SANDIA REPORT
SAND2022-11350
Printed September 2022

Sandia National Laboratories

Software Verification Toolkit (SVT): Survey on Available Software Verification Tools and Future Direction

Nickolas A. Davis, Taylor E. Berger, Arthur McDonald, Joey B. Ingram, James D. Foster, Katherine A. Sanchez

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

Software Verification Toolkit (SVT): Survey on Available Software Verification Tools and Future Direction

Davis, Nickolas A
Department 9364
Sandia National Laboratories
Albuquerque, NM 87185-9999
nadavi@sandia.gov

Berger, Taylor E
Department 9362
Sandia National Laboratories
Albuquerque, NM 87185-9999
teberge@sandia.gov

Foster, James D
Department 9362
Sandia National Laboratories
Albuquerque, NM 87185-9999
jdfost@sandia.gov

Ingram, Joey B
Department 5555
Sandia National Laboratories
Albuquerque, NM 87185-9999
jbingra@sandia.gov

McDonald, Arthur
Department 2494
Sandia National Laboratories
Albuquerque, NM 87185-9999
amcdon@sandia.gov

Sanchez, Katherine
Department 9362
Sandia National Laboratories
Albuquerque, NM 87185-9999
ksanche@sandia.gov

SAND2022-11350

ABSTRACT

Writing software is difficult. However, writing complex, well tested and designed, and functionally correct software is incredibly difficult. An entire field of study is devoted to the validation and verification of software to address this problem, and in this paper we analyze the landscape of currently available third party software. We have divided our analyses into three separate subsections with regards to software validation: formal methods, static analysis, and test generation. Formal verification is the most complex method in which to validate software correctness, but also the most thorough as it truly validates the mathematical validity of the source code. Static analysis generally is relegated to abstract syntax tree traversal techniques to find errors related to faulty software such as memory leaks or stack overflow issues. Automatic test generation is similar in implementation to static analysis, but pushes a bit further in verifying the boundedness of function inputs and outputs with regards to annotated or parsed criteria. The crux of this report is to analyze and describe the software tools that implement these techniques to validate and verify software. Pros and cons related to installation, utilization, and capabilities of the frameworks are described, and reproducible examples are provided with a focus on usability. The initial survey concluded that the most interesting tools of note are Z3, Isabelle/HOL, and TLA+ with regards to formal verification; and Infer, Frama-C, and SonarQube with regards to static analysis. With these tools in mind, a final conjecture is provided that describes future avenues of utilizing these tools for developing a verification framework to assist in validating existing software at Sandia National Laboratories.

ACKNOWLEDGMENT

Thanks to Jon Aytac and Kirk Landon for their input and discussion towards various existing literature and tools available for software verification. Thanks to Tu-Thach Quach for overseeing the development of this project and providing feedback on project direction and for proof-reading the report.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This document is a high level overview of various formal verification tools, static analysis tools, and test generation tools that had actively maintained git repositories associated with the tooling and that had enough documentation to support testing. The intended audience of this report is anyone interested in a brief, high level and usability focused description of the included tools, with an additional focus on the applicability of the analyzed tools with regards to how they can be used to validate existing and new software at Sandia.

EXECUTIVE SUMMARY

Commonly accepted methods of verifying software include the creation of automated tests, code reviews, manual systems testing, and high level static code analysis. While these methods are useful at finding bugs and issues with software, these can't be considered catch-all solutions, as they are particularly bad at guaranteeing that a given piece of software executes properly against a given specification without a chance at failure. This isn't a new problem, as many tools exist that assist in formally validating and more rigorously verifying software systems. However, the majority of these tools are not easy to use, as they require extensive knowledge of mathematical semantics surrounding formal proofs and are usually implemented in a disjoint manner with respect to the software that is being validated against.

This paper aims at taking a large subset of available, well documented, and well maintained software validation solutions and analyzing them with regards to usability, validation capabilities, and robustness. Combined, these analyses will provide an easy to digest, overhead view of the software verification landscape and exist as a resource with which other interested parties at Sandia can gain a quick entry level understanding to how these tools can be utilized to validate software. For every tool analyzed a thorough description is provided to give a general understanding of the purpose of the tool and the types of validation it is attempting to perform. Additional examples will follow, as well as a quick analysis of the pros and cons of a specific tool and some generalized ratings describing the tool's usability.

Our survey concluded that the most interesting tools of note are Z3, Isabelle/HOL, and TLA+ with regards to formal verification; and Infer, Frama-C, and SonarQube with regards to static analysis. These decisions were made on the basis of usability and documentation, with a large focus on non-subject matter expert adoption in verifying software. With regards to usability and future work, it is recommended that an annotation system be developed that is language agnostic. This system can be implemented on top of an Intermediate Representation of source software with the goal of creating a multi-language annotation based software verification system that developers can use to apply contract based verification to software in an inline manner.

ACRONYMS AND TERMS

Acronym/Term	Definition
API	Application Programming Interface: A set of functions that define how to access certain features of a system, application, or service.
ASCL	ANSI/ISO C Specification Language: a behavioral specification language for C programs that can express a wide range of functional properties.
AST	Abstract Syntax Tree: A specific form of an IR that represents code in a tree type format, designating language constructs as nodes in a tree.
BNF	Bachus-Naur Form: A metasyntax notation for context-free grammars used to describe the syntax of languages in computing.
C++	A powerful general purpose programming language generally used for lower level application development.
CI	Continuous Integration: the practice of merging code changes into a central repository and automatically building and testing the software as it is integrated.
GUI	Graphical User Interface: The interface between a user and a computer, usually implies non-text based interfaces.
HOL	Higher Order Logic: A family of interactive theorem proving systems using higher-order logics and implementation strategies.
IR	Intermediate Representation: A language representation that lies between the source language syntax and assembly/machine code.
JBMC	Java Bounded Model Checker: A bounded model checker for java programs that checks runtime exceptions and user defined assertions.
JSON	Javascript Object Notation: A lightweight data-interchange format designed to be easy to read and write.
PBT	Property Based Testing: create properties that define correctness of code test and then use said properties to verify that the code is correct.
PVL	Program Verification Language: A vercors specific java-based verification language.
SAT	Boolean Satisfiability Problem: A problem associated with determining if there exists a set of inputs that satisfies a given Boolean formula.
SML	Standard ML: A general purpose modular functional programming language with compile-time type checking and type inference.
SMT	Satisfiability Modulo Theories: A problem associated with determining whether a mathematical formula is satisfiable (a formula is true under some assignment of values to its variables).
SVT	Software Verification Toolkit: the framework discussed throughout the report upon which the analyzed tools are being selected for inclusion within.

VST	Verified Software Toolchain: a set of tools for proving the functional correctness of C programs.
XML	Extensible Markup Language: A simple text-based format for representing structured information.
XOR	Exclusive Or: A Boolean mathematical property that is only True when one of two input values is True.

1. INTRODUCTION

Software is ubiquitous, and its impact spans many national security applications ranging from the power grid to space systems. In this setting, a seemingly minor flaw, such as an integer overflow or a buffer overwrite, in the implementation of a critical algorithm can lead to catastrophic results. For decades, developers have relied on various combinations of manual quality and assurance, automated testing with code coverage, and code reviews to show that a given set of software is bug free. While these are good strategies, and good software engineering practices, they can give software developers a false sense of true implementation correctness. But this area of software verification is not completely novel: there exist many software tools that aim to verify and validate existing software systems. These tools usually fall into one of three different categories: formal verifiers, static analyzers, and test generators. Formal verifiers attempt to prove or disprove the correctness of a given function or set of software by generating mathematical definitions relative to a specific contract. Static analyzers lexically and semantically parse existing code bases and attempt to automatically reason about verifiable states of software, such as possible integer overflow or buffer overflow exceptions. Test generation works similarly to static analysis, in that it attempts to parse existing code and then generate test cases that satisfy a certain code coverage requirement. While each of these types of tools are all implemented differently, they all share the same base goal of more rigorously verifying the correctness of software. The following sections of this report dive further into the specific definitions of these categories, as well as the analyses of many different types of tools that belong to these categories.

2. FORMAL VERIFICATION

Formal verifiers fall into a rigorous category of program verification as they seek to prove or disprove the correctness of software systems. Many different types of formal verifiers exist, each with a different application domain and set of problems that they attempt to solve. However, the majority of these verifiers all utilize formal proofs enacted upon abstract mathematical models of some system. In the context of SVT, these systems usually do not produce a 1-to-1 mapping between source code written in a programmatic language and the domain language that the formal verifier actually uses. This makes devising a formal proof that existing software is valid a very time intensive and rigorous process: care must be taken to ensure that what the formal verifier is proving matches the algorithm's intent. Also, the concepts of compilers and architectures make the formal verification of actual software even more difficult, as the verification of an algorithm written in a specific language adds additional abstraction. Combined, all of these problems make the formal verification of existing software difficult. However, just because it is difficult does not mean we should push formal verification to the wayside. Because formal verification attempts to prove the algorithms correct, it is a much more stable solution system than static analysis or test generation: formal verification guarantees that software properties will hold under all conditions. With the goal of verifying software in mind, we have analyzed multiple existing formal verification systems that are publicly available for use. Many of these systems operate with their own functional domain language, while others use either annotations or custom parsers to

translate language syntax into formally verifiable chunks. Below are the analyses of these different formal verification systems.

2.1. Coq

The Coq proof assistant is a platform for writing and executing formal proofs. It includes a formal language to write mathematical definitions, theorems/lemmas, and algorithms for the semi-interactive development of machine-checked proofs. It is mostly used for certification and verification of properties of programming languages and the formalization of Mathematics.

2.1.1. Description

Coq is based on a logic framework called Calculus of Inductive Constructions [33]. A proof is constructed and checked by Coq, which allows [3]:

- definition of mathematical and programming objects
- mathematical theorems/lemmas and software specs
- development of formal proofs for these theorems/lemmas with an interactive GUI
- checking of proofs computationally

In order to assist in writing proofs, Coq comes with built in tactics. Tactics are ways to specify how to transform the proof state. Tactics may prove a goal or, more commonly, transform a goal into one or more subgoals (called reduction). A user writes a proof by executing a series of tactics to reduce the Theorem or Lemma until no more goals are needed to be solved.

Coq has been used in a number of verification projects, including a verified C compiler [2] and in a machine checked formal proof of the Four Color Theorem [13].

2.1.2. Examples

We present two examples to demonstrate the use of Coq. First, Listing 1 is an example of basic Conjunction in Propositional Logic. I.e. If A is true and B is true, then A AND B is true. In the proof, the built in coq tactics *intros*, *split*, and *apply* are used to prove the goals of the theorem.

The second example in Listings 2 and 3 is more in-depth and shows the proof of a programmatic algorithm. This example sets up some data types, notation, and support functions and then proves a list membership function correct.

```
(*
Define our Theorem:
  For every proposition A and B, if A is true and B is true, then A AND B is true
*)
Theorem basic_conjunction : forall (A B : Prop),
  A -> B -> A /\ B.

(* Prove it *)
Proof.
  intros.      (* establish assumptions *)
  split.       (* split our goal A /\ B into 2 subgoals *)
  - apply H.   (* apply first hypothesis - A is true *)
  - apply H0.  (* apply second hypothesis - B is true *)
Qed. (* there are now no more (sub)goals to prove *)
```

1 Coq Proof of Basic Conjunction

2.1.3. SVT Applicability

Because of the interactive nature of Coq, its use as a validation and verification tool for SVT is limited. There are tools to translate C and Java code to Coq specifications, but writing the Proofs is still a process that requires a programmer to use Coq Tactics to solve goals and complete the proofs.

The learning curve and time needed to write proofs in Coq is also significantly high. A proof of a simple algorithm such as a Mergesort requires dozens of lines of definitions and specifications before even getting to the actual Proof.

Table 2-1 Coq Pros and Cons

Pros	Cons
Formal verifier	Steep learning curve
Easy to install with good documentation	Requires interaction with programmer
CoqIDE provides a nice GUI for the language	
Coq Tactics aid in building proofs	

2.1.4. Final Thoughts

Coq is a very powerful tool for formal verification of software. It is used extensively in academic and industry research and has been applied to the verification of software systems and used to prove several well-known theorems. However, using Coq to build proofs to verify software is an interactive process that requires a user with deep knowledge of both the Coq language as well as

```
(* Setup all definitions, types, functions needed *)

(* Define a data type fruit *)
Inductive fruit :=
| Apple : fruit
| Banana : fruit
| Orange : fruit
| Pear : fruit
| Mango : fruit
| Peach : fruit
| Kiwi : fruit.

(* Define a list of type fruit *)
Inductive fruit_list :=
| Nil : fruit_list
| Cons : fruit -> fruit_list -> fruit_list.

(* Define what the head and tail of the list are *)
Definition head (l : fruit_list) (default : fruit) : fruit :=
  match l with
  | Nil => default
  | Cons f _ => f
  end.
Definition tail (l : fruit_list) : fruit_list :=
  match l with
  | Nil => Nil
  | Cons _ rest => rest
  end.

(* create some notation short hands for empty list, list with one item, list with head/tail *)
Notation "[ ]" := Nil.
Notation "[ f ]" := (Cons f Nil).
Notation "[ f1 ; f2 ; .. ; fn ]" := (Cons f1 (Cons f2 .. (Cons fn Nil) .. )).


(* define a function that checks if 2 fruits are equal *)
Definition eq_fruit (f1 f2 : fruit) : bool :=
  match (f1, f2) with
  | (Apple, Apple) => true
  | (Banana, Banana) => true
  | (Orange, Orange) => true
  | (Pear, Pear) => true
  | (Mango, Mango) => true
  | (Peach, Peach) => true
  | (Kiwi, Kiwi) => true
  | (_, _) => false
  end.
```

2 Coq List Membership Example Setup

```
(*
   define a function that checks if a fruit is a member of a fruit_list
   we will use Coq to prove this function correct below
*)
Fixpoint is_member (f : fruit) (l : fruit_list) : bool :=
  match l with
  | [] => false
  | Cons h t => eq_fruit f h || is_member f t
  end.


Inductive ListMember : forall (f : fruit) (l : fruit_list), Prop :=
| Member_head : forall f l, ListMember f (Cons f l)
| Member_tail : forall f f' l, ListMember f l -> ListMember f (Cons f' l).

(*
   propose and prove a Lemma that says if the eq_fruit function returns true
   for 2 fruits x and y, then x and y are equal
*)
Lemma eq_fruit_lemma : forall  x y, eq_fruit x y = true -> x = y.
Proof.
  intros x y.
  case x; case y; compute; (reflexivity || intros H; inversion H).
Qed.


(*
   now we are ready to prove is_member function is correct
*)
Theorem is_member_correct : forall f l, is_member f l = true -> ListMember f l.
Proof.
  intros f l.
  induction l.
  - compute.
    intro H; inversion H.
  - simpl.
    case_eq (eq_fruit f f0).
    + intros.
      assert (f = f0).
      apply eq_fruit_lemma.
      apply H.
      rewrite H1.
      apply Member_head.
    + simpl.
      intros.
      apply Member_tail.
      apply IHl.
      apply H0.
Qed.
```

3 Coq List Membership Example Proof

the theoretical background in Formal Methods. Both of these require a steep learning curve. The interactive nature also does not allow Coq to be used in a continuous integration pipeline.

