ISSN 0361-7688, Programming and Computer Software, 2020, Vol. 46, No. 8, pp. 712–730. © Pleiades Publishing, Ltd., 2020.

Analysis of Correct Synchronization of Operating System Components

P. S. Andrianov

Ivannikov Institute for System Programming of the RAS, Moscow, Russia

e-mail: andrianov@ispras.ru

Received February 13, 2020; revised March 10, 2020; accepted April 2, 2020

Abstract—Most of the software model checker tools do not scale well on complicated software. Our goal was to develop a tool, which provides an adjustable balance between precise and slow software model checkers and fast and imprecise static analyzers. The key idea of the approach is an abstraction over the precise thread interaction and analysis for each thread in a separate way, but together with a specific environment, which models effects of other threads. The environment contains a description of potential actions over the shared data and synchronization primitives, and conditions for its application. Adjusting the precision of the environment, one can achieve a required balance between speed and precision of the complete analysis. A formal description of the suggested approach was performed within a Configurable Program Analysis theory. It allows formulating assumptions and proving the soundness of the approach under the assumptions. For efficient data race detection we use a specific memory model, which allows to distinguish memory domains into the disjoint set of regions, which correspond to a data types. An implementation of the suggested approach into the CPAchecker framework allows reusing an existed approaches with minimal changes. Implementation of additional techniques according to the extended theory allows to increase the precision of the analysis. Results of the evaluation allow confirming scalability and practical usability of the approach.

DOI: 10.1134/S0361768820080022

1. INTRODUCTION

Verification of a multithreaded program is always a much more complicated task then verification of sequential program. Precise computation of all possible interleavings leads to a state explosion. Thus, most of the verification tools try to perform different kinds of optimizations: partial order reduction [1, 2], counter abstraction [3] and others. Anyway, most state-of-the-art tools do not scale well on real-world software. That is confirmed by the software verification competition [4]. Concurrency benchmarks based on Linux device drivers[1] cause significant difficulties for any software model checker tool.

The other side of model checking is static analysis that is targeted on fast finding bugs without any confidence in the final verdict. Such tools apply different filters and unsound heuristics to speed up the analysis and thus can not prove the correctness. Our goal was to develop a tool, which becomes a golden mean between precise and slow software model checkers and fast and imprecise static analyzers. It should be easily configured and targeted to a particular task.

The main idea of the suggested approach is following. As the verification object is a large multithreaded program, we do not consider all possible thread interleavings and consider every thread separately. In this case a state of every thread becomes *partial*, i.e. they do not contain information about other threads and, therefore, can not describe a complete state of the program. A possible thread interaction is over approximated by a set of actions, which the threads can perform over the set of shared data, including synchronization primitives. Thus, an approximation of possible actions, or *effects* is constructed simultaneously for all threads. We will call it an *environment*. Though the environment is single for all threads, it does not mean that all its effects could be applied to a thread, as for every effect there are conditions of its application. The conditions depends on the analysis. Moreover, the conditions may include requirements on certain operators of a program or thread states.

Precision of the environment defines precision and speed of the whole tool. The precision may be increased by combining different verification techniques. An implementation was performed on the top of CPAchecker [5] framework, which provides a rich set of verification approaches. A thread-modular approach [6–9] can also be implemented as a one more technique, which will be integrated into CPAchecker framework, appending the traditional set of techniques, i.e. CEGAR algorithm [10] and predicate abstraction [11].

Software model checkers usually have the following two stages:

(1) Constructing of a set of reachable states.

[1] sv-benchmarks/c/ldv-linux-3.14-races/ directory

(2) Check a property over the set of the reached states.

The two steps may be performed sequentially or in parallel and even the reached set construction may be driven by the checked property. For example, some static verification tools are checked a potential race condition while adding a following reached state. Such a tool may stop the analysis in case of detecting an error. This approach is slow and not practical for data race detection, but it is successfully used for solving a reachability task or detecting memory leaks.

Data race detection approaches usually are based on a Lockset algorithm [12]. We use a more intelligent way to detect data races: a potential data race is a pair of two *compatible* abstract transitions, which modifies the same memory. Compatibility here means that two partial abstract transitions may start from a single global state. Thus, the precision of such definition corresponds to a level of abstraction. So, a lock abstraction leads to a classical Lockset algorithm. Predicate analysis with BnB memory model [13–15] improves the precision for dealing with pointer accesses and allows to keep soundness under reasonable assumptions.

We evaluate our approach on a set of benchmarks, based on Linux device drivers. They are prepared by Klever, a framework for verification of large software systems [16, 17] which divides a large codebase into separate verification tasks (usually, one or two Linux modules) and prepares an environment model.

Our contribution is:

– CPA with transition abstractions – an extension of CPA theory to be able to describe thread-modular approach;

– Thread Modular CPA – an implementation of thread-modular approach in terms of extended CPA theory, which allows to combine a thread-modular approach with other approaches, like predicate abstraction;

– a tool, which was successfully evaluated on benchmarks, based on Linux device drives.

The rest of the paper is organized as follows. In Section 2 we present challenges for the state-of-the-art verifiers and our high-level solutions. Section 3. Section 4 introduce a program model and basic definitions. The next 4 sections are related to the efficient computation of abstraction of the program. Section 5 describes an extension of CPA theory. Section 6 presents a thread modular analysis in terms of CPA. Sections 7–12 contain extended CPAs for thread modular approach: predicate analysis and lock analysis. In Section 13 we discuss some specifics of data race detection with our approach. Section 14 presents the results of our approach on two benchmark sets: software verification competition (SV-COMP) and Linux device drivers. Section 15 gives a brief overview of similar works.

2. MOTIVATING EXAMPLE

Consider an example[2] from SV-COMP'19 [4]. The verification task is based on a real data race[3]. The task is more than 7 kLOC and contains 4 artificially created threads: one thread for basic platform callbacks, one for interrupt handling, one for power management operations of the driver and one main thread, which performs init-exit operations. All kernel mutexes and spinlocks are replaced by a pthread_mutexes. The known data race is encoded as a reachability task in a following way:

```
tmp = tspi->rst;
assert(tmp == tspi->rst);
```

The detailed results may be found on the SV-COMP web-site[4]. Most of the state-of-the-art verifiers faced with the problems:

– CBMC: "pointer handling for concurrency is unsound – UNKNOWN";

– CPAchecker: "Unsupported feature: BDD-analysis does not support arrays";

– SMACK: "Exception thrown in lockpwn";

– yogar-cbmc: "out of memory";

– Ultimate: "Ultimate could not prove your program".

The main challenge for a verifier in the example is a large number of operations in all threads, many of them are over the same data, which means they affect each other. It leads to the set of following subproblems, which are usually ignored for small artificial examples:

(1) Analysis of multithreaded programs should be precise enough to produce a small number of false alarms, but efficient enough to be able to solve the task. Many efficient analyses do not support complicated data structures (e.g. BDD analysis [18], analysis of explicit values [19]). And vice versa precise approaches have problems with long paths (e.g. Predicate analysis [20], BMC approach [21]).

(2) Efficient encoding of synchronization primitives. Many approaches encode locks as variables, which are simultaneously checked and assigned (e.g. [7, 9]). This encoding is mixed with other variables and complicates the general analysis.

Note, the race condition in the benchmarks is encoded as reachability task, thus it is a hint for verifier, which memory accesses are buggy. In practice a verifier does not know a precise location of the data race, thus it has to check all potential memory accesses. This complicates the task much more. That is why we made efforts to develop an approach, which will be able to solve the corresponding task.

[2] https://github.com/sosy-lab/sv-benchmarks.git, sv-benchmarks/c/ldv-linux-3.14-races/linux-3.14–drivers–spi–spi-tegra20-slink.ko.cil.i

[3] https://patchwork.kernel.org/patch/9915305/

[4] https://sv-comp.sosy-lab.org/2019/results/results-verified/META_ConcurrencySafety.table.html

```
volatile int g = 0;
volatile int d = 0;
Thread1 {
   g = 1;
   d = 1;
   ...
}
Thread2 {
   if (d==1) {
     g = 2;
   }
}
```

Fig. 1. An example of code.

3. OVERVIEW OF THE APPROACH

As we have already discussed, a data race detection method may be divided into two stages: construction of a set of reached states and checking of requirements, particularly, searching of pairwise states, which form the data race. An analysis of a program may be performed by a sequential combination of the phases or their parallel execution. Note, the requirement check usually takes little time, as it represented by a condition over a state. An example of check may be absence of states of a special kinds (reachability task), absence of special pair of states (data race detection task). Construction of the set of reached states usually leads to problems related to efficiency. Thus, further we will concentrate of this task.

Consider a simple program, which has only two threads (Fig. 1). This is a model example, which contains an implicit synchronization between threads: the first thread initializes some data (in this case, a global variable g), then sets a flag, encoding, that the data are ready. The second thread can read the data only after the flag was set, thus there is no data race in this example (Fig. 2).

Classical methods of model checking have to check all possible interleavings of two threads. A number of states grows rapidly even in a simple example and with different optimizations (Fig. 3). And so-called "combinatorial explosion" happens, which leads to resource exhaustion. Thus, classical methods of model checking can not prove the correctness of large real-world software.

Simple methods of static analysis try to compute a quick overapproximation of thread interaction, so-called thread effect. However, they are not able to consider complicated dependencies between shared variables. So, for the example in Fig. 1 there is only information, that the global variable d may be set into one. With such an approximation the simple kind of analysis has to conclude, that there is a potential data race in the example.

A suggested approach is based on a well known thread-modular approach. The approach considers

```
Thread1 Thread2
 g = 1;
 d = 1;
        [d == 1]
         g = 2;
```

Fig. 2. An example of possible thread execution.

each thread separately but in a special environment, which is constructed also during the analysis. The environment computation is based on the analysis of all threads, as every thread is a part of the environment for other threads. For each thread, a set of its actions, which may affect other threads, is collected. The actions include modification of shared variables, acquiring synchronization primitives and so on. Precision of the environment strongly affects the precision of the whole analysis. However, there is the main question, how to compute and represent the environment efficiently.

In sequential analysis, there is a successful technique, which allows reducing the number of considered program states – an abstraction. It allows to abstract from minor details of a program and considers general (abstract) state. Each abstract state may correspond to a set of real (concrete) program states. This idea allows to significantly increase the efficiency of the analysis.

The key idea of the suggested approach is an extension of abstraction not only to the program states but also to the operations of a thread. Adjusting the level of the abstraction, it is possible to choose a balance between speed and precision of the tool.

Figure 4 shows a part of the Abstract Reachability Graph (ARG) for the first and the second threads from the example (Fig. 1). There is no interaction considered, so this is not a final step of the analysis. The transitions are based on a simple Value Analysis [19], which tracks only explicit values of variables. A transition contains an abstract state and an abstract operation. The first abstract state in both threads contains information that both global variables (d and g) are equal to zero. After performing an operation $g = 1$ (transition #A1) the value of g is updated into 1 in the second abstract state (transition #A2).

After constructing an ARG for two threads separately, we need to consider the influence of threads to each other, i.e. to construct an environment. For every thread operation, we compute its *projection* – a representation of operation in a thread for other threads as an environment. For example, modification of local variables can not affect on other threads, so the corresponding projection is empty. Modification of a global variable may affect other threads, so the projection may be equal to the original transition or overapproximate it, for example, by abstraction from a precise assigned value. A projection may contain not only

information about an action but also a condition for performing this action, so-called *guard*. Consider a transition #A1. We may represent the corresponding projection in the following way: if the value of g equals zero, it may be changed to one. In other words, the projection consists from two parts: a *guard* ($[g == 0]$) and an *action* ($g \to 1$). The guard corresponds to a predecessor abstract state and an action corresponds to an operation.

