Predicate Abstraction in Program Verification: Survey and Current Trends*

Jakub Daniel[1] **and Pavel Parízek**[2]

1 **Charles University in Prague**
daniel@d3s.mff.cuni.cz
2 **Charles University in Prague**
parizek@d3s.mff.cuni.cz

Abstract

A popular approach to verification of software system correctness is model checking. To achieve scalability needed for large systems, model checking has to be augmented with abstraction. In this paper, we provide an overview of selected techniques of program verification based on predicate abstraction. We focus on techniques that advanced the state-of-the-art in a significant way, including counterexample-guided abstraction refinement, lazy abstraction, and current trends in the form of extensions targeting, for example, data structures and multi-threading. We discuss limitations of these techniques and present our plans for addressing some of them.

1998 ACM Subject Classification D.2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases program verification, model checking, predicate abstraction, refinement

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.1

1 Introduction

Software systems are nowadays ubiquitous and therefore it is important that they work correctly. An erroneous behavior may lead to loss of lives, money, and resources especially in safety-critical systems that are used, for example, in the domains of transportation and healthcare. It is necessary that as many errors as possible are detected and fixed before the deployment of a given system. We need techniques and tools able to reason about the behavior of programs in order to check the program safety and find errors. One approach to detection of errors is the use of program verification. The state-of-the-art verification techniques work for simple programs, but it is desirable that they scale also to large programs, have a good performance, and require little manual effort (automation).

In this paper, we provide an overview of selected recent contributions in the area of program verification that address these challenges. A very popular approach to program verification is model checking [15], which decides whether a given program satisfies a desired property by exhaustively exploring its state space. The state space of non-trivial programs is generally too large, and therefore an abstraction of some kind is needed to reduce its size. In particular, predicate abstraction [22] has been a subject of extensive research (e.g., [5, 11, 24, 28, 29]). We introduce the reader to successful verification techniques based on predicate abstraction and discuss possible directions of their further improvement.

We use the example program in Figure 1 throughout the paper to illustrate the basic principles of checking program safety with predicate abstraction. The program consists

* This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S.

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 1–9

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
1: procedure MAIN()
2:     a ← an arbitrary array of integers
3:     i ← FINDGREATER(a, 10)
4:     assert i < LENGTH(a) ⟹ a[i] > 10

