DOI: 10.15514/ISPRAS-2023-35(3)-14

Predicate Abstraction Refinement in Thread-Modular Analysis

V.P. Rudenchik, ORCID: 0009-0000-6719-2594 <rudenchik@ispras.ru>
P.S. Andrianov, ORCID: 0000-0002-6855-7919 <andrianov@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Thread-modular approach over predicate abstraction is an efficient technique for software verification of complicated real-world source code. One of the main problems in the technique is a predicate abstraction refinement in a multithreaded case. A default predicate refiner considers only a path related to one thread, and does not refine the thread-modular environment. For instance, if we have applied an effect from the second thread to the current one, then the path in the second thread to the applied effect is not refined. Our goal was to develop a more precise refinement procedure, reusing a default predicate refiner to refine both: a path in a current thread and a path to an effect in the environment. The idea is to construct a joined boolean formula from these two paths. Since some variables may be common, a key challenge is to correctly rename and equate variables in two parts of the formula to accurately represent the way threads interact. It is essential to get reliable predicates that can potentially prove spuriousness of the path.
The proposed approach is implemented on top of CPAchecker framework. It is evaluated on standard SV-COMP benchmark set, and the results show some benefit. Evaluation on the real-world software does not demonstrate significant accuracy increase, as the described flaw of predicate refinement is not the only reason of false positive results. While the proposed approach can successfully prove some specific paths to be spurious, it is not enough to fully prove correctness of some programs. However, the approach has further potential for improvements.

Keywords: static verification; predicate abstraction; thread-modular analysis.

For citation: Rudenchik V.P., Andrianov P.S. Predicate Abstraction Refinement in Thread-Modular Analysis. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 3, 2023. pp. 187-204. DOI: 10.15514/ISPRAS-2023-35(3)-14.

Уточнение предикатной абстракции при раздельном анализе потоков

В.П. Руденчик, ORCID: 0009-0000-6719-2594 <rudenchik@ispras.ru>
П.С. Андрианов, ORCID: 0000-0002-6855-7919 <andrianov@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Комбинация анализа с раздельным рассмотрением потоков (Thread-Modular analysis) и предикатной абстракции является эффективной техникой верификации реального программного обеспечения. Одним из недостатков этой техники является уточнение предикатной абстракции при анализе многопоточных программ. В классической процедуре уточнения абстракции рассматривается только путь в одном потоке, и окружение Thread-Modular анализа не уточняется. Например, при применении эффекта из второго потока к первому путь к эффекту во втором потоке не уточняется.

Целью нашей работы была разработка более точной процедуры уточнения абстракции, которая бы переиспользовала имеющуюся процедуру уточнения абстракции и позволяла бы уточнять и путь в анализируемом потоке, и путь в окружении. Основная идея заключается в построении совместной логической формулы для двух путей. Так как имена переменных разных потоков могут совпасть, необходимо корректно переименовать и приравнять некоторые переменные для того, чтобы формула правильно отражала связи между потоками. Это позволяет получить предикаты, необходимые для доказательства недостижимости пути.

Предложенный подход был реализован на базе инструмента статической верификации CPAchecker. Подход был оценен на стандартном наборе задач SV-COMP и показал небольшое улучшение. Для больших программ улучшений в результатах не наблюдалась, так как описанный недостаток анализа не является единственной причиной ложноположительных результатов. Предложенный подход может успешно доказать недостижимость некоторых путей, то этого может быть недостаточно для доказательства корректности программы. Однако подход обладает дальнейшим потенциалом для совершенствования.

Ключевые слова: статическая верификация; предикатная абстракция; анализ с раздельным рассмотрением потоков.

Для цитирования: Руденчик В.П., Андрианов П.С. Уточнение предикатной абстракции при раздельном анализе потоков. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 187–204 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–14.

1. Introduction

Program verification is a process of checking if a program satisfies certain requirements. In static verification a program or its model is analyzed without actually running the code. There are multiple tools for program verification that implement various techniques targeted at different types of tasks. One of them is a reachability problem – a task of determining if a given point in a program is reachable. For reachability problem verification process can be broken down into two separate parts: 1) building a set of reached states; 2) checking if target state is in this set. While the second part is relatively simple, the first part is complex and resource-intensive. Various techniques and optimizations are developed to solve it. One of such approaches is abstraction.

There are many different types of analyses, which implement different kinds of abstractions. Using several abstractions at once can make analysis more efficient, especially for complicated pieces of code. CPA (Configurable Program Analysis) [1-2] was introduced as an approach of unifying different techniques for software verification (including abstractions). It allows combining different kinds of abstractions in various ways, so they can be used simultaneously and construct a more accurate model of a program.

In software verification approaches a model of a program is automatically extracted from the source code. It may not be accurate enough to prove certain properties of a program. Constructing more complex models is not always resource-efficient. This problem can be solved by using algorithms of iterative model refinement such as CEGAR [3], which refines abstractions using a counterexample. The algorithm iteratively refines the abstraction until it achieves a level of precision suitable for proving a specific property. Further, we will consider predicate abstraction [4], which assigns to each state a predicate that limits possible values of variables in the state.

Multithreaded programs traditionally cause additional problems for software verification. Classic approaches, which consider different combinations of thread interleavings, quickly result in state space explosion. There are other approaches, for example, Thread-Modular approach [5]–[7], which considers each thread separately in combination with some environment. The environment is constructed automatically during the verification process, and may be unique for every process. Thread-Modular approach demonstrates good performance and precision for industrial software as a target code. However, as we use abstraction technique, we need to have a refinement procedure. This presents a challenge, as the threads may interact with each other, for example, they may operate with the same shared variables or use local variables with the same names.

The paper presents a way of refining predicate abstraction in Thread-Modular approach. In Thread-Modular an error path is a path in a one thread, as threads are analyzed separately. However, it may contain effects from other threads, and there are paths to the effects in other threads. We introduce an efficient way to construct a joined boolean formula for two different thread paths. The idea is to rename local variables to avoid matching and add specific equalities of shared variables to represent dependencies of values of shared variables in different threads. Constructing a joined formula allows reusing a basic predicate refinement procedure to refine multiple paths all together. However, practical implementation poses some technical problems such as hanging caused by repeated analysis of the same path.

A current limitation of the approach is complicated thread interleavings. For example, if the analyzed thread interleaves with the second one that is also affected by the third one, the proposed approach might not be effective.