Let us return to the Example 4. After computation of transitions in threads separately, we have to construct the environment, i.e. set of projections. Figure 5 presents computed projections for the first thread.

There are two transitions, which may affect the global variables: #A1 and #A2. We compute the corresponding projections: #P1 and #P2. Then every projection has to affect the second thread, i.e. apply to all possible (according to the guard) transitions of the second thread. We apply the projection #P1 to the transition #B1, as the state is *compatible* with the guard of the projection. Only after that, we may apply the projection #P2 to the new transition #B2, which requires g to be equal to one. And only then the second thread may go throw a new transition #B3, which discovers new paths. Note, the figure presents only projections for the first thread, in complete ARG there should be also projections for the second thread as well.

For data race detection we have to find two transitions, which modify the same variable. The example has potential candidates: #A1 and #B4. Now, we should check, if the two transitions may be executed in parallel. That means, that the corresponding abstract states must be a part of one global state, i.e. they must be *compatible*. In this case, the partial states are contradicting each other, as one has $g \to 0$ and the other $g \to 1$. So, the corresponding transitions can not be executed simultaneously. Thus, we conclude there is no data race for g. Note, there is a data race for d (transitions #A2 and #B2), but it may be considered as lock-free synchronization, and a potential race is a part of its implementation.

The suggested approach provides a lot of possible options and configurations for targeting to a particular task. A projection may be represented by more or less precise abstraction. Several projections may be joined altogether or considered separately. An example of more abstract transitions is presented in Fig. 6.

The two initial projections (#P1 and #P2) are joined into a single one (#P3). Usually, it leads to losing some information, for example, here we lost a precise value of variable g. The action of projection also becomes more complicated. The second thread can not identify a precise value of the variables, as both of the variables are now equal to zero or one. The simple kind of analysis operates only with single explicit values of variables and both variables are considered to be equal to any random value. Now, transitions #A1 and #C3 become compatible, which means, that the race is reported on variable g.

The level of abstraction strongly affects the precision of the analysis and its speed, but the analysis always remains sound.

4. PRELIMINARY DEFINITIONS

In this section, we present preliminary definitions of a parallel program and reachable concrete states in the program.

We restrict the presentation to a simple imperative programming language, where all operations are assignments, assumptions, acquire/release synchronization operations and thread creates. We denote all operations in a program as *Ops*.

A parallel program is represented by a Control Flow Automaton (CFA) [22], which consists of a set L of program locations (modeled by a program counter, *pc*), and a set $G \subseteq L \times Ops \times L$ of control-flow edges (modeling the operation that is executed when control flows from one program location to another). There is a thread create operation in *Ops* which creates a thread with an identifier from the set T and the thread starts from some location in L. The set of program variables that occur in assignment and assumption operations from *Ops* is denoted by X having values from $\mathbb{Z}$. The parts of X, containing local and global variables, are denoted by X^{local} and X^{global} respectively. Acquire/release operations are defined over a set of lock variables S having values from $T \cup \{\perp_T\}$, where $t \in T$ means that the lock is acquired by a thread t and $\perp_T$ means that the lock is not acquired.

A *concrete state* of a parallel program is a quadruple of (c_{pc}, c_l, c_g, c_s), where

(1) A mapping $c_{pc} : T \to L$ is a partial function from thread identifiers to locations.

(2) A mapping $c_l : T \to C^{local}$ is a partial function from thread identifiers to assignments to local variables C^{local}, where $C^{local} : X^{local} \to \mathbb{Z}$ assigns each local variable its value.

(3) $c_g : X^{global} \to \mathbb{Z}$ is an assignment of values to global variables.

(4) $c_s : S \to T \cup \{\perp_T\}$ is an assignment of values to lock variables.

A set of all concrete states of a program is denoted by C.

We define a (labeled) transition relation $\xrightarrow{g,t} \subseteq C \times G \times T \times C$, where an edge $g \in G$ and a thread $t \in T$. Note, there is a special ε-transition from every state to itself: $\forall c \in C,\ t \in T : c \xrightarrow{\varepsilon,t} c$. This transition is

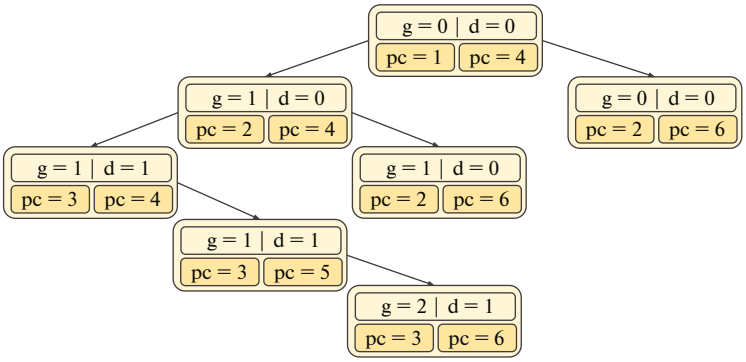

Fig. 3. Construction of interleaving set.

used only to convenient description of the next state in case there is no normal transition.

We define a set of concrete transitions as $\mathcal{T} = C \times G \times T$. An element $\tau \in \mathcal{T}$ is a triple $\tau = (c, g, t)$. We will write $\tau_1 \to \tau_2$ if $\exists c_3 \in C : c_1 \xrightarrow{g_1,t_1} c_2 \xrightarrow{g_2,t_2} c_3$. We denote $Reach_{\to}(\tau) = \{\tau' \mid \exists \tau_1, \ldots, \tau_n \in \mathcal{T}.\tau \to \tau_1 \to \ldots \to \tau_n = \tau'\}$.

A transition $c \xrightarrow{g,t} c'$ will be formally defined later for each $g = (l, \cdot, l') \in G$. Anyway, every correct transition fulfills the following requirements:

(1) Transition starts in state $c : t \in dom(c_{pc}) \wedge c_{pc}(t) = l$.

(2) Program counter of thread t proceeds to l': $t \in dom(c'_{pc}) \wedge c'_{pc}(t) = l'$.

(3) Each transition $c \xrightarrow{g,t} c'$ may change local parts of the state only for a thread t. Thus, local parts for the other threads remain unchanged, formally $\forall t' \in T : (t' \neq t) \wedge (t \in dom(c_{pc}) \cap dom(c'_{pc})) \Rightarrow c'_{pc}(t') = c_{pc}(t') \wedge c'_l(t') = c_l(t')$. Note, there are no restrictions on changes in global parts of the state.

We denote by $eval(c, t, expr)$ a value of expression $expr$ over variables in $X^{local} \cup X^{global}$ with values from state $c \in C$ and thread $t \in T$.

Further we will define semantics of operators in a program: assignment, assumption, acquire/release of locks and thread create operator. For every operator we have to define how it changes a concrete state of a program. We do not emphasize requirements 1–3, which are hold for every transition.

4.1. Assumptions

For an assumption edge $g = (l, assume(expr), l') \in G$, $t \in T$, $l, l' \in L$ a transition $c \xrightarrow{g,t} c'$, $c, c' \in C$ exists, if

– $c = c'$ – transition does not change the state.

– $eval(c, t, expr) \neq 0$ – expressions evaluated to non zero value.

4.2. Assignments

For an assignment edge $g = (l, assign(expr), l') \in G$, $t \in T$, $l, l' \in L$ a transition $c \xrightarrow{g,t} c'$, $c = (c_{pc}, c_l, c_g, c_s)$, $c' = (c'_{pc}, c'_l, c'_g, c'_s) \in C$ exists, if

– $dom(c) = dom(c')$ – transition does not change the set of threads.

– If $x \in X^{local}$ then $c'_l(t)(x) = eval(c, t, expr)$ and $c'_l(t)(x') = c_l(t)(x')$ for $x' \in X^{local} : x' \neq x$. Otherwise $c'_l(t) = c_l(t)$.

– If $x \in X^{global}$ then $c'_g(x) = eval(c, t, expr)$ and $c'_g(x') = c_g(x')$ for $x' \in X^{global} : x' \neq x$. Otherwise $c'_g(t) = c_g(t)$.

– $c'_s = c_s$ – locks stay unchanged.

4.3. Synchronization Operations

We define a synchronization primitives *acquire/release*. We assume that $acquire(s)$ operation in a thread $t \in T$, where $s \in S$ is a lock variable, has a semantics: if $s = \bot_T$

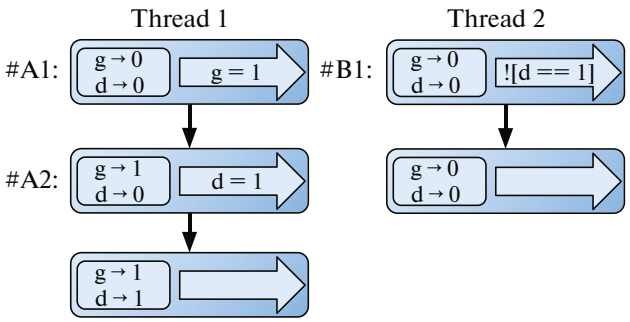

Fig. 4. Abstract transitions for two threads without any interaction.

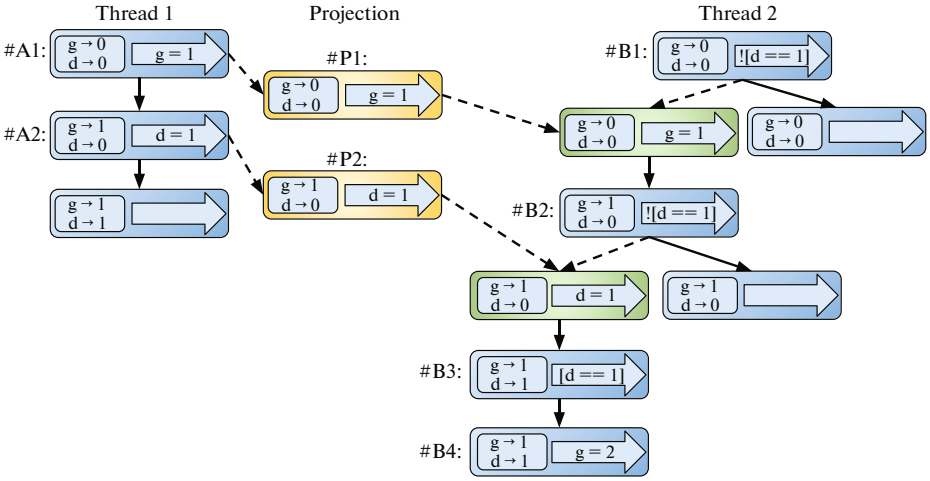

Fig. 5. Application of projections to the second thread.

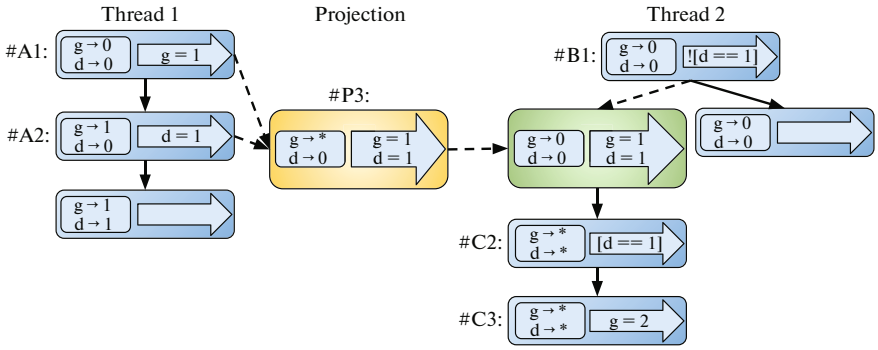

Fig. 6. Application of projections to the second thread.

in previous state then $s = t$ in the successor state and the operation performed atomically in a single edge (similar to [7]).