5: function FINDGREATER(a, c)
6:     for 0 ≤ j < LENGTH(a) do
7:         if a[j] > c then
8:             return j
9:     return LENGTH(a)
```

Figure 1 Example program

of the procedure MAIN, which creates an array of an arbitrary length, calls the function FINDGREATER, and subsequently asserts a specific property of the returned value i. The function FINDGREATER takes array a and constant c as arguments, and by iterating over the array a it finds an index j such that the array element $a[j]$ is greater than c. If no such index exists it returns the length of the array a. There are two locations, associated with lines 4 and 7, where the program execution may fail. The assertion at line 4 is violated if the value of the variable i points to an element of the array a whose value is less then or equal to 10. An error might arise also when attempting to access elements outside the bounds of the array at line 7.

The rest of the paper is structured in the following way. In Section 2, we give an overview of the state-of-the-art verification techniques based on predicate abstraction, and describe a procedure that can be used to verify the example program in Figure 1. In Section 3, we discuss limitations of the state-of-the-art techniques and the corresponding challenges. Finally, we present our plans and goals in this research area, including our current work to date (Section 4). Note that although in this paper we focus only on techniques related to predicate abstraction, there are many other approaches to program verification such as bounded model checking [13], and techniques tailored to efficient detection of errors of specific types, such as wrong usage of data structures and pointers [7, 20].

2 Overview

All the techniques presented in this section apply model checking to abstract programs. Their general idea is to automatically construct the most coarse-grained abstraction of an input program that is sufficient to prove the program safe. Such an abstraction captures all feasible behaviors of the program and also some infeasible behaviors.

First, we describe the basic procedure for model checking with predicate abstraction in Section 2.1. This is followed by a description of techniques that build on the basic procedure. We focus on verification techniques that can be divided into the following four categories: counterexample-guided abstraction refinement (Section 2.2), lazy abstraction (Section 2.3), combinations of multiple approaches (Section 2.4), and techniques targeting data structures, concurrency, and modular design (Section 2.5). The order of the subsections follows roughly the chronological development of the respective techniques.

2.1 Model Checking with Predicate Abstraction

The use of predicate abstraction enables efficient reasoning over an abstract state space that is much smaller than the original concrete one, because each abstract state represents a possibly large set of concrete states. Each abstract state corresponds to a specific valuation of *abstraction predicates* that express relationships between program variables.

In order to verify that the example program (Figure 1) is safe, it is necessary to determine whether there exists an execution under which any of the error locations is reached. The first

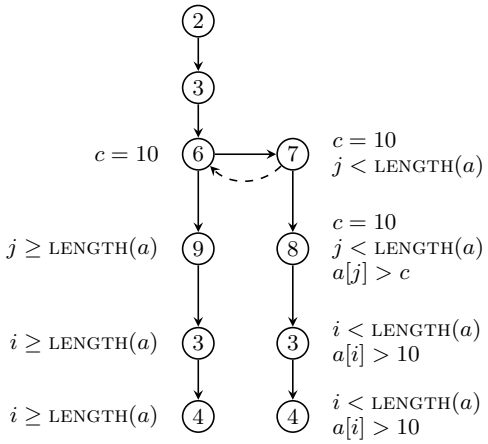

Figure 2 Abstract state space

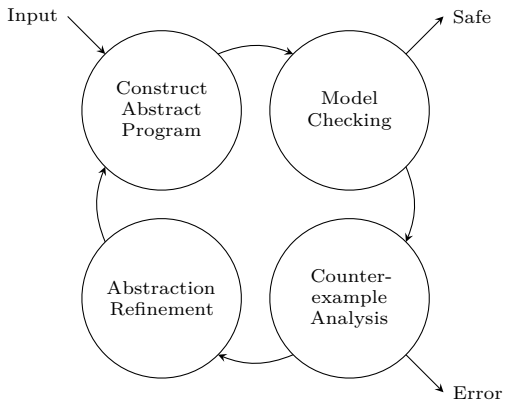

Figure 3 CEGAR loop

step is to reduce the size of the state space to be explored by constructing an abstraction in the form of a boolean program [5, 17]. A set of abstraction predicates must be defined and given as input to the construction of the abstract boolean program. In the case of the example program, the set of abstraction predicates necessary to prove its safety includes the following: $i <$ LENGTH(a), $a[i] > 10$, $j \geq 0$, $j <$ LENGTH(a), $a[j] > c$, and $c = 10$. Although the predicates can be defined manually, the general aim is to design automated techniques for their inference and construction of the abstraction.

Now, we describe a run of the model checking procedure over the example program in Figure 1. The states of the abstract program correspond to different valuations (*true*, *false*, or *unknown*) of the abstraction predicates at different program locations. Figure 2 shows the whole reachable abstract state space. Each node represents an abstract state — it is labeled with the corresponding program location (source code line number) and predicates that evaluate to *true*.