Experiments show that the approach allows refining more paths than the default predicate refinement procedure. It can successfully prove absence of errors for a certain number of tasks. However, the benefit is shown mostly on small artificial tests, as large real-world examples have a complicated thread interaction. Thus, even if the proposed predicate refinement procedure is able to remove some infeasible paths from abstractions, there are still other spurious paths due to other reasons, which do not allow to prove the correctness.

The main contributions are:

- an approach for environment refinement in predicate abstraction;
- implementation of the approach on top of the CPAchecker framework[1]. The source code is already merged in the main branch.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to the theory. Section 3 contains a motivation example with a description of the problem. The proposed solution is presented in section 4. In section 5 some implementation features are described. Evaluation details are given in section 6, and section 7 contains brief information about related work.

2. Preliminaries

2.1 Software model checking

We consider a multithreading program as target software. This is a program, which contains more than one execution thread. The threads can operate with *local* variables, which are available only to specific threads, and *shared* variables, which are available to all threads. We do not specify any interface, like, POSIX, ARINC, or other, as it is irrelevant to our analysis.

Further, we consider software model checking approach for static verification. Such approaches allow the automatic extraction of a formal model from the source code and check it against predefined specifications or *properties*.

One of such properties is *reachability*. If a specific *error* state is reachable, then the property is violated and the program is incorrect. Accordingly, if no error state is found, the program is considered to be correct.

Another possible property is absence of *data races* [8]. Theoretically, it can be expressed via reachability [9], however in practice it is more efficient to consider it separately. Further, we will consider only reachability problem, as it is simpler. However, it is possible to apply the proposed refinement procedure for verification of other properties. Also, we do not consider any specifics of weak memory models [10].

[1] https://gitlab.com/p.andrianov/cpachecker/

2.2 Abstractions

As mentioned above, instead of analyzing a program itself we analyze a model of a program. Traditionally, a model of a program is a graph built upon Control Flow Automaton (CFA). The edges represent program operators from CFA and the states represent program memory, including location from CFA (pc) and assignment of values to all variables. The states are called *concrete* ones.

Even for a one integer counter possible values are numerous. Real-world software contains thousands of variables, and using *concrete* states in analysis leads to combinatorial explosion of a state-space. One of the ideas to reduce the number of considered states is *abstraction*. Abstract states represent multiple concrete data states. There are many different kinds of possible abstractions. Our approach is based on predicate one, so, further we will consider it. In predicate abstraction [3] an abstract state contains predicates over program variables. For example, abstract state $(x = 0)$ represents many concrete states, including $(x \to 0, y \to 0)$, $(x \to 0, y \to 1)$, $(x \to 0, y \to 2)$, etc. It constrains x to have a value of zero, but does not specify values of other variables. The same way abstract state $(x \geq 0) \land (y \leq 1)$ constraints variables x and y in the way defined by the predicates.

An operator *transfer* allows to build a next abstract state for a parent state and program operation (control flow edge). In predicate abstraction the operator *transfer* is the strongest postcondition of the parent state and program operation. A set of states, which are reachable by a *transfer* from some initial state, is a *reached set*. Note, that the *reached set* is a set of abstract states, and potentially, some abstract states may represent those concrete states, which are impossible in a real execution of a program. This is, because an abstraction is an *overapproximation* of a program. This is necessary for the soundness of an analysis i.e. in order for the program to not be falsely considered correct. *Reached set* is usually represented by Abstract Reachability Graph (ARG).

Abstraction is built with a certain *precision*: high *precision* means more precise abstraction. *Precision* is formally defined by an analysis. In predicate analysis a *precision* π is a set of predicates, which are used in constructing predicate abstract states. The lowest (the weakest) predicate precision is an empty set $\pi = \emptyset$. Predicate abstraction with the empty precision will contain only trivial predicate states $\top$, which corresponds to formula **True**. They represent any concrete state.

And how can the precision be changed? For example, if the abstraction is not precise enough and contains spurious paths, there is a need to refine it. This question will be addressed in the following section.

2.3 Refining predicate abstractions with CEGAR

As we have already described, abstraction is an overapproximation of a program, so, it may omit some details. Because of such imprecision, a program can be falsely considered incorrect. Therefore, there should be a way to refine the abstraction.

Counter-Example Guided Abstraction Refinement (CEGAR) [3] is an approach for increasing precision of an abstraction. It iteratively refines an abstraction using *counterexamples*. In case of reachability problem, counterexample is a path to an error state. Let us consider the way CEGAR refines the abstraction.

First, an initial abstraction (a set of reached states) is built with a given precision. By default the initial precision is set to the lowest precision, i.e to the empty one, meaning the abstraction is built imprecisely.

Then we should check if an error state is present in the abstraction. For the initial abstraction it means just syntactical reachability, as there are no valuable predicates. If the error state is unreachable, the program is correct and the analysis finishes.

If the error state is present in the abstraction, it does not mean that it is reachable in the program since the abstraction can be imprecise. The counterexample (a path to this state) needs to be checked for feasibility precisely. If the error path is feasible in a precise model, the program is incorrect and the analysis finishes. If the error path is infeasible in the precise model, abstraction needs to be recomputed with the new precision provided by CEGAR. That is a default CEGAR loop. There are two points of interest here: how the counterexample is checked for precise feasibility and how new precision is obtained. Further we will consider these issues in case of predicate abstractions.

In predicate abstraction a *path formula* is calculated in order to check the counterexample for feasibility. Path formula is a conjunction of predicates that correspond to path operators. For instance, if a path contains three consecutive operators: an assignment operator $a = 1$, a conditional operator *if* $(a \geq 0)$ and another assignment operator $b = 1$, the corresponding path formula is $a = 1 \land a \geq 0 \land b = 1$. There is no contradiction in the formula, so it is satisfiable. Formula $a = 1 \land a < 0 \land b = 2$ corresponds to operators $a = 1$, *if* $(a < 0)$ and $b = 2$. This formula is unsatisfiable.

Satisfiability of a path formula is equivalent to existence of such input data (initial values of variables) that the error state is reachable. The satisfiability of the formula is checked by a specific external tool – SAT solver. If the formula is satisfiable, then the error state is considered reachable and analysis ends. Feasibility of the path in the abstraction but not in the program means that the abstraction is not precise enough and needs to be refined.

