# Predicate Abstractions Memory Modeling Method with Separation into Disjoint Regions

[1] A. Volkov <arvolkov@inbox.ru>
[2] M. Mandrykin <mandrykin@ispras.ru>
[1] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia
[2] Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

**Abstract.** Software verification is a type of activity focused on software quality control and detection of errors in software. Static verification is verification without the execution of software source code. Special software – tools for static verification – often work with program's source code. One of the tools that can be used for static verification is a tool called CPAchecker. The problem of the current memory model used by the tool is that if a function returning a pointer to program's memory lacks a body, arbitrary assumptions can be made about this function return value in the process of verification. Although possible, the assumptions are often also practically very improbable. Their usage may lead to a false alarm. In this paper we give an overview of the approach capable of resolving this issue and its formal specification in terms of path formulas based on the uninterpreted functions used by the tool for memory modeling. We also present results of benchmarking the corresponding implementation against existing memory model.

**Keywords:** memory model; predicate abstractions; static verification

**DOI:** 10.15514/ISPRAS-2017-29(4)-13

**For citation:** Volkov A., Mandrykin M. Predicate Abstractions Memory Modeling Method with Separation into Disjoint Regions. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 4, 2017, pp. 203-216. DOI: 10.15514/ISPRAS-2017-29(4)-13

## 1. Introduction

Software verification is a type of activity focused on software quality control and detection of errors in software [1]. Static verification is a verification without the execution of software source code.

Special software – tools for static verification – often work with program's source code. Depending on the tools used for static verification it is possible to conduct analysis of the source code to search for errors in program's behavior.

One of the tools that can be used for static verification is a tool called CPAchecker. It takes program's source code as an input, creates a CFA (control-flow automaton) and uses it to run the analysis. One of the analyses the instrument is capable of is a reachability analysis. In this paper we consider reachability properties that can be expressed as checking if the call to an error function is reachable. Its strong side is that the CPA (configurable program analysis) [2] concept allows to use a composition of several analyses for program verification. The tandem of Value Analysis and Predicate Analysis produces good results in terms of verification precision / verification time ratio.

## 2. Definitions and notations

We will call a model of program's memory or just a memory model a strategy of organization and representation of program's memory. By region we will refer to the set of lvalues with the following restriction: if two lvalues are taken from two different regions they necessarily reference disjoint memory locations [3]. For example, different regions may be safely assigned to the lvalues referring distinct structure fields under the following conditions:

- the fields do not occur as an argument to the address taking operator (&);
- the fields do not become targets of some pointers by the usage of pointer type conversion or address arithmetic.

The situation when a program's error state is reachable due to the imprecisions of abstraction employed in the analysis is called a false alarm.

## 3. CPAchecker's memory model

Existing memory model employed by Predicate Analysis of the CPAchecker tool uses uninterpreted functions. Each of those functions has only a name and a number of arguments. If $f(x)$ is an uninterpreted function, $a$ and $b$ are any of its arguments for which $a = b$ is true then $f(a) = f(b)$[4]. Uninterpreted functions in the CPAchecker tool are used to establish a correspondence between a memory location and the value stored at this memory location. Depending on the type of the expression different uninterpreted functions should be used.

Existing memory model of the CPAchecker tool uses typed regions. This means that all lvalues of the same type exist in the same region. However, a large number of lvalues of the same type is present in any big enough program written in the C programming language. This leads to the addition of a big number of logical constraints for each event of a pointer's memory update. The constraints express checks for potential equality of the updated lvalue to each memory location in the region. Those checks allow to determine precisely what memory should also be updated but noticeably increase the length of path formulas.

The problem of the current memory model used by the tool is that if a function returning a pointer to program's memory lacks a body, arbitrary assumptions can be made about its return value in the process of verification. In other words, it is considered possible for this pointer to point at any lvalue in the region. Although possible, this situation is also practically very improbable. In those cases it is hard to determine if a path leading to an error label really does or doesn't exist. One of the approaches capable of resolving this issue suggests the introduction of smaller regions that divide a bigger typed region.