For an edge $g = (l, acquire(s), l') \in G$ and $t \in T$, $l, l' \in L$, we have a transition $c \xrightarrow{g,t} c'$, $c = (c_{pc}, c_l, c_g, c_s)$, $c' = (c'_{pc}, c'_l, c'_g, c'_s) \in C$, if $c_s(s) = \perp_T$ and (we acquire a lock s by assigning a value t to s and changing location in thread t to l')

– $dom(c) = dom(c')$ – transition does not change the set of threads.

– $c'_l(t) = c_l(t)$ – local variables are unchanged.

– $c'_g = c_g$ – global variables are unchanged.

– $c'_s(s) = t$ and $\forall s' \in S : s' \neq s \Rightarrow c'_s(s') = c_s(s')$.

We assume that $release(s)$ operation in a thread $t \in T$ has a semantics of assigning a value $\perp_T$ to s if $s = t$.

For an edge $g = (l, release(s), l') \in G$ and $t \in T$, l, $l' \in L$, we have a transition $c \xrightarrow{g,t} c'$, $c = (c_{pc}, c_l, c_g, c_s)$, $c' = (c'_{pc}, c'_l, c'_g, c'_s) \in C$, if $c_s(s) = t$ and

– $dom(c) = dom(c')$ – transition does not change the set of threads.

– $c'_l(t) = c_l(t)$ – local variables are unchanged.

– $c'_g = c_g$ – global variables are unchanged.

– $c_s(s) = \perp_T$ and $\forall s' \in S : s' \neq s \Rightarrow c'_s(s') = c_s(s')$.

4.4. Thread Creation

We define semantics of $thread\_create(l_\nu)$ such that the current thread proceeds to the location after *thread_create* and new thread is created with the new identifier $\nu \in T$ which is added to local parts of the state. The starting location of thread ν is $l_\nu \in L$.

Note, that we consider programs with unbounded thread creation, as *thread_create* may occur in a loop.

For an edge $g = (l, thread\_create(l_\nu), l')$, we have a transition $c \xrightarrow{g,t} c'$, $c = (c_{pc}, c_l, c_g, c_s)$, $c' = (c'_{pc}, c'_l, c'_g, c'_s) \in C$, $\nu \in T$, if

$-\ \nu \notin dom(c) \wedge dom(c') = dom(c) \cup \{\nu\}$ – a thread ν is added to the set of threads.

$-\ c'_{pc}(t) = l'$ – program counter proceeds to l' in the current thread.

$-\ c'_{pc}(\nu) = l_\nu$ – program counter of the new thread is l_ν.

$-\ c'_l(\nu) = c_l(t)$ – we inherit local part of the state from t.

$-\ c'_l(t) = c_l(t)$ and $c'_g = c_g$ and $c'_s = c_s$ – all the other variables stay unchanged.

We do not consider join operations, as they complicate the theory description. Nevertheless, it is possible to add the operations into the theory.

5. CONFIGURABLE PROGRAM ANALYSIS WITH ABSTRACT TRANSITIONS

In the original CPA theory [5, 22], an abstract state represents a set of concrete states of a program. In our theory, an abstract state is a partial one, so it may not represent any concrete state of a program. That is why the concretization function in our theory differs from the original one. Our concretization function is defined on a set of elements.

As a consequence, an abstract transition is also a partial one. Thus the analysis can not guarantee, that succeeding concrete transitions are reachable by one step of an abstract transition. In the general case, the analysis needs k steps. In the thread-modular approach $k = 2$: the analysis performs a usual transition in a thread and then spread it to all others as a transition in an environment. It takes two iterations of the algorithm.

Now we formally define a *Configurable Program Analysis with Transition Abstractions* $\mathbb{D} = (D, \Pi, merge, stop, prec, \rightsquigarrow)$. It consists from an abstract domain D of abstract elements, a set of precisions Π, a merge operator *merge*, a termination check *stop*, a precision adjustment function *prec* and transfer relation $\rightsquigarrow$.

(1) The *abstract domain* $D = (\mathcal{T}, \mathcal{E}, [\![\cdot]\!])$ is defined by the set $\mathcal{T}$ of concrete transitions, $\mathcal{T} \subseteq C \times G \times T$, the semi-lattice $\mathcal{E}$ of abstract transitions and a concretization function $[\![\cdot]\!]$. The $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ consists of the (possibly infinite) set E of abstract domain elements, a top element $\top \in E$, a bottom element $\bot \in E$, a partial order $\sqsubseteq \subseteq E \times E$ and a function $\sqcup$: $E \times E \to E$ (join operator). The function $\sqcup$ yields the least upper bound for two lattice elements, and symbols $\top$ and $\bot$ denote the least upper bound and greatest lower bound of the set E respectively.

The concretization function $[\![\cdot]\!] : 2^E \to 2^{\mathcal{T}}$ assigns to each set of abstract transitions $R \subseteq E$ its meaning, i.e. the set of concrete transitions that it represents. Note, that we use concretization on sets of transitions instead of a single transition. Thus we have

$$\forall R \subseteq E : [\![R]\!] \supseteq \bigcup_{e \in R} [\![\{e\}]\!]$$

meaning that the summary knowledge for the set of partial transitions may be bigger than union of knowledge for the single (partial) transition.

(2) The set Π of *precision* determines the possible precisions of the abstract domain. The program analysis uses precision elements to keep track of different precisions for different abstract elements. A pair (e, π) is called abstract element e with precision π. The operators on the abstract domain are parametric in the precision. For $R \subseteq E \times \Pi$ we denote $[\![R]\!] = [\![\bigcup_{(e,\pi)\in R} \{e\}]\!]$.

(3) The *transfer* relation $\rightsquigarrow$: $E \times \Pi \times 2^E \times E$ assigns to each partial transition $\hat{e}$ with precision π possible new abstract transition e' which is abstract successor of $\hat{e}$. Note, the result may depend on reached elements $R \subseteq E$. We write $(\hat{e}, \pi) \overset{R}{\rightsquigarrow} e'$ if $(\hat{e}, \pi, R, e) \in \rightsquigarrow$.

Let us denote $Reach^k$ as

$$\begin{aligned} \forall R \subseteq E : Reach^0(R) &= R \\ \forall k \geq 1 : Reach^{k+1}(R) &= \\ \bigcup_{e \in Reach^k(R)} \{e' \mid e \rightsquigarrow e'\} &\cup Reach^k(R) \end{aligned} \tag{1}$$

The requirement on the transfer states that $Reach^k$ over-approximates the concrete transitions:

$$\exists k \geq 1 : \forall R \subseteq E : [\![Reach^k(R)]\!] \supseteq \bigcup_{\tau \in [[R]]} \{\tau' \mid \tau \to \tau'\} \tag{2}$$

The requirement 2 weakens the requirement on *transfer* operator in the classical CPA theory. It means, that we may produce corresponding concrete transitions not after one step of abstract transition (classical theory), but after k steps. For thread-modular approach $k = 2$ as we will see later.

(4) The merge operator $merge : E \times E \times \Pi \to E$ weakens the second parameter using the information of the first parameter and returns a new abstract element of the precision that is given as the third parameter. The merge operator has to fulfill the following requirement:

$$\forall e, e' \in E, \pi \in \Pi : e' \sqsubseteq merge(e, e', \pi) \tag{3}$$

(5) The termination check operator $stop : E \times 2^{E \times \Pi} \times \Pi \to \mathbb{B}$ checks if the abstract element given as the first parameter with the precision given as the third parameter, is covered by the set of abstract elements given as the second parameter. The termination check can, for example, go through the elements of the set R that is given as second parameter and search for a single element that subsumes ($\sqsubseteq$) the first parameter. The termination check has to fulfill the following requirements:

$$\begin{aligned} &\forall e \in E, R \subseteq E, \pi \in \Pi : \\ &stop(e, R, \pi) \forall \hat{R} \subseteq E : [\![\{e\} \cup \hat{R}]\!] \subseteq [\![R \cup \hat{R}]\!] \end{aligned} \tag{4}$$

(6) The *precision adjustment* function $prec : E \times \Pi \times 2^{E\times\Pi} \to E \times \Pi$ computes a new abstract element and a new precision, for a given abstract element with precision and a set of abstract elements with precision. The precision adjustment function may perform widening of the abstract element, in addition to a change of precision. The precision adjustment function has to fulfill the following requirement:

$$\forall e, e' \in E, \pi, \pi' \in \Pi, R \subseteq E \times \Pi : \\ (e', \pi') = prec(e, \pi, R) e \sqsubseteq e' \quad (5)$$

The precision domain Π, termination check *stop*, merge operator *merge*, precision adjustment *prec* are the same as in the original CPA theory. The main algorithm, which computes a set of reached abstract transitions, also stays the same except for the extension of the transfer operator.

Data: a configurable program analysis with partial states $\mathbb{D}$, initial abstract transition e_0 with precision $\pi_0 \in \Pi$, a set *reached* of elements E, a set *waitlist* of elements E

Result: a set of reachable states *reached*

$waitlist := \{(e_0, \pi_0)\}$;

$reached := \{(e_0, \pi_0)\}$;

while $waitlist \neq \varnothing$ do

  pop (e, π) from $waitlist$;

  for e' in $(e, \pi) blue^{reached}_{\rightsquigarrow} e'$) do

    $(\hat{e}, \hat{\pi}) = prec(e', \pi, reached)$;

    for each $(e'', \pi'') \in reached$ **do**

      $e_{new} = merge(\hat{e}, e'', \hat{\pi})$;

      if $e_{new} \neq e''$ **then**

        $waitlist := waitlist \backslash \{(e'', \pi'')\} \cup \{(e_{new}, \pi'')\}$

        $reached := reached \backslash \{(e'', \pi'')\} \cup \{(e_{new}, \pi'')\}$

      end

    end

    if $!stop(\hat{e}, reached, \hat{\pi})$ **then**

      $waitlist := waitlist \cup \{(\hat{e}, \hat{\pi})\}$;

      $reached := reached \cup \{(\hat{e}, \hat{\pi})\}$;

    end

  end

end

Algorithm 1. Algorithm $CPA(\mathbb{D}, e_0, \pi_0)$.

For the algorithm, the main Theorem 1 may be proven even with weaker requirements. The proof replicates the proof of the classical theorem.

Theorem 1. (*Soundness*) *For a given configurable program analysis with thread abstractions $\mathbb{D}$ and an initial abstract state e_0 with precision π_0, Algorithm CPA computes a set of abstract states that overapproximates the set of reachable concrete states*:

$$[\![CPA(\mathbb{D}, e_0, \pi_0)]\!] \supseteq Reach_{\rightarrow}([\![\{e_0\}]\!]).$$

Note, several CPAs are used in practice for analysis of source code. The set of CPAs has a tree structure, where a root CPA provides its operators to Algorithm 7. Different CPAs interact each other for increasing the precision of the whole analysis. Section 6 presents such a root CPA, which requires a nested CPA. Section 7 presents a CPA, which implements a parallel composition of analyzes. Examples of nested CPAs are presented in Sections 8–12. Section 13 describes the data race detection approach, which is based on the considered CPAs.

Some auxiliary CPAs (CallstackCPA, AutomatonCPA), which do not implement any kind of analysis, may be applied without changes comparing to the origin version of the theory. Thus, they will not be described further. CPAs, which implements different kinds of analysis (analysis of explicit values, predicate analysis, etc.) should be modified to support transitions in environment. These CPAs will be described in the corresponding sections.