The way precision is extracted from a spurious counterexample depends on the abstraction. Moreover, there are different ways to refine predicate or any other abstraction. We are using Craig interpolation [11] to extract predicates from an unsatisfiable path formula. There is an interpolation theorem, which claims that for any logical formulas φ, ψ such that $\varphi \land \psi \equiv \bot$ there exists logical formula ρ, called an *interpolant*, such that every non-logical symbol in ρ occurs both in φ and ψ, $\varphi \rightarrow \rho$ and $\psi \land \rho \equiv \bot$.

In practice, we use interpolating solvers such as MathSAT [12], Z3 [13], or CVC5 [14], to calculate the interpolants. Being a conjunction of predicates, an unsatisfiable path formula can be split in two parts, usually in multiple ways, to satisfy the precondition of the theorem. Solver extracts multiple interpolants from a path formula, those interpolants are then added to precision. Note that interpolants are not the only way to extract new precision.

It is important to mention one of the optimizations for efficient abstraction rebuild. It is called *lazy abstraction* [15]. The main idea is to rebuild not all abstraction after refinement, but to identify the changed parts and reconstruct only them. During the refinement procedure a *refinement root* is identified. This is the state, which is a common parent of all changed subtrees in the reached set. The subtree is removed after refinement, and the analysis continues from the *refinement root*.

One more optimization, which also should be mentioned, is Adjustable Block Encoding (ABE) [16]. Its main idea is to avoid reconstructing predicate formulas in every state. Instead, formulas are constructed for every block that is composed of multiple states. Because of it, interpolants are usually not set for every abstract state. We do not need to describe this optimization in detail since it is irrelevant to our work.

This concludes an overview of predicate abstraction refinement with CEGAR. So far, we have only considered a path in a single thread. It is not immediately obvious how this refinement procedure can be applied to an analysis of multithreaded programs where a path contains operators from different threads. In the following section we describe an approach to analysis of multithreaded programs that can be combined with CEGAR.

2.4 Thread-Modular Analysis

Thread-Modular analysis [5]–[7] is an approach for verification of multi-threaded programs. Unlike algorithms that rely on complete enumeration of possible thread interleavings, Thread-Modular analysis uses an abstraction of thread interactions. It analyzes each thread individually with consideration of an *environment*, which is a model (abstraction) of possible effects that threads can have on each other [7]. The more accurate the environment is, the more precise analysis is going to be. And less accurate and more abstract models can be used for analyzing large programs for which brute-force approaches are not applicable.

Interactions of threads can be formally described in terms of *projections*. A *projection of an operation* is an effect that the operation can have on other threads or an overapproximation of such effect. A projection can also contain a condition under which its effect can be applied. For instance, assigning a value to a local variable does not affect other threads, so a projection of this operation is empty. Now let us consider an assignment $x = 0$ to a global variable x. Its projection may contain the same assignment $x = 0$. Alternatively, a projection may be more abstract and contain assignment $x = *$, meaning “the thread can change a value of variable x to anything”. Therefore, environment can be defined as a set of projections of all operators in the program.

While analyzing each individual thread, Thread-Modular analysis builds projections of every operator of this thread. The projections are part of the environment for other threads. After the primary analysis of each thread, Thread-Modular analysis considers an effect of the environment. For that purpose it checks each projection from the environment and each state in other threads if they are *compatible*, i.e. if the projection can be applied to the state. In predicate abstraction two predicate abstract states are considered compatible if a conjunction of their predicates looks satisfactory. If a projection and a state are compatible, the effect of the projection is applied to the state which results in creation of a new state called *applied state*. Projections express an effect of other threads, and applied states contain the effect, which is applied to the particular state in the current thread.

Applied states and the states that are reachable from them by operator *transfer* are added to the *reached set*. The state to which the projection was applied is considered to be a parent state of the applied state. Because of this, new paths are created that represent how threads interact with each other. Note that the applied state may be the same as the parent state, meaning the effect does not change anything.

An illustration of the approach is given in *Fig. 1*. There is a part of ARG representing the first thread and a part of ARG representing the second thread. Assignment operator $x = 0$ that follows state *B* from the second thread can be projected.

If the new *projection* is compatible with the state A from the first thread, it can be applied to the state *A*. The new *applied* state corresponds to application of the effect $x = 0$ to the first thread. The analysis continues in the first thread from the new *applied* state.

As Thread-Modular approach considers threads separately, the error path is also a path in a separate thread. However, the path may contain different effects, representing the thread interaction. The next section shows the problem during refinement of paths in the Thread-Modular case.

3. Motivating example

Let us consider the program in the *Fig. 2*. It contains two threads *thread*1 and *thread*2, both can change values of global variables *a* and *b*. The first thread assigns the value of 1 to variables *a* and *b* with mutex protection. Then it releases the mutex and checks that the value of *b* has not changed.

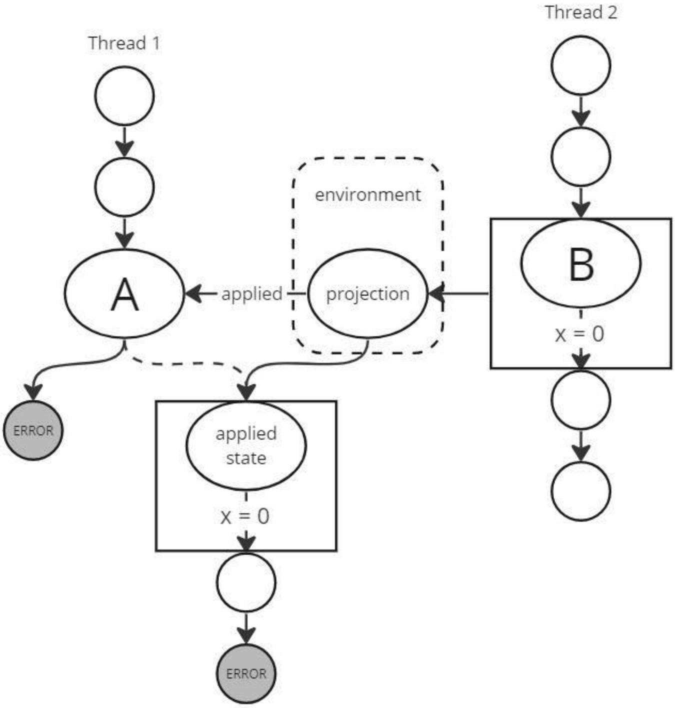
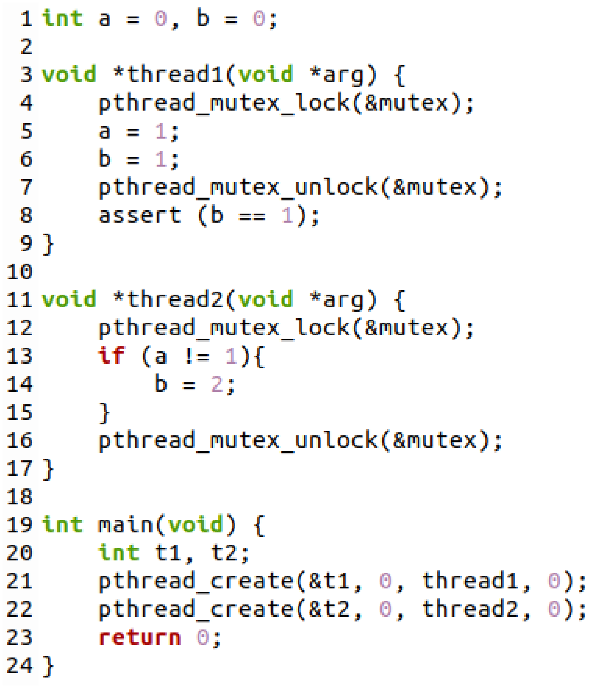


