HAL
open science

Automated Generation of Loop Invariants using Predicate Abstraction

Krishnamani Kalyanasundaram, Claude Marché

To cite this version:

Krishnamani Kalyanasundaram, Claude Marché. Automated Generation of Loop Invariants using Predicate Abstraction. [Research Report] RR-7714, INRIA. 2011, pp.32. inria-00615623

HAL Id: inria-00615623

https://inria.hal.science/inria-00615623v1

Submitted on 19 Aug 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Automated Generation of Loop Invariants using Predicate Abstraction

Kalyanasundaram K. — Claude Marché

N° 7714

July 2011

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7714--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche SACLAY - ÎLE-DE-FRANCE

Automated Generation of Loop Invariants using Predicate Abstraction

Kalyanasundaram K. *† , Claude Marché *†

Thème : Programmation, vérification et preuves
Équipe-Projet ProVal

Rapport de recherche n° 7714 — July 2011 — 30 pages

Abstract: Program verification is a challenging task that requires several techniques for addressing the different issues that arise because of program syntax, semantics and in many cases, the kind of properties that are to be established. Static analysis is one of the techniques that has anchored its presence in the verification of industrial scale softwares. However, no one technique is enough to combat the complexity of today's software systems. A combination of techniques is the only way forward in order to achieve the confidence levels that are required in safety-critical softwares. Frama-C is one such platform that combines various program analyses and verification techniques. It consists of a bunch of tools that operate on user-annotated C programs and generates verification conditions that would establish the correctness of the input programs. These verification conditions are automatically discharged by a set of automated provers. The annotations provided by the user along with the program include function contracts, assertions and loop invariants. Of these annotations, loop invariants are of special interest as writing a correct and useful loop invariant is as challenging as verifying the program itself. In this article, we describe the techniques we have developed for generating these loop invariants automatically to reduce the burden on the user. Our techniques are based on *predicate abstraction*, a well known abstract interpretation technique for abstract model-checking. We demonstrate the potential of our technique in a multi-prover verification of C-programs as implemented in Frama-C platform.

Key-words: Proof of Programs, Formal Specification, Loop Invariants, Predicate Abstraction

This work is partly supported by the *U3CAT* (ANR-08-SEGI-021, http://frama-c.com/u3cat/) project of the French national research organization (ANR)

* INRIA Saclay - Île-de-France, F-91893
† Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

Génération automatique d'invariants de boucle par abstraction booléenne

Résumé : La vérification de programmes est une tâche difficile qui nécessite plusieurs techniques pour traiter les différentes questions qui se posent en raison de la syntaxe du programme, la sémantique et dans de nombreux cas, le type de propriétés qui doivent être établies. L'analyse statique est l'une des techniques qui a obtenu quelques succès dans la vérification de logiciels de nature industrielle. Cependant, aucune technique n'est suffisante pour combattre la complexité des systèmes logiciels d'aujourd'hui. Une combinaison de techniques est la seule façon d'avancer en vue d'atteindre les niveaux de confiance qui sont nécessaires en matière de sécurité des logiciels critiques. Frama-C est un exemple de plate-forme qui combine plusieurs techniques d'analyse et de vérification. Elle se compose d'un ensemble d'outils qui fonctionnent sur des programmes C annotés et génère des conditions de vérification qui établissent la correction des programmes fournis. Ces conditions de vérification sont automatiquement validées par un ensemble de prouveurs automatiques. Les annotations fournies par l'utilisateur avec le programme comprennent les contrats de fonction, les assertions et les invariants de boucle. Parmi ces annotations, les invariants de boucle sont les plus ardus à déterminer car le bon choix d'un invariant correct et utile est aussi difficile que de vérifier le programme lui-même. Dans cet article, nous décrivons les techniques que nous avons développé pour générer ces invariants de boucles automatiquement pour réduire la tâche de l'utilisateur. Nos techniques sont basées sur l'abstraction booléenne (*Predicate Abstraction* en anglais), une technique bien connue dans le cadre de l'interprétation abstraite et du *model-checking*. Nous démontrons le potentiel de notre technique par une mis en œuvre dans l'environnement Frama-C.

Mots-clés : Preuve de programmes, Spécification formelle, Invariants de boucle, Abstraction booléenne

1 Introduction

Softwares are the most ubiquitous things on this planet today. They have become indispensable for carrying out our day-to-day activities. Their omnipresence has also made it a necessity to ensure their correctness and has hence opened up many challenging research opportunities. Traditionally the correctness of most industrial-size software is ensured through *testing* wherein a set of input test patterns is fed to the software or system under test and the output is observed to check if it is the desired one. Exhaustive testing is infeasible given the complexity of today's software. Further testing can identify the presence of bugs, but can not ensure the absence of them. A recent set of techniques, under the umbrella term *formal methods* has become prominent in industry. These techniques combine the clarity and rigour of mathematical modelling and reasoning, with efficient computer algorithms, thereby automating the process, for reasoning about the correctness of software and hardware systems. The advances in research in satisfiability (SAT) solvers [MMZ+01] and Binary Decision Diagrams (BDDs) [Bry86] and more recently, SMT solvers [DNS05] greatly aided the task of modelling systems and tools for the automated analysis of such models. Despite the development of such tools, there are numerous challenges due the growing complexity of software and systems they reside on.

Safety critical software systems that are being used in nuclear power plants, avionics or automobile systems require a high level of confidence. This level of confidence can not be assured by non-exhaustive testing. Formal methods, though expensive and difficult than testing is a reliable approach for verifying the correctness of such safety critical systems. There are a bunch of formal verification techniques which are applicable for different systems (like hardware designs, programming languages, etc.) and offer different degrees of automation. One of the most widely used technique in industry is *model checking* due to its high degree of automation. Model checking techniques involve traversing the finite state transition system of the input system to check if every state or a set of states satisfy a required property. Other techniques like theorem proving are effective and efficient but the degree of automation offered by theorem provers is not comparable with model checking techniques.

However, given the complexity of industrial software, even with the use of state-of-the-art SAT solvers model checking techniques do not scale up. In order to overcome the state space explosion problems, the models are usually abstracted — by considering parts of the model that are relevant for verification thereby making the model simpler, and the model checking techniques were applied to the simplified *abstract* model. The original model is referred to as *concrete* model. Abstractions were hand-crafted, and the success of the technique heavily relied on coming up with 'good' abstractions. *Predicate abstraction* is one of the abstraction techniques that enabled computation of abstractions automatically, given a concrete model and a set of predicates (information about the concrete model). This process is usually embedded within an abstraction refinement framework, wherein a conservative abstraction of the original design, amenable to verification techniques like model checking, is constructed to begin with. This abstraction is iteratively refined till the property of interest about the system is proved correct or a real bug is discovered. The most popular abstraction refinement frameworks use a propositional SAT solver or a theorem prover for constructing abstractions. Both approaches have their advantages as well as disadvantages.

The abstraction generated by predicate abstraction techniques are usually much simpler than the concrete model and can be readily subject to model checking. Since the abstract model is composed of Boolean variables and transitions over them, model checking is much more effective over the abstract model. If a property is true of the abstract model, it is also true of the concrete model. If

a property does not hold, the model checker generates a counterexample. A counterexample is a sequence of transitions from an initial abstract state and ending in an abstract state that violates the property. Note that because of loss of some information due to abstraction, this counterexample may not be a real counterexample, but just an artifact of abstraction process. The counterexample is analyzed to check if it is real, by simulating it on the concrete model. This phase is called Simulation. This is performed by mapping back the states in the abstract counterexample trace to the concrete model, to see if the behaviour represented by the abstract counterexample is feasible in the actual concrete model. If simulation succeeds, then the counterexample is real, and the concrete model has a corresponding path leading to an error state. If simulation fails, then the counterexample is said to be a spurious counterexample. The abstraction has to be refined in order to eliminate the spurious behaviour of the model. The refinement phase consists of analyzing the spurious counterexample in order to figure out more details in order to refine the abstract model. This iterative process of abstraction, model checking, simulation and refinement are carried out till the property is proved or a real counterexample encountered. This process is referred to as Counterexample Guided Abstraction Refinement [CGJ+03], more commonly dubbed as CEGAR. The promise of this approach saw instant and reasonable success in industry [BMMR01] and is a widely adopted formal verification technique today. However, the technique still suffers from scalability issues when applied over a wide range of verification problems.