For these reasons, Coq is not recommended for use in SVT.

Table 2-2 Coq Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Excellent	Very Easy	Very Difficult	Fair	Good	Good	Poor

2.2. HOL

The HOL interactive theorem prover is a proof assistant for higher-order logic: a programming environment in which theorems can be proved and proof tools implemented. Built-in decision procedures and theorem provers can automatically establish many simple theorems, though users usually end up having to manually prove the "difficult" theorems themselves. More information on HOL can be found at [4].

2.2.1. Description

HOL is a theorem prover for higher-order logic. Fundamentally it is not strictly designed to be applied to existing software solutions. HOL theorems are proved by defining theorems and then building upon lemmas in order to satisfy the conditions associated with proving the initial theorem. An example involves proving Euclid's theory of prime numbers: that for every prime number there exists a prime number greater than itself. First, you must define what a prime number is, then build upon that definition by proving useful theorems about the divides relation. These relations include theorems about division by 1, division transitivity, and division properties related to addition and multiplication. Induction proofs can be performed to verify that factorial division properties coincide with liberal usage of the rewriter and *metis_tac* [27] to unwind definitions. At that point, it is possible to then associate other lemmas needed to finish a proof. This tree/stack based approach involving lemma definition and cleverly crafting building blocks to prove complex theorems is the basis of how HOL is designed to work. More information on the usability of HOL can be found in the manual here [28].

2.2.2. Examples

HOL utilizes SML as its system modeling language. HOL builds upon an SML language implementation to provide additional verification features and capabilities. An example of a simple factorial related proof is shown in Listing 4. Here we can see principles of induction, usage of the re-writer, and lemma expansion to see how the proof is validated and verified. From this example we can see how HOL could be used to programmatically verify mathematical conditions or properties. It can also be seen via Listing 5 how we might devise a custom data type and continuously prove properties about functions involving that data type.

```
'!m p. 0 < m ==> m divides FACT (m + p)'
  suffices_by metis_tac[LESS_EQ_EXISTS] >>
Induct_on 'p' >| [
  rw[] >> Cases_on 'm' >| [
    fs[],
    rw[FACT, DIVIDES_LMUL, DIVIDES_REFL]
  ],
  rw[FACT, ADD_CLAUSES] >> rw[DIVIDES_RMUL]
]
```

4 Factorial

```
datatype 'a tree = Lf | Nd of ('a tree * 'a * 'a tree);

Definition size_def:
  (size Lf = 0) /\ (size (Nd l _ r) = 1 + size l + size r)
End
```

5 Tree

2.2.3. SVT Applicability

When it comes to validating or verifying already existing software, the HOL system has minimal applicability. Taking abstract lemmas and theorems and delegating them to software that is running on a targeted architecture could create many potential abstraction errors. There is no straightforward way to map software to an HOL theorem engine in general, as you can't truly verify that a given HOL theorem corresponds to a code dataset. The amount of work and research required to generate some sort of software to HOL theorem system would be out of scope for this work.

2.2.4. Final Thoughts

HOL is a well documented and simple to install formal verification engine. However, it is rooted in SML and not conducive to verifying already existing software. Additionally the learning curve

Table 2-3 HOL Pros and cons

Pros	Cons
Formal verifier, so guarantees mathematical correctness	Not language agnostic
Easy to install	Incredibly steep learning curve
Very well documented	Not immediately usable for verifying existing software

Table 2-4 HOL Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Excellent	Easy	Very Difficult	Poor	Very Good	Excellent	Poor

is incredibly steep and few tools were found to directly use HOL. With all of this in mind, HOL is not recommended for inclusion in SVT proper; the work required in order to create the necessary tooling needed to validate existing programs is great.

2.3. Lean

Lean is a theorem prover and programming language that started at Microsoft Research in 2013 [16].

2.3.1. Description

Lean aims to bridge the gap between interactive and automated theorem proving by providing automated tools and methods that support user interaction and the construction of fully-specified proofs. It is based on a logical system known as dependent type theory. Specifically, it is based on the Calculus of Constructions with inductive types [16]. Programming in Lean primarily involves defining types and writing functions. Some of the features of Lean include [19]:

- Type inference
- First-class functions
- Pattern matching
- Type classes
- Extensible syntax
- Hygienic macros
- Dependent types
- Metaprogramming framework
- Multithreading
- Verification

For more information on Lean, see [16, 21].

2.3.2. Examples

(Note: the examples in this section were taken from [21].

As previously mentioned, Lean programming primarily involves defining types and functions. For example, a type that defines the natural numbers might look like the code snippet in Figure 2-1, which defines the type inductively with a constant `zero` and a unary function `succ`. Given this type definition, a function can be written to add two natural numbers together through recursion in the second argument, which is demonstrated in Figure 2-2.

```
inductive nat : Type
| zero : nat
| succ : nat → nat

```

Figure 2-1 Lean type to define the natural numbers

```
def add : nat → nat → nat
| m nat.zero     := m
| m (nat.succ n) := nat.succ (add m n)

```

Figure 2-2 Lean function to add two integers

In addition to types and functions, Lean allows the user to make assertions about objects and then prove those assertions through the use of properties. For example, the notion of an even natural number can be expressed through the property defined in Figure 2-3. Given the definition of `even`, it can be used to check if a given natural number satisfies the property (i.e., is even; as in the first two examples of Figure 2-4) and can also be used to generate compound statements that are also properties (as in the last two examples of Figure 2-4).

```
def even (n : ℕ) : Prop := ∃ m, n = 2 * m

```

Figure 2-3 Lean property that expresses an even natural number

```
> check even 10
> check even 11
> check ∀ n, even n ∨ even (n + 1)
> check ∀ n m, even n → even m → even (n + m)

```

Figure 2-4 Lean example use of even property and generating compound statements

The `even` property defined above can also be used to prove existential statements. For example, in order to prove that the sum of two even numbers is also even, it is possible to utilize a compound statement generated from the `even` property, as shown in Figure 2-5. The user defines what is given and the assumptions that are made (via `take` and `assume`). The command `simp` calls on Lean's built-in simplifier to prove the assertion after the `have` keyword, using the two facts labeled `hk` and `hl`, and the distributivity of multiplication over addition (using another theorem called `mul_add` that is not shown in this section). The simplifier is what is known as a conditional term rewriting system and it rewrites subterms into a specific form until it can no longer rewrite any of the subterms. The use of `simp` in this example demonstrates a path to utilizing automated reasoning within Lean to help generate a proof. The above examples are all simple use cases associated with Lean. For more details and advanced examples, please refer to [21].

```
theorem even_add : ∀ m n, even m → even n → even (n + m) :=
take m n,
assume ⟨k, (hk : m = 2 * k)⟩,
assume ⟨l, (hl : n = 2 * l)⟩,
have n + m = 2 * (k + l),
    by simp [hk, hl, mul_add],
show even (n + m),
    from ⟨_, this⟩

```

Figure 2-5 Lean proof that the sum of two even numbers is also even

2.3.3. SVT Applicability

Lean has a steep learning curve and requires the developer to write the proof for a given piece of code, which is time consuming and possibly error prone. The resulting proof could also result in information loss (i.e., Lean may not be able to represent all features associated with the implementation). Also, the authors of the language admit that it is a research project that is still under heavy development and should be treated "as is" [20]. Finally, it does not seem that many tools exist for Lean in the space of verifying software for other languages, which may limit its use to programs written in Lean.

Table 2-5 Lean Pros and cons

Pros	Cons
Formal verifier, so guarantees mathematical correctness	Not language agnostic
Easy to install	Incredibly steep learning curve
Well documented	Not immediately usable for verifying existing software

2.3.4. Final Thoughts

Although Lean is a powerful programming language and proof assistant, it is not recommended for use by SVT. It has a steep learning curve and it is not well-suited for automated techniques. Additionally, there are other proof assistants that are more mature and would likely be a better fit, if a proof assistant is required.

Table 2-6 Lean Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Excellent	Excellent	Poor	Fair	Good	Good	Poor

2.4. Isabelle/HOL

Isabelle is a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus. Isabelle was originally developed at the University of Cambridge and Technische Universität München, but now includes numerous contributions from institutions and individuals worldwide. More information on Isabelle can be found at [14].

2.4.1. Description

Isabelle/HOL is a theorem prover for higher-order logic. There are additional subsets of Isabelle that implement theorem provers for different logic order, such as first order logic (Isabelle/FOL). Fundamentally it is not strictly designed to be applied to existing software solutions, as its domain language Isar (a superset of SML) is the main interface for creating and writing proofs and software. Isabelle/HOL theorems are proved by defining lemmas which build upon other lemmas in order to achieve a hierarchical theorem definition with an associated proof. Lemmas are derived and then solved by applying either intrinsic knowledge of the proof language and subsystems associated with Isabelle/HOL and Isar, or by using Sledgehammer [15]. Sledgehammer attempts to auto-resolve lemma goals by applying various solvers to find and build associated goal solutions. One such solver is Z3, a system that was explored in-depth as a separate tool. The Automated solving system provided by Sledgehammer is one of the strengths of Isabelle/HOL, as it provides a very easy way to decompose lemma goal steps in an automated manner. Simultaneously, Sledgehammer abstracts away important proof writing logic, and may leave the user crippled in the case where a solution for a given step fails to be found. Excluding the automatic goal resolution system, Isabelle/HOL is very similar to the other formal method solvers and provides lots of additional tooling that can be used to generate code (in Scala/Haskell) and export/import public theorems.

2.4.2. Examples

The following examples showcase some simple functionality of Isabelle/HOL as well as the Isar syntax. Listing 6 shows how division can be implemented and proven, as well as how a property of factorials can be devised. Sledgehammer allows us to automatically solve various stages of the proof, and induction is applied automatically when it is determined to be required.

Onto a more practical example, it is possible to implement element removal from a list as shown in Listing 7. Here a function is defined that takes a singular value a list of values and returns a list. The result is a list that no longer contains any elements matching the provided value. With this definition, it is possible to prove various properties of the function itself and the parameters and return values.

The final example shows how a new data structure can be defined and how functions can be defined to operate on that data structure and then various properties can be proven. In Listing 8 we define a tree type data structure consisting of a root node and two sub nodes which can either

```
theory divides
  imports HOL.Divides Main HOL.Factorial
begin

fun divides :: "nat \<Rightarrow> nat \<Rightarrow> bool"
  where "divides x y = (x mod y = 0)"


value "divides (4::nat) (2::nat)"
value "divides (5::nat) (2::nat)"
value "(5::int) dvd (5::int)"

lemma divides_le: "!m n. divides m n \<Longrightarrow> m \<le> n \<or> (n = 0)"
  apply auto
  by (metis (mono_tags, lifting) mod_by_0 zero_neq_one)

lemma divides_refl: "!m. divides m m"
  apply auto
  done

lemma divides_lmul: "!d a x. divides d a \<Longrightarrow> divides d (x * a)"
  apply auto
  done

lemma divides_rmul: "!d a x. divides d a \<Longrightarrow> divides d (a * x)"
  apply auto
  done


lemma fact_div_self: "!m. 0 < m \<Longrightarrow> divides m (fact m)"
  apply (induct m)
   apply auto


lemma fact_divides: "!m p. 0 < m \<Longrightarrow> divides m (fact (m + p))"
  apply (induct p)
   apply auto


end
```

6 Divides

```
theory remove
  imports Main
begin
fun remove:: "'a \<Rightarrow> 'a list \<Rightarrow> 'a list"
  where
"remove x [] = []" |
"remove x (y#ys) = (if x=y then (remove x ys) else y#(remove x ys))"


value "remove (2::int) [1,2,3]"


lemma "\<not> (List.member (remove e l) e)"
  apply (induct l)
  apply auto
  apply (simp add: member_rec(2))
  by (simp add: member_rec(1))

lemma count_remove: "(length l) = (length (remove e l)) + (count_list l e)"
  apply (induct l)
   apply auto
  done

lemma "(length (remove e l)) = (length l) \<longrightarrow> \<not> (List.member l e)"
  apply (induct l)
   apply auto
  apply (simp add: member_rec(2))
  apply (metis Suc_n_not_le_n count_remove le_add_same_cancel1 zero_le)
   apply (metis Suc_n_not_le_n count_remove le_add_same_cancel1 zero_le)
  by (simp add: member_rec(1))


export_code remove in Scala
export_code remove in Haskell

end
```

7 Remove

```
theory tree
  imports Main
begin

datatype 'a tree= Leaf | Node 'a "'a tree" "'a tree"
definition "myTree= Node (1::int) (Node 2 Leaf Leaf) (Node 3 Leaf Leaf)"

fun numNodes:: "'a tree \<Rightarrow> nat"
where
"numNodes Leaf = 0" |
"numNodes (Node e t1 t2) = 1+(numNodes t1)+(numNodes t2)"

fun subTree:: "'a tree \<Rightarrow> 'a tree \<Rightarrow> bool"
where
"subTree Leaf Leaf = True" |
"subTree t Leaf = False" |
"subTree t (Node e t1 t2) = ((t=(Node e t1 t2)) \<or> (subTree t t1) \<or> (subTree t t2))"

value "numNodes myTree"
value "subTree (Node 2 Leaf Leaf) myTree"

lemma subNum: "(subTree t1 t2) \<longrightarrow> (numNodes t1) \<le> (numNodes t2)"
  apply (induct t2)
  apply auto
  by (metis numNodes.elims subTree.simps(2))

end
```

8 Tree

be leaves or additional trees. Then, we define two functions to operate on this structure, one that counts the number of nodes, and one that returns whether the tree specified is a subtree of the supplied tree. With these functions we can then prove that the number of elements in the subtree will be less than the number of elements in the root tree using induction and sledgehammer.