The process of verification of this program starts at the entry point of MAIN (line 2) with all the predicates valuated to *unknown*. The program statements are executed symbolically until the invocation of FINDGREATER at line 3. At this point, the valuation of $c = 10$ is set to *true*, and the program execution continues at line 6, where branching takes place; one branch corresponds to the variable j pointing beyond the bounds of the array a, and the second branch corresponds to j pointing to some element of the array. The verification procedure needs to explore both state space branches. Suppose the procedure takes the first branch, so that the program execution continues to line 9 (left side of Figure 2). After returning to MAIN, the abstraction predicate $i <$ LENGTH(a) valuates to *false*, and therefore the assertion at line 4 holds. Next, the verification procedure explores the second branch, i.e. the body of the loop, and the program execution reaches line 7 (right side of Figure 2). The loop invariant $0 \leq j <$ LENGTH(a) ensures that only array elements at valid indices are accessed. Again, both branches of the `if`-statement have to be explored. We describe only the branch in which the body of the `if`-statement is executed because the other leads to an already visited state (through the dashed arc). In this branch, both the abstraction predicates $a[j] > c$ and $c = 10$ valuate to *true* at line 8. After returning to MAIN, the value of j is assigned to the variable i, hence $a[i] > 10$ is *true* and the assertion is satisfied. All the traces in the abstract program have been explored without reaching an error state, and thus

the example program is safe.

One of the key requirements for automation of such verification procedure is that effects of individual statements on valuation of abstraction predicates are encoded precisely. A natural approach is to use logic formulæ. Obviously, the choice of a suitable logic plays an important role and affects the precision and complexity of the verification technique that relies on it. The building blocks of commonly used theories of the first-order logic include: equality logic, linear integer arithmetic, and uninterpreted functions. SMT solvers are often used to decide validity of formulæ that capture effects of program statements. However, invocations of an SMT solver are typically expensive, and therefore some of the techniques that we describe later try to minimize the number of invocations performed during the verification of a given program.

2.2 Counterexample-Guided Abstraction Refinement

The first technique for automated program verification based on predicate abstraction that we describe is *Counterexample-Guided Abstraction Refinement* (CEGAR) [16]. Figure 3 shows the main loop of the algorithm, which consists of these four steps: construction of an abstract program, model checking, analysis of a counterexample, and refinement of the abstraction. The initial abstraction does not consider any data values and the control-flow at branching points is entirely non-deterministic. Such an abstraction usually permits spurious executions, which do not correspond to real executions of the original program. This is true especially for infeasible counterexamples, which are eliminated by abstraction refinement. Note that the abstract program is constructed from scratch in each iteration.

In the rest of this section, we describe the individual steps of the main loop:

1. An abstract boolean program is constructed for the given input program. The boolean program contains boolean variables that represent values of the abstraction predicates, and captures the effects of statements from the original program on the values of predicates. Every assignment statement in the original program is modeled in the abstract program by an assignment of boolean values to variables representing individual predicates. Branching conditions are modeled by the boolean variable that represents the corresponding abstraction predicate. If the value of the predicate is *unknown* or no such abstraction predicate is defined, a non-deterministic choice is used.
2. A model checker is used to verify the abstract program. In case an error state is reachable, the model checker produces a counterexample in the form of a trace that represents the program execution from the initial state to the error state. If the abstract program is safe then the original program is *safe* as well.
3. The next step is to analyze the counterexample (error trace). The error trace is associated with a trace formula, which is constructed by conjoining subformulæ that express the semantics of individual statements in the trace. Satisfiability of this formula is checked to determine feasibility of the counterexample. If there exists a satisfying assignment to variables of the trace formula, then we get *a real counterexample*, which is then reported to the user.
4. Infeasible counterexamples are eliminated through refinement of the abstraction. New predicates are inferred based on the counterexample, and used in the next iteration to improve precision of the generated abstraction, so that the trace representing the infeasible counterexample is no longer permitted.

The verification procedure terminates when a real counterexample is encountered (step 3),

or the program is proven safe (step 2). CEGAR is the basis of a large number of automated approaches to program verification, some of which we describe in the next three sections.

2.3 Lazy Abstraction

