-TR-2001-21, Microsoft Research (January 2002)
5. Beyer, D., Chlipala, A., Henzinger, T., Jhala, R., Majumdar, R.: The Blast query language for software verification. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 2–18. Springer, Heidelberg (2004)
6. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: Checking memory safety with blast. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 2–18. Springer, Heidelberg (2005)
7. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)

8. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008). IEEE Computer Society Press, Los Alamitos (2008)
9. Bodden, E., Hendren, L.J., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with tracematches. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 22–37. Springer, Heidelberg (2007)
10. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
11. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 238–252. ACM, New York (1977)
13. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: SIGSOFT 2006/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 117–127. ACM, New York (2006)
14. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. SIGPLAN Not. 37(1), 58–70 (2002)
15. Holzmann, G.: The Spin Model Checker, Primer and Reference Manual. Addison-Wesley, Reading (2003)
16. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 2nd edn. Addison-Wesley, Reading (2000)
17. Pasareanu, C.S., Pelánek, R., Visser, W.: Predicate abstraction with under-approximation refinement. Logical Methods in Computer Science 3(1) (2007)
18. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Transactions on Software Engineering 28(11), 1056–1076 (2002)
19. Poch, T.: Distributed behavior protocol checker. Master's thesis, Charles University in Prague, Czech Republic (2006)
20. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering 10(2), 203–232 (2003)

Appendix

The spinlock locking rule prescribes proper alternation of calls to the functions `spin_lock_irqsave` and `spin_unlock_irqrestore`. Listing of this rule specified as behavior protocol follows:

```
( spin_lock_irqsave; spin_unlock_irqrestore )*
```

Specified in the BLAST specification language:

```
global int lockStatus = 0;
event {
```

```
  pattern { spin_lock_irqsave($?, $?); }
  guard { lockStatus == 0 }
  action { lockStatus = 1; }
}
event {
  pattern { spin_unlock_irqrestore($?, $?); }
  guard { lockStatus == 1 }
  action { lockStatus = 0; }
}
```

The DMA rule prescribes specific ordering of calls to the DMA helper functions. For example, other helper functions should be called after the function `claim_dma_lock` and before `release_dma_lock`. According to the source code comments, the `set_dma_`*xxx* and `get_dma_`*xxx* functions expect a preceding call to `clear_dma_ff`. All these function are to be called with the specific DMA channel disabled. Specification of the rule using behavior protocol follows:

```
(
  claim_dma_lock;
  (
    (
      disable_dma;
        (
          clear_dma_ff;
          (
            set_dma_mode +
            set_dma_addr +
            set_dma_count +
            get_dma_residue
          )*
        ) + NULL
      )
    )
    +
    enable_dma
  )*;
  release_dma_lock
)*
```

The DMA rule in the Blast specification language:

```
global int dmaStatus = 0;
event {
  pattern { $? = claim_dma_lock(); }
  guard { dmaStatus == 0 }
  action { dmaStatus = 1; }
}
event {
  pattern { disable_dma($?); }
```

```
  guard { dmaStatus == 1 }
  action { dmaStatus = 2; }
}
event {
  pattern { enable_dma($?); }
  guard { dmaStatus > 1}
  action { dmaStatus = 1; }
}
event {
  pattern { clear_dma_ff($?); }
  guard { dmaStatus == 2 }
  action { dmaStatus = 3; }
}
event {
  pattern { set_dma_mode($?, $?); }
  guard { dmaStatus == 3 }
}
event {
  pattern { set_dma_addr($?, $?); }
  guard { dmaStatus == 3 }
}
event {
  pattern { set_dma_count($?, $?); }
  guard { dmaStatus == 3 }
}
event {
  pattern { $? = get_dma_residue($?); }
  guard { dmaStatus == 3 }
}
event {
  pattern { release_dma_lock($?); }
  guard { dmaStatus > 0 }
  action { dmaStatus = 0; }
}
```