6. THREAD-MODULAR ANALYSIS

In section, we present a CPA, which implements thread-modular functionality. The main task of the CPA is computation of a potential effect for environment, i.e. a projection, and also an application the projections to the corresponding thread. ThreadModularCPA requires a nested CPA with extended set of operators: an operator of projection, an operator of compatibility check and and operator of composition. Thus, first, we define an extended inner CPA with the new operators.

6.1. Extended Inner CPA for Thread-Modular Analysis

A definition of CPA for thread-modular analysis is extended with three new operators: $\mathbb{I} = (D_I, \Pi_I, \rightsquigarrow_I, merge_I, stop_I, prec_I, compatible_I, \cdot|_p, compose_I)$.

An abstract domain $D_I = (\mathcal{T}_I, \mathcal{E}_I, \oplus_I)$ consists of a set of concrete transitions $\mathcal{T}_I$, a semi-lattice of abstract transitions $\mathcal{E}_I$ and a composition operator $\oplus_I$. Note, an inner analysis is required to define not a concretization function $[\![\cdot]\!]$, but a composition operator $\oplus$ because the thread-modular approach requires unified schema for calculation of concrete states.

As we have already discussed all states and transitions are partial, so they may not be directly related to concrete states and transitions. To get a complete transition we should get a composition of a set of partial transitions, which represent all available threads. Compatible partial transitions can be composed into a

complete concrete transition with an operator $\oplus : E \times T \times 2^{E \times T} \to 2^{\mathcal{T}}$. It returns a set of concrete transitions, which is represented by given partial transitions.

$\oplus$ is required to be consistent with the semi-lattice. So, if one abstract transition is less than the other, the composition with the same set must not get a larger set of concrete transitions.

A compatibility operator $compatible_I : E \times E \to \{true, false\}$ checks if two partial transitions can be started from a common complete predecessor.

Projection operator $\cdot|_p$: $E \to E$ projects a transition in a thread to another thread. For example, a projection may contain modifications of global variables and misses changes of thread-local data.

$compose_I : E \times E \to E$ combines two abstract transitions into a single one. It applies an abstract edge from one transition to an abstract state of the other transition.

Further we will use operator $apply_I$ as a combination of the three operators: $\cdot|_p$, $compose_I$ and $compatible_I$:

$$\forall e, e' \in E : apply(e, e') = \begin{cases} compose_I(e, e'|_p), & \text{if } compatible_I(e, e'|_p) \\ \bot, & \text{else.} \end{cases} \quad (6)$$

Thus, *apply* means that the transitions may be composed only if they are compatible. The operator produces an applied transition, which is also called a transition in the environment because it represents the effect of the environment.

6.2. Thread-Modular CPA

Define a thread-modular CPA $\mathbb{TM} = (D_{TM}, \Pi_{TM}, \rightsquigarrow_{TM}, merge_{TM}, stop_{TM}, prec_{TM})$, which is based on an inner CPA $\mathbb{I} = (D_I, \Pi_I, \rightsquigarrow_I, merge_I, stop_I, prec_I, compatible_I, \cdot|_p, compose_I)$.

(1) The abstract domain $D_{TM} = (\mathcal{T}, \mathcal{E}, [\![\cdot]\!]_P)$.

A complete semi-lattice $\mathcal{E} = \mathcal{E}_I$ is equal to inner analysis one.

For thread-modular analyses a concretization function $[\![\cdot]\!]$ means all possible compositions of partial transitions:

$$\forall R \subseteq E : [\![R]\!]_P = \bigcup_{k=1}^{\infty} \bigcup_{\substack{e_0, e_1, \ldots, e_k \in R \\ t_0, t_1, \ldots, t_k \in T}} \bigoplus_I \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_k \\ t_k \end{pmatrix} \right\} \right) \quad (7)$$

(2) Transfer relation computes the next transitions, after that it applies all reached transitions as transitions in an environment to the new one and applies a new transition as a transition in an environment to all reached transitions.

Data: preceding transition e_0 with precision $\pi_0 \in \Pi$

Result: a set of succeeding transitions *result*

result := $\varnothing$;

for each $\hat{e} : e_0 \overset{R}{\rightsquigarrow}_I \hat{e}$ **do**

  result := *result* $\cup \{\hat{e}\}$;

  for each $e' \in$ *reached* **do**

    result := *result* $\cup \{apply(e', \hat{e})\}$;

    result := *result* $\cup \{apply(\hat{e}, e')\}$;

  end

end

return result

Algorithm 2. *transferTM*(e_0, π_0, *reached*)

(3) $\Pi_{TM} = \Pi_I$

(4) $merge_{TM} = merge_I$.

(5) $stop_{TM} = stop_I$.

(6) $prec_{TM} = prec_I$.

7. COMPOSITE ANALYSIS

A composite analysis is used to combine different kinds of analysis altogether. Examples of the inner CPAs will be present further.

CompositeCPA $\mathbb{C} = (D_{\mathbb{C}}, \Pi_{\mathbb{C}}, \rightsquigarrow_{\mathbb{C}}, merge_{\mathbb{C}}, stop_{\mathbb{C}}, prec_{\mathbb{C}}, compatible_{\mathbb{C}}, \cdot|_p, compose_{\mathbb{C}})$ operates with inner analyses $\Delta_i = (D_{\Delta_i}, \Pi_{\Delta_i}, \rightsquigarrow_{\Delta_i}, merge_{\Delta_i}, stop_{\Delta_i}, prec_{\Delta_i}, compatible_{\Delta_i}, \cdot|_p, compose_{\Delta_i})$

(1) $D_{\mathbb{C}} = D_{\Delta_1} \times \ldots \times D_{\Delta_n}$

$$\forall e_0, \ldots, e_m \in E_{\mathbb{C}}, \forall 0 \le i \le m : e_i = (e_i^1, \ldots, e_i^n)$$

$$\bigoplus_{\mathbb{C}} \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_m \\ t_m \end{pmatrix} \right\} \right) = \bigoplus_{\Delta_1} \left(\begin{pmatrix} e_0^1 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1^1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_m^1 \\ t_m \end{pmatrix} \right\} \right) \cap \ldots \cap \bigoplus_{\Delta_n} \left(\begin{pmatrix} e_0^n \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1^n \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_m^n \\ t_m \end{pmatrix} \right\} \right)$$

(2) $\Pi_{\mathbb{C}} = \Pi_{\Delta_1} \times \ldots \times \Pi_{\Delta_n}$

(3) Transfer relation applies inner transfers to corresponding part of the state.

$$\forall e_1, e_2 \in E : e_1 \rightsquigarrow_{\mathbb{C}} e_2 \Leftrightarrow \forall 1 \le i \le n : e_1^i \rightsquigarrow_{\Delta_i} e_2^i$$

(4) Merge operator calls inner merge operators: $\forall e_1, e_2 \in E, \pi \in \Pi : merge_{\mathbb{C}}(e_1, e_2, \pi) = (merge_{\Delta_1}(e_1^1, e_2^1, \pi^1), \ldots, merge_{\Delta_n}(e_1^n, e_2^n, \pi^n)$.

(5) Stop operator calls inner stop operators: $\forall e \in E, R \subseteq E, \pi \in \Pi : stop_{\mathbb{C}}(e, R, \pi)\ \exists \hat{e} \in R\ \forall 1 \le i \le n : stop_{\Delta_i}(e^i, \{\hat{e}^i\}, \pi^i)$.

(6) $prec_{\mathbb{C}}(e, R, \pi) = (prec_{\Delta_1}(e^1, R, \pi_1), \ldots, prec^E_{\Delta_n}(e^n, R, \pi^n))$.

(7) $compatible_{\mathbb{C}}(e_1, e_2) = compatible_{\Delta_1}(e_1^1, e_2^1) \wedge \ldots \wedge compatible_{\Delta_n}(e_1^n, e_2^n)$.

(8) $e|_p = (e^1|_p, \ldots, e^n|_p)$.

(9) $compose_{\mathbb{C}}(e_1, e_2) = (compose_{\Delta_1}(e_1^1, e_2^1), \ldots, compose_{\Delta_n}(e_1^n, e_2^n))$.

8. LOCATION ANALYSIS

This section presents a simple Location Analysis $\mathbb{L} = (D_L, \Pi_L, \rightsquigarrow_L, merge_L, stop_L, prec_L, compatible_L, \cdot|_p, compose_L)$ which tracks abstract locations. The analysis extends the origin LocationCPA to be able to support thread-modular analysis.

(1) $D_L = (\mathcal{T}, \mathcal{C}_L, \oplus_L)$.

An abstract transition consists of an abstract state $s \in E_L^S$ and an abstract edge $q \in E_L^T$.

E_L^S is a set of abstract program locations, which is mapped to concrete CFA locations with function $loc : E_L^S \to 2^L$. $\top_L^S$ means, the analysis does not know what is a particular program location. Formally, $loc(\top_L^S) = L$. In general case analysis is able to operate with abstract locations, which express several concrete locations, but further we describe a simple version of analysis, which considers only single locations: $\forall s \in E_L^S : s = \top_L^S \vee s = \bot_L^S \vee loc(s) = l \in L$. Defined $\mathcal{C}_L^S$ is a flat lattice, meaning the two different locations are not comparable.

$$\forall s_1, s_2 \in E_L^S:$$
$$s_1 \sqsubseteq s_2 \Leftrightarrow s_1 = \bot_L^S \vee s_2 = \top_L^S$$
$$s_1 \sqcup s_2 = \begin{cases} s_1, & \text{if} \quad s_2 = \bot_L^S \\ s_2, & \text{if} \quad s_1 = \bot_L^S \\ \top_L^S, & \text{if} \quad s_1 \neq \bot_L^S \wedge s_2 \neq \bot_L^S \end{cases}$$

Abstract edge is based on a ordinary CFA edge, containing only source location and destination location. $E_L^T \subseteq E_L^S \times E_L^S$.

$$\forall q_1, q_2 \in E_L^T : q_{1,2} = (s_{1,2}, d_{1,2})$$
$$q_1 \sqsubseteq q_2 \Leftrightarrow loc(s_1) \subseteq loc(s_2) \wedge loc(d_1) \subseteq loc(d_2)$$
$$q_3 = q_1 \sqcup q_2 \Leftrightarrow loc(s_3)$$
$$= loc(s_1) \cup loc(s_2) \wedge loc(d_3) = loc(d_1) \cup loc(d_2)$$

Define an auxiliary operator to check if partial abstract transitions can be composed into a complete one.

$$\begin{gathered} \forall e_0, e_1, \ldots, e_n \in E, e_i = (s_i, q_i), q_i = (s_i^{src}, s_i^{dst}): \\ C_{check}(e_0, \{e_1, \ldots, e_n\}) \\ = loc(s_0) \cap loc(s_0^{src}) \cap \ldots \cap loc(s_n^{src}) \neq \varnothing \\ \wedge loc(s_1^{dst}) \cap \ldots \cap loc(s_n^{dst}) \neq \varnothing \end{gathered} \tag{8}$$

The requirement 8 claims that all partial transitions may be related to the same concrete transitions: they have the same source location and the same destination location.

The most complicated part is the definition of composition operator.

$$\forall e_1, \ldots, e_n \in E, \quad e_i = (s_i, q_i), \quad q_i = (s_i^{src}, s_i^{dst}) :$$
$$\bigoplus\nolimits_L \left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{ \begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_n \\ t_n \end{pmatrix} \right\} \right)$$
$$= \begin{cases} \left\{ \begin{array}{l|r} (c, g, t_0) \in \mathcal{T} & c_{pc} = \begin{Bmatrix} t_0 \to l_0, \\ \ldots \\ t_n \to l_n \end{Bmatrix}, \\ c = (c_{pc}, c_l, c_g, c_s) & s_i \neq \top_L^S \; l_i = s_i \\ & g \in G, g = (l_0, op, l_1): \\ & l_1 \in loc(s_0^{dst}) \cap \ldots \cap loc(s_n^{dst}) \end{array} \right\}, \\ \varnothing, \quad \text{otherwise} \end{cases} \tag{9}$$
$$\text{if} \quad C_{check}(e_0, \{e_1, \ldots, e_j\})$$

(2) $\Pi_L = \{\pi_0\}$. The location state does not depend from the precision, so there is only one dummy precision element.