Deductive verification of programs, although has been around for more than four decades, is gaining interest among academic researchers [FM07] and industry [BCD+05, FLL+02]. The approach mainly involves three ingredients. (1) the program as well as the desired properties about it are expressed as logical formulas which when proven valid proves the program correct, with respect to the set of properties (2) the validity of the formulas itself is proven by deduction in a logical calculus and (3) the most desired ingredient is to automate the process as much as possible. One major advantage is the ability to precisely model the semantics of the program. Most widely used frameworks handle a subset of the underlying programming language and treat programs annotated with pre and post-conditions, assertions, etc. These are written in a general purpose specification language. The program statements are transformed into logical expressions using logic formalisms like Hoare's logic, Dijkstra's weakest pre-condition computation or computation of strongest post-condition. The result is a set of formulas, the *verification conditions* which are discharged by a bunch of provers.

Although the verification conditions generated above could be discharged automatically by the state-of-the-art provers, annotating the program itself has to be done manually. These annotations include function contracts, pre and post-conditions, assertions, loop invariants, etc. Of these, assertions and function contracts could be written by the developer who knows the intended behaviour of the program. However, writing loop invariants is a challenging and more involved task. In particular we want to be able to write loop invariants that are useful in proving the post-condition. As writing a loop invariant itself is involved, writing those that are useful in proving the post-condition is a Herculean task. There is a tremendous amount of research in loop invariant generation some of which could be employed to ease the process of writing loop invariants for deductive program verification. This is exactly the motivation of our research.

In this report we propose techniques based on predicate abstraction for automatically generating complex loop invariants, that would aid in proving the correctness of post-conditions. We advocate predicate abstraction techniques because (1) it has been studied extensively by the model checking community and has been highly successful in CEGAR frameworks, (2) choosing predicates is much easier and less involved than writing loop invariants, although choosing the best set of predicates is

going to be challenging and (3) it could benefit from the information derived from other complementary techniques like using numerical abstract domains or program analysis engines. We have implemented the proposed techniques as plugins in the Frama-C [Fra08] framework for deductive verification of C programs. The preliminary experimental results are promising and they demonstrate the potential of this approach. The current implementation along with the examples of this report are available at URL `http://proval.lri.fr/agen/`.

The report is organized as follows: In Section 2, we describe the research work that are related to loop invariant generation techniques as well as the framework we use. In Section 3, we describe our underlying verification framework for C programs, namely Frama-C. Predicate abstraction techniques forms the core of our approach and we introduce it in Section 4. Our approach is discussed in detail in Section 5. We illustrate the use of our approach for complex C data-structures in Section 6. Section 7 provides heuristics for choosing predicates. We illustrate our approach with a set of tricky examples in Section 8 and conclude the report mentioning possible directions of future research in Section 9.

2 Related Work

Interest in automating the generation of loop invariants is more than a decade old. As with program verification, the problem is undecidable. There are several techniques that range from the use of numerical abstract domains [Jea], constraint solving based approaches [Gul09] and also template-based techniques for invariant generation [Mic03, GR09]. A more recent approach [SDDA11] advocates transformation of the input program loop in order to use the earlier approaches more effectively. More popular techniques [BCD+05, Lei08, Wei11] that work within the deductive verification framework which is also similar to our setting use SMT solvers or theorem provers incrementally in order to generate loop invariants.

In this section, we describe the work that are closest to our approach, in order to compare the relative strengths and weaknesses so that the comparisons remain meaningful and effective. In particular, we describe the approaches which use predicate abstraction based techniques [BCD+05, FQ02, Wei11] as well as approaches [Jea, GR09, SDDA11] that could be orthogonal to predicate abstraction based approaches. Approaches [FQ02, BCD+05, Lei08] follow similar techniques as in [FM07, FM04] where in the annotated program is transformed into an intermediate language and verification conditions are generated for the program in this intermediate language. [FQ02] uses strongest post-condition semantics for the intermediate language where as the other tools use weakest pre-condition semantics. [FQ02, BCD+05, Lei08, Wei11] use predicate abstraction techniques for automatically inferring some of the annotations given a set of predicates, whereas [FM07, FM04] do not offer any automation. INTERPROC [Jea] employs numerical abstract domains, while [GR09] uses template based invariant generation techniques. We describe the approaches closest to our approach in detail and briefly describe the other approaches to an understandable extent. We begin with the description of Extended Static Checker for Java and Boogie.

ESC Java: The *Extended Static Checker* for Java, more commonly known as ESC Java, is a technique based on predicate abstraction for verification of Java programs. The input Java program is annotated with function contracts, pre and post-conditions and assertions. The basic technique

$$
\begin{array}{rcl}
A, B \in Stmt & :== & \textbf{assert } e \\
 & | & \textbf{assume } e \\
 & | & x := e \\
 & | & A; B \\
 & | & A \,[\!]\, B \\
 & | & \{P, I\}\textbf{while } e \textbf{ do } B \textbf{ end} \\
x \in Var & & (variables) \\
e \in Expr & & (expressions) \\
I \in Formula & & (logical\ formula) \\
P \subseteq I & & (loop\ predicates)
\end{array}
$$

Figure 1: ESC Java: Guarded Command Language

S	Norm(Q, S)	Wrong(Q, S)
$x := e$	$\exists x'.x = e(x \leftarrow x') \wedge Q(x \leftarrow x')$	false
assert P	$Q \wedge P$	$Q \wedge \neg P$
assume P	$Q \implies P$	$false$
A $[\!]$ B	$Norm(Q, A) \vee Norm(Q, B)$	$Wrong(Q, A) \vee Wrong(Q, B)$
A ; B	$Norm(Norm(Q, A), B)$	$Wrong(Q, A) \vee Wrong(Norm(Q, A), B)$
$\{P, I\}$**while** e **do** B **end**	$Norm(Q, desugar(S))$	$Wrong(Q, desugar(S))$

Figure 2: ESC Java — Guarded Command Language — Formal Semantics

behind this approach [FQ02] is to translate each method and associated specifications of a Java program into a simple intermediate language — a guarded command language, as shown in Figure 1.

This guarded command language is used to generate verification conditions (VC). The formal semantics of the guarded command language is that of strongest post-condition. For an execution of statement S from an initial state satisfying formula Q, $Norm(Q, S)$ denotes all post-states in which execution would terminate normally and $Wrong(Q, S)$ denotes the states in which execution would fail with an assert. The strongest post-condition semantics for the above guarded command language is shown in Figure 2.
where $desugar(S)$ is

$$
\begin{array}{ll}
desugar\ (\{P, I\} & \textbf{while } e \textbf{ do } B \textbf{ end}\ ) = \\
 & \textbf{assert } I; havoc(targets(B)); \textbf{assume } I; \\
 & ((\textbf{assume } e; B; \textbf{assert } I; \textbf{assume } false) \\
 & [\!]\textbf{assume } \neg e)
\end{array}
$$

```
/*@ requires a != null && b != null */
/*@ requires a.length == b.length  */
/*@ ensures \result == a.length || b[\result]  */
/*@ loop_predicate spot == a.length, b[spot], spot < i */
int find (int [] a, boolean [] b) {
  int spot = a.length; int i = 0;
  while (i < a.length) {
    if ((spot == a.length) && (a[i] != 0))
      spot = i;
    b[i] = (a[i] != 0);
    i++;
  }
  return spot;
}
```

Figure 3: A Simple Java Program

spot = a.length; i = 0;
((**assume** (i $<$ a.length);
((**assume** (spot == a.length && a[i] != 0); spot = i) []
(**assume** ¬(spot == a.length && a[i] != 0))
b[i] = (a[i] != 0); i = i + 1)) []
(**assume** ¬(i $<$ a.length))

Figure 4: Desugaring of $\{P, I\}$ **while** e do B

For example, let us consider the **while** loop in the Java program of Figure 3 (borrowed from [FQ02]). Given the set of predicates $P : \{$ `spot` $==$ `a.length`, `b`[`spot`], `spot` $<$ `i`$\}$ the loop reduced to the guarded command language is shown in Figure 4.