## 4. B&B memory model

### 4.1 Memory model overview

B&B memory model was proposed by Richard Bornat and had been based on the work of Rod Burstall [5], [6]. It is used in Frama-C verification tool in Jessie plugin which is capable of performing verification of the C programs. In its foundation are assumptions that can introduce regions of smaller sizes instead of having very big one for a type. These assumptions state that if struct data type fields never occur as arguments to the address taking operator (&) in program's source code then those fields can be placed to separate regions. Otherwise they must belong to the same region as the normal pointers of the same type.

This memory model has some flaws. It does not take into account that the struct fields can be accessed through address arithmetic and pointer conversions. It also needs mentioning that some overhead costs are required for region support. Taking into account the pros and cons of the model it is possible to say that the B&B memory model looks promising.

### 4.2 Formal specification

For ease of specification we will assume the following:

- variables can only be of struct s * types;
- struct s fields can only be of int type;
- struct s has *n* fields: struct s { int f1, f2, …, fn; };

Program's memory location can be represented by an lvalue expression like pointer dereference. To model changes to the program's state when assignments to lvalues arise the CPAchecker tool uses uninterpreted functions [4].

We assume absence of pointer arithmetic and restrict pointer dereferences to the applications of the arrow operator ($p \to f_i$), where *p* is a pointer to the struct type and $f_i$ is one of the struct fields).

Let ϒ be a set of uninterpreted functions. It consists of the uninterpreted function *G* that is used for accessing a memory location in global region, a finite number of uninterpreted functions $F_i$, where each function $F_i$ represents the state of the memory region corresponding to lvalues of the form $b \to f_i, i = \overline{1,n}$ and the uninterpreted

function *undef_ptr* with zero arity that models the usage of the program's functions returning an unknown pointer.

Let $B(e)$ be an uninterpreted function used for global memory location modeling and $B_i(e)$, $i = \overline{1,n}$ — a finite set of uninterpreted functions used for memory location modeling in regions corresponding to $F_i$ uninterpreted functions. For address representation it is suggested to use expressions like *a*, where *a* is a variable. The axioms of the memory model (positivity of addresses and their non-intersection within one region) can be represented as follows:

- $a > 0$ ;
- $B(a) = k$ , where *k* is a unique number for each such variable.

The tool uses SSA representation to model the varying state of program variables and memory regions. In this representation usage of a name splits into usages of its versions. Each time an assignment happens to a program variable or a memory region represented by the corresponding variable or uninterpreted function in the path formula, the version number (index) of that variable or an uninterpreted function increases.

Let *Index* : $\Upsilon \to \mathbb{N}$ be a mapping of a set of uninterpreted functions ϒ to a numerical set of their indices.

Let $Alloc\colon \Upsilon \to Addrs$ be a mapping of a set of uninterpreted functions ϒ to the set of subsets of memory locations $Addr$: $Addrs = 2^{Addr}$.

We will use a supplementary function *mem_upd*:

$$mem\_upd(p, f, m', m) = \bigwedge_{a \in Alloc(f)} ((p = a) \vee (f_{m'}(a) = f_m(a)))$$

that defines a check for address equality for all of the lvalues in the same region as pointer *p* (locations in the *Alloc*(*f*) region are modeled by the uninterpreted function *f*, *m* = *Index*(*f*) is a current version of *f* and *m* ' = *m* + 1 is a new version).

We define $\omega(s, f_i)$ as a constant offset of a field $f_i$ from the base address of struct type variable $s$. Because we assume that there is only one structure type struct $s$ in our programs, $\omega(s, f_i)$ can be made just $\omega(f_i)$.

In B&B memory model implemented on top of CPAchecker's existing memory model the operator of a strongest post-condition is defined as $SP(op(\phi)) = \phi \wedge \Gamma(op)$, where $\phi$ is a symbolic abstract state and constraints $\Gamma(op)$ are defined by table 1.