(3) The transition $e \overset{R}{\rightsquigarrow}_L e'$ exists, if transition in a thread changes the location according to the abstract edge. More formally. Let $e = (s, q)$, $e' = (s', q')$, $q = (pred, suc)$, $q' = (pred', suc')$. $loc(s) \cap loc(pred) \neq \varnothing, \exists g' \in G : g' = (l_1, op, l_2) \wedge l_1 \in loc(pred') \cap loc(s') \wedge l_2 \in loc(suc')$ and $s' = suc$.

(4) The merge operator does not join states: $merge_L(e, e', \pi) = e'$.

(5) The termination check considers abstract states individually: $stop_L(e, R, \pi) = (e \in R)$.

(6) The precision is never adjusted: $prec_L(e, \pi, R) = (e, \pi)$.

(7) $e|_p^L = e$.

(8) $\forall e, e' \in E, e = (s, q), e' = (s', q') : compose_L(e, e') = \tilde{e} = (s, \hat{q})$, where $\hat{q} = (s, s)$, as transition in thread do not change the state.

(9) $\forall e_1, e_2 \in E : compatible(e_1, e_2) \equiv true$

9. THREAD ANALYSIS

We define a simple Thread Analysis $\mathbb{T} = (D_T, \Pi_T, \rightsquigarrow_T, merge_T, stop_T, prec_T, compatible_T, \cdot|_p, compose_T)$ which tracks thread identifiers.

The Thread Analysis inherits the limitations of [6] and restricted to the programs with bounded thread creation. We suppose that the program has a finite number of threads identified by the locations where they are created, i.e. $T \subseteq L$ and for $thread\_create(pc_v)$ we always create a thread with identifier pc_v. Note, that the other analyses are not bounded.

(1) The domain D_T is based on the flat lattice for the set of threads T: $D_T = (C \times G \times T, \mathcal{E}_T, \oplus_T)$, with $\mathcal{E}_T = \mathcal{E}_T^S \times \mathcal{E}_T^T$. $E_T^S = T \cup \{\bot_T^S, \top_T^S\}$, $\bot_T^S \sqsubseteq_T^S t \sqsubseteq_T^S \top_T^S$ and $t \neq t' \Rightarrow t \not\sqsubseteq_T^S t'$ for all elements $t, t' \in T$ (this implies $\bot_T^S \sqcup_T^S t = t$, $\top_T^S \sqcup t = \top_T^S$, $t \sqcup_T^S t' = \top_T^S$ for all elements $t, t' \in T$, $t \neq t'$)

$$E_T^T = \{ \bot_T^T, \varepsilon, \top_T^T \} \cup G$$

Define an auxiliary operator to check if partial abstract transitions can be composed into a complete one.

$$\forall e_1, \ldots, e_n \in E, \quad e_i = (s_i, q_i), \quad s_i \in E_T^S, \quad q_i \in E_T^T \quad C_{check}(\{e_1, \ldots, e_n\}) = \forall k \neq m : s_k \neq s_m \tag{10}$$

Now define a composition operator $\oplus_T$

$$\forall e_0, e_1, \ldots, e_n \in E, \quad e_i = (s_i, q_i), \quad t_0, t_1, \ldots, t_n \in T, \quad t_i \neq t_j$$
$$\oplus_T\left(\begin{pmatrix} e_0 \\ t_0 \end{pmatrix}, \left\{\begin{pmatrix} e_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} e_n \\ t_n \end{pmatrix}\right\}\right) \tag{11}$$
$$= \begin{cases} \left\{ (c, g, t_0) \in \mathcal{T} \;\middle|\; \begin{matrix} dom(c_{pc}) = \{s_1, \ldots, s_j\} \\ dom(c_l) = \{s_1, \ldots, s_j\} \\ g \in G, t_0 = s_0 \end{matrix} \right\} \\ \quad (c_{pc}, c_l, c_g, c_s) \in C \\ \varnothing, \quad \text{otherwise} \end{cases}$$
$$\text{if} \quad C_{check}(\{e_0, \ldots, e_n\})$$

(2) There is only one empty precision: $\Pi_T = \{\varnothing\}$.

(3) The transfer relation $\rightsquigarrow_T$ has the transfer $\tau \rightsquigarrow_T \tau'$, $\tau = (s, g)$, $g = (\cdot, op, \cdot)$ if

– $op \neq tc_{child}$ (the syntactical successor in the CFA without considering the semantics of the operation op), $s = s'$, $g' \in G$.

– $op = tc_{child}(l_v)$ then $s' = l_v$, $g' \in G$

(4) The merge operator does not combine elements when control flow meets: $merge_T(e, e', \pi) = e'$.

(5) The termination check considers abstract states individually: $stop_T(e, R, \pi) = (e \in R)$.

(6) The precision is never adjusted: $prec_T^E(e, \pi, R) = (e, \pi)$.

(7) Definition of the projection:

$$\forall e \in E, e = (s, g) : e|_p = \begin{cases} e, & \text{if} \quad g \notin G \\ (s, \varepsilon), & \text{otherwise} \end{cases}$$

(8) $\forall e, e' \in E_P, e = (s, q)$, $e' = (s', q') : compose_P(e, e') = \tilde{e} = (s, q')$.

(9) $\forall e_1, e_2 \in E : compatible_P(e_1, e_2) = C_{check}(e_1, e_2)$.

10. PREDICATE ANALYSIS

In section, we present a commonly used Predicate Analysis [20] with transition abstraction. Predicate transitions consist of two parts: a predicate state and a predicate abstract edge, which can be expressed by a normal CFA edge or a formula, encoding an operation. However, the formulas should be renamed to avoid the local variables collision.

Let $\mathcal{P}$ be a set of formulas over program variables in a quantifier-free theory $\mathcal{T}$. Let $\mathcal{P} \subseteq \mathcal{P}$ be a set of predicates. Let $v : X \to \mathbb{Z}$ is a mapping from variables to values. Define $v \vDash \varphi$, where v is called model of φ.

Let us define renaming of variables $\theta : X \to X'$ from the names X to X' which is applicable to formulas $\theta(\varphi)$. We denote as

$$\theta_{X,i} = \begin{cases} x \mapsto x\#i, & \text{if} \quad x \in X \\ x \mapsto x, & \text{otherwise.} \end{cases}$$

$$\theta_{X,i}^{-1} = \begin{cases} x\#i \mapsto x, & \text{if} \quad x \in X \\ x \mapsto x, & \text{otherwise.} \end{cases}$$

Define $(\varphi)^\pi$ – the boolean predicate abstraction of formula φ.

Define $SP_{op}(\varphi)$ – strongest postcondition of φ and operation op.

We define Predicate Analysis $\mathbb{P} = (D_P, \Pi_P, \rightsquigarrow_P, merge_P, stop_P, prec_P, compatible_P, \cdot|_p, compose_P)$ which can tracks the validity of predicates over program variables.

It consists of the following components.

(1) The abstract domain $D_P = (\mathcal{T}, \mathcal{E}_P, \oplus_P)$.

$\mathcal{T} = C \times G \times T$. $\mathcal{E}_P = (E_P, \top_P, \bot_P, \sqsubseteq_P, \sqcup_P)$. An abstract transition consists of an abstract state $s \in E_P^S$ and an abstract edge $q \in E_P^T$, so $E_P = E_P^S \times E_P^T$ and $\mathcal{E}_P = \mathcal{E}_P^S \times \mathcal{E}_P^T$.

$E_P^S = \mathcal{P}$, so a state is a quantifier free formula. The top element $\top_P^S = true$, and bottom element $\bot_P^S = false$. The partial order $\sqsubseteq_P^S \subseteq E_P^S \times E_P^S$ is defined as $e \sqsubseteq_P^S e' \Leftrightarrow$

$e \Rightarrow e'$. The join $\sqcup_P^S$: $E_P^S \times E_P^S \to E_P^S$ yields the least upper bound according to partial order.

Abstract edge $q \in E_P^T$ is an action, expressed by a formula, or an ordinary CFA edge: $E_P^T = E_P^S \cup G$.

We do not provide a formal definition of $\oplus_P$, as it requires a lot of space. The basic idea is that it returns a set of concrete transitions, which correspond to a formula (strongest post-condition). The more tricky part of definition is a check C_{check}, whether the set of partial transitions corresponds to a single concrete one. For a transition in the thread e_0 with normal CFA edge $q_0 \in G$ and transitions in environment $e_1, \ldots, e_n$ the check consists of two parts:

(a) for the state s_0 of e_0 and all states of transitions in environment $s_1, \ldots, s_n$ there exists a model v;

(b) the edge q_p of projection $e_0|_p$ is covered by abstract edges q_j of transitions in the environment $q_p \sqsubseteq q_j$.

Formally,

$$\begin{gathered}\forall e_0, e_1, \ldots, e_n \in E_P, \\ e_i = (s_i, q_i), \quad s_i \in E_P^S, \quad q_i \in E_P^T \\ C_{check}(e_0, \{e_1, \ldots, e_n\}) \\ = \exists v : v \vDash s_0 \wedge \theta_{X^{local},1}(s_1) \wedge \ldots \wedge \theta_{X^{local},n}(s_n) \\ \wedge (q_0 \in G \wedge \forall 0 < j \le n : q_j \notin G \wedge e_0|_p = (s_p, q_p) : q_p \sqsubseteq q_j)\end{gathered} \tag{12}$$

(2) The set of precisions $\Pi_P = 2^{\mathscr{P}}$ models a precision for an abstract state as a set of predicates.

(3) Merge operator may have several modifications, for example,

(a) $merge_{Join}$ merges both parts of the transition:

$$\begin{gathered}\forall e, e' \in E_P, \pi \in \Pi_P, e = (s, g) : \\ merge_P(e, e', \pi) \\ = \begin{cases}(s \vee s', g), & \text{if} \quad g = g', \quad g \in G \\ e', & \text{if} \quad g \in G \wedge g' \in \mathscr{P} \vee g \in \mathscr{P} \wedge g' \in G \\ (s \vee s', g \vee g'), & \text{if} \quad g, g' \in \mathscr{P}\end{cases}\end{gathered} \tag{13}$$

(b) $merge_{Eq}$ merges only abstract edges for equal (or covered) states.

(c) $merge_{Sep}$ does not merge elements.

(4) The termination check if e is covered by another state in the reached set: $stop_P(e, R, \pi) = \exists e' \in R : (e \sqsubseteq e')$.

(5) The precision adjustment function constructs predicate abstraction over predicates in precision π: $prec_P(e, \pi, R) = e^{\pi} = (s^{\pi}, q)$.

(6) The transfer relation $e \rightsquigarrow_P (e', \pi)$, $e = (s, g)$, $e' = (s', \cdot)$. As Predicate analysis does not track relevant edges, it returns all possible ones.

– For $g \in G$ we have the transfer $e \rightsquigarrow_P (e', \pi)$ with $g = (\cdot, op, \cdot)$, if

$$s' = \begin{cases}(SP_{op}(e))^{\pi}, & \text{if} \\ op = assign(w, expr) \vee op = assume(expr) \\ s, & \text{otherwise.}\end{cases}$$

– For $g = \hat{\varphi} \in \mathscr{P}$ the transfer computes a formula: $s' = \hat{\varphi} \wedge e$.