These examples showcase some of the features of Isabelle/HOL that can be used to derive formally correct definitions of programs. Deriving these definitions and proofs is not a simple process, however, and it requires extensive knowledge of the Isabelle/HOL system and language.

2.4.3. SVT Applicability

By itself, Isabelle/HOL has minimal applicability when it comes to verifying/validating already existing software in a given language. Taking abstract lemmas and theorems and delegating them to software that is running on a targeted architecture could result in many abstraction errors. Verifying that a hand-crafted Isabelle/HOL theorem corresponds to a given software dataset is not trivial. Creating a way to map a code dataset in a specific language to the Isabelle/HOL theorem engine would be a massive undertaking: and even still this solution would still require the user to

Table 2-7 Isabelle-HOL Pros and cons

Pros	Cons
Formal verifier, guaranteed mathematical correctness	Command line/batch style usage is awkward to use
Powerful GUI	Not language agnostic
Very well documented	Steep learning curve, though the GUI and auto-completion/syntax highlighting help greatly
Code generation tools are robust (Haskell/Scala)	Not immediately usable for verifying existing software

Table 2-8 Isabelle/HOL Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Excellent	Easy	Difficult	Fair	Very Good	Excellent	Poor

write their own proofs to guarantee correctness of the parsed code. Direct incorporation of Isabelle/HOL with SVT without some sort of intermediate system is not recommended. That said, there are other tools that exist that can be utilized to convert C source code into Isabelle/HOL statements via AST parsing and translation (via AutoCorres). An analysis of AutoCorres is provided in a separate section.

2.4.4. Final Thoughts

Isabelle/HOL, as a formal verification engine, is powerful and incredibly robust. The tooling associated with, and the language Isar, make it a very useful tool for verifying proofs about systems and functions. Additionally, its inherent capability to generate Haskell/Scala code from written proofs is very powerful. There may be some merit in the derivation of a Haskell -> other language compiler/interpreter, as that would allow a direct method in which to create validated code in other languages. That said, the compiler/interpreter would need to be formally verified as well to ensure consistency between translated code. As an alternative to validating existing software, it might be prudent to consider creating new software from verified systems like Isabelle/HOL.

2.5. Z3

Z3 was initially developed by Microsoft's research team nearly 15 years ago as a bit-precise SMT/SAT solver that included memory and vector features. Since that time, it has developed into one of the premier SMT/SAT solvers in the formal software verification domain. Z3's

```
1 int factorial(int p) {
2     int result = 1;
3
4     for (int i = 1; i < p; i++) {
5         result *= i;
6     }
7
8     return result;
9 }
```

9 Bad Factorial Example

documentation can be found at their github [40] and an in-depth overview of Z3's features can be found here [31].

2.5.1. Description

Z3's primary input are formulas in the SMTLIB2 format, which is a lisp-like representation of SMT statements. It also has bindings for multiple other languages with python, C/C++, and Java being among them. Those interfaces allow the user to utilize language-specific syntax to interact with the backend of Z3. However, Z3's data types do not integrate with the base types of its supported language bindings. This is demonstrated in the examples.

Z3 utilizes a number of different techniques, reference as "tactics" by Z3, and solvers to generate a set of inputs that satisfy user-created SMT/SAT statements. Z3 is flexible in its possible representations of data inputs that should cover any user's program input (Algebraic Data Types, floating point precision, infinite precision of rational numbers, bit-vectors, arrays, etc).

As Z3 is a formal verifier, the user's program must be transcribed into Z3 statements so the tactics and solvers can be applied and an answer may be found. There is no way to directly tie a program's execution or statements to the type of statements Z3 operates on so the user must accurately translate an already written program into Z3 statements.

2.5.2. Examples

Consider the example in Listing 9 of some code written to compute the factorial of a number, written in C++. It has a small mistake on line 4, where by setting the for-loop's exit condition to $i < p$, it excludes the result from being multiplied by p.

If we wanted to use Z3's C++ language bindings to uncover the error in the code, we have to translate the function accurately into Z3 statements. First, we'll define a helper function that we can use to assert formal statements about code in Listing 10.

The *prove* function takes a single expression we want to prove is true using an already-defined Z3 solver. On line 9, we are adding the negation of the expression we want to prove because we want Z3's tactics and solvers to find a set of variable assignments that would satisfy the negation. If Z3

```
1
2  #include "z3++.h"
3
4  #ifndef PROVE_EXMAPLE_H
5  #define PROVE_EXAMPLE_H
6
7  void prove(z3::solver& s, z3::expr p) {
8      s.push();
9      s.add(!p);
10
11     std::cout << "\tConjecture:" << p << std::endl;
12
13     if (s.check() == z3::unsat) {
14         std::cout << "\tproved." << std::endl;
15     }
16     else {
17         std::cout << "\tConjecture failed to prove. Counterexample: " << std::endl;
18         std::cout << s.get_model() << std::endl;
19     }
20     std::cout << "=============" << std::endl;;
21     s.pop();
22 }
23
24 #endif
```

10 Prove Helper

can find those value assignments, then we have a counter-example that disproves the expression p and Z3 can return the counterexample to the user (in this case, we just print it to the console).

Using the *prove* function and Z3's C++ language bindings, we can transform the *factorial* function into the code in Listing 11

Transcription of already-written code into Z3 statements has the result of duplicating variable declarations (lines 10 and 14) so Z3 knows about variables and can define the domain of each (lines 11 and 22).

On lines 25 and 26, we define a integer constant that can assume all integer values in the range $[1, p]$. Then on line 27, we finally posit that for any value that *all_p* can take on, the result should be evenly divisible by that value. However, as this is an incorrect implementation, we should expect to see a failure after the program is run in Listing 12.

The output is in SMTLIB2 format and can take a little bit to understand, but the short version is SMTLIB2 treats everything like a function, including constants. SMTLIB2 statements follow the form `(define-fun <identifier> <params...> <domain> <value>)` and constants can be thought of as functions that take zero arguments over a specified domain. So, we can see in the output that the result value was calculated to be `24` which is is obviously not divisible by 5 (as the `div0` and `mod0` state)

The pre-conditions and post-conditions sections (lines 9-12 and 21-23) are an example of the user ensuring the function is operating on input it deems as valid and produces output within a valid

```cpp
#include "z3++.h"
#include "prove.h"


int factorial(int p) {
    z3::context context;
    z3::solver s(context);

    // pre-conditions
    z3::expr z3p = context.int_const("p");
    s.add(z3p == p);
    prove(s, z3p > 0);

    z3::expr z3result = context.int_const("result");
    int result = 1;

    for (int i = 1; i < p; i++) {
        result *= i;
    }

    // post-conditions
    s.add(z3result == result);
    prove(s, z3result > 0);

    z3::expr all_p = context.int_const("all_p");
    s.add(all_p <= p && all_p > 0);
    prove(s, z3result % all_p == 0);

    return result;
}

int main() {
    std::cout << "Bad Factorial" << std::endl << factorial(6) << std::endl;
}
```

11 Bad Factorial Example

```
1   Bad Factorial
2     Conjecture:(> p 0)
3     proved.
4 ============
5     Conjecture:(> result 0)
6     proved.
7 ============
8     Conjecture:(= (mod result all_p) 0)
9     Conjecture failed to prove. Counterexample:
10 (define-fun all_p () Int
11 5)
12 (define-fun p () Int
13 5)
14 (define-fun result () Int
15 24)
16 (define-fun div0 ((x!0 Int) (x!1 Int)) Int
17 4)
18 (define-fun mod0 ((x!0 Int) (x!1 Int)) Int
19 4)
20 ============
21 24
```

12 Bad Factorial Output

range. This can help find domain issues if, for example, the user called the *factorial* function with a value that would overflow an 32 bit integer. In the above output, we can see that both the pre and post conditions were satisfied. However, if we instead called *factorial* with a parameter of 30, the output would instead show 13.

Which indicates our specified post-condition failed. Now, we can fix the for-loop's domain to properly include the value of p in Listing 14 (lines 15-17): Now that we have fixed the bounds of the for-loop, the resulting program's output would yield the output in Listing 15.

By themselves, the Z3 statements written into the function do nothing – they must be compiled and executed for them to evaluate their correctness. This means that Z3 can only prove the correctness of a program that has been concretely executed (in our case by calling $factorial(5)$). This can lead a user into a false sense of security if care is not taken to create the proper proof. Consider the previous *factorial* example called with a parameter of 6 instead of 5. In the incorrectly written version (Listing 11), the output is shown in Listing 16.

Above, we've proved the incorrect version (Listing 11) is indeed incorrect, but because the code was executed over a case where `prove(s, z3result % all_p == 0)` does not properly catch this error, the user could incorrectly assume they proved their code correct. This is where Z3 starts to show a little bit of a weakness. Because Z3 requires the code to execute for it to determine correctness, users will struggle to implement proofs that rely on inductive algorithm properties.

```
  Bad Factorial
        Conjecture:(> p 0)
        proved.
============
        Conjecture:(> result 0)
        Conjecture failed to prove. Counterexample:
(define-fun p () Int
  30)
(define-fun result () Int
  (- 1241513984))
============
        Conjecture:(= (mod result all_p) 0)
        Conjecture failed to prove. Counterexample:
(define-fun all_p () Int
  3)
(define-fun p () Int
  30)
(define-fun result () Int
  (- 1241513984))
(define-fun div0 ((x!0 Int) (x!1 Int)) Int
  (- 413837995))
(define-fun mod0 ((x!0 Int) (x!1 Int)) Int
  1)
============
-1241513984
```

13 Worse Factorial Output

```cpp
#include "z3++.h"
#include "prove.h"


int factorial(int p) {
    z3::context context;
    z3::solver s(context);

    // pre-conditions
    z3::expr z3p = context.int_const("p");
    s.add(z3p == p);
    prove(s, z3p > 0);

    z3::expr z3result = context.int_const("result");
    int result = 1;

    for (int i = 1; i <= p; i++) {
        result *= i;
    }

    // post-conditions
    s.add(z3result == result);
    prove(s, z3result > 0);

    z3::expr all_p = context.int_const("all_p");
    s.add(all_p <= p && all_p > 0);
    prove(s, z3result % all_p == 0);

    return result;
}

int main() {
    std::cout << "Good Factorial" << std::endl << factorial(5) << std::endl;
}
```

14 Good Factorial Example

```
  Good Factorial
    Conjecture:(> p 0)
    proved.
============
    Conjecture:(> result 0)
    proved.
============
    Conjecture:(= (mod result all_p) 0)
    proved.
============
120
```

15 Good Factorial Output

```
1  Bad Factorial
2    Conjecture:(> p 0)
3    proved.
4 ============
5    Conjecture:(> result 0)
6    proved.
7 ============
8    Conjecture:(= (mod result all_p) 0)
9    proved.
10 ============
11 120
```

16 False Factorial Output

Table 2-9 Z3 Pros and cons

Pros	Cons
High pedigree	Struggles with inductive proofs
Active development	Not language agnostic
Language bindings for all targeted SVT languages	Precise translation of already-written code is necessary
Support for bitvectors, floats, and Abstract Data Types	
Good documentation	

2.5.3. SVT Applicability

Z3 has the bindings necessary for the languages SVT is interested in. However, because it falls into the formal verifier category, it will have limited applications for SVT. Any code will have to be translated to Z3 statements, either STMLIB2 or source language bindings, before Z3's processing tool will be able to make any assertions about it.

One option to use Z3, is to develop or use language specific parsers so SVT can modify the abstract syntax tree to create Z3 statements that are mechanically the same as the source code. This tends to be what many of the automatic software proof tools that use Z3 as a backend do. However, if SVT develops language parsers, maintaining those parsers after SVT's main development period is over will become the primary concern.

Another option is to use LLVM's intermediate representation [23] as a language-agnostic, stable interface for SVT to translate a program into Z3 statements. This would reduce the amount of maintenance to keep SVT relevant for longer and expand SVT's applicability to the myriad of languages that can be transformed to LLVM's intermediate representation. However, mapping any incorrect logic statements back to the original source code might prove to be difficult.

Table 2-10 Z3 Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Excellent	Very Easy	Neutral	Fair	Excellent	Excellent	Excellent

2.5.4. Final Thoughts

Due to Z3's pedigree in the formal verification domain, it is hard to ignore how trusted this software is when looking for SVT's applicable tools. The most promising use for Z3 in SVT would be to develop a translation of LLVM's IR into Z3 statements with an additional annotation based comment syntax. However, the scale of developing an LLVM IR parser and translator might be a task outside the scope of SVT's effort unless a significant amount of effort is dedicated to it. It is recommended that SVT considers Z3's applicability to SVT's goals.

2.6. TLA+

TLA+ is a high-level language for modeling programs and systems: especially concurrent and distributed ones. It's based on the idea that the best way to describe things precisely is with simple mathematics. TLA+ and its tools are useful for eliminating fundamental design errors, which are hard to find and expensive to correct in code. More information on TLA+ can be found here [22].

2.6.1. Description

TLA+ is a language for modeling software above the code level and hardware above the circuit level. It has an IDE used to create models and then check them. TLA+ is based on mathematics, and its syntax does not resemble a programming language. PlusCal, a different syntax for the TLA+ model checker, can be used to create TLA+ verified models and is more program like in nature. TLA+ has a heavy focus on formally verifying the correctness of multi-threaded models and applications. TLA+ translates a single model file into verifiable components. Each TLA+ file contains a set of variables and a single algorithm. The variables define constructs, including sets, values, and tuples. The algorithms section defines the specification functions, algorithms, and what the specification is actually trying to accomplish. Together, these components are applied to a model, which defines parameters, the behavior to check (one of deadlock, invariant, or temporal properties), how to run the model, and the behavioral specification. The model setup defines what is being checked in terms of the specification.