The construction of the full abstract boolean program from scratch in each iteration of the CEGAR loop is costly. More recent techniques [24, 28] use an incremental approach to abstraction refinement. Their key differences from the techniques described in the previous section is that (1) both construction and refinement of the abstraction are performed on-the-fly during model checking, and (2) the refinement does not affect already explored parts of the abstract state space, which saves a large computational effort. The basic idea is to iteratively unroll a control-flow graph of the given program into an *abstract reachability graph* (ARG). The nodes of ARG correspond to abstract states and edges reflect the program control-flow. At the beginning of the verification procedure, the ARG contains only the initial state. In each iteration, the procedure expands a leaf state and adds its children states into the graph. If an error state is added into the ARG, the corresponding error trace is analyzed for feasibility. The error may be either real, in which case it is reported and the procedure terminates, or spurious. In the case of a spurious error, the abstract states along the corresponding trace are refined with new predicates in order to eliminate the error. The new predicates for each location on the trace are obtained using interpolants [25, 28]. The locality of the newly added predicates makes it possible to keep the abstract state space reasonably small, as the verification procedure refines the precision only where necessary. The procedure uses a *covering* relation to ensure that, in each step, it explores only those abstract states that have not yet been processed before.

When unrolling the ARG, the abstraction may be constructed either in an eager or lazy fashion. The eager approach [25] computes the reachable abstract states using an *abstract post operator*, which involves multiple SMT queries. Blast [9] is an example of a verification tool that uses lazy predicate abstraction with an eager construction of the abstraction. The lazy approach [28], called Impact, overapproximates all abstract states with *true* and postpones the construction of a precise abstraction to the refinement step.

2.4 Combinations of Multiple Approaches

Lazy abstraction is an efficient technique as long as a relatively coarse abstraction is sufficient to find a real error or prove safety of the given program. Otherwise, the verification procedure has to perform a large number of refinement steps, which is computationally expensive. Loops in the program code, in particular, often require a complete unrolling of the ARG and numerous refinement steps, which reduce the benefits of abstraction. In this section, we describe some techniques that address this problem and improve scalability.

Yogi [23] is a tool that implements the Dash algorithm [6], which combines testing and verification in order to achieve better performance. The Dash algorithm works in an iterative fashion, and maintains two data structures: (1) an abstraction of the input program, which serves for proving the program's safety, and (2) a collection of already executed tests. At the start of each iteration, the algorithm inspects the current abstraction to see if there is any abstract error trace (counterexample). For the counterexample, Dash identifies the longest prefix that is covered by previously executed tests, and attempts to construct a new test that follows the prefix and reaches the next state in the counterexample after the prefix. If there is such a test, either its execution confirms the presence of a real error, or it guides abstraction refinement along the spurious counterexample. Otherwise, if such a test does not exist, the

last abstract state of the prefix is refined by adding predicates that are derived from the transition between the prefix and the rest of the trace. An advantage of this approach is that it is computationally much less expensive for two reasons: (1) execution of tests may save many refinement steps, and (2) predicates are derived without the use of SMT.

UFO [2] is a verification framework, which combines a configurable forward computation of the program abstraction with usage of interpolants to achieve sufficient precision. Depending on the particular configuration, the abstraction may be constructed in an eager or lazy manner (as in lazy abstraction). UFO differs from the previously mentioned approaches in that it does not construct a separate trace formula for every trace that reaches an error location. Instead, it captures multiple traces in the ARG with a single formula, and thus significantly reduces the number of necessary refinement steps. A satisfying assignment of the formula's variables yields a real counterexample, whereas unsatisfiability of the formula enables the use of interpolants to refine the abstraction. In this way, the ARG is refined globally rather than along a single trace at a time, and each abstract state is refined as in lazy abstraction.