The key contribution of this approach is the use of predicate abstraction techniques to infer loop invariants automatically. The algorithm also implements heuristics to guess predicates that could be used for invariant generation. Further, one could generate universally quantified loop invariants. The basic operation for loop invariant generation is the computation of abstraction of a set of reachable states, iteratively. The initial approximation is obtained by computing the set of reachable states at loop entry C. Further approximations enlarge this set by executing the loop body B once from the current approximation. Given a set of predicates $p_1, p_2, \cdots, p_n$, and a set of states given by formula ϕ, an abstraction of ϕ is computed by making theorem prover queries. One of the advantages of this approach over SLAM [BMMR01] is the ability to generate universally quantified invariants. One possible disadvantage may be the use of theorem prover for computing predicate abstraction. If an SMT solver or a SAT solver is used in its place, the exponential number of theorem prover queries gets replaced by a single SAT or SMT instance.

Boogie: Boogie [BCD+05] is a research project at Microsoft Research and it is aimed at Spec# programs. In its basic approach it is very similar to ESC Java. BoogiePL is used as an intermediate language. BoogiePL includes procedures, mutable variables, pre- and post-conditions. The

$$\begin{aligned} WP(\textbf{assert}\ e, Q) &= e \wedge Q \\ WP(\textbf{assume}\ e, Q) &= e \implies Q \\ WP((A; B), Q) &= WP(A, WP(B, Q)) \\ WP(A \,[\!]\, B, Q) &= WP(A, Q) \wedge WP(B, Q) \end{aligned}$$

Figure 5: BoogiePL: Weakest Pre-condition Semantics

invariant generation using abstract interpretation employed in Boogie is very similar to that of ESC/Java tool. The difference between the two approaches is that BoogiePL also includes declarations for mathematical functions and axioms. The semantics of the intermediate language BoogiePL is weakest pre-condition semantics in contrast to strongest post-condition semantics of the guarded command language of ESC Java. The verification conditions are generated by computing weakest pre-condition

$$wp(S, true) \tag{1}$$

where S is the program statement. The semantics of BoogiePL are given in Figure 5.
In order to generate VC for a loop, such as

$$LoopHead:\ \textbf{assert}\ I;\ S;\ \textbf{goto}\ LoopHead;$$

it is transformed into the following:

$$\begin{gathered} x_1^0 := x_1; \cdots; x_n^0;\ \textbf{assert}\ I; \\ \textit{havoc}\ x_1, \cdots, x_n;\ \textbf{assume}\ \ I; \\ S;\ \textbf{assert}\ I;\ \textbf{assume false}; \end{gathered}$$

Further, the abstract interpretation engine of Boogie has implementations for various abstract domains — polyhedra abstract domain, heap succession domain for heaps, abstract domains for constant propagation and dynamic type analysis. The basic idea is to have a combination of abstract domains so that one can trade-off precision and efficiency. The BoogiePL with invariants is passed on to the VC generator. The invariant generation algorithm of Boogie is rather involved. The core is the use of a lemmas-on-demand theorem prover. The basic idea is to pass ¬(1) to a theorem prover. If it is UNSAT then S is valid. If it is SAT the theorem prover returns a *monome* — a conjunction of atomic formulas that makes the formula SAT. This step utilizes a DPLL-based SMT solver, solving a Boolean abstraction of ¬(1), and it is similar to finding minimum unsatisfiable core in SMT solvers. Once a monome is found, an algorithm to extract facts from the monome is invoked. This algorithm can switch between various abstract domains in order to tune the quality of the facts, the invariants generated. The verification flow of BoogiePL is depicted in Figure 6.

An advantage of Boogie is that the precision can be tuned by varying the abstract domains. Further, the use of SMT solver for computing the monome could be efficient. But the predicates discovered may not be the best ones and the fact generation algorithm might get expensive if it runs into many iterations. Boogie claims that the VCs generated by Boogie are more compact than what is generated by ESC.

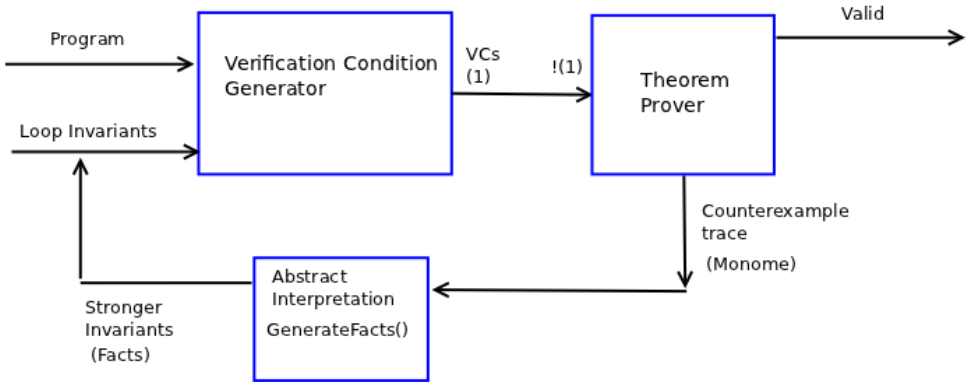


Figure 6: BoogiePL: Verification Flow

The KeY System: The KeY verification system [SW07] uses a combination of deductive verification and abstract interpretation. The main difference in this approach with respect to ESC/Java and Boogie is that the three main components — the abstract interpretor, the VC generator and the theorem prover are all integrated within one program logic. The approach works on the program logic JavaCard DL (Dynamic Logic). It extends first order predicate calculus with $[p]$ (box) and $\langle p \rangle$ (diamond) operators for program statement p. $[p]\psi$ means that if execution of p terminates in a state s then ψ holds in state s. $\langle p \rangle$ additionally requires that p does terminate.

Formulas are of the form $(\phi \implies [p]\psi)$. The JavaCard DL calculus includes proof rules similar to sequent calculus for programming constructs of Java. Basically, this approach works by starting with a formula of the form $(\phi \implies [p]\psi)$. The initial ϕ is $true$. Starting from $(true \implies [p]\psi)$ where p is the program fragment which might contain loops. p is symbolically executed by applying the rules of JavaCard DL. This gradually transforms $(true \implies [p]\psi)$ into a form $(\phi_1 \implies [p]\psi)$. The left hand side of the implication is considered as a candidate for loop invariant. At a certain point (technically, just after the application of `loopMerge` rule) predicate abstraction is applied. The predicates P are heuristically chosen — the loop guard, parts of the post-condition and all parts of the invariant candidate (left hand side of $(\phi_1 \implies [p]\psi)$) accumulated before the first unfolding of the loop. A conjunction of predicates $p \in P$ such that $(\phi_i \implies p)$ is computed which is the desired loop invariant.

Other Approaches: There are other approaches to automatically generate loop invariants. Template based invariant inference is employed in [SG09]. It is based on [CSS03] and extends it for quantified invariants using SMT solvers instead of specialized non-linear solvers. The user provides a template, guessing the shape of the invariant to be discovered, along with the predicates. The problem of invariant discovery is reduced to a problem of finding the optimal solutions for unknowns (in the template) over user defined predicates. The invariant inference algorithm fills in the template with the predicate valuations discovered. This instantiated template becomes the loop invariant. Guessing invariant templates might be as challenging as guessing the right predicates, or in certain cases the invariant itself. For in certain examples, the invariants could be of different shapes and hence working with just a single template might introduce a restriction. Approaches as in INTER-

PROC [Jea] static analyzer use abstract interpretation techniques with numerical abstract domains (the Apron library). The input is a program written in a small imperative language with loops and recursive procedures. It performs forward or backward analysis (or a combination) discovering facts (invariants) at each program point. The output is a program annotated with invariants. A new approach described in [SDDA11] transforms the input program by rewriting multi-phase loops into a sequence of loops and then applies the techniques of abstract interpretation. The main drawback of techniques like [Jea], as mentioned in [SDDA11], is the inability to generate useful loop invariants for programs with multi-phase loops, and hence the need for transformation of the input program.

3 Verification of C Programs

C programs are still widely used, mostly in embedded softwares. The embedded softwares are in turn widely used in safety-critical applications. Hence, there is a need to ensure the correctness of the programs used in these applications up to the highest confidence level. Exhaustive testing is clearly ruled out because of the complexity of the softwares and also because of the fact that testing does not ensure the absence of faults, though it could figure out the presence of it. The only way is to do a rigorous mathematical treatment using formal verification tools in order to ensure the correctness of these programs.