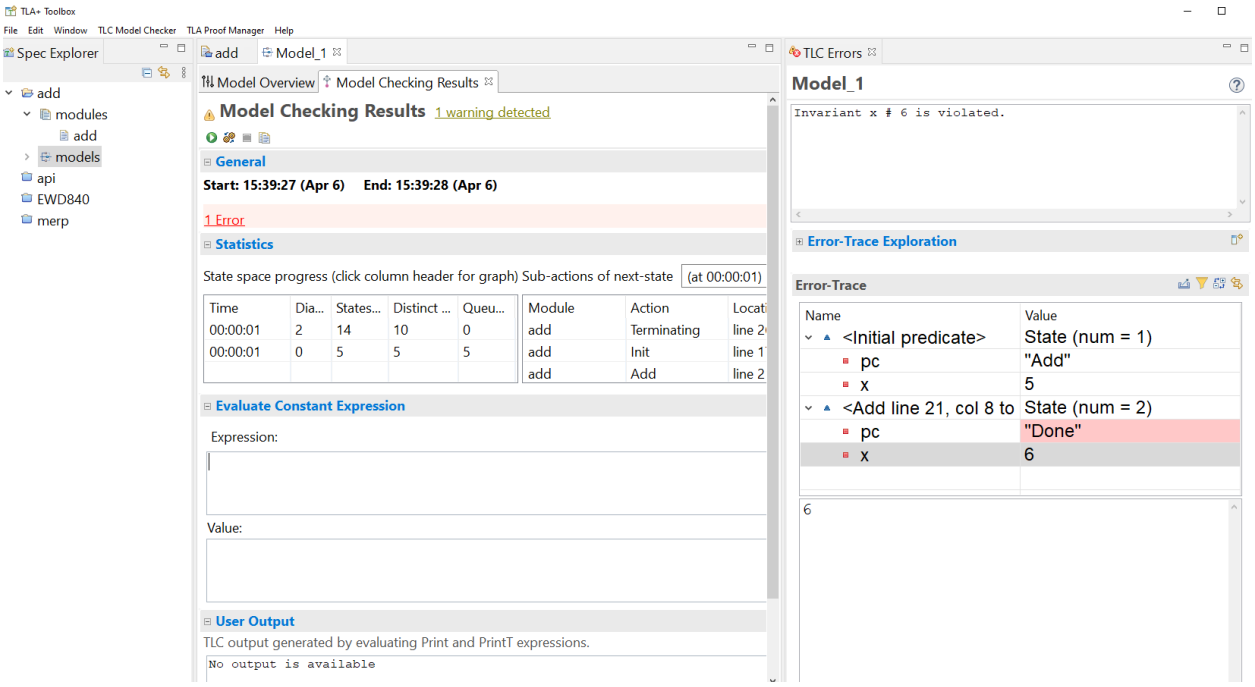
2.6.2. Examples

The first example in Listing 17 showcases a simple addition function that takes an input, adds one, and then returns. This example is fairly contrived, but serves as a good entry point to the PlusCal

```
1  ---------------------------- MODULE add ----------------------------
2
3  EXTENDS Integers
4
5  (* --algorithm example
6  variables x \in 1..5
7  begin
8   Add:
9     x := x + 1;
10 end algorithm; *)
11
12 \* BEGIN TRANSLATION (chksum(pcal) = "1fda2b85" /\ chksum(tla) = "864bc52f")
13 VARIABLES x, pc
14
15 vars == << x, pc >>
16
17 Init == (* Global variables *)
18         /\ x \in 1..5
19         /\ pc = "Add"
20
21 Add == /\ pc = "Add"
22        /\ x' = x + 1
23        /\ pc' = "Done"
24
25 (* Allow infinite stuttering to prevent deadlock on termination. *)
26 Terminating == pc = "Done" /\ UNCHANGED vars
27
28 Next == Add
29            \/ Terminating
30
31 Spec == Init /\ [][Next]_vars
32
33 Termination == <>(pc = "Done")
34
35 \* END TRANSLATION
36 ====
```

17 Add Example

Figure 2-6 TLA+: Model of Add Example



syntax, the TLA+ translation, as well as how to use the model checker. In Figure 2-6, an example model is run with the invariant of $x \# 6$, which of course fails when run since $x == 5 + 1 = 6$.

The second example, Listing 18, showcases a simple multi-threaded http GET/PUT algorithm that is querying an API endpoint for some data collection information, getting a single data item, and then potentially modifying it. The issue is that this API endpoint has a set rate limit that it can support: too many queries within a certain time-frame will result in an error. *get_collection* makes an arbitrary number of calls, while *get_put* will make either 1 or 2 calls. *reset_limit* will modify the total number of API calls available at specific intervals, and the *make_calls* macro will make calls until the limit is reached after which it will wait for *reset_limit* to run. With all of these processes combined in parallel, it showcases a potential implementation of the communication logic necessary to work with an available API. The actual business logic required to complete the transactions is not a part of the example and is not what TLA+ is interested in proving. It is possible to extend this example to model more complex use cases where the clients are distributed and require shared knowledge of how many calls are available for the API.

2.6.3. SVT Applicability

On its own, TLA+ isn't directly usable at verifying already existing software. TLA+ could be used to design models around existing software in an attempt to retroactively check for bugs, but there is currently no automation available for performing that task. TLA+ is a much more approachable formal verification system than the other systems analyzed. It has an amazing tutorial, and while its mathematical syntax isn't strictly mappable to existing programming languages, this syntax is less verbose and abstract than other formal language systems. In fact,

```
1  ---- MODULE api ----
2  EXTENDS Integers, TLC
3  (* --algorithm api
4  variables made_calls = 0, max_calls \in 5..10;
5
6  macro make_calls(n)
7  begin
8    await made_calls <= max_calls - n;
9    made_calls := made_calls + n;
10   assert made_calls <= max_calls;
11 end macro;
12
13 process reset_limit = -1
14 begin
15   Reset:
16     while TRUE do
17       made_calls := 0;
18     end while
19 end process
20
21 process get_collection = 0
22 begin
23   Request:
24     make_calls(1);
25     either goto Request
26     or skip
27     end either;
28 end process;
29
30 process get_put \in 1..3
31 begin
32   Call:
33     with c \in {1, 2} do
34       make_calls(c)
35     end with;
36 end process;
37
38 end algorithm; *)
```

18 Api Example

```
\* BEGIN TRANSLATION (chksum(pcal) = "7931a8d8" /\ chksum(tla) = "f1891ac2")
VARIABLES made_calls, max_calls, pc

vars == << made_calls, max_calls, pc >>

ProcSet == {0} \cup (1..3)

Init == (* Global variables *)
        /\ made_calls = 0
        /\ max_calls \in 5..10
        /\ pc = [self \in ProcSet |-> CASE self = 0 -> "Request"
                                        [] self \in 1..3 -> "Call"]

Request == /\ pc[0] = "Request"
           /\ made_calls' = made_calls + 1
           /\ Assert(made_calls' <= max_calls,
                     "Failure of assertion at line 9, column 3 of macro called at line 15, column 5.")
           /\ \/ /\ pc' = [pc EXCEPT ![0] = "Request"]
              \/ /\ TRUE
                 /\ pc' = [pc EXCEPT ![0] = "Done"]
           /\ UNCHANGED max_calls

get_collection == Request

Call(self) == /\ pc[self] = "Call"
              /\ \E c \in {1, 2}:
                   /\ made_calls' = made_calls + c
                   /\ Assert(made_calls' <= max_calls,
                             "Failure of assertion at line 9, column 3 of macro called at line 25,
                                 column 7.")
              /\ pc' = [pc EXCEPT ![self] = "Done"]
              /\ UNCHANGED max_calls

get_put(self) == Call(self)

(* Allow infinite stuttering to prevent deadlock on termination. *)
Terminating == /\ \A self \in ProcSet: pc[self] = "Done"
               /\ UNCHANGED vars

Next == get_collection
        \/ (\E self \in 1..3: get_put(self))
        \/ Terminating

Spec == Init /\ [][Next]_vars

Termination == <>(\A self \in ProcSet: pc[self] = "Done")

\* END TRANSLATION
====
```

19 Api Example

Table 2-11 TLA+ Pros and Cons

Pros	Cons
Formally verifies algorithms using mathematical notations	Domain language only, no direct program mapping
Includes a robust and well documented IDE	Requires adjustment to learn the syntax and how to use
Very simple to install; runs with java	Possibly less rigorous/powerful compared to other formal verifiers.
Designed specifically to work with highly parallel, multithreaded algorithms	
Tutorial is very through	
Has multiple industry applications	

TLA+ seems to have been designed with a focus on usability, as it includes a separate language called PlusCal that is more imperative in nature which can be compiled into TLA+ specifications. TLA+ has direct links to studies and industrial use cases where it has been shown to reduce bugs in deployed systems. Currently the only downside is that the mapping between TLA+ models and the actual code is not one to one. While there is a Python module called PlusPy that generates python code from TLA+ models, it is out of date and only works for a subset of TLA+ models. If code could either be generated from TLA+ models, or code could be converted into TLA+ models then TLA+ would be directly applicable.

2.6.4. Final Thoughts

Of the formal verification languages and toolkits, TLA+ is one of the most usable and practical. The simplified semantics of PlusCal, combined with the IDE and excellent tutorial make it an amazing entry point into the formal verification ecosystem. However, while it may be easier to use than other formal verification systems, do not take that to imply that it is not powerful; TLA+ is highly capable when it comes to formally validating concurrent algorithms.

While TLA+ may be one of the more approachable formal verification systems, it still falls victim to the same issues that plague the other formal verifiers. These issues are namely related to the fact that TLA+ models don't map directly to real code, and that there is still a high learning curve required to utilize the tool correctly.

Utilizing TLA+ directly in SVT will be difficult: this would essentially require writing an annotation system that transforms into TLA+ algorithms, or the creation of a parser that could automatically convert functions into TLA+ algorithms. Either implementation would require extensive amounts of work to complete. However, the approachability of TLA+ is not to be understated: if a specific formal verification tool needed to be chosen and utilized this would be a great option.

Table 2-12 AutoCorres Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Excellent	Easy	Difficult	Fair	Good	Excellent	Poor

2.7. AutoCorres

AutoCorres is a tool that assists reasoning about C programs in Isabelle/HOL. In particular, it uses Norrish's C-to-Isabelle parser to parse C into Isabelle, and then abstracts the result to produce a result that is (hopefully) more pleasant to reason about. AutoCorres is a tool that can be used to formally validate C programs by directly converting C syntax into Isabelle/HOL statements that can then be formally proven using theorem solvers. More information on AutoCorres can be found here [8].

2.7.1. Description

Autocorres is a C to Isabelle/HOL compiler/parser: it converts valid C programs into Isabelle/HOL statements which can then be reasoned about and proven directly correct by writing Isabelle/HOL theorems using the Isabelle/HOL GUI. By itself, Autocorres does little to formally validate or verify existing C software, it is simply a means to convert C software into Isabelle/HOL statements to make the verification process much more straightforward. Autocorres provides a one-to-one mapping between C software and Isabelle/HOL theorems, greatly enhancing the ability to formally verify C programs. Autocorres has a few limitations due in part to the architectural requirements related to C programs:

- 3 architectural platforms are supported: ARM, X64, and RISCV64.
- Heap/Stack memory abstractions and global variables
- Word size and typing rules
- Function scope (function scope callee depth when compiling the translation)

Due to the architectural limitations, Autocorres generated statements may not be validated on specific architectures that are not supported. Additionally, the software itself is assuming that the C compilation into SIMPL by the C-to-Isabelle parser is correct (this bypasses any issues raised by different assemblies created by different compilers, like intel, gcc, and clang). Overall Autocorres provides an interesting method in which various C programs can be proven correct in a logical sense by directly compiling C files into various Isabelle/HOL contexts.

2.7.2. Examples

Listings 20 and 21 showcase the conversion of a C function that computes the maximum value of two integers into an Isabelle/HOL file that is then loaded as part of a proof showcasing that the max function operates correctly. Autocorres is able to take the C input file and convert it into a series of Isabelle/HOL statements. In this case the max function is easily proven correct by using reflection with respect to the Isabelle/HOL built-in max function.

```c
int max(const int a, const int b) {
  if (a > b) {
    return a;
  }
  return b;
}
```

20 Max Example

```
(*
 * Copyright 2020, Data61, CSIRO (ABN 41 687 119 230)
 *
 * SPDX-License-Identifier: BSD-2-Clause
 *)

theory Max
imports
  "AutoCorres.AutoCorres"
begin

external_file "hand_written/max.c"

(* Parse the input file. *)
install_C_file "hand_written/max.c"

(* Abstract the input file. *)
autocorres [ ts_force pure = max ] "hand_written/max.c"

(* Generated theorems and proofs. *)
thm simple.max'_def simple.max'_ac_corres

context simple begin

(* Show generated "max'" function matches in-built "max". *)
lemma "max' a b = max a b"
  unfolding max'_def max_def
  by (rule refl)

end

end
```

21 Max Example Theorem

```c
int gcd(int a, int b) {
  int c;
  while (a != 0) {
      c = a;
      a = b % a;
      b = c;
  }
  return b;
}
```

22 Gcd Example

In another case, Listings 22 and 23 show the conversion of a C function that computes the greatest common divisor of two values. In this case, the gcd function isn't easily proven via reflection. What must be done instead is a proof through induction against the built-in gcd function. Even in this case we can see the Isabelle/HOL knowledge required in order to prove even this simple function correct. The syntactic translation of the C code to Isabelle/HOL is incredibly powerful and helpful, but it still requires a high level knowledge of proving systems with Isabelle/HOL.

2.7.3. SVT Applicability

Autocorres is an interesting tool that allows users to prove C software correct. However, C software can only be proven correct after an extensive process involving the manual transcribing and definition of Isabelle/HOL theorems. Autocorres essential removes the abstract process of writing theorems against software by directly translating C code into a provable syntax. That said, the entire process is still incredibly manual, requires extensive domain knowledge of the Isabelle/HOL proof system, and is not easy to automate. Direct incorporation of Autocorres with SVT without some soft of method in which to automate the creation of the Isabelle/HOL theorem definitions is not recommended. Writing Isabelle/HOL theorems and proofs is still too far above the skill ceiling of most developers. That said, if it were possible to describe the goal of a function/application and then generate the Isabelle/HOL proofs from that definition there could be some potential for the simplification of verification from the developer's standpoint. It may be worth looking into researching the automatic creation of Isabelle/HOL proofs/theorems from macros/formatted comments prepended before C software that is formally verified.