Another notable framework is CPACHECKER [11], which supports custom *configurable program analyses* (CPA) [10]. The definition of each analysis consists of an abstract domain, transfer relation, merge operator, and a stop operator. The merge operator specifies if and how to merge abstract states when two control-flow paths meet. The stop operator detects whether a newly reached state is already covered and does not have to be explored again. Multiple custom analyses can be put together to form a combined analysis. The verification algorithm of CPACHECKER performs a simple reachability analysis parametrized with a given CPA. It is implemented in the tool with the same name [11], which provides the CPA's with a compact interface to other necessary pieces of modern program verification frameworks, such as the input parser and SMT solvers.

2.5 Data Structures, Concurrency, and Modularity

In this section, we provide an overview of recent notable advancements within verification techniques towards support of realistic programs. This support is important for broader practical applicability, precision, and scalability of program verification. Specifically, we show techniques that can handle some of the following aspects of realistic programs: heap and data structures, object-oriented constructs, modular design (libraries), and concurrency.

Basically, every larger program uses data structures such as arrays, linked lists, and trees. This makes verification more challenging because more complex logics have to be used for reasoning about such programs. There exist verification techniques that address arrays [3, 26] and data structures of other kinds [14, 27].

The approach proposed in [3] focuses on precise reasoning over arrays of unknown length. Interesting properties of arrays and their elements (e.g., whether a given array is sorted) can be expressed only using quantifiers. The basic idea of [3] is to compute an overapproximation of the set of states backward-reachable from error states [21]. Then, the task of verifying safety of a given program reduces to a check for an empty intersection with the set of initial states. Spurious errors are eliminated through lazy abstraction. This approach was implemented in the SAFARI tool [4].

Modular verification is a popular approach to achieve scalability. Parts of the program can be analyzed separately in order to reduce the total cost of the verification. In particular, library code can be analyzed just once, even in the case of methods that are called at multiple locations in a given program. Authors of [27] propose to replace the calls of library methods that manipulate data structures with summary transitions that are derived from

their specifications. The WHALE algorithm [1] computes method summaries, which are necessary for the proof of correctness, using an iterative approach based on interpolation.

A difficult challenge related to programs with multiple threads is the need to model concurrent updates of shared data. The first step towards overcoming this challenge was proposed in [19]. The central idea is to perform the CEGAR loop on a parallel composition of multiple instances of the boolean program that abstracts the input program. Predicates referring just to thread-local variables are modeled with local boolean variables (a fresh copy per thread is used), and predicates involving only shared variables are modeled with shared boolean variables. In the case of predicates that refer to a mixture of local and shared variables, this verification procedure also uses local boolean variables to represent such predicates but each update of such a variable is broadcast to all the threads. To compute the correct value to be broadcast from an active thread t_a to a given thread t, the procedure uses predicates associated with t_a, predicates associated with t, and all shared predicates. In contrast, the approach proposed in [30] extends the IMPACT algorithm with support for concurrency, and uses partial order reduction [15] to achieve reasonable performance.

3 Limitations and Challenges

Our survey of the state-of-the-art verification techniques based on predicate abstraction indicates that great advancements have been made in the last decade, but the techniques still have certain limitations in terms of their support of features used in realistic programs, performance, scalability, and automation. In particular, they are applicable only to programs of a moderate size (up to tens of thousands of source code lines in C) and they can handle only programs with a limited usage of data structures.

The comparison [12] of the two fundamental approaches to lazy abstraction [24, 28] analyzes the effects of their individual features on performance. The authors implemented and compared several different configurations of the approaches using the same framework. Results show that the two approaches have similar performance in configurations that involve larger block encodings [8]. The general conclusion is that less refinement steps and cheaper covering tests lead to better performance.