Fig. 1. Thread-Modular approach



```c
1 int a = 0, b = 0;
2
3 void *thread1(void *arg) {
4     pthread_mutex_lock(&mutex);
5     a = 1;
6     b = 1;
7     pthread_mutex_unlock(&mutex);
8     assert (b == 1);
9 }
10
11 void *thread2(void *arg) {
12     pthread_mutex_lock(&mutex);
13     if (a != 1){
14         b = 2;
15     }
16     pthread_mutex_unlock(&mutex);
17 }
18
19 int main(void) {
20     int t1, t2;
21     pthread_create(&t1, 0, thread1, 0);
22     pthread_create(&t2, 0, thread2, 0);
23     return 0;
24 }
```

Fig. 2. Example of a program

The second thread checks if the value of *a* has changed and if it has not, then it changes the value of *b* to 2; all while the mutex is locked. The error label (assertion in line 8) is not reachable, because change of the variable *b* is allowed only in case of $a \neq 1$. However, analyzing the program with CPAchecker using Thread-Modular analysis with default predicate refinement returns a counterexample, meaning the error label is feasible. The reason this is happening is the inability to refine the predicate abstraction.

First, the analysis constructs a path to the error state. The path is in the first thread, as the error state (assert in line 8) is in the first thread. Initially, the predicate precision in empty, the path corresponds to operators $a = 1$ in line 5, $b = 1$ in line 6, and *assert* in line 8 and does not contain any effects.

The corresponding path formula: $(a = 1) \wedge (b = 1) \wedge (b \neq 1)$. It is unsatisfiable, because the value of *b* is not considered in the abstraction. So, the abstraction is successfully refined and the interpolant $b = 1$ is added to the predicate precision.

In the next iteration of the analysis another path is constructed. The path is in the first thread and it contains an application of the effect $b = 2$ (line 14) from the second thread right before line 8. The path corresponds to succession of operators $a = 1$ (line 5), $b = 1$ (line 6), $b = 2$ (line 14, thread 2), and *assert* in line 8. Actually, this effect cannot be applied since the operation $b = 2$ can only be executed if *a* does not equal 1 (line 13) but the value of variable *a* before line 8 is equal to 1.

The path is spurious, abstraction is not precise enough, because it does not contain any predicate over value of *a*. And the abstraction is supposed to be refined. But default refiner fails to prove that the effect cannot be applied.

The counterexample is shown in *Fig. 3* (highlighted in dark color). State A corresponds to the line 8, before assertion check. State B corresponds to the line 14 with operation $b = 2$. So, the projection represents the effect from operation $b = 2$ for other threads. It is indeed feasible as a path in a single thread if the projection is applied. But the projection could not have been applied. Default predicate refiner refines only a path to the error state, and it does not check the projection, state from which it was projected (state B in *Fig.3*) or a path to that state. Predicate abstraction of the second thread is not refined, and it stays not precise enough to exclude the application of the projection. Because of that, spurious counterexample is not ruled out.

If the predicate precision contained predicates $a == 1$ and $a \neq 1$ then state A would contain predicate $a == 1$ and both state B and the projection would contain predicate $a \neq 1$. That would make state A and the projection incompatible and the projection would have been applied. The question is, how to obtain such predicates.

4. Proposed solution

4.1 An approach overview

Let us consider a path to an error state in an abstraction. This is a path in a single thread, and it contains an applied effect, meaning it is affected by another thread. Let the path in the single thread be reachable in the abstraction. If the effect cannot been applied, the path is technically unreachable. One would naturally expect a refiner to detect the unreachability of the path and construct a more precise environment in which the effect would not be applied. However, the default refiner lacks the capability to do so as it only refines the path in a single thread and does not refine the environment. It is unable to prove that the effect cannot be applied. Therefore, the analysis considers the path feasible and the error label can be falsely recognized as reachable.

The problem arises, as the default refinement procedure considers only thread abstraction and misses the environment. So, we need an efficient way to refine two parts of the abstraction (thread and environment) together. And it means that the counterexample now consists of two parts: a path in thread (main path) and a path in an environment. If the two paths are spurious, we need to obtain interpolants that can potentially prove the incompatibility of these paths, and add them to precision. The next step is to determine an imprecise part of the abstraction and rebuild it with new precision.