### 4.3 Example

The following program will be considered correct if we use either of the memory models. Γ constraints in terms of B&B memory model for the program are shown in table 2. Path formula can be made as a conjunction of all formulas in Γ column of the table 2. It is *unsat* in terms of either of the memory models. This means that the tool cannot go by this path (i.e. won't consider it as a potential error trace candidate).

```
struct s { int f1, f2; };
struct s * p1;
struct s * p2;
p1 = alloc();
p2 = alloc();
p1 -> f1 = 6;
p2 -> f2 = 5;
assume(p1 -> f1 == p2 -> f2);
```

*Table 1. Γ constraints creation rules*

| Operation ($op$) | $Index$ | $Alloc$ | Base address index $k'$ | Γ constraints |
|---|---|---|---|---|
| Variable allocation on stack<br>struct s * p; | No changes | $A'$ - new variable,<br>$Alloc'(G) = A' \cup Alloc(G)$ | $k'$ - new index | $p = A' \wedge A' > 0 \wedge B(A') = k'$ |
| Heap variable allocation<br>p = alloc() | $l'$ - new index for $G$,<br>$l = Index(G)$,<br>$Index' = Index \setminus \{G \to l\} \cup \{G \to l'\}$ | $A', A'_i$ - new variables,<br>$Alloc'(G) = A \cup Alloc(G)$<br>$Alloc'(F^i) = A'_i \cup Alloc(F^i), i = \overline{1,n}$ | $k', k'_i$ - new indices, $i = \overline{1,n}$ | $G_{l'}(p) = A' \wedge A' > 0 \wedge B(A') = k'$<br>$\wedge\ mem\_upd(p,G,l,l')$<br>$\bigwedge_{i=\overline{1,n}}((G_{l'}(p)+\omega(f_i)) = A'_i \wedge A'_i > 0 \wedge B^i(A'_i) = k'_i)$ |
| p=*undef_ptr*() | $l'$ - new index for $G$,<br>$l = Index(G)$,<br>$m'$ - new index for *undef_ptr*,<br>$m = Index(undef\_ptr)$,<br>$Index' = Index \setminus (\{G \to l\} \cup \{undef\_ptr \to m\}) \cup \{G \to l'\} \cup \{undef\_ptr \to m'\}$ | No changes | No changes | $G_{l'}(p) = undef\_ptr_m \wedge mem\_upd(p,G,l,l')$ |
| p→ $f_i$ = e | $m'$ - new index for $F^i$, $m = Index(F^i)$,<br>$Index' = Index \setminus \{F^i \to m\} \cup \{F^i \to m'\}$ | No changes | No changes | $F^i_{m'}(G_l(p) + \omega(f_i)) = \Gamma(e) \wedge mem\_upd(G_l(p) + \omega(f_i), F^i, m', m)$, where $l = Index(G)$ and $\Gamma(e)$ can be computed using the following rules:<br>$\Gamma(const) : const;$<br>$\Gamma(p2 \to f_j) :\ F^j_k(G_l(p2) + \omega(f_j))$, where $k = Index(F^j)$, $l = Index(G)$;<br>$\Gamma(e_1\ op\ e_2)$, $op \in \{‘+’, ‘-’, ‘*’, ‘\backslash’\}$:<br>$\Gamma(e_1)\ op\ \Gamma(e_2)$. |
| assume(p) | No changes | No changes | No changes | $\Gamma(p)$ for predicate $p$ can be computed as following:<br>$\Gamma(const) : const$;<br>$\Gamma(s) : G_l(s)$, where $l = Index(G)$;<br>$\Gamma(s \to f_i) : F^i_m(G_l(s) + \omega(f_i))$, where $m = Index(F^i)$, $l = Index(G)$;<br>$\Gamma(p1 == p2) : \Gamma(p1) == \Gamma(p2)$;<br>$\Gamma(p1 < p2) : \Gamma(p1) < \Gamma(p2)$;<br>$\Gamma(p1 <= p2) : \Gamma(p1) \leq \Gamma(p2)$;<br>$\Gamma(p1 \parallel p2) : \Gamma(p1) \vee \Gamma(p2)$;<br>$\Gamma(p1\ \&\&\ p2) : \Gamma(p1) \wedge \Gamma(p2)$;<br>$\Gamma(!p) : \neg\Gamma(p)$. |