2.7.4. Final Thoughts

AutoCorres is a very powerful tool that can be used to verify the correctness of C programs. That said, it still requires an extensive knowledge of Isabelle/HOL in order to utilize fully and correctly. For C system programs that truly require complete formal correctness this tool would be of great help. For standard users, however, the complexity of the tool itself and the requirement of learning Isabelle/HOL makes it difficult to recommend.

```
(*
 * Copyright 2020, Data61, CSIRO (ABN 41 687 119 230)
 * SPDX-License-Identifier: BSD-2-Clause
 *)
(* Perform imports similarly to the Max theorem proof *)
context simple begin

lemma gcd_wp [wp]:
     "\<lbrace> P (gcd a b) \<rbrace> gcd' a b \<lbrace> P \<rbrace>!"
   (* Unfold definition of "gcd'". *)
   apply (unfold gcd'_def)

   (* Annoate the loop with an invariant and measure. *)
   apply (subst whileLoop_add_inv [where
       I="\<lambda>(a', b') s. gcd a b = gcd a' b' \<and> P (gcd a b) s"
      and M="\<lambda>((a', b'), s). a'"])

   (* Solve using weakest-precondition. *)
   apply (wp; clarsimp)
    apply (metis gcd.commute gcd_red_nat)
   using gt_or_eq_0 by fastforce

lemma monad_to_gets:
     "\<lbrakk> \<And>P. \<lbrace> P \<rbrace> f \<lbrace> \<lambda>r s. P s \<and> r = v s \<rbrace>!;
          empty_fail f \<rbrakk> \<Longrightarrow> f = gets v"
   apply atomize
   apply (monad_eq simp: validNF_def valid_def no_fail_def empty_fail_def)
   apply (rule conjI)
    apply clarsimp
    apply (drule_tac x="\<lambda>s'. s = s'" in spec)
    apply clarsimp
    apply force
   apply clarsimp
   apply (drule_tac x="\<lambda>s'. s' = t" in spec)
   apply clarsimp
   apply force
   done

lemma gcd_to_return [simp]:
     "gcd' a b = return (gcd a b)"
   apply (subst monad_to_gets [where v="\<lambda>_. gcd a b"])
     apply (wp gcd_wp)
     apply simp
    apply (clarsimp simp: gcd'_def)
    apply (rule empty_fail_bind)
     apply (rule empty_fail_whileLoop)
     apply (clarsimp simp: split_def)
    apply (clarsimp simp: split_def)
   apply (clarsimp simp: split_def)
   done
end
```

23 Gcd Example Theorem

Table 2-13 AutoCorres Pros and cons

Pros	Cons
Formally verifies C code, which is incredibly powerful and useful	Only works with a C programming language
Directly converts C code into Isabelle/HOL Isar statements	Still requires proofs to be written in Isabelle/HOL
Used to verify the SEL4 kernel, so it's well tested	Only works on select architectures
	Very difficult to install, lots of third party packages
	Not the best documentation

Table 2-14 AutoCorres Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Poor	Difficult	Difficult	Poor	Fair	Excellent	Poor

This tool is an interesting case study, however. Writing a parser to convert existing code into a formal verification language/functional may lead to interesting solutions. If there were a method to more easily prove the correctness of the translated functions this tool could be very useful.

2.8. Tamarin

The Tamarin prover is a security protocol verification tool that supports both falsification and unbounded verification in the symbolic model. Security protocols are specified as multiset rewriting systems and analyzed with respect to (temporal) first-order properties and a message theory that models Diffie-Hellman exponentiation and XOR, combined with a user-defined rewriting theory that has the Finite Variant Property, which includes subterm-convergent theories. More in-depth tutorials and information on Tamarin can be found at [7].

2.8.1. Description

The Tamarin prover is a powerful tool for the symbolic modeling and analysis of security protocols. It takes as input a security protocol model, specifying the actions taken by agents running the protocol in different roles (e.g., the protocol initiator, the responder, and the trusted key server), a specification of the adversary, and a specification of the protocol's desired properties. Utilizing these inputs, Tamarin can then automatically construct a proof that the protocol fulfils its specified properties. From the users perspective, the difficulty in proving their protocol comes in devising the correct models and functions using Tamarin's expressive language based on multiset rewriting rules. These rules represent the knowledge of the adversary, messages on the network, information about the system, and the protocol's state. Tamarin, similar to

Isabelle-HOL and other formal verification systems, provides a fully automated solver mode that uses heuristics to guide the proof generation. In the case where automated proving fails, there is also an interactive mode. Tamarin has been used to model and validate multiple protocols from various domains including ARPKI and TLS.

Writing a theory in Tamarin is a multistep process with the following steps derived in order:

- Define cryptographic primitives: incorporate functions that satisfy foundational properties IE hashing, asymmetric-encryption
- Declare multiset rewriting rules that model the protocol: these rules operate on system state represented as multisets of facts.
- Write the properties to be proven (lemmas) that satisfy desired security properties.

2.8.2. Examples

In Listing 24 the cryptographic primitives are defined via the inclusion of the *builtins* identifier. The inclusion of the specified *builtins* provides hashing, asymmetric encryption and decryption algorithms, and the definition of a private key. Following this are a set of *rules* which are a multiset of facts which store system state given by their arguments. The first three rules model the private key infrastructure of the system. The first rule models the registration of a public key, *ltk*. The premise states that a fresh name is generated, *ltk*, which is the new private key, and generates a public name *A* for the agent owning the key-pair. Upon generation of this key, the conclusion then associates the Agent with the private key and also associates the Agent with its public key *pk(ltk)* Adversaries are allowed to retrieve the key with the following rule which reads a public key database entry and then concludes with sending the public key to the network. The next rule models the compromise of private keys by reading a private key database entry, using an *action fact* to determine which agents have been compromised in a trace, and then sending the private key to the adversary. With the infrastructure model defined, it is now possible to model the protocol. The first rule models the client sending a message, the second models receiving a response, and the third models the server receiving the message and responding.

Listings 25 models the security properties of the modeled rules and cryptographic primitives. These properties are denoted by lemmas, and the first of these lemmas is on the secrecy of the session key from the client's point of view. The lemma *Client_session_key_secrecy* says that it cannot be that a client has set up a session key *k* with a server *S* and the adversary learned k unless the adversary performed a long-term key reveal on the server *S*. The second lemma *Client_auth* specifies client authentication, which states that for all session keys *k* that the clients have setup with a server *S*, there must be a server that has answered the request or the adversary has previously performed a long-term key reveal on *S*. These lemmas operate upon the rules defined previously, and utilize the state properties and trace information generated when executing the rules as modeled. The next lemma strengthens the authentication property to include injective authentication by utilizing the freshness of the data as a conjunctive to prove a uniqueness claim on the session keys. The final lemma ensures that the model can run to completion; this assists in proving that the previous lemmas aren't holding simply because the models are not executable

```
theory FirstExample
begin

builtins: hashing, asymmetric-encryption

// Registering a public key
rule Register_pk:
    [ Fr(~ltk) ]
  -->
    [ !Ltk($A, ~ltk), !Pk($A, pk(~ltk)) ]

rule Get_pk:
    [ !Pk(A, pubkey) ]
  -->
    [ Out(pubkey) ]

rule Reveal_ltk:
    [ !Ltk(A, ltk) ]
  --[ LtkReveal(A) ]->
    [ Out(ltk) ]

// Start a new thread executing the client role, choosing the server
// non-deterministically.
rule Client_1:
    [ Fr(~k)           // choose fresh key
    , !Pk($S, pkS)     // lookup public-key of server
    ]
  -->
    [ Client_1( $S, ~k )     // Store server and key for next step of thread
    , Out( aenc(~k, pkS) )   // Send the encrypted session key to the server
    ]

rule Client_2:
    [ Client_1(S, k)     // Retrieve server and session key from previous step
    , In( h(k) )         // Receive hashed session key from network
    ]
  --[ SessKeyC( S, k ) ]-> // State that the session key 'k'
    []                     // was setup with server 'S'

// A server thread answering in one-step to a session-key setup request from
// some client.
rule Serv_1:
    [ !Ltk($S, ~ltkS)                              // lookup the private-key
    , In( request )                                // receive a request
    ]
  --[ AnswerRequest($S, adec(request, ~ltkS)) ]-> // Explanation below
    [ Out( h(adec(request, ~ltkS)) ) ]             // Return the hash of the
                                                   // decrypted request.
```

24 Tamarin Example Protocol Rules

```
lemma Client_session_key_secrecy:
  " /* It cannot be that a  */
    not(
      Ex S k #i #j.
        /* client has set up a session key 'k' with a server'S' */
        SessKeyC(S, k) @ #i
        /* and the adversary knows 'k' */
      & K(k) @ #j
        /* without having performed a long-term key reveal on 'S'. */
      & not(Ex #r. LtkReveal(S) @ r)
    )
  "

lemma Client_auth:
  " /* For all session keys 'k' setup by clients with a server 'S' */
    ( All S k #i.  SessKeyC(S, k) @ #i
       ==>
         /* there is a server that answered the request */
       ( (Ex #a. AnswerRequest(S, k) @ a)
         /* or the adversary performed a long-term key reveal on 'S'
            before the key was setup. */
       | (Ex #r. LtkReveal(S) @ r & r < i)
       )
    )
  "

lemma Client_auth_injective:
  " /* For all session keys 'k' setup by clients with a server 'S' */
    ( All S k #i.  SessKeyC(S, k) @ #i
       ==>
         /* there is a server that answered the request */
       ( (Ex #a. AnswerRequest(S, k) @ a
            /* and there is no other client that had the same request */
           & (All #j. SessKeyC(S, k) @ #j ==> #i = #j)
       )
         /* or the adversary performed a long-term key reveal on 'S'
            before the key was setup. */
       | (Ex #r. LtkReveal(S) @ r & r < i)
       )
    )
  "

lemma Client_session_key_honest_setup:
  exists-trace
  " Ex S k #i.
        SessKeyC(S, k) @ #i
      & not(Ex #r. LtkReveal(S) @ r)
  "

end
```

25 Tamarin Example Protocol Lemmas

Table 2-15 Tamarin Pros and cons

Pros	Cons
Formal protocol verifier, very robust	Not language agnostic
Easy to use and install	Difficult to learn
Very well documented	Not useful for verifying software
Excellent way of modeling attack vectors on security protocols	

Table 2-16 Tamarin Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Excellent	Very Easy	Difficult	Poor	Very Good	Excellent	Poor

and haven't entered a dead end state. Using the *exists-trace* keyword, we can show that the trace on the *SessKeyC* was executed when the server hasn't been compromised.

From this point, it is possible to prove the protocol theory from the command line, or utilize the graphical user interface to analyze individual components of the theory in a stepwise manner.

2.8.3. SVT Applicability

Tamarin is an interesting case study, but ultimately it has little use for inclusion with SVT. At its core, Tamarin is a protocol verifier, not a software verifier. It takes descriptions about security protocol models and transforms those into provable statements that verify the conditions of that protocol. This isn't amenable to the problem of verifying existing software.

2.8.4. Final Thoughts

Tamarin is an interesting case study for verifying algorithmic protocol correctness and vulnerabilities. However, it is not useful for validating software correctness; rather Tamarin is more rooted in the algorithmic side of protocol robustness, correctness, and interoperability. For those interested in validating protocol correctness, Tamarin is a very powerful and optimal choice. It is possible that the fundamental building blocks of Tamarin could be utilized to create a generic algorithm verifier. Something like this would be more akin to the other types of formal verifiers we have already analyzed at this point.

```c
#include <stddef.h>

unsigned sumarray(unsigned a[], int n) {
    int i; unsigned s;
    i = 0;
    s = 0;
    while (i < n) {
        s += a[i];
        i++;
    }
    return s;
}

unsigned four[4] = {1,2,3,4};

int main(void) {
    unsigned int s;
    s = sumarray(four, 4);
    return (int)s;
}
```

26 C program for Sum Array

2.9. Verifiable C

Verifiable C is a language and logic system that assists in building Coq specifications to prove the correctness of C programs. It is based on a version of Hoare logic called Higher-Order Impredictive Concurrent Separation Logic. This logic was developed because it was determined that Hoare logic was not very good at verifying programs with pointers, concurrent programs, or higher-order object-oriented patterns. Verifiable C is part of the VST [2].

2.9.1. Description

The clightgen tool provided in the VST translates C code into ASTs, which are expressed in the Coq formal proof language. This AST output can then be used by a programmer to write proofs to formally prove the C programs. In general, it is not necessary to know the details of the generated AST [2]. For this reason, and for brevity, we have not included the generated code in this paper. Instead, the example below, taken from [1] shows how a programmer would set up and prove a C program using Coq and the AST.

2.9.2. Examples

In this section, we will show how Verifiable-C and the VST are used to write Coq proofs for a C program that sums the values in an array. Listing 26 is the C code we would like to prove correct.

```
1   Require Import VST.floyd.proofauto. (* Import the Verifiable C system *)
2   Require Import VST.progs.sumarray. (* Import the AST of this C program *)
3   Instance CompSpecs : compspecs. make_compspecs prog. Defined.
4   (* Vprog contains a list of global variable type-specifications *)
5   Definition Vprog : varspecs.  mk_varspecs prog. Defined.
6   Definition sum_Z : list Z -> Z := fold_right Z.add 0.
7
8   (* Define a theorem about the sequence definition and prove it *)
9   Lemma sum_Z_app : forall a b, sum_Z (a++b) = sum_Z a + sum_Z b.
10  Proof.
11    intros.
12    induction a; simpl; lia.
13  Qed.
```

27 Functional Model of Adding a Sequence

After running the Verifiable-C tool clightgen on the above C code, a new file containing all of the Coq inductive data structures describing the syntax trees of the sumarray program is produced. To prove the correctness of sumarray, the first step is to use Coq to write a functional model of adding a sequence, then an API specification of adding up an array in the C programming language. This code is shown in Listings 27 and 28.