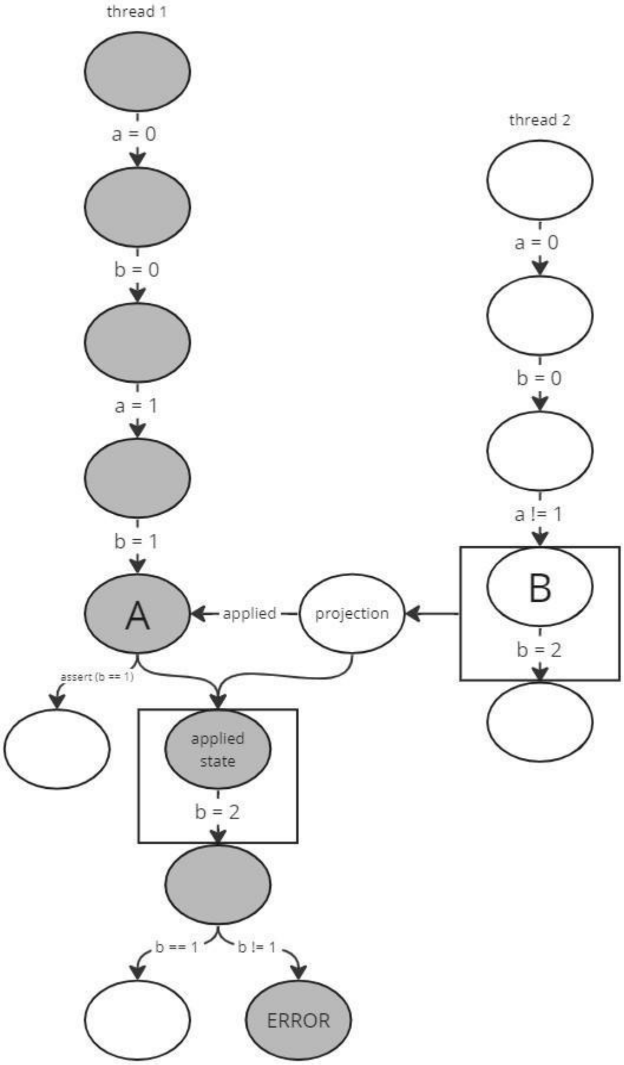


Fig. 3. Counterexample

One of the options to refine two paths is to develop a new refiner specifically for this task. However, this approach would lead to a considerable amount of code duplication, since only the refinement target is changed, not the refinement technique itself. Instead, we choose to extend an existing approach, and refine two paths altogether by reusing an existing refiner. While reusing a large piece of code is generally practical and efficient, it requires addressing certain issues to ensure successful code reuse. Since the input of a default refiner is a single path, the two paths need to be joined into one to be refined by it. Moreover, names of local variables may overlap, and global variables may appear in both paths, so they need to be carefully renamed in order to avoid false dependencies. Although the interpolation procedure stays the same, we still need some post-processing of obtained interpolants. Now we present the approach in more detail.

Consider an instance of a projection depicted in *Fig. 3*, where the projection originates from state B and is applied to state A. We consider two paths: the first is a path to state A and the second is the path to state B. The paths are reconstructed using ARG relations. A path formula, which is a conjunction of predicates that correspond to program operators, is constructed for each path, as it is performed in default predicate analysis. In order to check simultaneous feasibility of these two paths, we check satisfiability of a conjunction of these two path formulas. Since the resulting formula is still a conjunction of predicates, it can be processed like a regular path formula of a single path. And then we request SAT solver about its satisfiability. However, the process is not as straightforward due to complications in joining the formulas.

4.2 Joining formulas

The path to the error state in *Fig. 3* contains multiple assignments to the same variable. For instance, $b = 0$ and $b = 1$ are successive assignments to variable b. If the path formula contained the unsatisfiable conjunction of the corresponding predicates $b = 0 \land b = 1$, it would be unsatisfiable regardless what other predicates it contains. Thus, path formulas are built with SSA indexation [17], which assigns an index to each variable that increments with each assignment. Variables with different indices are considered different. And since each variable is only assigned a value once, there are no collisions in path formula caused by multiple assignments.

SSA indexation can cause problems when joining formulas. Each thread (path) has its own SSA indexation. That means that a global variable can have multiple overlapping sets of indices, one for each thread. In a joined formula two instances of the same global variable from different threads but with equal indices will be considered as the same variable. This can cause unexpected dependencies. This problem can be solved by renaming variables in one of the threads.

For instance, we rename global variable *b* in the second (environment) formula to *env_b*. Adding a special symbol, which is not permitted in a variable name in real code, to the variable ensures that the newly renamed variable does not coincide with any other variable.

However, renaming loses relation between two threads, and we need to artificially restore it. Values of global variables at the point of projection application in both threads must be equal. In the opposite case, for example, if a global variable *b* in one thread is equal to 1 and in the second thread the same variable *b* is equal to 2, it means that the two states are incompatible. In order for a path formula to reflect that, we need to add variable equalities. Each global variable with the latest index in one thread is considered equal to this global variable with the latest index in the other thread. The equalities are then added to the joined path formula as new predicates in a conjunction. That ensures that the formula reflects relation between threads.

Another problem occurs if formulas contain local variables. There can be two local variables in different threads with identical names. When joined into one path formula they can potentially be treated as one global variable, which can affect satisfiability of the formula. To avoid that, all local variables of one of the two threads should be renamed. For instance, similarly to global variables, we rename local variable i in the second (environment) formula to *env_i*. However, we do not add any variable equalities for the local variables.

The resulting formula accurately represents two paths and a relation between them. If the formula is satisfiable then the two paths are considered feasible simultaneously and the error state is reachable. If this formula is unsatisfiable then the two paths are not feasible simultaneously and abstraction needs to be refined. The default Craig interpolation can be used to get interpolants. Usually, a path formula can be split into parts φ and ψ such that $\varphi \land \psi \equiv \bot$ in multiple ways. Interpolation is then performed for each partition to obtain more potentially useful predicates. The joined path formula is no exception. It is a conjunction of predicates and interpolants are extracted from it just like from any other path formula.

Let's take a closer look at predicates that are obtained during the interpolation. Let's consider a projection *proj* that was applied after state *A* and that was projected from the state *B*. Let μ_1 and μ_2 be path formulas for the paths to *A* and *B* respectively. If $\mu_1 \land \mu_2 \equiv \bot$ (meaning *proj* could not have been applied) then Craig interpolation theorem can be applied for such unsatisfiable conjunction. Therefore, there exists a predicate ρ_1 such that every non-logical symbol in ρ_1 occurs both in μ_1 and

μ_2, $\mu_1 \Rightarrow \rho_1$ and $\mu_2 \land \rho_1 \equiv \bot$. Since $\mu_2 \land \rho_1$ is an unsatisfiable conjunction, there exists predicate ρ_2 such that every non-logical symbol in ρ_2 occurs both in μ_2 and ρ_1, $\mu_2 \Rightarrow \rho_2$ and $\rho_1 \land \rho_2 \equiv \bot$.