*Table 2. Example build of path formula for the correct program*

| Path instruction | *Index* | *Alloc* | $k'$ | Γ |
|---|---|---|---|---|
| struct s * p1; | {G→1, $F^1$ →1, $F^2$ →1} | $Alloc$(G)={$A_1$} | 1 | $p1 = A_1 \wedge A_1 > 0 \wedge B(A_1) = 1$ |
| struct s * p2; | {G→1, $F^1$ →1, $F^2$ →1} | $Alloc$(G)={$A_1$, $A_2$} | 2 | $p2 = A_2 \wedge A_2 > 0 \wedge B(A_2) = 2$ |
| p1 = alloc(); | {G→2, $F^1$ →1, $F^2$ →1} | $Alloc$(G)={$A_1$, $A_2$, $A_3$} $Alloc(F^1) = \{A_4\}$ $Alloc(F^2) = \{A_5\}$ | 3,4,5 | $G_2(p1) = A_3 \wedge A_3 > 0 \wedge B(A_3) = 3$ $\wedge (G_2(p1)+\omega(f_1)) = A_4 \wedge A_4 > 0 \wedge B(A_4) = 4$ $\wedge (G_2(p1)+\omega(f_2)) = A_5 \wedge A_5 > 0 \wedge B(A_5) = 5$ |
| p2 = alloc(); | {G→3, $F^1$ →1, $F^2$ →1} | $Alloc$(G)={$A_1$, $A_2$, $A_3$,$A_6$} $Alloc(F^1) = \{A_4, A_7\}$ $Alloc(F^2) = \{A_5, A_8\}$ | 6,7,8 | $G_3(p2) = A_6 \wedge A_6 > 0 \wedge B(A_6) = 6$ $\wedge (G_3(p2)+\omega(f_1)) = A_7 \wedge A_7 > 0 \wedge B(A_7) = 7$ $\wedge (G_3(p2)+\omega(f_2)) = A_8 \wedge A_8 > 0 \wedge B(A_8) = 8$ |
| p1→f1 = 6; | {G→3, $F^1$ →2, $F^2$ →1} | $Alloc$(G)={$A_1$, $A_2$, $A_3$,$A_6$} $Alloc(F^1) = \{A_4, A_7\}$ $Alloc(F^2) = \{A_5, A_8\}$ | 8 | $F_2^1(G_3(p1)+\omega(f_1)) = 6$ $\wedge\ mem\_upd(G_3(p1)+\omega(f_1), F^1, 2, 1)$ |
| p2→f2 = 5; | {G→3, $F^1$ →2, $F^2$ →2} | $Alloc$(G)={$A_1$, $A_2$, $A_3$,$A_6$} $Alloc(F^1) = \{A_4, A_7\}$ $Alloc(F^2) = \{A_5, A_8\}$ | 8 | $F_2^2(G_3(p2)+\omega(f_2)) = 6$ $\wedge\ mem\_upd(G_3(p2)+\omega(f_2), F^2, 2, 1)$ |
| assume(p1→f1 == p2→f2) | {G→3, $F^1$ →2, $F^2$ →2} | $Alloc$(G)={$A_1$, $A_2$, $A_3$,$A_6$} $Alloc(F^1) = \{A_4, A_7\}$ $Alloc(F^2) = \{A_5, A_8\}$ | 8 | $F_2^1(G_3(p1)+\omega(f_1)) = F_2^2(G_3(p2)+\omega(f_2))$ |

Why the conjunction is *unsat*?