Now we are ready to write our proofs. To prove the whole program correct:

- Collect the function and API specifications together into Gprog
- Prove that each function satisfies its API
- Put everything together and prove the entire program

Listing 29 is a proof of the sumarry function. This Lemma, body_sumarray, is saying, *In the context of Gprog and Vprog, the function f_sumarray satisfies the specification sumarray_spec.*

Finally, proofs of the main function and for the entire C program are shown in Listing 30.

2.9.3. SVT Applicability

Verifiable-C and the Verified Software Toolchain are a set of tools to assist in writing Coq proofs to verify C programs. Like Coq, the learning curve is quite high, as the user must be knowledgeable in the Coq language, as well as the mathematical foundations of Formal Methods. Verifiable-C is also only applicable to programs written in the C language, which limits its use in SVT.

2.9.4. Final Thoughts

While Verifiable-C includes some nice tools to assist in proving C programs correct using the Coq proof assistant, the development of specifications and the actual proofs is still a manual process that requires the user to be well-versed in Coq. Like Coq, Verifiable-C is not recommended for use in SVT.

```
Definition sumarray_spec : ident * funspec :=
DECLARE _sumarray
  WITH a: val, sh : share, contents : list Z, size: Z
  PRE [ (tptr tuint), tint ]
    PROP(readable_share sh;
         0 <= size <= Int.max_signed;
         Forall (fun x => 0 <= x <= Int.max_unsigned) contents)
    PARAMS(a; Vint (Int.repr size))
    SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)
  POST [ tuint ]
    PROP()
    RETURN(Vint (Int.repr (sum_Z contents)))
    SEP(data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a).

Definition main_spec :=
DECLARE _main
  WITH gv : globals
  PRE  [] main_pre prog tt gv
  POST [ tint ]
    PROP()
    LOCAL (temp ret_temp (Vint (Int.repr (1+2+3+4))))
    SEP(TT).

(* Packaging the API spec all together. *)
Definition Gprog : funspecs :=
        ltac:(with_library prog [sumarray_spec; main_spec]).
```

28 API Specification for the C Program

Table 2-17 Verifiable C Pros and cons

Pros	Cons
Formal verifier, guaranteed mathematical correctness	Only applicable to C programs Steep learning curve Not immediately usable for verifying existing software

Table 2-18 Verifiable C Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Good	Neutral	Difficult	Fair	Good	Excellent	Poor

```
Lemma body_sumarray: semax_body Vprog Gprog f_sumarray sumarray_spec.
Proof.
start_function.
(* the [forward] tactic does symbolic execution of a single line in the function body *)
forward.  (* i = 0; *)
forward.  (* s = 0; *)
(* use [forward_while] to execute the function's while loop .*)
(* [forward_while] takes an assertion – in this case an existential (EX) *)
forward_while
 (EX i: Z,
   PROP  (0 <= i <= size)
   LOCAL (temp _a a;
          temp _i (Vint (Int.repr i));
          temp _n (Vint (Int.repr size));
          temp _s (Vint (Int.repr (sum_Z (sublist 0 i contents)))))
   SEP   (data_at sh (tarray tuint size) (map Vint (map Int.repr contents)) a)).
(* [forward_while] leaves four subgoals to solve *)
(* subgoal 1 *)
Exists 0. (* instantiate EX with 0 *)
entailer!. (* solve 0 <= n *)
(* subgoal 2 *)
entailer!. (* type checking on the while loop, i.e. execution is computationally possible *)
(* subgoal 3 – body of the while loop *)
assert_PROP (Zlength contents = size). {
  entailer!. do 2 rewrite Zlength_map. reflexivity.
}
(* execute the loop body *)
forward. (* x = a[i] *)
forward. (* s += x; *)
forward. (* i++; *)
 (* Reached the end of the loop body – prove that the _current precondition_  entails the loop
      invariant. *)
 Exists (i+1).
 entailer!. simpl.
 f_equal.
 rewrite (sublist_split 0 i (i+1)) by lia.
 rewrite sum_Z_app. rewrite (sublist_one i) by lia.
 autorewrite with sublist. normalize.
 simpl. rewrite Z.add_0_r. reflexivity.
(* sbugoal 4 – After the loop *)
forward.  (* return s; *)
 (* Prove that the postcondition of the function body
    entails the postcondition required by the function specification. *)
entailer!.
autorewrite with sublist in *.
autorewrite with sublist.
reflexivity.
Qed.
```

29 Proof of sumarray Function

```
1   Definition four_contents := [1; 2; 3; 4].
2
3   Lemma body_main:  semax_body Vprog Gprog f_main main_spec.
4   Proof.
5   start_function.
6   forward_call (*  s = sumarray(four,4); *)
7     (gv _four, Ews,four_contents,4).
8    repeat constructor; computable.
9   forward. (* return s; *)
10  Qed.
11
12  Lemma prog_correct:
13  semax_prog prog tt Vprog Gprog.
14  Proof.
15    prove_semax_prog.
16    semax_func_cons body_sumarray.
17    semax_func_cons body_main.
18  Qed.
```

30 Proof of main function

3. STATIC ANALYSIS

Much less rigorous than formal verification, static analysis involves the parsing of existing code and applying various detection strategies in an attempt to find errors with software. Whereas formal verification concerns itself with the actual correctness of a given algorithm, static analysis is only concerned with whether or not a given algorithm will produce errors. These types of errors can range from integer overflows and divide by zero errors to multi-threading conflicts and memory boundedness issues. While static analysis tools don't concern themselves with algorithmic correctness, they are still very useful at finding issues with software. Many static analysis tools exist: some are standalone pieces of software, while others are integrated with existing solutions, such as compilers or IDEs. While many of these tools find similar issues within different programming languages, their methods for detection and implementation techniques differ. It is these differences that make each static analysis tool unique in issue detection and usability. The following sections enumerate the various static analysis tools analyzed for the purposes of incorporation into SVT.

3.1. Infer

Infer is a static code analysis tool for Java, C/C++, and Objective-C. Developed by Facebook, Infer is run continuously to verify select properties of their code and to check for errors [9]. Facebook claims that hundreds of bugs every month are found by Infer and fixed before app deployment.

3.1.1. Description

Infer uses a novel approach to code analysis based on Separation Logic and Bi-Abduction [10]. Separation Logic facilitates reasoning about mutations to computer memory. It is based on the separating conjunction $P * Q$, which asserts that P and Q hold for separate parts of memory. I.e., $P * Q$ is true of some heap of memory if it can be split into two sections, one which makes P true and the other which makes Q true. The separating conjunction differs from the Boolean conjunctions as $P * P \neq P$ while $P \wedge P = P$.

Bi-abduction is a form of logical inference for separation logic which automates the key ideas about local reasoning. Bi-abduction allows a large program to be broken down into small, independent analyses of its procedures. Therefore, if a section of code is changed, when the full program is analyzed again, only the code change needs to be re-analyzed. This process is called incremental analysis.

There are two main phases of an Infer scan:

1. **Capture Phase**
   In the first phase, Infer translates the input source code files into Infer's own intermediate representation language. The files get compiled like normal using the language's standard

```c
#include <stdlib.h>

struct Person {
    char fname[20];
    char lname[20];
};

void memory_leak_bug() {
    struct Person *p = malloc(sizeof(struct Person));
}

int main()
{
    return 0;
}
```

31 Memory Leak

compiler, and then are translated to the Infer intermediate representation language to be used in the next phase.

2. **Analysis Phase** In this phase, Infer analyzes each function and method separately, rather than scanning an entire source code file. If a bug is found in a method, then analysis is stopped for that method, but continues for other methods and functions. Errors are displayed on the standard out and also in a report text file. Bugs are filtered and Infer shows the ones that are most likely to be real.

3.1.2. Examples

Listing 32 shows the output of running Infer on a small example C program 31. Infer was able to catch two errors, a memory leak where memory had not been freed after allocation, and a "Dead Store" error where a variable is never used. The second example, shown in Listing 33, is the output of running Infer on a real Sandia developed piece of software.

3.1.3. SVT Applicability

As a static code analyzer, Infer takes a different approach than other tools such as SonarQube. It would be interesting to compare it to one of the other popular static code analysis tools and compare the results. If Infer reports errors that the other scan does not (or vice versa) then perhaps running both would be even more effective at catching issues in the build process.

Other features that make Infer a good candidate for including in the SVT are:

Integration with CI Infer has good documentation and recommendations for integrating it into a Continuous Integration (CI) workflow, something SVT would want to take advantage of if the decision was made to use Infer in the toolkit.

```
Capturing in make/cc mode...
Found 1 source file to analyze in /infer-examples/c/infer-out
1/1 [##########################################################################################] 100% 42.393ms

test_prgrm.c:10: error: Dead Store
  The value written to &p (type Person*) is never used.
   8.
   9. void memory_leak_bug() {
  10.       struct Person *p = malloc(sizeof(struct Person));
            ^
  11. }
  12.

test_prgrm.c:10: error: Memory Leak
  Pulse found a potential memory leak. Memory dynamically allocated at line 10 by call to 'malloc',
is not freed after the last access at line 10, column 5.
   8.
   9. void memory_leak_bug() {
  10.       struct Person *p = malloc(sizeof(struct Person));
            ^
  11. }
  12.


Found 2 issues
  Issue Type(ISSUED_TYPE_ID): #
    Memory Leak(MEMORY_LEAK): 1
      Dead Store(DEAD_STORE): 1
```

32 Example Infer Output

```
Capturing using compilation database...
Starting linting, translating 5 files
5/5 [##########################################################################################] 100% 2.638s

Found 5 source files to analyze in /infer-examples/sandia/leap/infer-out
5/5 [##########################################################################################] 100% 1.155s

WriteCache.cpp:113: error: Uninitialized Value
  The value read from result was never initialized.
  111.    DasTime::getSysTime(m_endTime);
  112.
  113.    if (result < 0) {
              ^
  114.      return false;
  115.    }


Found 1 issue
                  Issue Type(ISSUED_TYPE_ID): #
  Uninitialized Value(UNINITIALIZED_VALUE): 1
```

33 Infer Output on Sandia Software

Ease of use and documentation The documentation for Infer is thorough and easy to follow, and installing and running Infer is straightforward.

Open source and widely used Facebook has made Infer open source and freely available, and it has been adopted by many leading software companies, including Microsoft, Amazon, and Uber. Outside contributors continue to enhance the capability of Infer by adding additional error types to scan for.

3.1.4. Final Thoughts

From our analysis, Infer is recommend for inclusion in the Software Verification Toolkit. The pros of using Infer outweigh the cons (see Table 3-1), and on initial research Infer is a good tool that takes a unique approach to static code analysis.

Table 3-1 Infer Pros and cons

Pros	Cons
Easy to install and use	Only supports Java and C-family languages
Very well documented	Does not check program correctness, just reports (possible) errors
Integrates well in a CI build pipeline	
Used extensively in Industry	
Free and open source	

Table 3-2 Infer Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Excellent	Very Easy	Very Easy	Good	Good	Good	Fair

3.2. Vercors

Vercors straddles the line between a static analyzer and formal verifier. It offers a wide arrays of static analysis tools that can catch typical programming errors but comes with an annotation language that can be used for formal verification of written software. It targets an audience interested in proving multi-threaded applications as "*partially correct*" using a permission-based Concurrent Separation Logic as its logical foundation. Its documentation, installation instructions, and tutorial can be found at [38].

3.2.1. Description

Vercors utilizes a backend known as Viper [39] which ultimately converts programs through a series of transformations to a format that can be fed into the Z3 formal verifier. It currently supports select versions of Java, C, and Cuda/OpenMP. It also processes their own Java-based language they call the Program Verification Language. Other languages can be added by specifying a language parser or parser generator like ANTLR [37].

As it is a static analysis tool, Vercors can operate on already-written code with little modifications necessary. If modifications are necessary to prove a user's algorithm correct, Vercors uses a combination of the source language, a JML-like annotation syntax, and source code comments to make assertions, test method contracts, and identify thread permissions during code analysis. Those extensions to a pure static analysis tool allows Vercors to assert algorithm correctness in a similar fashion to a formal verifier.

Although Vercors boasts support for multiple languages, its feature set primarily focuses on Java and their custom language, PVL. Their documentation will explicitly mention that a feature is unavailable for C, Cuda and/or OpenMP.

3.2.2. Examples

Vercors as a static analyzer includes many of the basic tools you'd expect in a decent static analysis tool. It has options to check for null-pointer dereferences, indexing arrays out-of-bounds, overflow, etc. These options can be selectively turned on with different executable flags, allowing the user to choose a combination of tools they want on a per-file basis.

In addition to the static analysis tools that come with Vercors, it supplies the user with an annotation language to support formal software verification. The annotation language roots itself in the source language, but follows a JML syntax so it is incompatible with other tools that use the JML syntax (comment blocks with an '@' indicate statements Vercors should parse). Additionally, the annotation language reserves a few more keywords for Vercors' functionality and proof statements. The simplest use case for those keywords allows the user to set up method contracts like the one in Listing 34.

If the function contracts are violated during the execution of Vercors' analysis, the analysis will report a failure (Listing 35).

In addition, the annotation language extends to user defined proofs and assertions (Listing 36). Users can define specific statements they wish to prove about the functions they are writing by adding JML with the *assert* keyword (Listing 37).