4 Our Research Goals

Our main goal is to create a framework for verification of multi-threaded Java programs that involve loops, recursion, and usage of data structures. We especially intend to investigate automated generation of abstraction predicates by combination of static analysis with dynamic analysis, which is known to handle programs with loops very well.

We have already done some initial work on using predicates to capture the state and behavior of data collections. In [29], we defined a predicate language for modeling lists, sets, and maps, and implemented the language in the J2BP tool that automatically generates abstract programs [31]. We also started investigating the capabilities of predicate abstraction in the context of on-the-fly state space traversal combined with a dynamic analysis. We implemented basic support for predicate abstraction into the tool called Abstract Pathfinder [18], which is an extension to the Java Pathfinder verification framework [32].

References

1 A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An Interpolation-Based Algorithm for Inter-procedural Verification. In Proceedings of VMCAI 2012, LNCS, vol. 7148.

2 A. Albarghouthi, A. Gurfinkel, and M. Chechik. From Under-Approximations to Over-Approximations and Back. In Proceedings of TACAS 2012, LNCS, vol. 7214.

3 F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy Abstraction with Interpolants for Arrays. In Proceedings of LPAR 2012, LNCS, vol. 7180.

4 F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: SMT-Based Abstraction for Arrays with Interpolants. In Proceedings of CAV 2012, LNCS, vol. 7358.

5 T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In Proceedings of PLDI 2001, ACM.

6 N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from Tests. In Proceedings of ISSTA 2008, ACM.

7 J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory Safety for Systems-Level Code. In Proceedings of CAV 2011, LNCS, vol. 6806.

8 D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software Model Checking via Large-Block Encoding. In Proceedings of FMCAD 2009, IEEE.

9 D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST. STTT, 9(5-6), 2007.

10 D. Beyer, T. A. Henzinger, and G. Theoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In Proceedings of CAV 2007, LNCS, vol. 4590.

11 D. Beyer and M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In Proceedings of CAV 2011, LNCS, vol. 6806.

12 D. Beyer and P. Wendler. Algorithms for software model checking: Predicate abstraction vs. Impact. In Proceedings of FMCAD 2012, IEEE.

13 A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, Bounded Model Checking. Advances in Computers, 58, Elsevier, 2003.

14 A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On Inter-Procedural Analysis of Programs with Lists and Data. In Proceedings of PLDI 2011, ACM.

15 E. M. Clarke, O. Grumberg, and D. A. Peled. Model Checking. MIT Press, 1999.

16 E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In Proceedings of CAV 2000, LNCS, vol. 1855.

17 E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-Based Predicate Abstraction for ANSI-C. In Proceedings of TACAS 2005, LNCS, vol. 3440.

18 J. Daniel, P. Parizek, and C. Pasareanu. Predicate Abstraction in Java Pathfinder. Java Pathfinder Workshop 2013, ACM SIGSOFT Software Engineering Notes, 39(1).

19 A. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-Aware Predicate Abstraction for Shared-Variable Concurrent Programs. In CAV 2011, LNCS, vol. 6806.

20 K. Dudka, P. Peringer, and T. Vojnar. Byte-Precise Verification of Low-Level List Manipulation. In Proceedings of SAS 2013, LNCS, vol. 7935.

21 S. Ghiraldi and S. Ranise. Backward Reachability of Array-based Systems by SMT solving: Termination and Invariant Synthesis. LMCS, 6(4), 2010.

22 S. Graff and H. Saidi. Construction of abstract state graphs with PVS. In Proceedings of CAV 1997, LNCS, vol. 1254.

23 B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: A New Algorithm for Property Checking. In Proceedings of FSE 2006, ACM.

24 T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In Proceedings of POPL 2002, ACM.

25 T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from Proofs. In Proceedings of POPL 2004, ACM.

26 R. Jhala and K. L. McMillan. Array Abstractions from Proofs. CAV 2007, LNCS, vol. 4590.

27 D. Kapur, R. Majumdar, and C.G. Zarba. Interpolation for Data Structures. In Proceedings of FSE 2006, ACM.

28 K. L. McMillan. Lazy Abstraction with Interpolants. CAV 2006, LNCS, vol. 4144.

29 P. Parizek and O. Lhotak. Predicate Abstraction of Java Programs with Collections. In Proceedings of OOPSLA 2012, ACM.

30 B. Watcher, D. Kroening, and J. Ouaknine. Verifying Multi-threaded Software with Impact. In Proceedings of FMCAD 2013, IEEE.

31 J2BP: Predicate Abstraction for Java, http://plg.uwaterloo.ca/~pparizek/j2bp

32 Java Pathfinder, http://babelfish.arc.nasa.gov/trac/jpf