1) In the existing memory model memory allocated for pointers $p1$ and $p2$ cannot intersect because it was allocated using the known $alloc()$ function (the corresponding path formula is not given).
2) In the given Γ constraints for this path (using the B&B model) the following contradicting elements are present:
   - $F_2^1(G_3(p1) + \omega(f1)) = F_2^2(G_3(p2) + \omega(f2))$;
   - $F_2^1(G_3(p1) + \omega(f1)) = 5$;
   - $F_2^2(G_3(p2) + \omega(f2)) = 6$.

Let's take a look at the example program below. In the program's source code there are calls to the function *undef_ptr*() that returns an unknown pointer. The pointer $p2$ is initialized using this function. Γ constraints in terms of B&B memory model for the program are shown in table 3. Path formula can be made as conjunction of all formulas in Γ column of the table 3.

```
void * undef_ptr();
struct s { int f1, f2; };
struct s * p1;
struct s * p2;
p1 = alloc();
p2 = undef_ptr();
p1 -> f1 = 6;
p2 -> f2 = 5;
assume(p1 -> f1 == p2 -> f2);
```

In B&B memory model $p1 \rightarrow f1$ and $p2 \rightarrow f2$ exist in the separate memory regions. In Γ constraints for this path the same contradicting elements as for the previous example are present. Thus, the update of one of them wouldn't affect the other one. Because of that the result of verification would be that the error state is unreachable (path formula is still *unsat*).

However, in the existing memory model fields $f1$ and $f2$ of struct *s* exist in the same memory region and it uses only one uninterpreted function for them (see table 2 in [4]). Memory for their base pointers $p1$ and $p2$ was allocated using known *alloc*() function and function *undef_ptr*() returning unknown pointer respectively. It cannot be confirmed that an update to a field $f2$ of the $p2$ wouldn't affect the access to the $f1$ struct field of $p1$. In the formula the location for field $f2$ of the $p2$ is $(G_3(p2) + \omega(f2))$ which is $undef\_ptr_1 + \omega(f2)$. Locations $(G_3(p1) + \omega(f1))$ and $(G_3(p2) + \omega(f2))$ exist in the same region and may be equal. Thus the formula is satisfiable. It means that the result of verification with existing memory model will be a reachable path to the program's error state.

Usually such situations in practice are false alarms because different fields of different structures do not normally intersect. Thus, the assumptions related to this behavior in the existing memory model aren't really incorrect but they are quite improbable in practice. Usage of the B&B memory model will be able to reduce the number of false alarms caused by these assumptions (continued in section 6).

*Table 3. Example build of path formula for the program with unknown memory function*