Predicates ρ_1 and ρ_2 are then added to precision. A part of the abstraction is reconstructed with the updated precision (see lazy abstraction). In the default refinement procedure the rebuilt part of abstraction does not include states in the environment, but in order to eliminate the infeasible paths a part of the environment also has to be reconstructed. Predicate ρ_1 is an implication of path formula μ_1 which resembles a path to state A. Since predicate state is built as the strongest postcondition of the path, predicate state of state A will contain predicate ρ_1 in the rebuilt ARG. Likewise, predicate state of state B will contain predicate ρ_2. Since $\rho_1 \land \rho_2 \equiv \bot$, states A and B are now considered incompatible, and the projection cannot be applied. That proves infeasibility of the counterexample.

Finally, let's see how the counterexample in *Fig. 3* is refined. The first path is the path to state A and its path formula is $a_1 = 0 \land b_1 = 0 \land a_2 = 1 \land b_2 = 1$. Note, the subscript here is an SSA index. The second path is the path to state B and its path formula is $a_1 = 0 \land b_1 = 0 \land a_1 \neq 1$. By renaming variables in the second formula we obtain $env\_a_1 = 0 \land env\_b_1 = 0 \land env\_a_1 \neq 1$. After that we join the two formulas and add variable equalities: $a_2 = env\_a_1 \land b_2 = env\_b_1$. The resulting formula is

$$\begin{gathered} a_1 = 0 \land b_1 = 0 \land a_2 = 1 \land b_2 = 1 \land \\ \land\, env\_a_1 = 0 \land env\_b_1 = 0 \land env\_a_1 \neq 1 \land \\ \land a_2 = env\_a_1 \land b_2 = env\_b_1 \end{gathered}$$

Precise extracted interpolants depend on the solver and block encoding (see ABE). In theory, we can obtain interpolants $a_2 = 1$ and $env\_a_2 \neq 1$. The variables in the interpolants are then reverted to their original names, in our case by removing the prefix. Resulting predicates $a = 1$ and $a \neq 1$ are added to precision. In the rebuilt abstraction predicate state of state A would contain predicate $a = 1$ and predicate state of state B would contain predicate $a \neq 1$. Since $a = 1 \land a \neq 1 \equiv \bot$, states A and B are now incompatible, meaning the projection cannot be applied. That proves infeasibility of the counterexample.

4.3 Limitations of the approach

In theory new interpolants must exclude a spurious error path from the abstraction. Actually, an error path may be found again due to different reasons: optimizations, errors, unsupported cases, and so on. To avoid infinite loops of CEGAR loop, there is a technique for detection of repeated counterexamples. The default predicate refinement procedure compares error paths from last two CEGAR iterations and if they are equal stops the analysis. However, there are some difficulties in thread-modular case.

First, paths with effects can be falsely deemed equal. The default technique for detection of repeated counterexamples considers paths equal if their edges are identical, i.e. if paths correspond to the same sequence of executed operators. This approach does not take into account paths to effects if there are effects applied. For instance, two similar paths, each with different effects applied to the same state, are considered equal. The issue leads to false errors. In our approach this issue is more crucial since the environment can be refined and a new path can differ from the previous one solely based on paths in the environment.

Secondly, reusing the refiner multiple times in a single CEGAR iteration can lead to losing information about repeated paths, potentially resulting in looping. The default refiner procedure is run multiple times for one counterexample with applied effects. Both the path to the error state itself and the pairs of main paths and paths in the environment are refined, all within the same iteration. That interferes with error path detection. Default refiner only caches one path from the previous refinement, and deletes it after comparing it with a next path. So, if a repeated counterexample contains an effect, the refinement procedure will be executed at least twice for it. The counterexample will be cached during the first execution but will be overwritten in the second one. As a result, the repetition of such a counterexample will go undetected, causing looping.

Caching all paths, which is an existing option, will not resolve the issue either. The same effect can be applied to the same state in different iterations. That means that the same joined path may be refined multiple times. However, that does not indicate repetition of counterexample and should not stop the analysis.

So far we have only considered a case where an error path contains only one applied projection that originates from a single state from the other thread. But in reality there might be several projections applied. If multiple projections are applied to the main path, meaning there are several effects applied to the first thread, we may iteratively check all of them one by one. If a main path and any path to one of these effects are not feasible together, the path is considered spurious and abstraction needs to be refined.

One more problem occurs when a projection is projected from multiple effects. Such projection can be created by the optimization which merges projections from different states into a single one. In that case all pairs of a path to each of these states and a path to the applied state are refined. In theory, the path should be considered spurious if at least one of the pairs of paths is infeasible simultaneously. But in reality, that projection merging optimization is not consistent with this theory. Because of this, we consider a projection application spurious if each path to each effect it was projected from is spurious.

Another problem occurs when projections are applied to the different threads. For example, one projection is applied to the first path, and a path to that projection in the second thread contains an effect from the third thread. The part of the environment that is important for the path to the projection will not be refined. The natural idea is to include recursion in the refinement process, but it is not yet clear if it would work somewhat effectively or work at all, considering other already existing limitations. The problem occurs when effects are applied not successively, multiple times and etc. Currently, this is a limitation of our approach.

5. Implementation features

The proposed approach was implemented on top of the CPAchecker framework as a separate predicate refiner. Its input is an error path in a main thread. First, the default refinement procedure is applied. If the path is spurious, the abstraction is refined with default predicate abstraction refinement procedure. It means that the contradiction is found in the path in one thread without any thread interaction. If the main path is feasible, it is analyzed with the proposed approach. For that purpose, we find all applied states in the path. An applied state is applied from a projection that can be projected from multiple states in another thread. For each such state the refiner checks feasibility of two paths: a path to the state in another thread and the main path.

It is important to note that the implemented approach differs from the presented theory. Theoretically, the first set of predicates should be obtained by interpolating a combination of two path formulas. That part is fully implemented in the actual code. However, the second set of predicates, in theory, should be obtained from interpolating a combination of path formula and the first set of predicates. Implementing this within the framework of the given task would be problematic. Given our decision to reuse an existing refiner which only input is a path, not a set of predicates; it would be quite a challenge to acquire these exact predicates. Nonetheless, the implemented method still has potential to prove infeasibility of a path.