To help prove more difficult statements, or programmatically define a variable for Vercors you plan to use later, Vercors allows for 'ghost code' sections. These sections allow users to define ghost variables or even functions that can be used by Vercors but are completely ignored in the normal compilation of a program.

```
// -*- tab-width:2 ; indent-tabs-mode:nil -*-
//:: cases function-contracts
//:: tools silicon
//:: verdict Pass

/*@
  requires a <= b;
  ensures \result == a*b;
 @*/
int ordered_mult(int a, int b) {
  return a * b;
}
```

34 Vercors C Static Analysis

```
 vct --silicon function_contracts.c
clang-12: warning: argument unused during compilation: '-nocudainc' [-Wu...
Could not find file: /usr/local/vercors/src/main/universal/res/config/ja...
Errors! (1)
=== function_contracts.c            ===
------------------------------------
  CallPreCondition:AssertionFalse
====================================
=== function_contracts.c            ===
------------------------------------
  caused by
====================================
The final verdict is Fail
```

35 Contract Violation

```
// -*- tab-width:2 ; indent-tabs-mode:nil -*-
//:: cases asserts
//:: tools silicon
//:: verdict Pass

/*@
 requires a < c;
 requires b < c;
 @*/
int min_mult(int a, int b, int c) {
    int tmp = a < b ? a : b;

    //@ assert tmp < c;
    return c * tmp;
}
```

36 Vercors Assert Statements

```
vct —silicon asserts.c
clang-12: warning: argument unused during compilation: '-nocudainc' [-Wu...
Could not find file: /usr/local/vercors/src/main/universal/res/config/ja...
Success!
The final verdict is Pass
```

37 Simple Pass

```
1
2  // -*- tab-width:2 ; indent-tabs-mode:nil -*-
3  //:: cases asserts
4  //:: tools silicon
5  //:: verdict Pass
6
7  /*@
8   requires a < c;
9   requires b < c;
10  @*/
11 int min_mult(int a, int b, int c) {
12     int tmp = a < b ? a : b;
13     //@ ghost int min = a < b ? a : b;
14     //@ assert tmp == min;
15
16     //@ assert tmp < c;
17     min = 10;
18     return c * min;
19 }
```

38 Vercors Assert Statements

```
// -*- tab-width:2 ; indent-tabs-mode:nil -*-
//:: cases permissions
//:: tools silicon
//:: verdict Pass


/*@
    requires size > 0;
    requires values != NULL;
    context \pointer(values, size, write);
    context (\forall* int i; 0 <= i && i < size; Perm(values[i], write));
@*/
void zero_array(int size, int pos, int* values) {
    for (int i = 0; i < size -1; i++) {
        values[i] = 0;
    }

}
```

39 Vercors Assert Statements

In Listing 38, we've defined another variable *min* that is only referenced in the Vercors annotation language. We can use that ghost variable to perform assertions about our code and extend our proofs to more complex logic. Those ghost blocks can extend to functions though care has to be made to not modify the external state of a function, that is ghost functions must be *pure* functions.

One word of warning though, ghost functions can be tricky to use correctly because Vercors is modifying the AST in-place. If a user accidentally refers to a ghost function or variable (for example $returnc * min$ in Listing 38), Vercors will not complain about the extra variable. However, the code will fail to compile with a regular compiler since *min* doesn't actually exist except in comments.

One strong aspect of Vercors is its permission-based assertions for multi-threaded function analysis. This does extend to single-threaded applications by enforcing read/write capabilities for data stored in mutable memory. Setting and consuming permissions handled directly in the annotation languages using specific Vercors keywords.

Unfortunately, since Vercors relies on its own parsers, more complex C programs that utilize includes can cause issues if the user is trying to use Vercors to prove something about a complex piece of code. For example, simply including the 'stdio' header for C programs causes Vercors to fail to parse the following example even though the stdio header has nothing to do with the assertions in the annotation language (Listing 41).

The $--cpp$ flag stand for the C pre-processor flag, which enables the *#include* to actually perform its includes substitutions. This will cause problems for already-written software that utilizes includes to break up the functionality into manageable chunks. If the $--cpp$ flag is turned off, it will limit the amount of proofs developers can perform over the larger executable to singular functions.

```
// -*- tab-width:2 ; indent-tabs-mode:nil -*-
//:: cases stdio-errors
//:: tools silicon
//:: verdict Pass

#include <stdio.h>


//@ requires a>0;
//@ ensures \result == 0;
int foo(int a) {
  return 0;
}
```

40 Vercors Assert Statements

```
vct --silicon --cpp stdio_errors.c
[abort] Exception class java.lang.NullPointerException while parsing stdio...
The final verdict is Error
```

41 Simple Pass

A few other inconveniences to mention starts with the error reports (especially with C) leaving much to be desired. When writing some of the examples above, if mistakes were made the error reporting output by Vercors is obtuse and often refers to the incorrect lines of code that actually caused the errors. Additionally, Vercors does not support the 'const' keyword making it a tool completely incapable of use for modern C programs.

3.2.3. SVT Applicability

As a static analysis tool, Vercors applicability lies mainly in the typical problems best solved through static analysis (e.g. memory safety). However, there are more feature rich static analysis tools out there that can perform as-good-as Vercors, or even better.

Vercors main draw is its out-of-the-box toolset is applicable to multi-threaded applications. Its permission logic for concurrent operations can find potential race-conditions and integrate that feature into the same tools used for formal verification.

The formal logic tools in Vercors is a welcome addition to the static analysis tools built into the tool. Without any effort, Vercors formal logic tools would be applicable to Java and its native language, PVL. The formal logic toolset in a C context revealed Vercors would report erroneous line numbers when it ran into errors making it difficult to recommend for C projects. The applicability for Vercors' formal tools in CUDA/OpenMP are unknown at this time.

One option for SVT to use Vercors is to write language specific parsers that transform the source language into a PVL syntax. This falls into the same pitfalls of any language-specific parser

Table 3-3 Vercors Pros and cons

Pros	Cons
Both static & formal analysis possible	Overall feels like beta or alpha software
Language options are extensible extensible	Feature support for Non-Java/PVL languages
Active development	Obtuse or incorrect error messages on failures
	Large number of foreign contributors might limit its high-side use

Table 3-4 Vercors Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Fair	Neutral	Very Easy	Fair	Poor	Fair	Poor

though and will introduce a maintenance cost that has to be paid any time a language releases a new version and difficulty in mapping any errors back to the original source language.

The Viper backend was developed by funding provided to ETH Zurich and has a large footprint of foreign contributors. For applications that exist on high-side only networks, this may require additional work to get the right approvals and vetting for SVT's use.

3.2.4. Final Thoughts

Unfortunately, Vercors' functionality and robustness leaves much to be desired. While its formal verification tools and feature sets fit well in the premise of the SVT ecosystem, Vercors doesn't deliver well on those tools outside of Java and PVL. Additionally, since adding new languages is performed by adding a new parser, a maintenance cost to update those parsers in tandem with the languages would be added to SVT if Vercors is included in SVT's toolbox.

At this time, Vercors is not recommended for SVT due to its fragility, language restrictions, and security concerns.

3.3. K Framework

The K Framework is a tool for designing and modeling programming languages and software/hardware systems. At the core of the K Framework is a programming, modeling, and specification language called K. The K Framework includes tools for compiling K specifications to build interpreters, model checkers, verifiers, associated documentation, and more [18].

3.3.1. Description

K is a rewrite based executable semantic framework in which programming languages, type systems, and formal analysis tools can be defined using configurations and rules. A K file consists of one or more requires or modules, and each module consists of one or more imports or sentences. Sentences consist of productions, which can define either syntax or rules. Syntax statements are called productions, and are introduced with the syntax keyword followed by sorts, followed by a ::=, and finally followed with the definition of one or more productions themselves separated by the | operator. Functions are given definitions using rules, which are defined by the rule keyword and contain at least one rewrite operator, =>. If a function is called with arguments that match the patterns on the left-hand side then the return value of the function is the pattern on the right-hand side. Additionally, variables can be used to match any pattern with the exception that the same name must match the same pattern.

K's grammar is divided into two components, the outer syntax and the inner syntax. The outer syntax refers to the parsing of requires, modules, imports, and sentences, IE the K grammar's syntax/semantics. Inner syntax refers to the parsing of rules and programs defined by the user. Rules are parsed within the context of a module, and modules are used to construct the grammar and perform syntax parsing. The K Framework parsing utilizes a modified variant of BNF notation which distinguishes between two types of production items: terminals and non-terminals. Terminals represent literal strings of characters that are part of the production syntax, whereas non-terminals represent a sort of name where the syntax of that production accepts any valid term that fits that position. These BNF grammar definitions allow the user to sculpt their own language semantics which can then be interpreted by the K Framework. Semantics rules can be created and used to parse programs and generate an intermediate AST representation known as Kore. K's just-in-time parser is a GLL parser, which means it can handle the full generality of context-free grammars, including ambiguous ones. Additionally, K allows users to define custom lexical tokens with the keywords syntax lexical using Flex to implement the lexical analysis. These tokens can be utilized in lexical regular expressions and used to generate a typed string in the AST, effectively allowing the developer to create their own language syntax in addition to semantics. In addition to the lexical analysis, K supports creating an ahead-of-time parser using the GNU Bison LR(1) parser generator.

3.3.2. Examples

Listing 42 shows an example BNF grammar definition of Boolean operations using the K Framework syntax. In this example, the module and import definitions are showcased, along with the interpreter formations of the Boolean implementation which allows for the interpreter to operate on the grammar. Most of components of the K Framework's grammar notation are on display in this example.

Additionally, when we run *kast* against the example provided in 43 we get the AST shown in Listing 44. With this, it is possible to see just how exactly the K Framework parser is generating the AST utilized by the K Framework when parsing an input file. For more complex grammars representing programming languages, it doesn't take much imagination to understand the

```
module BOOLEAN-EXAMPLE-SYNTAX
  imports BOOL-SYNTAX

  syntax Bool ::= "(" Bool ")" [bracket]
                > "!" Bool [function]
                > left:
                  Bool "&&" Bool [function]
                | Bool "^" Bool [function]
                | Bool "||" Bool [function]
                | Bool "–>" Bool [function]
endmodule

module BOOLEAN-EXAMPLE
  imports BOOLEAN-EXAMPLE-SYNTAX
  imports BOOL

  rule ! B => notBool B
  rule A && B => A andBool B
  rule A ^ B => A xorBool B
  rule A || B => A orBool B
  rule A –> B => A impliesBool B
endmodule
```

42 Boolean Example

```
    (true && false) || (true && true) –> !false || !true ^ (true && (true || false))
```

43 Boolean Evalutation

```
    `_^__BOOLEAN-EXAMPLE-SYNTAX_Bool_Bool_Bool`(
        `_||__BOOLEAN-EXAMPLE-SYNTAX_Bool_Bool_Bool`(
            `_–>__BOOLEAN-EXAMPLE-SYNTAX_Bool_Bool_Bool`(
                `_||__BOOLEAN-EXAMPLE-SYNTAX_Bool_Bool_Bool`(
                    `_&&__BOOLEAN-EXAMPLE-SYNTAX_Bool_Bool_Bool`(#token("true","Bool"),#token("false","
                        Bool"))
                    ,`_&&__BOOLEAN-EXAMPLE-SYNTAX_Bool_Bool_Bool`(#token("true","Bool"),#token("true","
                        Bool")))
                ,`!__BOOLEAN-EXAMPLE-SYNTAX_Bool_Bool`(#token("false","Bool")))
            ,`!__BOOLEAN-EXAMPLE-SYNTAX_Bool_Bool`(#token("true","Bool")))
        ,`_&&__BOOLEAN-EXAMPLE-SYNTAX_Bool_Bool_Bool`(#token("true","Bool")
            ,`_||__BOOLEAN-EXAMPLE-SYNTAX_Bool_Bool_Bool`(#token("true","Bool"),#token("false","Bool")))
                )
```

44 Boolean AST

```c
// KCC can also detect buffer overflows within the subobjects of an aggregate
// type. Here we see a struct declared with an array followed by an integer.
// It is correct to assume that the struct is laid out sequentially in memory.
// However, nonetheless, it is a buffer overflow to access the 32nd index of
// this array because it was declared with size 32. This means that we are
// able to detect if a buffer overflow might leak sensitive information
// contained elsewhere in a struct that also contains an array.

struct foo {
  char buffer[32];
  int secret;
};

int idx = 0;

void setIdx() {
  idx = 32;
}

int main() {
  setIdx();
  struct foo x = {0};
  x.secret = 5;
  return x.buffer[idx];
}
```

45 3-array-in-struct

complexity behind an associated AST and how it may be possible to inject code verification components into a given language's semantics.

The C example from 45 and the output of the file when compiled with rv-match 47, the c-semantics compiler generated using the k-framework, shows what is possible using the k-framework parser utilities. Here we can see that the K Framework compiler, kcc, is able to detect an incorrect pointer dereference past the scope of the array. Additionally, when kcc is run on 46 no issues are detected. Unlike the previous example this output fails to find the issue. This is probably due in part to the lack of a reference to the C standard, as well as the inclusion of linked libraries. Still, it is promising that it is possible to detect non-library based static issues with the K Framework generated c parser.

3.3.3. SVT Applicability

The K Framework has a lot of potential with regards to statically analyzing software. It provides a unified AST and underlying interface for manipulating multiple programming languages. The only caveat is that utilizing the K Framework involves creating a BNF based parser for each language. Additionally, these parsers will require constant maintenance and updates when new language features are deployed. However, the work required in maintaining the parser instead of some library to analyze the software would appear to be much less. Additionally, utilizing the K

```c
#include <stdlib.h>

struct Person {
    char fname[20];
    char lname[20];
};

void memory_leak_bug() {
    struct Person *p = malloc(sizeof(struct Person));
}

int main()
{
    return 0;
}
```

46 Memory Leak

```
   Dereferencing a pointer past the end of an array:
   > in main at 3-array-in-struct.c:24:3

 Undefined behavior (UB-CER4):
      see C11 section 6.5.6:8 http://rvdoc.org/C11/6.5.6
      see C11 section J.2:1 items 47 and 49 http://rvdoc.org/C11/J.2
      see CERT-C section ARR30-C http://rvdoc.org/CERT-C/ARR30-C
      see CERT-C section ARR37-C http://rvdoc.org/CERT-C/ARR37-C
      see CERT-C section STR31-C http://rvdoc.org/CERT-C/STR31-C
      see MISRA-C section 8.18:1 http://rvdoc.org/MISRA-C/8.18
      see MISRA-C section 8.1:3 http://rvdoc.org/MISRA-C/8.1
```

47 array-in-struct compiled output

Framework has the added benefit of providing a single interface in which an intermediate format could be analyzed and validated. Depending on the underlying intermediate format capabilities it may even be possible to stretch into the realm of formal verification through analysis of the intermediate format or through the AST.

The main downside to the K Framework at this point is its lack of support. Currently the only language with a valid parser still being updated for it is the C programming language. Compared to other tools, like ANTLR, which has much more mainstream support, it becomes difficult to recommend using the K Framework.