| Path instruction | *Index* | *Alloc* | $k'$ | Γ |
|---|---|---|---|---|
| struct s * p1; | {G→1,<br>$F^1$ →1,<br>$F^2$ →1,<br>*undef_ptr*→1} | $Alloc$(G) ={$A_1$} | 1 | $p1 = A_1 \wedge A_1 > 0 \wedge B(A_1) = 1$ |
| struct s * p2; | {G→1,<br>$F^1$ →1,<br>$F^2$ →1,<br>*undef_ptr*→1} | $Alloc$(G) ={$A_1$, $A_2$} | 2 | $p2 = A_2 \wedge A_2 > 0 \wedge B(A_2) = 2$ |
| p1 = alloc(); | {G→2,<br>$F^1$ →1,<br>$F^2$ →1,<br>*undef_ptr*→1} | $Alloc$(G) ={$A_1$, $A_2$, $A_3$}<br>$Alloc(F^1)$ = {$A_4$}<br>$Alloc(F^2)$ = {$A_5$} | 3,4,5 | $G_2(p1) = A_3 \wedge A_3 > 0 \wedge B(A_3) = 3$<br>$\wedge (G_2(p1)+\omega(f_1)) = A_4 \wedge A_4 > 0 \wedge B(A_4) = 4$<br>$\wedge (G_2(p1)+\omega(f_2)) = A_5 \wedge A_5 > 0 \wedge B(A_5) = 5$ |
| p2=*undef_ptr*(); | {G→3,<br>$F^1$ →1,<br>$F^2$ →1,<br>*undef_ptr*→2} | $Alloc$(G) ={$A_1$, $A_2$, $A_3$}<br>$Alloc(F^1)$ = {$A_4$}<br>$Alloc(F^2)$ = {$A_5$} | 5 | $G_3(p2) = undef\_ptr_1$<br>$\wedge\ mem\_upd(p2,G,3,2)$ |
| p1→f1 = 6; | {G→3,<br>$F^1$ →2,<br>$F^2$ →1,<br>*undef_ptr*→2} | $Alloc$(G) ={$A_1$, $A_2$, $A_3$}<br>$Alloc(F^1)$ = {$A_4$}<br>$Alloc(F^2)$ = {$A_5$} | 5 | $F_2^1(G_3(p1)+\omega(f_1)) = 6$<br>$\wedge$<br>$mem\_upd(G_3(p1)+\omega(f_1), F^1,2,1)$ |
| p2→f2 = 5; | {G→3,<br>$F^1$ →2,<br>$F^2$ →2,<br>*undef_ptr*→2} | $Alloc$(G) ={$A_1$, $A_2$, $A_3$}<br>$Alloc(F^1)$ = {$A_4$}<br>$Alloc(F^2)$ = {$A_5$} | 5 | $F_2^2(G_3(p2)+\omega(f_2)) = 6$<br>$\wedge$<br>$mem\_upd(G_3(p2)+\omega(f_2), F^2,2,1)$ |
| assume(p1→f1 == p2→f2) | {G→3,<br>$F^1$ →2,<br>$F^2$ →2,<br>*undef_ptr*→2} | $Alloc$(G) ={$A_1$, $A_2$, $A_3$}<br>$Alloc(F^1)$ = {$A_4$}<br>$Alloc(F^2)$ = {$A_5$} | 5 | $F_2^1(G_3(p1)+\omega(f_1)) = F_2^2(G_3(p2)+\omega(f_2))$ |

## 5. Implementation notes

The creation of memory regions is an automated process. In CPAchecker verification tool CFA (control-flow automaton) is used as an inner representation of the program. It is sufficient to go through it and find in it all of the struct field accesses. This allows to distinguish those fields that don't have their address taken somewhere in the program.

In the implementation we do not take into consideration the possibility of field accesses through pointer arithmetic and through the usage of pointer conversions because of the high improbability of such field accesses in program's source code.

## 6. Experiments

To determine the efficiency of B&B memory model implementation in comparison to existing memory model of the CPAchecker tool a number of launches were performed on the predefined sets of Linux kernel modules. To use the implemented memory model one must have:

- CPAchecker verification tool with revision number 23271 or higher from the branch trunk;
- option cpa.predicate.useMemoryRegions should be set to 'true'.
- The following experiments were made using the revision *trunk:*23271 of the tool.

### 6.1 False alarm set

The review of error traces obtained during the verification of Linux kernel 3.14 allowed to determine situations when reachability of error state was present due to updates to same-typed pointers' memory. This set consists of those 26 kernel modules that caused false alarms due to the updates to pointer's memory. The goal of this experiment was to find out what effect the usage of B&B memory model will have on the tools precision. Tables 4 and 5 hold information about changes of the tool's verdicts.

*Table 4. B&B applicability*

| | B&B could help | B&B could not help |
|---|---|---|
| B&B helped | 10 | 0 |
| B&B did not help | 0 | 16 |

*Table 5. Verdict changes*

| False alarm → Safe | False alarm → Unsafe | False alarm → False alarm* |
|---|---|---|
| 3 | 5 | 2 |

* - different error trace and cause of Unsafe

## 6.2 Linux 4.2-rc1 kernel modules

A set of Linux kernel drives (version 4.2-rc1) was selected to study the efficiency of B&B memory model implementation in comparison to the existing memory model of the CPAchecker tool.