(7) Definition of the projection:

$$\begin{gathered}\forall e \in E_P, \quad e = (s, g) : \\ e|_p = \begin{cases}e, & \text{if} \quad g \notin G \\ (\theta_{X^{local},env}(s), \theta_{X^{local},env}(SP_{op}(s))), & \text{otherwise}\end{cases}\end{gathered} \tag{14}$$

The projection represents how the transition looks as an environment. The local variables are renamed to avoid name collision. Thus, only predicates over global variables stay valuable. The state (the first part of the transition) represents an abstraction over global part of a thread state. And the edge (the second part of the transition) corresponds the concrete operation to the global changes.

(8) $\forall e, e' \in E_P, e = (s, q)$, $e' = (s', q') : compose_P(e, e') = \tilde{e} = (s, q')$.

(9) $\forall e_1, e_2 \in E_P, e_i = (s_i, q_i)$, $s_i \in E_P : compatible_P(e_1, e_2) = \exists v : v \vDash \theta_{X^{local},1}(s_1) \wedge \theta_{X^{local},2}(s_2)$

Compatibility check means that the transitions may be composed into the global one. And this is impossible, if the global predicates are inconsistent. Thus, we should check, if there exists a single model of two partial formulas.

11. VALUE ANALYSIS

Define a Value Analysis $\mathbb{V} = ((D_V, \Pi_V, \rightsquigarrow_V, merge_V, stop_V, prec_V, compatible_V, \cdot|_p, compose_V)$, which tracks explicit values. Unlike predicate analysis Value analysis stores only explicit values of variables, so it can not handle more complicated dependencies between variables, for example, inequalities.

The Value Analysis consists from the following components:

(1) Abstract state of the analysis is a mapping from a variables to their values: $\forall s \in E_V^S, s : X \to \mathscr{Z}$, where $\mathscr{Z} = \mathbb{Z} \cup \{\bot_Z, \top_Z\}$. Thus, a set of abstract states E_V^S is a flat lattice over integers. A top element $\top_V^S = \{v \mid \forall x \in X : v(x) = \top_Z\}$ is a mapping with each variable has an arbitrary value. A bottom element $\bot_V^S = \{v \mid \exists x \in X : v(x) = \bot_Z\}$ is a mapping, where at least one variable has no value. The state is unreachable for real execution of a program. An order is trivial: any two states (not equal to $\top_V^S$ or $\bot_V^S$) are not comparable.

An operator C_{Check} checs if there a common mapping for global variables: $\forall s_0, \ldots, s_n \in E_V^S : C_{Check}(\{s_0,$

$..., s_n\}) = \forall x \in X^g, \exists z \in Z : \forall 0 \le i \le n : z = s_i(x) \vee s_i(x) = \top_Z$. Denote the mapping as $\hat{v}_g$.

Now, define a composition operator:

$$\forall s_1, ..., s_n \in E_V^S :$$
$$\oplus_S^S\left(\begin{pmatrix} s_0 \\ t_0 \end{pmatrix}, \left\{\begin{pmatrix} s_1' \\ t_1 \end{pmatrix}, ..., \begin{pmatrix} s_n' \\ t_n \end{pmatrix}\right\}\right)$$
$$= \begin{cases} c \in C \mid \forall x \in X^g : \hat{v}_g(x) \neq \top_Z \Rightarrow c_g(x) = v_g(x)\}, \\ \text{if} \quad C_{Check}(\{s_0, ..., s_n\}) \\ \varnothing, \quad \text{otherwise} \end{cases}$$

The definition of the operator $\oplus_S^S$ definitely satisfy the requirements ?, ?, as an upper element ($\sqsubseteq_V^S$) means only $\top_V^S$, and thus, widen the set of concrete states.

A set of abstract edges contains a set of normal CFA edges and transitions in environment, which are defined by changes of global variables: $E_V^T = 2^{X \to \mathbb{Z}} \cup G$. An identical transition $\varepsilon = \varnothing$ is an empty mapping, meaning no variable changes its state.

(2) Precision of the Value analysis contains a set of tracked variables: $\Pi_V = 2^X$.

(3) The transfer relation $\rightsquigarrow_V$ contains a transition $e \overset{R}{\rightsquigarrow_V} e'$, with $e = (s, q)$, $e' = (s', \top_V^T)$, if

$- q \in G, q = (\cdot, assume(expr), \cdot')$

$$\forall x \in X : s'(x)$$
$$= \begin{cases} \bot_Z, \quad \text{if} \quad a \in \mathbb{Z}.(x \to a) : (expr \neq 0)_{/s} \\ a, \quad \text{if} \quad \exists! a \in \mathbb{Z}.(x \to a) : \\ (expr \neq 0)_{/s} \vee s(x) = c \\ \top_Z, \quad \text{otherwise} \end{cases}$$

Here $(expr)_{/s}$ means an interpretation of the expression *expr* over variables in X for an abstract assignment s. And expression $(x \to a) : (expr \neq 0)_{/s}$ means, that the value of a of the variable x satisfies the interpretation.

$- q \in G, q = (\cdot, assign(w, expr), \cdot')$

$$\forall x \in X : s'(x) = \{expr_{/s}, \text{ if } x = ws(x), otherwise$$

$- q \in G$ in other cases the state is not changed: $s' = s$.

$- q : X \to Z$ means that we have a transition, which changes the definite variables. In the general case the successor

$$\forall x \in X : s'(x) = \begin{cases} q(x), \quad \text{if} \quad x \in dom(q) \\ s(x), \quad \text{otherwise} \end{cases}$$

(4) Merge operator does not join the states: $merge_V(e, e', \pi) = e'$. The condition 3is evidently fulfilled.

(5) Stop operator consider states separately: $stop_S(e, R, \pi) = (\exists e' \in R \wedge e \sqsubseteq e')$.

(6) Precision adjustment computes a new abstract state, limiting the assignments only by variables in a precision: $prec_V(e, \pi, R) = (e_\pi, \pi)$.

(7) $\forall e_1, e_2 \in E_V : compatible_V(e_1, e_2) = C_{check}(\{s_1, s_2\})$.

(8) Operator $compose(e_1, e_2)$ is a default one: $\forall e_1, e_2, e_i = (s_i, q_i) : compose(e_1, e_2) = (s_1, q_2)$.

(9) A transition in environment may affect only on global variables: $\forall e \in E_V, e = (s, q) : e|_p = (s^{global}, q^{global})$. Here a mapping s^{global} means only a part, which is related to a global variables.

12. LOCK ANALYSIS

We define Lock Analysis $\mathbb{S} = (D_S, \Pi_S, \rightsquigarrow_S, merge_S, stop_S, prec_S, compatible_S, \dashv_p, compose_S)$ which tracks the set of acquired locks (synchronization variables) for each thread.

It consists of the following components.

(1) The abstract domain $D_S = (C \times G \times T, \mathcal{E}_S, \oplus_S)$ uses semi-lattice $\mathcal{E}_S = \mathcal{E}_S^S \times \mathcal{E}_S^T$. $E_S^S = 2^S \cup \{\top^E, \bot^E\}$ is a superset of synchronization variables, $\bot_S^S \sqsubseteq ls \sqsubseteq_S^S \top_S^S$ and $ls \subseteq ls' \Rightarrow ls \sqsupseteq_S^S ls'$ for all elements ls, $ls' \subseteq S$ (this implies $\bot_S^S \sqcup_S^S ls = ls$, $\top_S^S \sqcup ls = \top_S^S$, $ls \sqcup_S^S ls' = ls \cap ls'$ for all elements $ls, ls' \subseteq S$, $ls \neq ls'$), and $E_S^T = \{\bot_S^T, \varepsilon, \top_S^T\} \cup G$.

(2) There is only one empty precision: $\Pi_S = \{\varnothing\}$.

(3) The *transfer* increases the number of stored locks in case it goes via *acquire* operator and decreases in case of *release*. Formally, the transfer relation $\rightsquigarrow_S$ has the transfer $\tau_S \rightsquigarrow \tau'$, $\tau = (ls, g)$, $g = (\cdot, op, \cdot')$ if

$- op \neg quire(s)$ and $s \notin ls \wedge ls' = ls \cup \{s\}$, $g' \in G$.

$- op = release(s)$ and $ls' = ls \backslash \{s\}$, $g' \in G$.

$- op = thread\_create(l_v)$ and $ls' = \varnothing$, $g' \in G$.

– otherwise, $ls = ls'$, $g' \in G$.

(4) The merge operator does not combine elements: $merge_S(e, e', \pi) = e'$.

(5) The termination check is true if exists state which contains less locks:

$$stop_S(e, R, \pi) = (\exists e' \in R \wedge e \sqsubseteq e').$$

(6) The precision is never adjusted: $prec_S(e, \pi, R) = (e, \pi)$.

(7) Definition of the projection: $\forall e \in E_S, e = (s, g) : e|_p = (s, \varepsilon)$.

Note, the transitions in environment (ε transitions) can not change the set of acquired locks, as the one

thread can not affect the acquired locks of the other thread. Thus, we have only one option for projection. Anyway, the projection strongly affects for compatibility check, as the threads can not acquire the same lock simultaneously.

(8) $\forall e, e' \in E_S$, $e = (s, q)$, $e' = (s', q') : compose_S(e, e') = \tilde{e} = (s, q')$.

(9) $\forall e_1, e_2 \in E_S : compatible_S(e, e') = (ls \cap ls' = \varnothing)$. The compatibility check is very close to basic Lockset algorithm. If there is the same lock in both threads, the operations can not be composed into the concrete one, as the threads can not acquire the same lock twice.

13. DATA RACE DETECTION

As we have already discussed, an approach for data race detection is divided into two steps: construction of a reached set and detection of pairs, which from a data race. The first subtask is solved by a set of considered CPAs.

Figure 7 presents an example of CPA configuration for data race detection. The set of CPAs contains ThreadModularCPA (Section 6), as a root one. It includes ARGCPA, which provides different relations between states, including "parent-child", "projected-projection" and others. CompositeCPA (Section 7) implements a parallel composition of nested CPA: LocationCPA (Section 8) models program counter, CallstackCPA provides interprocedural analysis, LockCPA (Section 12) tracks acquired locks, ThreadCPA (Section 9) tracks thread creation points, PredicateCPA (Section 10) implements predicate analysis. Note, this is not the only configuration. Moreover, different tasks require different configuration of CPAs.

An Abstract Reachability Graph (ARG) is constructed with help of the set of CPAs. It consists from the following transitions, which are reachable with a particular level of an abstraction. Thus, they may be not reachable in a real execution of a program. The next step is to find pairs, which form data races.

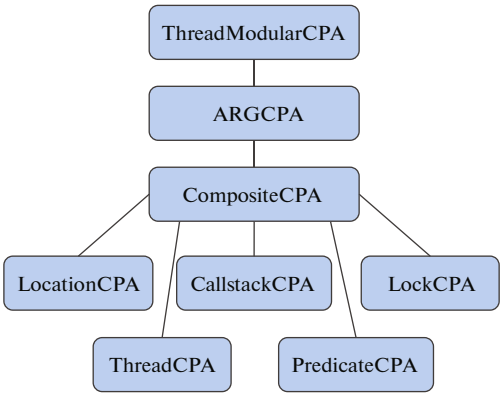

Fig. 7. An example of CPA configuration.

Usually, data race is considered to be a situation where simultaneous accesses to the same memory take place from different threads, and one of the accesses is a write one. Here are two main issues: how to detect the same memory in a static way and how to detect simultaneous accesses. Further, we will discuss the two features of our approach.