Frama-C [Fra08] is a verification framework that combines a set of program analysis engines in order to reason about the correctness of C-programs. The basic idea behind the architecture of Frama-C is derived from the ideas in [FM04] and [FM07]. Frama-C borrowed the multi-prover verification paradigm to C programs and included techniques for static analysis of C code. The input is an annotated C program that conforms to ACSL [BFM+08] syntax. Annotations include function contracts, pre and post-conditions, loop invariants and assertions. The verification is usually achieved translating the annotated program into an intermediate language — Jessie [Mar07] which generates a set of verification conditions using weakest pre-condition computation. These verification conditions (VCs) are discharged by a bunch of automated provers. The VCs when proved correct establish the correctness of program behaviour. The flow of the Frama-C/Jessie/Why tool chain is shown in Figure 7.

The input C programs are manually annotated by the user. Of ACSL annotations, loop invariants are of special interest. The topic of loop invariants has been researched for decades in order to find techniques to generate them without much manual intervention. In some cases, writing the loop invariant could be more challenging than verifying the program itself. In this report, we suggest innovative ways of generating loop invariants automatically with the help of user-defined predicates (hints). The techniques suggested use predicate abstraction employing SMT solvers thus scaling up the algorithms to larger examples, in an efficient manner.

4 Predicate Abstraction

Abstraction techniques help in taming the complexity of automated formal verification, by simplifying the original program, henceforth referred to as the *concrete program*. *Predicate Abstraction* [GS97] is one of the most widely used abstraction techniques, particularly in automated abstraction-refinement frameworks that are more popularly dubbed CEGAR [CGJ+03]. In this tech-

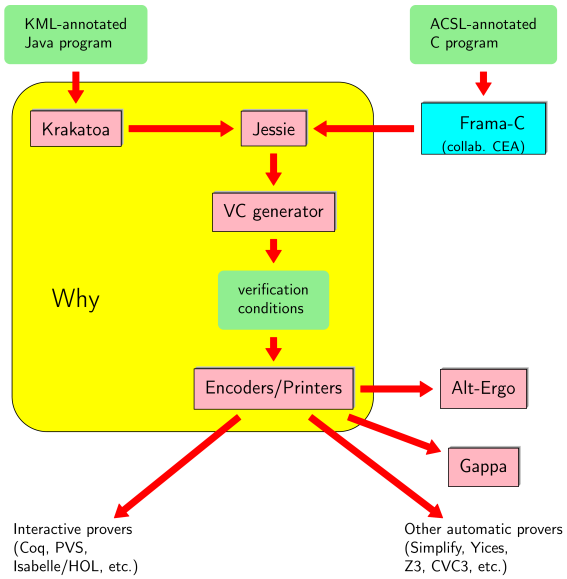


Figure 7: Combined Why/Frama-C Architecture

nique, the behaviour of a set of predicates is tracked with respect to the behaviour of the concrete system. Predicates $\alpha(\vec{x})$ are relations over the variables of the concrete program $\vec{x}$. Each predicate $\alpha_i(\vec{x})$ is represented by a Boolean variable b_i in the abstracted version, henceforth referred to as the *abstract program*. Intuitively, each such b_i helps in partitioning the concrete state space into two — one in which b_i holds and the other in which it does not. Since predicates correspond to Boolean variables, the abstract program obtained in this manner is a Boolean program and is much more amenable to model-checking techniques. Thus, each abstract state corresponds to a set of concrete states where the corresponding predicate holds. Two abstract states are related if and only if there exists a transition between two sets of concrete states in the corresponding concrete program. Abstractions computed in this manner are conservative, i.e., if the abstract program is free of bugs, so is the concrete program. On the other hand, if there is a bug in the abstract program, there may not necessarily be a bug in the concrete program. A bug in the concrete program may be *spurious*, introduced as a result of loss of precise information during abstraction process.

For our purposes of description, we can consider the system to be defined by an initial state and state transitions. A state is defined as the values of the state variables. Let $\vec{x}$ be the set of variables of the concrete program and let $\Phi_c(\vec{x})$ be a first order logic formula describing the concrete program — concrete initial states and concrete transitions. Predicate abstraction computes the propositional formula

$$\Phi_a(\vec{b}) \;\equiv\; \exists \vec{x}.\, (\Phi_c(\vec{x}) \;\wedge\; \bigwedge_i b_i \leftrightarrow \alpha_i(\vec{x})) \tag{2}$$

4.1 Predicate Abstraction as Quantifier Elimination

From equation 2, we see that computation of abstraction involves just quantifier elimination. If the equation is cast as a SMT problem, any SMT solver that is tuned to give us a set of satisfying assignments — an `AllSMT` procedure, could solve the above equation. The set of assignments we thus obtain characterizes the abstract program. Henceforth we refer to the predicate abstraction procedure `Solve` which is shown below:

Algorithm 1 Predicate Abstraction: Quantifier Elimination

1: **function** `Solve`$(T(s, s'), b_i, b'_i)$
2: $\vec{x} := Var(s)$;
3: $\vec{x'} := Var(s')$;
4: $r_\alpha := \exists \vec{x}, \vec{x}'.((b_i \leftrightarrow \alpha(\vec{x})) \wedge (b'_i \leftrightarrow \alpha(\vec{x}')) \wedge T(s, s'))$
5: **return** r_α
6: **end function**

5 Predicate Abstraction for Loop Invariant Generation

Loop invariant generation is an interesting and challenging problem that has been tacked by computer scientists for years. There has been several approaches for computing loop invariants, as described in Section 2. We tackle loop invariants in the context of Frama-C, the main goal being automated discovery of loop invariants for annotating C programs, with the user having to provide just a few hints - the predicates. Most of these predicates could also be extracted from the program itself.

We concentrate on predicate abstraction techniques as they have a demonstrated potential in ensuring scalability and making program analysis feasible. We propose to use techniques of predicate abstraction in order to automate the process of generating loop invariants. We use a SMT solver as an abstraction engine for computing the abstractions as described in the previous section. We implement two techniques for generating loop invariants from a given set of predicates. The first technique computes invariants at every program point, and is referred to as *abstraction at program points*. A second technique computes invariants by considering the path programs at the loop head. We shall refer to this as a *abstraction at loop heads*. The following section describes the techniques in greater detail.

5.1 Abstraction at Program Points

We deviate from the usual predicate abstraction procedures that abstract the entire concrete program *en masse*. Instead we compute the abstraction with respect to a given set of predicates at each program point of the concrete program. The valuations of the chosen predicates at each program point indicate precise information at that program point. For reasons of ease of explanation, we assume that we are given a set of predicates. In Section 7 we give some insights into discovering predicates automatically for use with such a procedure.

```
int main()
{
  int x = 0;
  int y = 1;
  while (x < 100) {
    x = x + y;
    y = x;
  }
  //@ assert x < 200;
  return 0;
}
```

Figure 8: A Simple C Program

We use the C function of Figure 8 as a running example for illustrating our approach. The CFG of this concrete program is shown in Figure 9. We assume that we are given the set of predicates: $P : \{(x < 200), (x < 100), (x + y < 200)\}$. We compute the abstraction at each program point in the CFG shown in Figure 9, w.r.t the given set of predicates P. Let us consider the first node in the CFG and its transition as shown in Figure 10(a).

The algorithm for constructing the abstract program by computing predicate abstraction at each program point on the concrete CFG is given below:

Algorithm 2 Abstract CFG Construction

procedure `BuildAbsCFG`($G : CFG$)
  $A : AbsCFG := empty$
  `Build`$(0, _)$
where
  function `Build`($i : node, v : valuation$)
    if $(i, v) \in A$ **then**
      return (* node already reached *)
    end if
    add state (i, v) in A
    for all j successor of i in G **do**
      $sol :=$ `Solve`(G, i, v, j)
      for all $v' \in sol$ **do**
        `Build`(j, v')
        add transition (i, v) to (j, v') in A
      end for
    end for
  end function
end procedure

The algorithm proceeds as follows. From the input concrete CFG G it builds the abstract CFG A to be constructed using predicate abstraction. The recursive procedure `Build`(i, v) builds the part of A starting from node (i, v) where i is a node of G and v denotes some valuation of the predicates.