One of the implementation features is refining two different combinations of paths. A main path and a path to an effect are concatenated in both possible ways and both combinations are refined. Solver extracts different predicates from these two constructed paths and both of these sets of predicates are necessary to prove spuriousness of the counterexample. Additionally, if two combinations of paths are refined, the already existing code provides correct refinement root (a root of the subtree in the ARG that is rebuilt with new precision).

As it has been established, repeated counterexample detection is a problem. The same error path can be rediscovered again and again, which leads to hanging. To solve it, we integrated detection of repeated counterexample into our refiner. It checks if the last two paths in main thread are equal and caches the main path to the error state until next iteration. That effectively prevents looping.

The previously mentioned issue of paths being falsely regarded as equal also requires a suitable solution. In default repeated counterexample detection paths are considered equal if the (ordered) sets of executed operators are equal. Comparing paths by states is problematic since it would require caching a considerable part of ARG. We implemented an enhanced method of comparing paths by edges. Apart from edges in main paths it also compares edges in all paths to applied effects. It allows differentiating between paths with effects more effectively, but does not completely eliminate the possibility of false repeated counterexample detection.

6. Evaluation

The proposed approach was evaluated on standard benchmark set SV-COMP[2]. The benchmark set contains 161 tasks from directories:

- *pthread/;*
- *pthread-C-DAC/;*
- *pthread-divine/;*
- *pthread-ext/;*
- *pthread-memsafety/;*
- *pthread-atomic/;*
- *pthread-complex/;*
- *pthread-driver-races/;*
- *pthread-lit/;*
- *pthread-nondet/.*

The tasks are mostly artificially created tests with about 1 KLoc and 2-3 worker threads. They may contain a specific synchronization, like atomics, Dekker algorithms and others. We evaluated the new approach against two existing ones.

- **Default**. The default predicate refiner, which refines only one error path without considering other threads.
- **Simple**. The simplified version of refinement that checks feasibility of every path (including paths to effects) separately. Thus, it is more precise than **Default**, as it is possible to exclude paths to infeasible effects.
- **Effect**. The proposed approach for simultaneous refinement of two paths.

The tool was run with the thread modular approach over predicate analysis. The following options were used:

- precise encoding of environment actions;
- SMTInterpol is used for SAT check and interpolation;
- support for the same threads in tests.

The experiments were performed on a machine with Intel® Core™ i5-8250U CPU @ 1.60GHz × 8 and 8 GB of RAM, using 4 CPU cores; with Ubuntu 22.04.2 LTS. Timeout was set to 5 minutes. The results are presented in *Table 1*.

Table 1. Evaluation on SV-COMP benchmarks

Approach	**Default**	**Simple**	**Effect**
Correct results:	50	38	44
• Correct true	20	20	22
• Correct false	30	18	22
Incorrect results	71	51	52
• Incorrect true	0	0	0
• Incorrect false	71	51	52
Unknowns	40	72	65
• Timeouts	12	6	12
• Repeated Counterexample error	0	41	40
• Other Unknowns	28	25	13
CPUtime, s	6040	3969	6780

The proposed approach was able to prove correctness of two tests, which both thread-modular analysis and the simplified version of presented approach falsely considered incorrect. The simplified version didn’t show any improved results.

The most frequently encountered error (both for **Effect** and **Simple**) was the repeated counterexample error, which indicates that the analyses recognized a counterexample as spurious but failed to refine the abstraction, leading to the counterexample being rediscovered. One possible explanation for this is that the obtained interpolants were insufficient to eliminate the path. Some errors were falsely reported due to the imperfect nature of repeated counterexample detection. At least three tests falsely reported a repeated counterexample error. The decreased amount of correct (and incorrect) false results is also caused by the repeated counterexample error.

As expected, the proposed approach is more time-consuming. Most of the extra time is spent on refining joined paths. The simplified version (**Simple**) averaged in less time than the default approach only because it reported repeated counterexample error almost immediately on several time-consuming tests.

The proposed approach was able to prove correctness of a motivation example (program in *Fig. 2*).

We also evaluated the approach on a benchmark set of more complicated tasks, based on Linux device drivers. Each task contains about 10 KLoc and about 5 threads. There are 7 such tasks. The proposed approach did not show any improvement, it mostly reported repeated counterexample error.

The reason for that is complicated thread interleavings. The proposed approach can refine specific paths, but eventually a path will be constructed that it cannot refine. A common example of such path is one where effects are applied to the different threads: one effect is applied from the second thread to the first thread and a path to that effect contains another effect application. However, in smaller tests, the predicates obtained during the first few iterations are typically enough to prevent such path from being constructed in the first place.

The evaluation results show benefit on a small subset of the benchmarks. The proposed approach did not show any improvement on complicated tests, since it is not targeted to analyze intricate thread interleavings. While it can successfully prove infeasibility of counterexamples, this is often, but not always, not enough to prove correctness of a program. It works in a reasonable time and has potential for future improvement, as the issue with repeated counterexamples is mostly technical.

7. Related work

There are different approaches to the analysis of multi-threaded programs. They have different features and performance.

Precise approaches, based on bounded model checking techniques, investigate different techniques to reduce state space. The examples of the optimizations are partial-order reduction [18], context bounding [19-20], etc. They consider thread interleavings, and they do not have such problems with environment refinement. We do not dive deep into BMC approaches, and concentrate on thread-modular ones.

Thread-modular approach was first suggested by [21] and a predicate abstraction was composed with a thread-modular approach in [22]. There was only one thread in several copies, so, the environment of the thread is formed by itself.

An extension of the thread-modular approach, which also uses an abstraction, is firstly presented in [23] and then implemented in TAR [5]. One of the main difference is underapproximation of the environment. So, the approach does not need environment refinement.

A similar approach was also implemented in Threader tool [24]. Threader uses over-approximation for an environment, based on Horn clauses.

A thread modular approach to formal verification was presented in [25]. The idea is to provide invariants for every process, which together imply the formal requirement.

8. Conclusion

The paper presents an approach for predicate refinement in case of Thread-Modular analysis. The basic idea is to join thread-parted formulas into a single one, and check its satisfiability to determine whether two paths are feasible simultaneously.

Refinement of two paths in combination provides higher precision for the analysis. Because of this, specific spurious paths can be eliminated and a program can be proven to be correct. The evaluation results show benefit on medium-sized programs. Large programs contain intricate thread interleavings and the proposed approach is not enough to prove their correctness.