The launch was performed for rule that checks correctness of functions working with `usb_get_*` and `usb_put_*` functions of `usb-system`. Launch results can be found in tables 6, 7.

Launch configuration:

- time limit – 15 minutes;
- memory limit – 15 Gb;
- number of CPU cores – 4;

The differences in the regions the models have led to the difference in program's paths that are covered by the tool. This explains Unsafe → Unknown, Unknown → Safe and Unknown → Unsafe transitions, where Safe means that program's error state is unreachable, Unsafe – error state is reachable, Unknown – timeout or runtime error. This experiment's results show that the improvement to the tool's precision is present while the verification speed remains competitive.

*Table 6. Linux 4.2-rc1 statistics*

| | Existing model | B&B |
|---|---|---|
| Verification time | 35.8 hours | 35.3 hours |
| Safe | 4245 | 4241 |
| Unsafe | 69 | 68 |
| Unknown | 161 | 166 |

*Table 7. Transitions*

| Existing model \ B&B model | Safe | Unsafe | Unknown |
|---|---|---|---|
| Safe | 4240 | 0 | 5 |
| Unsafe | 0 | 67 | 2 |
| Unknown | 1 | 1 | 159 |

## 6.3 SV-COMP'17 DeviceDrivers64

This set contains files from the DeviceDrivers64 set of the international competition on software verification SV-COMP'17. It consists of 2795 modules of different Linux kernel versions. Launch results can be found in tables 8, 9, 10.

Launch configuration:

- time limit – 15 minutes;
- memory limit – 15 Gb;

- number of CPU cores – 4;

Several of the transitions from the incorrect results can be explained by the difference in models' choice of pointer's may-aliases. The same modules were present in the earlier mentioned False alarm set. Several transitions to Unknown can be explained by the additional overhead costs required for B&B usage to the verification tasks on the verge of timeout.

*Table 8. DeviceDrivers64 statistics*

| Memory models | Existing | B&B |
|---|---|---|
| Total number of files | 2795 | 2795 |
| Correct results | 1791 | 1780 |
| Error state unreachable | 1524 | 1522 |
| Error state reachable | 267 | 258 |
| Incorrect results | 7 | 5 |
| Missed errors | 4 | 4 |
| False alarms | 3 | 1 |
| Unknown | 997 | 1010 |

*Table 9. Time for DeviceDrivers64 set*

| Memory models | Existing | B&B |
|---|---|---|
| Total time | 143.6 hours | 143.1 hours |
| Time for correct results | 14.9 hours | 14.1 hours |
| SMT solver time | 10500 sec (2.9 hours) | 12400 sec (3.4 hours) |
| SMT solver time for correct results | 660 sec | 605 sec |

*Table 10. Transitions*

| Existing model \ B&B model | Correct results | Incorrect results | Unknown |
|---|---|---|---|
| Correct results | 1775 | 0 | 16 |
| Incorrect results | 2 | 5 | 0 |
| Unknown | 3 | 0 | 994 |

## *7. Conclusion*

This paper proposes the specification of B&B memory model and its region-based reasoning in terms of uninterpreted functions. Its implementation on top of existing memory model of the CPAchecker verification tool provides better verification precision while the verification speed remains competitive. The implementation was included in the official repository of the CPAchecker static verification tool.