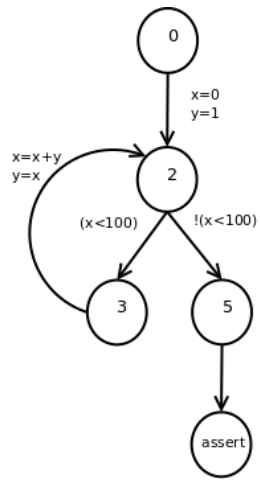


Figure 9: Concrete Program CFG

To start with `Build`() is called with initial index '0' and an empty valuation ('_') as its parameters. Recursively, for each successor node j of node i in G the algorithm computes the abstract reachable states by a call to `Solve`() – predicate abstraction step. Since the number of Boolean predicates we use are finite, we will always have finite number of satisfiable solutions. And for each of these solutions v', a recursive call to `Build`() is made with the successor node j as the first parameter and the solution v' as the second parameter. The transitions $((i, v), (j, v'))$ are added to the abstract CFG A. Since we compute all possible solutions given a set of predicates, the abstractions we compute are precise. In other words, given a set of predicates we compute all possible abstract transitions corresponding to concrete transitions. Each such satisfying assignment v' is a valuation of the set of predicates whose concretization $(\bigwedge_{(i,v')\in A} \gamma(v'))$ is essentially an invariant at node i in the CFG G.

For illustrating the above algorithms, let us consider the first node in the concrete CFG of our running example in Figure 9. The fragment of the concrete CFG is shown in Figure 10(a). The node labelled '0' corresponds to s in Algorithm 1, and the node labelled '1'corresponds to s'. The assignments to variables x and y corresponds to the transition relation $T(s, s')$. Computing the predicate abstraction as described in Equation (2), we require to solve the following quantifier elimination problem:

$$\begin{aligned}
&\exists x_0, x_1, y_0, y_1.((x_1 = 0 \wedge y_1 = 1) \wedge \\
&(b_0 \leftrightarrow (x_0 < 200) \wedge b'_0 \leftrightarrow (x_1 < 200)) \wedge \\
&(b_1 \leftrightarrow (x_0 < 100) \wedge b'_1 \leftrightarrow (x_1 < 100)) \wedge \\
&(b_2 \leftrightarrow (x_0 + y_0 < 200) \wedge b'_2 \leftrightarrow (x_1 + y_1 < 200)))
\end{aligned}$$

The above formula could be fed to an SMT solver capable of also providing satisfying assignments to a first-order formula. We project the assignments on the set of variables that are of interest to us, namely the Boolean variables $\{b_0, b_1, b_2\}$ and their corresponding next state variables $\{b'_0, b'_1, b'_2\}$. We use Alt-Ergo SMT solver which was suitably modified for our requirements. Each v' of the possible set of assignments assignments denoted sol in Algorithm 2 for the above formula

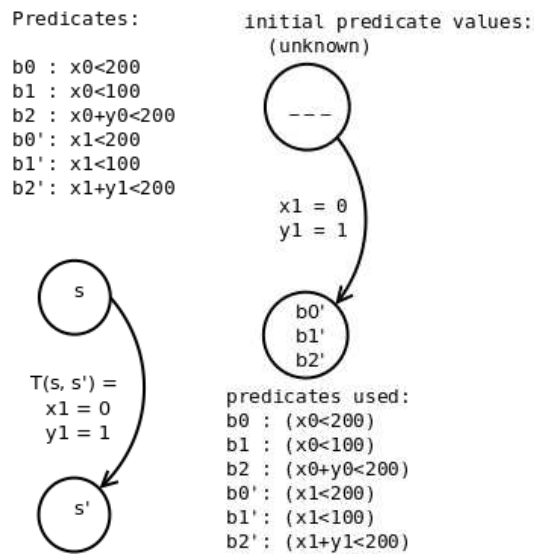


Figure 10: Abstraction at program point

as given by the solver – $\{b'_0 = \top, b'_1 = \top, b'_2 = \top\}$ is subsequently used as the initial valuation of the predicates $\{b_0 = \top, b_1 = \top, b_2 = \top\}$ for computing predicate abstractions for the successive node in the CFG. The algorithm recursively calls itself with the new node ($j = 1$) and updated valuation of the predicates (v') as shown in Line 13 in Algorithm 2. The abstract CFG is built from the solutions of calls to $Solve()$ and it is shown in Figure 11.

In the final abstract CFG A, the concretization of the predicates at any node denotes the invariant at the corresponding node. In our example, the loop head corresponds to node 2 (as shown in Figure 9). We compute the disjunction of the predicate valuations at the nodes indexed 2 to be $I : (b_0 \wedge b_1 \wedge b_2) \vee (\neg b_0 \wedge b_1 \wedge \neg b_1)$. The concretization of I, i.e., $\gamma(I)$ is the desired loop invariant in the concrete CFG G. Hence,

$$((x < 200) \wedge (x < 100) \wedge (x + y < 100)) \vee$$
$$(\neg(x < 200) \wedge (x < 100) \wedge \neg(x + y < 100))$$

is the required loop invariant.

5.2 Abstraction at Loop Heads

The technique described in the previous section works fine for most of the examples. However, there are certain cases where the background abstraction engine - the SMT solver does not have enough information to compute the precise abstract state. An instance of such an example is shown in Figure 12(a).

An elegant approach would be to abstract only at the loop heads. Let us define a *path program* to be the sequence of program statements along a path starting from a particular program point

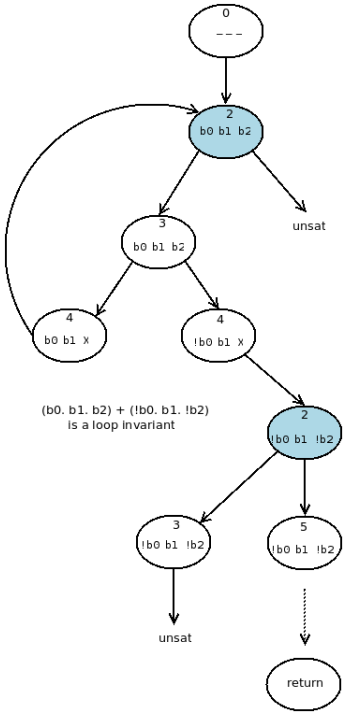
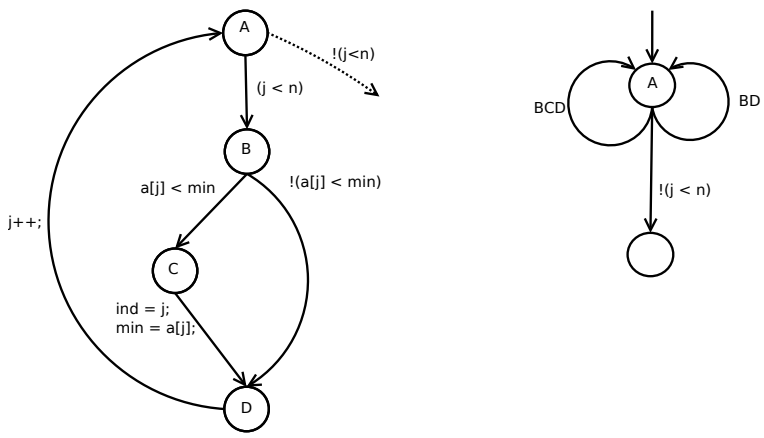


Figure 11: Final Abstract CFG



(a) Insufficient Information at node-D

(b) Path Programs

(say a loop head) and terminating at the same program point (for example, loop iteration) or at a different point (for example, loop termination). In figure 12(a), ABCDA is a path program and so are ABCD and ABD. At each loop head we compute the path programs fanning out of the loop head and abstract each of these path programs using the same technique as above. In contrast to the previous technique which does not record the information at the sink of two paths, this technique records all the information along any particular program path, thereby enabling precise computation of abstractions.

Let us illustrate the technique with a simple C program that computes the minimum element of the input array. The C program and the corresponding path programs in the CFG are shown below:

```
int searchmin (int a[], int n)
{
  int j, min, ind;
  //@ ghost int r;
  ind= 0;
  min = a[0];
  j = 1;
  while (j < n) {
    if (a[j] < min) {
      ind= j;
      min = a[j];
    }
```

```
    j++;
  }
}
return min;
```

Computing predicate abstractions for the path programs ABCDA, ABDA and AE yields the following abstract CFG. As with our previous technique we can extract the loop invariant by concretizing the Boolean formula over predicates at the loop node. In this case, the invariant turns out to be:

$$\forall r.\neg(0 \leq r < j) \wedge (0 < j) \wedge (n \geq 1) \wedge (0 \leq ind < n) \wedge (min = a[ind])$$
$$\vee(0 \leq r < j) \wedge (a[r] \geq min) \wedge (0 < j) \wedge (n \geq 1) \wedge (0 \leq ind < n) \wedge (min = a[ind])$$

which is sufficient to prove the post-condition - $\forall \mathtt{x}.(\mathtt{0} \leq \mathtt{x}) \wedge (\mathtt{x} < \mathtt{n}) \implies (\mathtt{min} \leq \mathtt{a}[\mathtt{x}])$) as well as the assertion - ($\mathtt{min} = \mathtt{a}[\mathtt{ind}]$). Intuitively, the different path programs capture the phase transitions within the loop. Each of the path programs ABCDA and ABDA corresponds to different phases of the loop, corresponding to the conditional ($\mathtt{a}[\mathtt{j}] < \mathtt{min}$) being *true* or *false* respectively. This technique is more relevant in programs involving multi-phase loops (described in Section 6.3) where other techniques [SDDA11] transform the input program in order to generate useful loop invariants.