The presented theory supports shared data, which are expressed only by global variables. In real-world software, there are a lot of operations with pointers, structure fields and so on. We are using BnB memory model, which divides memory into a disjoint set of regions [13–15]. The region corresponds to a special data type or to a special structure field in case of the field was not addressed. The memory model has a certain number of limitations. First of all, it does not support address arithmetic and casting, which reduces the soundness. Then, there may be false alarms for general data types, like integer, as there are a lot of accesses to it.

In case of we found a pair of accesses to the same memory region, we need to check a possibility of simultaneous access to it. We use compatibility notion here. Compatibility here means the two partial abstract states may be a part of a single state. Or, in other terms, one transition can be applied to another and vice versa, which means the two operations may be executed in parallel. Our approach for static race detection is a generalization of Lockset [12], which claims a data race as two accesses with disjoint sets of locks. One of the limitations of the Lockset approach is the absence of support of other synchronization primitives. We use *compatible* operator to identify the potentially parallel operations. As compatibility check is based on different kinds of analyzes, including Lock analysis, Predicate analysis, and others, it is more precise, than the Lockset.

The data race detection algorithm consists of the following steps:

(1) compute a complete set of reached abstract transitions (Algorithm 1);

(2) for every reached transition extract a memory region it accesses to;

(3) for every memory region try to find a compatible pair of transitions, which form a race condition;

(4) check every potential data race for feasibility and refine a predicate abstraction if necessary [20].

Note, that algorithm of refinement the abstraction (Counterexample Guided Abstraction Refinement, CEGAR) was reused without significant modifications. However, it allows to perform refinement only local transitions in a single thread. So, it can not detect contradiction between different threads. It is not a limitation of the approach, and in case of extending

the CEGAR algorithm it is possible to obtain more precise results.

14. EVALUATION

We have implemented Thread-Modular Analysis with transition abstraction as a composition of following analyses:

– Location Analysis $\mathbb{L}$ (Section 8),

– Callstack Analysis $\mathbb{CS}$ (tracks function callstack),

– Thread Analysis $\mathbb{T}$ (Section 9),

– Lock Analysis $\mathbb{S}$ (Section 12),

– Predicate Analysis $\mathbb{P}$ (Section 10) with options:

• $Merge_{Join}$ – merge state and abstract edge of abstract transition.

• $Merge_{Eq}$ – merge abstract edges if states are equal.

• $Merge_{Sep}$ – not merging.

– Value Analysis $\mathbb{V}$ (Section 11).

The benchmark sets were launched on a set of machines with a GNU/Linux operating system (x86_64-linux, Ubuntu 18.04), Intel Xeon E3-1230 v5, 3.40 GHz. We used default SV-COMP limits: 15 min of CPU time and 15 GB of memory.

14.1. SV-COMP Benchmark Set

The approaches to compare:

(1) Variants of thread-modular analysis with abstract transition.

(a) Variants of merge for Predicate Analysis.

(i) **MergeJoin**. ($\mathbb{L}$, $\mathbb{CS}$, $\mathbb{T}$, $\mathbb{S}$, $\mathbb{P}$), $Merge_{Join}$.

(ii) **MergeEq**. ($\mathbb{L}$, $\mathbb{CS}$, $\mathbb{T}$, $\mathbb{S}$, $\mathbb{P}$), $Merge_{Eq}$.

(iii) **MergeSep**. ($\mathbb{L}$, $\mathbb{CS}$, $\mathbb{T}$, $\mathbb{S}$, $\mathbb{P}$), $Merge_{Sep}$.

(b) **Value**. Value Analysis. ($\mathbb{L}$, $\mathbb{CS}$, $\mathbb{T}$, $\mathbb{S}$, $\mathbb{V}$).

(2) **Threading**. ThreadingCPA described in [18] uses original CPA theory and considers all possible interleavings.

Discussion. The thread-modular approach does not produce incorrect true results that confirms the soundness of the approach.

MergeJoin shows better performance than MergeSep configuration. This is mostly because of a large number of environment steps, which should be performed for each transition in the environment. MergeJoin combines them together and applies them at once. This allows for saving a large amount of time. Anyway, the MergeSep configuration allows avoiding some imprecision due to separate state exploration.

A simple value configuration is a fast analysis, but sometimes it has to explore all possible values of a variable, which are numerous, and thus it fails into a timeout.

Classic analysis (**Threading**) is sound and precise, so it does not produce incorrect verdicts at all. However it requires a lot of resources, this is the main disadvantage of the approach. **Threading** is able to solve only 1 of 7 real-world benchmarks, which are based on Linux device drivers. The thread-modular approaches (MergeJoin,MergeEq, MergeSep) solve 5 of 7.

Most of the new true verdicts proved by the thread-modular approach (26 of 27 for the **MergeJoin**) were not proved by **Threading**. That is one of the contributions of the approach.

The thread-modular approach has many incorrect false verdicts. Most of them are due to unsupported atomic constructions like compare-and-swap. In some cases, we do not support happens-before ordering by thread creation (the child thread can not interfere parent before creation). One more minor problem is the current limitation of a refinement procedure, which does not allow to discover of interpolants to the other thread. A small number of benchmarks are sensitive for exploring interleavings, which is a limitation of the thread-modular approach.

Table 1. Evaluation on SV-COMP benchmarks

Approach	MergeJoin	MergeEq	MergeSep	Value	Threading
False verdicts	1026	1027	763	963	720
Correct false	805	806	548	752	720
Incorrect false	221	221	215	211	0
True verdicts	27	28	28	30	165
Correct true	27	28	28	30	165
Incorrect true	0	0	0	0	0
Unknowns	29	27	297	89	197
CPU time (s)	16800	15900	278000	75300	93700
Wall time (s)	10200	9 630	258000	64900	67500

14.2. Data Race Detection in Device Drivers

The set of benchmarks, based on Linux device drivers, was prepared by Klever tool, a framework for verification of large software systems [16, 17]. It divides a large codebase into separate verification tasks. For the Linux kernel, a verification task consists of one Linux module. Then Klever automatically prepares an environment model, which includes a thread model, a kernel model, and operations over the module. After the preparation of a verification task, Klever calls a verification tool via a common interface – BenchExec [23].

The benchmarks are checked for data race conditions as it was described in Section 13. Note, in SV-COMP benchmarks are reachability tasks. Thus, we can not evaluate SV-COMP participants on the new set, because they do not support data race detection.

We chose the *drivers/net/* subsystem of Linux kernel 4.2.6, for which Klever prepared 425 verification tasks. We compared the following configurations:

(1) **Base**. **MergeJoin** configuration from the previous subsection.

(2) **Havoc**. **MergeJoin** configuration from the previous subsection without any predicate over global variables in predicate precision. It means, we abstract from value of global variables and consider them to be arbitrary changed.

Discussion. A false verdict means, that there is at least one potential data race condition. **Havoc** configuration is a bit faster and less precise, as it can not prove 8 true modules, but is able to find more unsafes. 6 of these false unsafes correspond to the missed true verdicts and are spurious due to imprecision. And 13 unsafes corresponds to unknowns in **Base** configuration, which means **Havoc** finds new unsafes.

We checked the 8 true modules, which are proved as correct. The precise approach proves, that the device can not be initialized in an appropriate way, and thus there is no race. Actually, this is a problem of the environment model (preparation of verification task), as it missed semantics of the data.

We analyzed some part of produced unsafes. There are several different race conditions per a verification task, we call them warnings. The true positive rate is about 42% (34 correct false and 47 incorrect false warnings). The main cause of false alarms are problems with memory model (>90%). The rest of false alarms are related to kernel specifics (for instance, interrupts handling), function pointer analysis, different synchronization primitives, and other specific cases. So, we have not faced problems due to imprecision in the thread-modular approach, similar to SV-COMP benchmarks. It confirms our thesis, that SV-COMP benchmarks contain mostly complicated, but artificial cases.

Table 2. Evaluation on drivers/net/ of Linux 4.2.6

Approach	Base	Havoc
False verdicts	6	26
True verdicts	262	254
Unknowns	157	145
CPU time (s)	137000	125000

We reported most of the true race conditions and they were confirmed by Linux maintainers. However, most of the bugs were found in ancient drivers and nobody wants to fix it. Only a couple of patches are applied to the upstream as a part of Google Summer of Code project.

15. RELATED WORK

The existing approaches to the analysis of multithreaded programs have different features and performance. On the one side, there are precise approaches, which can prove the correctness of the program under certain assumptions. Starting from the bounded model checking, most of the approaches investigate different techniques to reduce state space. The examples of the optimizations are partial-order reduction [1], context bounding [24, 25], etc.

An attempt to abstract from an irrelevant environment is a thread-modular approach, which was first suggested by [6]. This version does not use any abstraction. A thread modular approach to formal verification was presented in [26]. The idea is to provide invariants for every process, which together imply the formal requirement. The evaluation is provided for two protocols for mutual exclusion. A predicate abstraction was composed of a thread-modular approach in [27]. The main distinction of the presented approach was that there was only one thread in several copies. Thus, the environment of the thread is formed by itself. Also, there were no synchronization primitives considered.

An extension of the thread-modular approach, which also uses an abstraction, is firstly presented in [28] and then implemented in TAR [7]. Our approach has the following main differences:

– TAR considers locks as ordinary variables. Our tool has a special Lock Analysis, which is composed of other analyses. That allows to avoid extra refinement iterations, as the analysis already handles it.

– TAR applies thread effects, which are precisely related to thread operators. Our tool provides a possibility to abstract (operator $\cdot|_p$) and to join (operator *merge*) different transitions. That may increase the speed of analysis but decrease its precision.

– TAR supports a fixed number of threads, whereas our approach supports unbound thread creation.

– For environment TAR uses under-approximation, and our tool – over-approximation.

A similar approach was also implemented in Threader tool [29]. Threader uses over-approximation for an environment, based on Horn closes. Similar to our approach, Threader can provide modular proofs,

but also it can search non-modular ones, in which complete interaction between threads is considered.

Many techniques provide techniques for sequentialization of the program to be able then to verify it with the aid of sequential tools [30–32]. One of the examples is WHOOP [33], which uses a sequentialization technique and does not consider thread interaction. Moreover, it strongly applies Lockset algorithm and has no way to extend the approach with other analyzes. The authors applied the tool as a front-end to a more precise verifier CORALL [34].

On the other side there exist many lightweight approaches, which can be applied to a large amount of code. Such techniques are determined by weak requirements for resources and low precision. The examples are RELAY [35] and Locksmith [36], which are evaluated on the Linux kernel source code. RELAY found thousands of warnings when analyzing the Linux kernel, and then employs unsound post-analysis filters. The tool does not consider the thread interaction at all.

In opposite to RELAY Locksmith considers thread creation points, but it does not precisely identify shared data and thread interactions. It computes a general future effect related to the whole thread. An experimental evaluation shows the tool has problems with scalability.

In [37] authors presented an extension of Andersen points-to analysis for multithreaded programs. The idea is similar to the thread-modular approach, they compute a set of operators, which can be executed in parallel, and apply the operators in other threads.

There are many specific approaches for efficient data race detection for a particular software or a property. For example, low-level software with nested interrupts [38], data race detection in FreeRTOS [39, 40], concurrent use-after-free bugs in Linux device drivers [41]. Such approaches demonstrate good results, but significantly base on property and code specifics. Our approach is pretending to be more general, nevertheless, it also may be improved by targeting on a particular code and a specific property.

16. CONCLUSIONS

The paper presents an approach for practical data race detection in complicated software. We extend an existing CPA theory and implement it in a new tool. The experiments show the benefit of the approach on large examples. On the other hand, small and complicated benchmarks are better solved with other approaches. Anyway, the approach is sound and may be improved and optimized in the future.