## References

1. V. Kuliamin, “Software verification methods,” Vserossiiskii konkursnyi otbor obzornoanaliticheskikh statei po prioritetnomu napravleniyu ”Informatsionnotelekommunikatsionnye sistemy” [Russian national competitive selection of review and analytical articles in priority direction ”Information and telecommunication systems”], 117 p., 2008 (in Russian).
2. D. Beyer, T. A. Henzinger, and G. Theoduloz, “Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis,” in Computer Aided Verification, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 504–518.
3. M. Mandrykin and A. Khoroshilov, “A memory model for deductively verifying Linux kernel modules,” A.P. Ershov Informatics Conference, the PSI Conference Series, 11th edition, 2017 (to appear).
4. M. Mandrykin and V. Mutilin, “Modeling Memory with Uninterpreted Functions for Predicate Abstractions,” Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 117–142 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-7
5. R. Bornat, “Proving pointer programs in Hoare Logic,” in Mathematics of Program Construction: 5th International Conference, MPC 2000, ser. Lecture Notes in Computer Science, R. Backhouse and J. N. Oliveira, Eds., vol. 1837. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 102–126.
6. R. Burstall, “Some techniques for proving correctness of programs which alter data structures,” Machine Intelligence, vol. 7, pp. 23–50, 1972.

## Метод моделирования памяти в предикатных абстракциях с разделением на непересекающиеся области

[1] *А.Р. Волков <arvolkov@inbox.ru>*
[2] *М.У. Мандрыкин <mandrykin@ispras.ru>*
[1] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1.*
[2] *Институт системного программирования РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*

**Аннотация.** Верификация программного обеспечения — вид деятельности, направленный на контроль качества программного обеспечения и обнаружения ошибок в нем. Статическая верификация - это один из способов верификации, который производится без выполнения исходного кода программы. Для статической верификации используется специальное программное обеспечение: инструменты статической верификации, которые часто работают с исходным кодом программы. Одним из таких инструментов является инструмент под названием CPAchecker. Проблема его текущей модели памяти заключается в том, что при встрече функции, возвращающей указатель на область памяти, у которой отсутствует тело, в процессе

верификации о ее возвращаемом значении могут быть сделаны произвольные предположения. Несмотря на то, что они теоретически допустимы, вероятность их выполнения на практике очень низка. Использование этих предположений может привести к ложному предупреждению в качестве результата верификации. В данной статье мы делаем обзор на один из подходов, благодаря которому можно избавиться от такой проблемы, а также предлагаем формальное описание данного подхода в терминах формул путей, содержащих неинтерпретируемые функции, которые инструмент использует для моделирования памяти программы. Также мы приводим результаты сравнительного анализа эффективности предложенной реализации относительно существующей модели памяти.

**Ключевые слова:** модель памяти; предикатные абстракции; статическая верификация.

**DOI:** 10.15514/ISPRAS-2017-29(4)-13

**Для цитирования:** Волков А.Р., Мандрыкин М.У. Метод моделирования памяти в предикаты абстракциях с разделением на непересекающиеся области. *Труды ИСП РАН*, том 29, вып. 4, 2017 г., стр. 203-216 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(4)-13

## Список литературы

1. В.В. Кулямин, “Методы верификации программного обеспечения” Всероссийский конкурсный отбор обзорно-аналитических статей по приоритетному направлению "Информационно-телекоммуникационные системы", 117 стр., 2008.
2. D. Beyer, T. A. Henzinger, and G. Theoduloz, “Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis,” in Computer Aided Verification, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 504–518.
3. M. Mandrykin and A. Khoroshilov, “A memory model for deductively verifying Linux kernel modules,” A.P. Ershov Informatics Conference, the PSI Conference Series, 11th edition, 2017 (to appear).
4. Мандрыкин М.У. и Мутилин В.С., “Моделирование памяти с использованием неинтерпретируемых функций в предикатных абстракциях” Труды ИСП РАН, том 27, вып. 5, 2015, стр. 117–142. DOI: 10.15514/ISPRAS-2015-27(5)-7
5. R. Bornat, “Proving pointer programs in Hoare Logic,” in Mathematics of Program Construction: 5th International Conference, MPC 2000, ser. Lecture Notes in Computer Science, R. Backhouse and J. N. Oliveira, Eds., vol. 1837. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 102–126.
6. R. Burstall, “Some techniques for proving correctness of programs which alter data structures,” Machine Intelligence, vol. 7, pp. 23–50, 1972.