6 Loop Invariant Generation for Complex Data Structures

6.1 Arrays and Quantified Invariants

The algorithms described in the previous section is also capable of generating more complex invariants, using the same procedure. In particular, it is capable of generating quantified loop invariants.

Let us start with the simplest example with arrays in order to illustrate this. We consider the C program which initializes all the elements of an input array to a particular value.

```
void initcheck(int *a, int n)
{
  int i=0;
  //@ ghost int j;
  while (i<n) {
      a[i] = 0;
      i++;
    }
}
```

We introduce a variable `j` which is quantified over. We introduce it as a `ghost` variable as it does not belong to the program. Let us assume that $P : \{\ \mathtt{b}_0 : (\mathtt{i} < \mathtt{n}), \mathtt{b}_1 : 0 \leq \mathtt{j} < \mathtt{i}, \mathtt{b}_2 : \mathtt{a}[\mathtt{j}] = 0\}$ is the set of user-defined predicates. Note that some of the predicates are over the variable `j`. Our technique generates the loop invariant $\neg b_1 \vee (b_1 \wedge b_2)$ which is the following in the concrete C program after adding the universal quantifier:

$$\forall j; \neg(0 \leq j < i) \vee ((0 \leq j < i) \wedge (a[j] = 0))$$

In short, the technique just introduces variables to be quantified over, and predicates involving such variables, and we prefix the generated invariant with an universal quantifier.

A more complicated example involving arrays is the following C program to find the minimum element of an array. The post-condition that we would like to establish is:

$$\forall \mathtt{x}.(0 \leq \mathtt{x} < \mathtt{n}) \implies (\mathtt{min} \leq \mathtt{a}[\mathtt{x}])$$

We shall use $P : \{(0 \leq \mathtt{r} < \mathtt{j}), (\mathtt{a}[\mathtt{r}] \geq \mathtt{min}), (\mathtt{min} = \mathtt{a}[\mathtt{ind}])\}$ as the set of user-defined predicates.

```
void searchmin (int a[], int n)
{
  int j, min, ind;
  //@ ghost int r;
  ind= 0;
  min = a[0];
  j = 1;
  while (j < n) {
    if (a[j] < min) {
      ind= j;
      min = a[j];
    }
    j++;
  }
}
```

The invariant generated, that also establishes the correctness of the post-condition is:

$$\begin{array}{ll} \forall r. & (0 \leq r < j)) \wedge (a[r] \geq min) \wedge (min == a[ind]) \\ \vee & \neg(0 \leq r < j) \wedge (min == a[ind]) \end{array}$$

The details on rewriting the generated loop invariant with universal quantification are described in Section 7.

6.2 Structures and Pointers

In addition to generating quantified invariants as illustrated in the previous section, the proposed technique can also handle complex datatypes that appear in C programs. Of particular interests are arrays, structures and pointers.

Structures Structures are handled by looking at the fields of the structure as arrays of structure instantiations. This is the so-called *component-as-array* memory model early proposed by Burstall and emphasized by Bornat in 2000 [Bor00].

Since we already know how to handle arrays, once this casting is done, the rest of the technique is straight forward. In order to illustrate this, let us consider the following C program that partitions a list: removing every element greater than an input value `v` from the input list `l` and returns the new list.

```
struct cell {
  int val;
  struct cell *next;
};

struct cell
*partition (struct cell *l, int v)
{
  struct cell *curr, *prev;
  struct cell *newl, *nextCurr;
  curr = l; prev = newl = NULL;
  while (curr != NULL) {
    nextCurr = curr->next;
    if (curr->val > v) {
      if (prev != NULL)
        prev->next = nextCurr;
      if (curr == l)
        l = nextCurr;
      curr->next = newl;
      //@ assert (curr != prev);
      newl = curr;
    }
    else
      prev = curr;
    curr = nextCurr;
  }
  return newl;
}
```

In the structure in the above C program has two fields, `val` and `next`, each of which could be seen as an array indexed by the different instantiations of the structure `cell` - namely `curr`, `prev`, `newl` and `nextCurr`. For example, `val` is seen as `val[curr]`, `val[prev]`, `val[newl]` and `val[nextCurr]`. So, an assignment to one of the fields of the structure is seen as an assignment to an index of the corresponding field array. For example, an assignment to a field of the form `curr→val = 0` would become `val[curr] = 0`, and an assignment of the form `prev→next = nextCurr` becomes `next[prev] = nextCurr`. One of the main reasons for modelling fields in this manner is to get the information from the CFG of the concrete program to the underlying SMT solver in a straight forward manner. As the underlying SMT solver does not understand structures and pointers, it should be recast in such a way as to not lose the intended semantics.

In this example, given the set of predicates P: $\{\mathtt{b}_0 : \mathtt{prev} = \mathtt{NULL}, \mathtt{b}_1 : \mathtt{prev} \rightarrow \mathtt{val} > \mathtt{v}\}$, the proposed techniques generates the following loop invariant:

$$b_0 \vee (\neg b_0 \wedge \neg b_1)$$

Pointers *Pointers* can be handled in a manner similar to structures described above. This is indeed how they are handled in the memory model of the Jessie plugin [Mar07, Hub08, Moy09] The main idea is to view the pointer to a type `t` as a C structure containing only one field of type `t`. For example, `int *p` would be viewed as

```
struct Sint {
  int Fint;
}p;
```

An assignment to a variable `v` of the form `v = *p` would thus become `v = p→Fint`. Consequently, this statement as described in the previous section boils down to `v = Fint[p]`. An assignment to a pointer of the form `*p = exp` becomes `p→Fint = exp`, which eventually becomes `Fint[p] = exp`.

In general, a pointer of type `<t>` of the form `<t> *p` is imagined to have the following C structure:

```
struct S<t> {
  int F<t>;
}p;
```

An assignment of the form `var = *p` involving a pointer thus becomes `var = p→F<t>` and an assignment to a pointer of the form `*p = exp` becomes `p→F<t> = exp`.

6.3 Disjunctive Invariants for Multi-phase Loops

A challenging case for loop invariant generation is the case where the program requires disjunctive invariant. This combined with *multi-phase* loops - where a *condition* within a loop evaluates to *true* during one iteration of the loop and to *false* during another iteration. The following C program borrowed from [SDDA11] illustrates this:

```
int main ()
{
  int x=0, y=50;
  while (x < 100) {
    if (x < 50)
      x++;
    else {
      x++;
      y++;
    }
  }
  //@ assert (y == 100);
  return 0;
}
```

The conditional $(x < 50)$ evaluates to *true* during some iterations of the while loop and for the remaining iterations, evaluates to *false*. Techniques involving abstract domains like INTERPROC computes loop invariants that are conjunctions of inequalities, but still can not prove the assertion. Abstraction refinement based techniques do succeed in proving the assertion, but go through an unacceptable number of iterations of the CEGAR loop. The technique proposed in [SDDA11] transforms the above program into an equivalent program with two while loops corresponding to the different phases of the conditional $(x < 50)$ and then uses the usual techniques of abstract interpretation for generating loop invariants, that would assist in proving the assertion right. Our technique based on predicate abstraction does not require any program transformation and still it could generate loop invariants that are strong enough to prove the assertion right. We use P :

$\{\mathtt{b_0} : \mathtt{x} \leq 50,\ \mathtt{b_1} : \mathtt{y} = 50,\ \mathtt{b_2} : \mathtt{x} \leq 100,\ \mathtt{b_3} : \mathtt{x} = \mathtt{y},\ \mathtt{b_4} : \mathtt{y} = 100\ \}$ as the set of predicates. The generated loop invariant is:

$$(\neg(x \leq 50) \wedge (\neg(y = 50) \wedge ((x \leq 100) \wedge (x = y)))) \vee ((x \leq 50) \wedge ((y = 50) \wedge ((x \leq 100) \wedge \neg(y = 100))))$$

which also helps in establishing the correctness of the assertion $(y = 100)$.