Thus, we may conclude, that the requirements to the new tools are fulfilled, as it is successfully applied to different software systems, including Linux device drivers. As in classical model checker approaches, there may be provided a guarantee of correctness under a certain conditions: requirement on CPA operators and condition of disjoint BnB regions.

The extended CPA theory allows to describe complicated kinds of analysis, including those ones, which are not covered by a thread-modular approach. Nevertheless, the theory does not cover all possible kinds of analysis, and, for example, efficient description of interleaving analysis will require an extension. However, this is a question for the further investigation.

One of the possible directions is to extend the thread-modular approach in such a way, that it considers some thread interaction. One of the ideas is to implement analysis, which can keep an adjustable balance between interleaving analysis and thread-modular one.

The other interesting practical improvement is the integration of different approaches into one tool. For example, a combination of fast thread-modular analysis as the first stage and precise classical analysis as the second stage, which may be implemented according to cooperative verification idea [42].

One of the weaknesses of the thread-modular approach is difficulties with the computation of a real path with interleaving. However, the path will be really helpful for the investigation and refinement of the abstraction. Thus, reconstruction of the full path from a path with transitions in the environment is in our future plans.

17. ACKNOWLEDGMENTS

The research was carried out with funding from the Ministry of Science and Higher Education of the Russian Federation (the project unique identifier is RFMEFI60719X0295).

REFERENCES

1. Abdulla, P., Aronis, S., Jonsson, B., and Sagonas, K., Optimal dynamic partial order reduction, *SIGPLAN Not.,* 2014, vol. 49, no. 1, pp. 373–384.
2. Godefroid, P., *Partial-Order Methods for the Verification of Concurrent Systems: an Approach to the State-Explosion Problem,* Berlin, Heidelberg: Springer-Verlag, 1996.
3. Basler, G., Mazzucchi, M., Wahl, T., and Kroening, D., Symbolic counter abstraction for concurrent software, in *Proc. 21st Int. Conf. on Computer Aided Verification, CAV'09,* Berlin, Heidelberg: Springer-Verlag, 2009, pp. 64–78.
4. Beyer, D., Automatic verification of C and Java programs: SV-COMP 2019, in *Tools and Algorithms for the Construction and Analysis of Systems,* Beyer, D. Huisman, M. Kordon, F. and Steffen, B., Eds., Cham: Springer Int. Publ., 2019, pp. 133–155.
5. Beyer, D., Henzinger, T.A., and Theoduloz, G., Program analysis with dynamic precision adjustment, in Proc. *23rd IEEE/ACM Int. Conf. on Automated Software Engineering, ASE 2008,* L'Aquila, Sept. 2008, pp. 29–38.

6. Flanagan C. and Qadeer, S., Thread-modular model checking, in *Proc. 10th Int. Conf. on Model Checking Software, SPIN'03,* Berlin, Heidelberg: Springer-Verlag, 2003, pp. 213–224.
7. Henzinger, T. A., Jhala, R., Majumdar, R., and Qadeer, S., *Thread-Modular Abstraction Refinement,* Berlin, Heidelberg: Springer, 2003, pp. 262–274.
8. Cook, B., Kroening, D., and Sharygina, N., Verification of boolean programs with unbounded thread creation, *Theor. Comput. Sci.,* 2007, vol. 388, no. 1–3, pp. 227–242.
9. Gupta, A., Popeea, C., and Rybalchenko, A., Threader, a constraint-based verifier for multi-threaded programs, in *Proc. 23rd Int. Conf. on Computer Aided Verification, CAV'11,* Berlin, Heidelberg: Springer-Verlag, 2011, pp. 412–417.
10. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., and Veith, H., Counter example-guided abstraction refinement, *Proc. CAV 2000: Computer Aided Verification,* Chicago, 2000, pp. 154–169.
11. Graf S. and Saidi, H., Construction of abstract state graphs with PVS, in *Computer Aided Verification,* Grumberg, O., Ed., Berlin, Heidelberg: Springer, 1997, pp. 72–83.
12. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T., Eraser: a dynamic data race detector for multi-threaded programs, *SIGOPS Oper. Syst. Rev.,* 1997, vol. **31**, no. 5, pp. 27–37.
13. Bornat, R., Proving pointer programs in Hoare logic, in *Proc. 5th Int. Conf. on Mathematics of Program Construction, MPC'00,* London: Springer-Verlag, 2000, pp. 102–126.
14. Burstall, R.M., Some techniques for proving correctness of programs which alter data structures, in *Machine Intelligence 7,* Michie, D., Ed., New York: American Elsevier, 1972, pp. 23–50.
15. Andrianov, P., Friedberger, K., Mandrykin, M., Mutilin, V., and Volkov, A., CPA-BAM-BnB: block-abstraction memoization and region-based memory models for predicate abstractions, in *Tools and Algorithms for the Construction and Analysis of Systems,* Legay, A. and Margaria, T., Eds., Berlin, Heidelberg: Springer, 2017, pp. 355–359.
16. Novikov, E. and Zakharov, I., Towards automated static veri_cation of GNU C programs, in *Perspectives of System Informatics,* Petrenko, A.K. and Voronkov, A., Eds., Cham: Springer Int. Publ., 2018, pp. 402–416.
17. Novikov, E. and Zakharov, I., Verification of operating system monolithic kernels without extensions, in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice,* Margaria, T. and Steffen, B., Eds., Cham: Springer Int. Publ., 2018, pp. 230–248.
18. Beyer, D. and Friedberger, K., A light-weight approach for verifying multithreaded programs with CPAchecker, in *Proc. 11th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2016) Telč, Czechia, Oct. 21–23, 2016,* Bouda, J., Holík, L., Kofroň, J., Strejček, J., and Rambousek, A., Eds., 2016, pp. 61–71.
19. Beyer, D. and Löwe, S., Explicit-state software model checking based on CEGAR and interpolation, in *Proc. 16th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2013, Rome, Italy, March 20–22, 2013),* Heidelberg: Springer-Verlag, 2013, pp. 146–162.
20. Beyer, D., Keremoglu, M.E., and Wendler, P., Predicate abstraction with adjustableblock encoding, *Proc. Formal Methods in Computer-Aided Design, FMCAD 2010,* Lugano, 2010.
21. Biere, A., Cimatti, A., Clarke, E.M., and Zhu, Y., Symbolic model checking without bdds, in *Proc. 5th Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems, TACAS'99,* London: Springer-Verlag, 1999, pp. 193–207.
22. Beyer, D., Henzinger, T.A., and Théoduloz, G., Configurable software verification: concretizing the convergence of model checking and program analysis, in *Proc. CAV,* Berlin, Heidelberg: Springer-Verlag, 2007, pp. 504–518.
23. Beyer, D., Löwe, S., and Wendler, P., Reliable benchmarking: requirements and solutions. *Int. J. Software Tools Technol. Transfer,* 2019, vol. **21**, no. 1, pp. 1–29.
24. Qadeer S. and Rehof, J., Context-bounded model checking of concurrent software, in *Tools and Algorithms for the Construction and Analysis of Systems,* Halbwachs, N. and Zuck, L.D., Eds., Berlin, Heidelberg: Springer, 2005, pp. 93–107.
25. Cordeiro, L., Morse, J., Nicole, D., and Fischer, B., Context-bounded model checking with esbmc 1.17, in *Proc. 18th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'12,* Berlin, Heidelberg: Springer-Verlag, 2012, pp. 534–537.
26. Cohen, A. and Namjoshi, K.S., Local proofs for global safety properties, *Formal Methods Syst. Des.,* 2009, vol. 34, no. 2, pp. 104–125.
27. Henzinger, T.A., Jhala, R., and Majumdar, R., Race checking by context inference, in *Proc. ACM SIGPLAN 2004 Conf. on Programming Language Design and Implementation, PLDI'04,* New York: ACM, 2004, pp. 1–13.
28. Malkis, A., Podelski, A., and Rybalchenko, A., Thread-modular verification is cartesian abstract interpretation, in *Proc. Theoretical Aspects of Computing – ICTAC 2006,* Barkaoui, K., Cavalcanti, A., and Cerone, A., Eds., Berlin, Heidelberg: Springer, 2006, pp. 183–197.
29. Gupta, A., Popeea, C., and Rybalchenko, A., Predicate abstraction and refinement for verifying multi-threaded programs, *SIGPLAN Not.,* 2011, vol. 46, no. 1, pp. 331–344.
30. Lal, A. and Reps, T., Reducing concurrent analysis under a context bound tosequential analysis, *Formal Methods Syst. Des.,* 2009, vol. **35**, no. 1, pp. 73–97.
31. La Torre, S., Madhusudan, P., and Parlato, G., Reducing contextbounded concurrent reachability to sequential reachability, in *Computer Aided Verification,* Bouajjani, A. and Maler, O., Eds., Berlin, Heidelberg: Springer, 2009, pp. 477–492.
32. Tomasco, E., Inverso, O., Fischer, B., La Torre, S., and Parlato, G., MU-CSeq: sequentialization of C programs by shared memory unwindings, in *Tools and Algorithms for the Construction and Analysis of Systems,* Ábrahám, E. and Havelund, K., Eds., Berlin, Heidelberg: Springer, 2014, pp. 402–404.

33. Deligiannis, P., Donaldson, A.F., and Rakamaric, Z., Fast and precise symbolic analysis of concurrency bugs in device drivers (t), in *Proc. 30th IEEE/ACM Int. Conf. on Automated Software Engineering, ASE'15,* Washington: IEEE Computer Soc., 2015, pp. 166–177.

34. Lal, A., Qadeer, S., and Lahiri, S.K., A solver for reachability modulo theories, in *Proc. 24th Int. Conf. on Computer Aided Verification, CAV'12,* Berlin, Heidelberg: Springer-Verlag, 2012, pp. 427–443.

35. Voung, J.W., Jhala, R., and Lerner, S., RELAY: static race detection on millions of lines of code, in *Proc. 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering, ESEC-FSE'07,* New York: ACM, 2007, pp. 205–214.

36. Pratikakis, P., Foster, J.S., and Hicks, M., LOCKSMITH: context sensitive correlation analysis for race detection, in *Proc. 27th ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI'06,* New York: ACM, 2006, pp. 320–331.

37. Di, P. and Sui, Y., Accelerating dynamic data race detection using static thread interference analysis, in *Proc. 7th Int. Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'16,* New York: ACM, 2016, pp. 30–39.

38. Kroening, D., Liang, L., Melham, T., Schrammel, P., and Tautschnig, M., Effective verification of low-level software with nested interrupts, *Proc. ACM Design, Automation & Test in Europe Conf. & Exhibition (DATE 2015),* Grenoble, March 2015, pp. 229–234.

39. Mukherjee, S., Kumar, A., and D'Souza, D., Detecting all high-level dataraces in an RTOS kernel, in *Verification, Model Checking, and Abstract Interpretation,* Bouajjani, A. and Monniaux, D., Eds., Cham: Springer Int. Publ., 2017, pp. 405–423.

40. Chopra, N., Pai, R., and D'Souza, D., Data races and static analysis for interrupt-driven kernels, in *Programming Languages and Systems,* Caires, L., Ed., Cham: Springer Int. Publ., 2019, pp. 697–723.

41. Bai, J.-J., Lawall, J., Chen, Q.-L., and Hu, S.-M., Effective static analysis of concurrency use-after-free bugs in Linux device drivers, in *Proc. USENIX Annu. Technical Conf. (USENIX ATC 19),* Renton, WA: USENIX Assoc., July 2019, pp. 255–268.

42. Beyer, D., Jakobs, M.-C., Lemberger, T., and Wehrheim, H., Reducer-based construction of conditional verifiers, in *Proc. 40th Int. Conf. on Software Engineering, ICSE'18,* New York: ACM, 2018, pp. 1182–1193.