While the results show potential of the approach, there is room for future improvement. Some ideas for future work include recursive application of the approach to paths in the environment and improving the detection of repeated counterexamples.

Overall, the approach presented in this paper can be used in analyzing small and medium-sized multithreaded programs. It can successfully prove the correctness of programs that it is targeted at. Its efficiency can be increased by resolving technical problems that arise in its implementation.

References

[1]. D. Beyer, T. A. Henzinger, and G. The´oduloz, “Configurable software verification: concretizing the convergence of model checking and program analysis,” in Proceedings of CAV, (Berlin, Heidelberg), pp. 504– 518, Springer-Verlag, 2007.

[2]. D. Beyer, T. Henzinger, and G. Theoduloz, “Program analysis with dynamic precision adjustment,” in Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on, pp. 29– 38, sept. 2008.

[3]. M. Mandrykin, V. Mutilin, and A. Khoroshilov, “Vvedenie v metod CEGAR – utochnenie abstraktsii po kontrprimeram [Introduction to CEGAR – Counter-Example Guided Abstraction Refinement],” Trudy ISP RAN [Proceedings of ISP RAS], vol. 24, pp. 219–292, 2013.

[4]. S. Graf and H. Saidi, “Construction of abstract state graphs with PVS,” in Computer Aided Verification (O. Grumberg, ed.), (Berlin, Heidelberg), pp. 72–83, Springer Berlin Heidelberg, 1997.

[5]. T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer, Thread-Modular Abstraction Refinement, pp. 262–274. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[6]. A. Gupta, C. Popeea, and A. Rybalchenko, “Threader: A constraint-based verifier for multi-threaded programs,” in Proceedings of the 23rd International Conference on Computer Aided Verification, CAV’11, (Berlin, Heidelberg), pp. 412–417, Springer-Verlag, 2011.

[7]. P. Andrianov, “Analysis of correct synchronization of operating system components,” vol. 46(8), p. 712–730, Programming and Computer Software, 2020.

[8]. P. Andrianov and V. Mutilin, “Scalable thread-modular approach for data race detection,” Frontiers in Software Engineering Education, pp. 371– 385, 2020.

[9]. D. Kroening and M. Tautschnig, “Cbmc – c bounded model checker,” vol. 8413, pp. 389–391, 04 2014.

[10]. J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig, “Software verification for weak memory via program transformation,” ESOP’13, (Berlin, Heidelberg), p. 512–532, Springer-Verlag, 2013.

[11]. W. Craig, “Three uses of the herbrand-gentzen theorem in relating model theory and proof theory,” Journal of Symbolic Logic, vol. 22, pp. 269– 285, Sep 1957.

[12]. R. Bruttomesso, A. Cimatti, A. Franze´n, A. Griggio, and R. Sebastiani, “The mathsat 4smt solver,” in CAV, pp. 299–303, 2008.

[13]. L. de Moura and N. Bjørner, “Z3: an efficient smt solver,” vol. 4963, pp. 337–340, 04 2008.

[14]. H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, Mohamed, M. Mohamed, A. Niemetz, A. No¨tzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, cvc5: A Versatile and Industrial-Strength SMT Solver, pp. 415–442. 01 2022.

[15]. T. A. Henzinger, R. Jhala, and R. Majumdar, “Lazy abstraction,” in Symposium on Principles of Programming Languages, pp. 58–70, ACM Press, 2002.

[16]. D. Beyer, M. E. Keremoglu, and P. Wendler, “Predicate abstraction with adjustable-block encoding,” in Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010, Lugano, October 20-23), pp. 189–197, FMCAD, 2010.

[17]. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” ACM Trans. Program. Lang. Syst., vol. 13, pp. 451–490, 10 1991.

[18]. P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, “Optimal dynamic partial order reduction,” SIGPLAN Not., vol. 49, pp. 373–384, jan 2014.

[19]. S. Qadeer and J. Rehof, “Context-bounded model checking of concurrent software,” in Tools and Algorithms for the Construction and Analysis of Systems (N. Halbwachs and L. D. Zuck, eds.), (Berlin, Heidelberg), pp. 93–107, Springer Berlin Heidelberg, 2005.

[20]. L. Cordeiro, J. Morse, D. Nicole, and B. Fischer, “Context-bounded model checking with esbmc 1.17,” in Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS’12, (Berlin, Heidelberg), pp. 534–537, Springer-Verlag, 2012.

[21]. C. Flanagan and S. Qadeer, “Thread-modular model checking,” in Proceedings of the 10th International Conference on Model Checking Software, SPIN’03, (Berlin, Heidelberg), pp. 213–224, Springer-Verlag, 2003.

[22]. T. A. Henzinger, R. Jhala, and R. Majumdar, “Race checking by context inference,” in Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI ’04, (New York, NY, USA), pp. 1–13, ACM, 2004.

[23]. A. Malkis, A. Podelski, and A. Rybalchenko, “Thread-modular verification is cartesian abstract interpretation,” in Theoretical Aspects of Computing – ICTAC 2006 (K. Barkaoui, A. Cavalcanti, and A. Cerone, eds.), (Berlin, Heidelberg), pp. 183–197, Springer Berlin Heidelberg, 2006.

[24]. A. Gupta, C. Popeea, and A. Rybalchenko, “Predicate abstraction and refinement for verifying multi-threaded programs,” SIGPLAN Not., vol. 46, pp. 331–344, Jan 2011.

[25]. A. Cohen and K. S. Namjoshi, “Local proofs for global safety properties,” Form. Methods Syst. Des., vol. 34, pp. 104–125, Apr. 2009.

Информация об авторах / Information about authors

Вероника Павловна РУДЕНЧИК – студентка 5 курса специалитета механико-математического факультета МГУ, лаборант ИСП РАН. Научные интересы: статическая верификация, анализ многопоточных программ.

Veronika Pavlovna RUDENCHIK – 5th year student of the Specialist’s program of MSU, faculty of mechanics and mathematics; laboratory assistant at ISP RAS. Research interests: software model checking, analysis of multithreaded software.

Павел Сергеевич АНДРИАНОВ – научный сотрудник ИСП РАН, кандидат физико-математических наук. Научные интересы: статическая верификация, параллельные программы.

Pavel Sergeevich ANDRIANOV – researcher in ISP RAS, Ph.D. Research interests: software model checking, parallel programs.