7 Predicate Discovery

Some of the predicates that might be useful for generating loop invariants using the proposed technique can be syntactically obtained from the program itself. We illustrate the discovery of simple predicates from the program text using Example 8. The example uses three predicates namely $\mathtt{b_0} : \mathtt{x} < 200$, $\mathtt{b_1} : \mathtt{x} < 100$ and $\mathtt{b_2} : \mathtt{x} + \mathtt{y} < 200$. The first of the predicates, $\mathtt{b_0} : \mathtt{x} < 200$ is the assertion to be proved. The predicate $\mathtt{b_1} : \mathtt{x} < 100$ is the loop guard. The third and the more interesting predicate $\mathtt{b_2} : \mathtt{x} + \mathtt{y} < 200$ is the weakest precondition of the assertion with respect to the program statements. Intuitively the weakest precondition of the post-condition (here, the assertion) gives the invariant at the head of the loop that would contribute to driving the assertion to *true*.

In the case of quantified invariants, we need predicates over quantified variables. For purposes of illustration, let us consider a C program for initialization of arrays, as shown in Example 8. A trivial post-condition that we would like to establish is that all the elements of the array are initialized to 0 when the function returns.

We follow the usual heuristics for discovering predicates for universally quantified invariants. First, we add the loop guard as one of the predicates. Hence, $\mathtt{b_0} : (\mathtt{i} < \mathtt{n})$ is a predicate. For the more interesting predicates, we explore the loop body. If the loop targets contain updates of array or fields, we introduce a variable to be quantified over. In the above example, the loop body contains array modification (`a[i] = 0` on Line 6). We introduce a variable `j` as a ghost variable (as on Line 4). This variable is not a part of the program, but is introduced to quantify over. We introduce new predicates referring to `j`, that indicate the scope of the integer variables the index of the array ranges over. In this case, the array index ranges from 0 to n. Hence, we introduce a predicate referring to `j`: $b_1 : 0 \leq \mathtt{j} < \mathtt{i}$. And finally, we introduce a predicate referring to the array update $\mathtt{b_2} : (\mathtt{a}[\mathtt{j}] = 0)$. With the three above mentioned predicates, the proposed technique discovers $\neg \mathtt{b_1} \vee (\mathtt{b_1} \wedge \mathtt{b_2})$ which with the intended universal quantifier is:

$$\forall \mathtt{j}; \neg(0 \leq \mathtt{j} < \mathtt{i}) \vee ((0 \leq \mathtt{j} < \mathtt{i}) \wedge (\mathtt{a}[\mathtt{j}] = 0))$$

The effectiveness of the loop invariant generated using predicate abstraction techniques heavily depends on the quality of the predicates used. While the user is allowed to add predicates of his choice, irrelevant predicates when added result in a bottleneck in the computation of abstraction. It is important to choose predicates wisely. Other techniques like Craig interpolants [Cra57] could also be used to infer localized predicates. But these techniques are beyond the scope of the topic of this report and hence ignored.

Simplification of Invariants We build BDDs incrementally for the formulas that constitute the invariants. Although variable ordering is an open problem in BDDs, most of the real-world examples would involve loop invariants as formulas over a small set of variables - a dozen or less, and for such

a case, ordering does not affect the performance drastically. On the other hand, we are able to get shorter and more understandable expressions for loop invariants after this simplification.

8 Experimental Evaluation

Let us consider other simple examples, in order to illustrate the potential of our approach in generating loop invariants.

Example 1: The following example has been borrowed from [FQ02] and cast as a C program.

```
int main (int a[], int b[], int length)
{
  int i;
  int spot = length;
  //@ ghost int j;
  for (i=0; i<length; i++) {
    if (spot == length)
      spot = i;
    if (a[i] != 0)
      b[i] = 1;
    else
      b[i] = 0;
  }
  return spot;
}
```

The variable `spot` is initialized to the length of the array. It is assumed that arrays `a` and `b` are of the same length. The program returns the index of the first non-zero element of `a` if such an element exists. In case there is no such element, it returns the length of the array. We try to establish the post-condition $\forall \mathtt{j}.(0 \leq \mathtt{j} < \backslash\mathtt{result}) \implies (\mathtt{b}[\mathtt{j}] = 0)$. We use $P : \{\ \mathtt{spot} == \mathtt{length},\ \mathtt{b}[\mathtt{spot}] \neq 0,\ \mathtt{spot} < \mathtt{i},\ \mathtt{i} \leq \mathtt{length},\ 0 \leq \mathtt{j} < \mathtt{i},\ \mathtt{a}[\mathtt{j}] \neq 0,\ \mathtt{b}[\mathtt{j}] \neq 0,\ \mathtt{j} < \mathtt{spot}\ \}$ as the set of predicates for abstraction. Our technique generates the following loop invariant which is sufficient to establish the correctness of the post-condition.

$$\neg(\mathtt{spot} == \mathtt{length}) \wedge (\mathtt{spot} < \mathtt{i}) \wedge (\mathtt{i} \leq \mathtt{length}) \wedge \neg(0 \leq \mathtt{j} < \mathtt{i}) \wedge \neg(\mathtt{spot} == \mathtt{length})$$
$$\vee(0 \leq \mathtt{j} < \mathtt{i}) \wedge \neg(\mathtt{a}[\mathtt{j}] \neq 0) \wedge \neg(\mathtt{b}[\mathtt{j}] \neq 0) \wedge \neg(\mathtt{j} < \mathtt{spot}) \wedge \neg(\mathtt{spot} == \mathtt{length})$$
$$\vee(\mathtt{a}[\mathtt{j}] \neg = 0) \wedge (\mathtt{b}[\mathtt{j}] \neg = 0) \wedge \neg(\mathtt{j} < \mathtt{spot}) \wedge \neg(\mathtt{spot} == \mathtt{length})$$
$$\vee(\mathtt{spot} == \mathtt{length}) \wedge (\neg(\mathtt{spot} < \mathtt{i}) \wedge (\mathtt{i} \leq \mathtt{length}) \wedge \neg(0 \leq \mathtt{j} < \mathtt{i}) \wedge (\mathtt{spot} == \mathtt{length})$$

Example 2: In the following C program, we would like to prove the assertion $(\mathtt{a} + \mathtt{b} = 3\mathtt{n})$ at the end of the loop. We use $P : \{\ 3\mathtt{i} = \mathtt{a} + \mathtt{b},\ 3\mathtt{n} = \mathtt{a} + \mathtt{b},\ 3(\mathtt{i} + 1) = \mathtt{a} + \mathtt{b},\ \mathtt{i} < \mathtt{n}\ \}$ as the set of user-defined predicates.

```
extern unsigned int nd();
void tricky(int n)
{
  int i = a = b = 0;
  int x;
```

```
  while (i<n)
    {
      x = nd();
      if (x != 0)
        {
          a = a + 1;
          b = b + 2;
        }
      else
        {
          a = a + 2;
          b = b + 1;
        }
      i++;
    }
  //@ assert (a+b == 3*n);
}
```

The generated loop invariant is:

$$(3\mathtt{i} = \mathtt{a} + \mathtt{b}) \wedge \neg(3\mathtt{n} = \mathtt{a} + \mathtt{b}) \wedge \neg(3(\mathtt{i} + 1) = \mathtt{a} + \mathtt{b}) \wedge (\mathtt{i} < \mathtt{n})$$
$$\vee(3\mathtt{n} = \mathtt{a} + \mathtt{b}) \wedge \neg(3(\mathtt{i} + 1) = \mathtt{a} + \mathtt{b}) \wedge \neg(\mathtt{i} < \mathtt{n})$$

which also helps in establishing the correctness of the assertion at the end of the loop.

Example 3: The following example is borrowed from a TACAS paper [GCNR08] and it illustrates a simple, but tricky to verify C program. In order to establish the correctness of the post-condition ($\mathtt{assert}(\mathtt{x} >= 4 \implies (\mathtt{y} > 2))$ we choose as predicates, the atomic predicates in the post-condition as well as the phase-changing condition $\mathtt{BN} == 1$. With the set of predicates $P : \{\mathtt{x} \geq 4, \mathtt{y} > 2, \mathtt{BN} == 1\}$ our invariant generation algorithm generates the following loop invariant:

$$((\neg(\mathtt{x} \geq 4) \wedge (\neg(\mathtt{y} > 2) \vee ((\mathtt{y} > 2) \wedge (\mathtt{BN} == 1)))) \vee ((\mathtt{x} \geq 4) \wedge (\mathtt{BN} == 1)))$$

which is sufficient to establish the correctness of the post-condition.

```
//@ behavior agen: ensures \true;
void foo(){
  int BN, x, y, w, z, arandom, brandom, crandom;
  x = y = w = z = 0;

  while(BN == 1) {
    if(BN == 1){
      x=x+1;
      y=y+100;
    }
    else{
      if(BN == 1){
        if(x>=4){
          x=x+1;
          y=y+1;
```

```
        }
      }
      else{
        if(y>10*w && z>=100*x){
        y=-y;
        }
      }
    }
    w=w+1;
    z=z+10;
  }
  //@ assert (x >= 4) ==> (y > 2);

  //@ for agen: assert (x >= 4);
  //@ for agen: assert (y > 2);
  //@ for agen: assert (BN == 1);
}
```

Example 4: Let us consider the following simple C program and try to generate loop invariants for it.

```
extern int nd();
void test(int n)
{
  int b;
  int i=0;
  while (i<n) {
    b = nd();
    if (b != 0)
    i++;
  }
  //@ assert (i == n) && (b != 0);
}
```

As predicates we choose the atomic predicates that appear in the post-condition (assertion) as well as the loop guard. So, the set of predicates we work with is $P : \{\ \mathtt{i} \leq \mathtt{n}, \mathtt{b} \neq 0, \mathtt{i} < \mathtt{n}\}$. The loop invariant generated using any of the two techniques is:

$$(\mathtt{i} \leq \mathtt{n}) \wedge ((\neg(\mathtt{b} \neq 0) \wedge (\mathtt{i} < \mathtt{n})) \vee (\mathtt{b} \neq 0))$$

which is sufficient to establish the post-condition.

Our approach is also scalable and gives promising results for industrial-size software as shown in the following table. We experimented with both variants of the predicate abstraction technique described in this report indicated by the columns labelled 'Loop' and 'Point' respectively, in the table below. The examples are borrowed from research papers on verification wherein the correctness of industrial code like Apache have been established. Most of the examples were borrowed from the benchmarks of InvGen. In the following table an entry 'Y' in the column 'Useful Invariant' indicates that the generated invariant was useful in proving the post-condition or assertion. A 'N' indicates that

the loop invariants generated were not strong enough to prove the post-condition. The 'Predicates' column indicates the number of predicates that were required to do so. The last column shows the time required (user time + system time) for computing the invariants. The experiments were conducted on a machine with 4GB of RAM, 2.4 GHz Intel Core 2 Duo processor running Linux (Ubuntu 11. 04) operating system.

Benchmark	Useful Invariant		# Predicates	Time (s)
	Loop	Point		
`apache_get_tag.c`	Y	Y	4	1.164
`NetBSD_loop_int.c`	Y	Y	3	0.512
`sendmail_mime_XX.c`	Y	Y	2	0.536
`sendmail_mime_fromqp.c`	Y	N	3	0.732
`svd_some_loop.c`	Y	N	7	16.329
`svd4.c`	Y	Y	7	11.741
`nested9.c`	Y	Y	3	0.884
`nest-if5.c`	Y	Y	4	2.524

9 Conclusions and Future Work

In this report, we proposed techniques for automated generation of annotations for deductive verification of C programs in the Frama-C verification framework. In particular we showed that predicate abstraction techniques can be effectively used for generating complex loop invariants automatically. We also provided heuristics for choosing effective predicates in order to improve the quality of the generated invariants. The experimental results show the potential of this approach.

The directions for future research is two-fold: (1) gathering information from other program analyzers like value analysis or data-flow analysis or abstract interpretation techniques and using the relevant information as predicates would be a better way of choosing predicates that are specific to certain program variables. As Frama-C already includes these techniques it should be just a matter of passing the information from one analyzer to another (loop invariant generator). Further the plugin for weakest precondition computation could be effectively used to discover new predicates as discussed in Section 7. (2) one could extract the abstract CFG of the input program and subject it to model checking techniques in a CEGAR framework [CGJ+03,CFG+10,Kal09] for verifying simple temporal properties of large programs. Other extensions could include use of the technique for generating other interesting annotations, like assertions at interesting program points, and improve the usability of the tool by providing constructs for adding predicates on the fly and computing incremental abstraction.

The research in predicate abstraction techniques is almost a decade old and still is gaining interest. Loop invariant generation techniques have been around for decades and is still seen as a challenging research problem. A combination of these two techniques as proposed in this report and also combining it with other techniques of interest as just discussed would enable cleaner software engineering, easier verification and eventually improve productivity of verification engineers in the long run.

References

[BCD+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs 0002, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[BFM+08] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2008. http://frama-c.cea.fr/acsl.html.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI 2001, pages 203–213, New York, NY, USA, 2001. ACM.

[Bor00] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35:677–691, August 1986.

[CFG+10] Alessandro Cimatti, Anders Franzen, Alberto Griggio, Krishnamani Kalyanasundaram, and Marco Roveri. Tighter integration of BDDs and SMT for predicate abstraction. In *Design, Automation & Test in Europe*, Dresden. Germany, March 2010. IEEE.

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. In *Journal of the ACM*, volume 50, pages 752–794, New York, NY, USA, September 2003. ACM.

[Cra57] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.

[CSS03] Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In *Proceedings of Computer Aided Verification*, CAV 2003, pages 420–432. Springer Verlag, 2003.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A Theorem Prover for Program Checking. *J. ACM*, pages 365–473, May 2005.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[FM04] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *ICFEM*, pages 15–29, 2004.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In *CAV*, pages 173–177, 2007.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL 2002, pages 191–202, New York, NY, USA, 2002. ACM.

[Fra08] The Frama-C platform for static analysis of C programs, 2008. http://www.frama-c.cea.fr/.

[GCNR08] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Automatically Refining Abstract Interpretations. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 443–458, Berlin, Heidelberg, 2008. Springer-Verlag.

[GR09] Ashutosh Gupta and Andrey Rybalchenko. InvGen: An Efficient Invariant Generator. In *CAV*, pages 634–640, 2009.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV 1997, pages 72–83, London, UK, 1997. Springer-Verlag.

[Gul09] Gulwani, Sumit and Srivastava, Saurabh and Venkatesan, Ramarathnam. Constraint-Based Invariant Inference over Predicate Abstraction. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 120–135, Berlin, Heidelberg, 2009. Springer-Verlag.

[Hub08] Thierry Hubert. *Analyse Statique et preuve de Programmes Industriels Critiques*. Thèse de doctorat, Université Paris-Sud, June 2008.

[Jea] B. Jeannet. The INTERPROC analyzer.

[Kal09] K. Kalyanasundaram. *CEGAR Using SMT Solvers for Predicate Abstraction*. PhD thesis, University of Trento, March 2009.

[Lei08] K. Rustan M. Leino. This is boogie 2, 2008.

[Mar07] Claude Marché. Jessie: an intermediate language for Java and C verification. In *Programming Languages meets Program Verification (PLPV)*, pages 1–2, Freiburg, Germany, 2007. ACM.

[Mic03] Michael Colón and Sriram Sankaranarayanan and Henny Sipma. Linear Invariant Generation Using Non-linear Constraint Solving. In *CAV*, pages 420–432, 2003.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[Moy09] Yannick Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, January 2009.

[SDDA11] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In *CAV*, page *To Appear*, 2011.

[SG09] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI 2009, pages 223–234, New York, NY, USA, 2009. ACM.

[SW07] Peter H. Schmitt and Benjamin Weiß. Inferring invariants by symbolic execution. In Bernhard Beckert, editor, *Proceedings, 4th International Verification Workshop (VERIFY'07)*, volume 259 of *CEUR Workshop Proceedings*, pages 195–210. CEUR-WS.org, 2007.

[Wei11] Benjamin Weiß. Predicate abstraction in a program logic calculus. *Science of Computer Programming*, 2011. To appear.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399