Table 3-5 K Framework Pros and cons

Pros	Cons
Very well documented parser and grammar syntax	Only language with a valid BNF grammar written is C
Useful abstraction for performing static analysis	Underlying APIs for manipulating the AST are more obtuse
Very usable with good examples	Installation of c-semantic parser is out of date
Capable of doing simple static analysis out of the box	Doesn't work at finding issues with linked c-libraries

Table 3-6 K Framework Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Excellent	Difficult	Difficult	Good	Good	Fair	Good

3.3.4. Final Thoughts

Due to the stagnated nature of the repository and the issues with the c-semantic analyzer, the K Framework is not recommended for incorporation into SVT. While the K Framework itself may not be suitable, the functionality it provides with regards to analyzing ASTs should be looked into further. Being able to analyze an intermediate format would allow for the capability to more easily analyze multiple programming languages. This would allow SVT to implement analysis tools on a common format, after which a focus could be placed on converting higher level languages into that intermediate format.

3.4. CBMC

The C Bounded Model Checker [5] is a bounded model checker originally designed to work with C/C++ code and was later expanded to work with Java bytecode as JBMC. It is actively being

developed on github [29] and has been used in high-profile code bases like Amazon's aws-c-common library to form proofs about code execution [30].

3.4.1. Description

CBMC is a bit-precise bounded model checker that can be used to statically analyze C/C++ programs using SAT solvers. CBMC enables users to add assertion-based statements directly into their C/C++ code so they may prove its correctness. It utilizes its own parser to transform the software with assertions into an AST and intermediate representation that can be interpreted by the SAT solvers (Listing 3-1) so counterexamples can be generated.

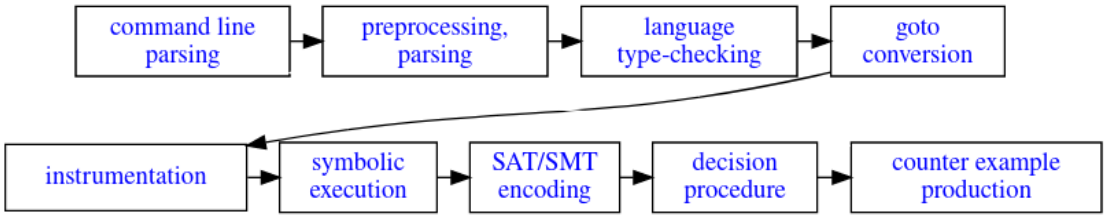


Figure 3-1 CBMC Processing Workflow [26]

It currently supports C89, C99, and partial C11 with most of the C++ features associated with the same C releases. By default, it uses a SAT solver based on MiniSat but does support the use of additional SAT solvers (Z3, Boolector, MathSAT, Yices 2), although they have to be installed separately. Once they are installed, CBMC can seamlessly use the other SAT solvers by specifying which one to use on the command line.

As CBMC is a static analysis tool, it comes well-prepared to deal with the average problems a static analysis tool would be expected to find. A variety of memory safety, exception handling, and undefined behavior checks can be turned on with individual command line flags. In addition to the static analysis checks, CBMC uses the user-specified assertions to further constrain the SAT solvers which helps bridge the gap between a pure static analysis tool and the formal verification of written software. Finally, the results from running CBMC can be output to the console or formatted in a structured format, either JSON or XML.

3.4.2. Examples

CBMC as a static analysis tool has collection of individual flags that can be turned on for directed checks on a single source file and requires no changes to the source file. Consider a program that deletes all elements from an array that match an input (Listing 48).

The program erroneously decrements the size of the array after shifting all the elements in the array down one index (lines 12,13). This would result in accessing memory outside the bounds of the array (line 4 when $i == size - 1$) and subsequently performs undefined behavior. If we run

```
1
2  void shift_down(const int size, const int pos, int* values) {
3      for (int i = pos; i < size; i++) {
4          values[i] = values[i+1];
5      }
6  }
7
8  void delete_element(int* size, int* values, int element) {
9
10     for (int i = 0; i < *size; ++i) {
11         if (values[i] == element) {
12             shift_down(*size, i, values);
13             *size = *size - 1;
14         }
15     }
16 }
17
18 int main() {
19   int size = 10;
20   int values[10] = {0, 1, 2, 3, 4, 5, 3, 7, 8, 9};
21
22   delete_element(&size, values, 9);
23 }
```

48 C++ Array Element Deletion

CBMC with the *–pointer-check* static analysis tool on, it does find the mistake (output trimmed, Listing 49).

In the previous example (Listing 48), CBMC defaulted to analyzing the code starting with a main function. If this were library code and did not have a main function, CBMC can start its analysis at any arbitrary function using the *–function* command line option, however, that function name must exist in a symbol table. Since a symbol table is required, C++ functions are significantly harder to specify even when CBMC allows you to dump the symbol table it identified (*–show-symbol-table*). In the time used to analyze CBMC's functionality, specifying a C++ function never seemed to work correctly (Listing 50).

However, the C version of the same program works just fine (Listing 51).

If an entry point is set to a function other than main and there is not enough information about how the program is used in the entry point, CBMC may enter an infinite loop as it tries to unroll loops to a natural stopping point. For these cases, CBMC allows users to set max unrolling depth with *–unwind <number>*. However, successful verification will depend on whether the issues with the code are tied to the depth of the search. As seen in Listing 51, the program was successfully verified despite still having issue on line 4.

In addition to the static analysis tools, CBMC has an included tool they call the CPROVER library. The CPROVER library is a series of function calls or macros that insert statements directly into the source code that can be used by a SAT solver to make assertions about code correctness. The CRPOVER suite elevates the tools in CBMC from just a static analysis tool to a

```
CBMC version 5.49.0 (cbmc-5.49.0-37-g637f13816) 64-bit x86_64 linux
Parsing delete_element_bad_pointer.cpp
Converting
Type-checking delete_element_bad_pointer
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 8 object bits, 56 offset bits (default)
Starting Bounded Model Checking
Unwinding loop delete_element(ptr_signed_int,ptr_signed_int,signe...
...

** Results:
delete_element_bad_pointer.cpp function delete_element
[delete_element.pointer_dereference.1] line 10 dereference failure:
  pointer NULL in *size: SUCCESS
... line 10 ... pointer invalid in *size: SUCCESS
... line 10 ... deallocated dynamic object in *size: SUCCESS
... line 10 ... dead object in *size: SUCCESS
... line 10 ... pointer outside object bounds in *size: SUCCESS
... line 10 ... invalid integer address in *size: SUCCESS
...

delete_element_bad_pointer.cpp function shift_down
[shift_down.pointer_dereference.1] line 4 dereference failure: pointer NULL in
     values[(signed long int)i]: SUCCESS
[shift_down.pointer_dereference.2] line 4 dereference failure: pointer invalid
     in values[(signed long int)i]: SUCCESS
[shift_down.pointer_dereference.3] line 4 dereference failure: deallocated
    dynamic object in values[(signed long int)i]: SUCCESS
[shift_down.pointer_dereference.4] line 4 dereference failure: dead object in
    values[(signed long int)i]: SUCCESS
[shift_down.pointer_dereference.5] line 4 dereference failure: pointer outside
     object bounds in values[(signed long int)i]: SUCCESS
[shift_down.pointer_dereference.6] line 4 dereference failure: invalid integer
     address in values[(signed long int)i]: SUCCESS
[shift_down.pointer_dereference.7] line 4 dereference failure: pointer NULL in
     values[(signed long int)(i + 1)]: SUCCESS
[shift_down.pointer_dereference.8] line 4 dereference failure: pointer invalid
     in values[(signed long int)(i + 1)]: SUCCESS
[shift_down.pointer_dereference.9] line 4 dereference failure: deallocated
    dynamic object in values[(signed long int)(i + 1)]: SUCCESS
[shift_down.pointer_dereference.10] line 4 dereference failure: dead object in
     values[(signed long int)(i + 1)]: SUCCESS
[shift_down.pointer_dereference.11] line 4 dereference failure: pointer
    outside object bounds in values[(signed long int)(i + 1)]: FAILURE
[shift_down.pointer_dereference.12] line 4 dereference failure: invalid
    integer address in values[(signed long int)(i + 1)]: SUCCESS

** 1 of 30 failed (2 iterations)
VERIFICATION FAILED
```

49 A Basic CBMC Static Analysis Check for Memory Safety

```
> cbmc delete_element_bad_pointer.cpp --function shift_down
CBMC version 5.49.0 (cbmc-5.49.0-37-g637f13816) 64-bit x86_64 linux
Parsing delete_element_bad_pointer.cpp
Converting
Type-checking delete_element_bad_pointer
Invalid User Input
Option: --function
Reason: couldn't find function with name 'shift_down' in symbol table
```

50 Attempting to target a C++ function by name

```
> cbmc delete_element_bad_pointer.c --function shift_down --unwind 10
CBMC version 5.49.0 (cbmc-5.49.0-37-g637f13816) 64-bit x86_64 linux
Parsing delete_element_bad_pointer.c
Converting
Type-checking delete_element_bad_pointer
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 8 object bits, 56 offset bits (default)
Starting Bounded Model Checking
Unwinding loop shift_down.0 iteration 1 file delete_element_bad_pointer.c line
     3 function shift_down thread 0
Unwinding loop shift_down.0 iteration 2 file delete_element_bad_pointer.c line
     3 function shift_down thread 0
Unwinding loop shift_down.0 iteration 3 file delete_element_bad_pointer.c line
     3 function shift_down thread 0
Unwinding loop shift_down.0 iteration 4 file delete_element_bad_pointer.c line
     3 function shift_down thread 0
Unwinding loop shift_down.0 iteration 5 file delete_element_bad_pointer.c line
     3 function shift_down thread 0
Unwinding loop shift_down.0 iteration 6 file delete_element_bad_pointer.c line
     3 function shift_down thread 0
Unwinding loop shift_down.0 iteration 7 file delete_element_bad_pointer.c line
     3 function shift_down thread 0
Unwinding loop shift_down.0 iteration 8 file delete_element_bad_pointer.c line
     3 function shift_down thread 0
Unwinding loop shift_down.0 iteration 9 file delete_element_bad_pointer.c line
     3 function shift_down thread 0
Not unwinding loop shift_down.0 iteration 10 file delete_element_bad_pointer.c
     line 3 function shift_down thread 0
Runtime Symex: 0.0074673s
size of program expression: 211 steps
simple slicing removed 0 assignments
Generated 0 VCC(s), 0 remaining after simplification
Runtime Postprocess Equation: 1.8363e-05s
VERIFICATION SUCCESSFUL
```

51 Running CBMC and Targeting the 'shift_down' Function

```
1  #include <stdio.h>
2
3  void shift_down(const int size, const int pos, int* values) {
4    __CPROVER_precondition(size > pos, "0-index position is less than 1-indexed size");
5    __CPROVER_precondition(size > 0, "Size is > 0");
6    __CPROVER_precondition(pos >= 0, "position is valid (> 0)");
7    __CPROVER_precondition(values != NULL, "values is a non-null array");
8    int i = pos;
9    for (i = pos; i < size; ++i) {
10     values[i] = values[i + 1];
11   }
12   __CPROVER_postcondition(i == size, "Iterated through the whole array");
13   __CPROVER_postcondition(values !=  NULL, "values hasn't changed");
14 }
15
16 void delete_element(int* size, int* values, int element) {
17   __CPROVER_precondition(size != NULL, "pointer to values array size is non-null");
18   __CPROVER_precondition(values != NULL, "values pointer is non-null");
19
20   int i = 0;
21   for (i = 0; i < *size; ++i) {
22     __CPROVER_assert(i < *size, "iteration is inside the length of the array");
23     if (values[i] == element) {
24       __CPROVER_assume(values[i] == element);
25       *size = *size - 1;
26       shift_down(*size, i, values);
27     }
28   }
29   __CPROVER_assert(i == *size, "Iterate through the whole array");
30
31   __CPROVER_assert(__CPROVER_forall {
32     int j;
33     (0 <= j && j < *size) ==> values[j] != element
34   }, "Value has been deleted");
35 }
```

52 CBMC Assertion Examples

combination of static analysis and formal verification. The statements also integrate into the source code and can directly reference the source variables to make assertions about their values (Listing 52).

There are more CRPOVER statements that can be used but a few of the major ones are used here. Functions can use both pre and post conditions to define method contracts and are asserted throughout the execution of the program every time that function is called (Listing 52, lines 4-7, 12-13, 17-18). Assertion statements regarding the expected variable values can prove the correctness or incorrectness of a specified program during its execution (Listing 52, lines 22, 29, 31). Assume statements can help restrict the domain of a variable and lead to a narrowed proof search space (Listing 52, line 24, but a more complex example can be seen in Listing 53). Additional logical quantifier CPROVER functions can help cover entire variable domains and

```
1 #include "good_delete_element.c"
2
3 void test() {
4     int size;
5     int values[10] = {0,1,2,3,4,5,6,7,8,9};
6
7     __CPROVER_assume(size == 10);
8     delete_element(&size, values, 3);
9 }
```

53 CBMC Assertion Main - Correct

make an assertion for each valid element in that domain (Listing 52, lines 31-34).

Defining an entry point outside of the main execution of the program allows CBMC to attempt to prove the correctness of the program over an entire domain of possible values instead of a specific execution (Listing 53 versus Listing 48). The assume statements (Listing 53) set up the domain of the parameter values pass into the *delete_element* function. In this case, the domain of *size* is tied specifically to the length of the array (Listing 54); anything else will cause CBMC to correctly report an error.

If a pre or post condition was violated during the execution path of the main program or during the analysis, CBMC will report those failures to the user along with the failing error message. For example, consider the case where a *NULL* pointer is passed into the *shift_down* function (Listing 55)

On line 12, we restrict the domain of *values* to be equal to *NULL* which violates the precondition of the *shift_down* function. The output reflects the violation and reports a failure of the pre and post conditions of the function.

One troubling downside for C/C++ programs analyzed with CBMC, is there does not seem to be a way to turn off the pre-processor instructions. This runs into problems when *#include* files contain unsupported C/C++ language features in the CBMC ecosystem. For example, even though Listing 57 contains no C++11 features, attempting to use CBMC on the source code causes CBMC's parser to error out (Listing 58) due to the included iostream header file containing an unsupported language feature. This will be especially troubling for SVT if CBMC is used in already written code that makes use of, or includes libraries that use the unsupported language features.

3.4.3. SVT Applicability

For SVT's purposes, the limiting factor of CBMC is going to be its support for C++ templates and language features beyond C++11. However, for older C/C++ programs, CBMC has the robustness and simple interface to integrate into those programs with relative ease. As long as the language restrictions are kept in mind from the start, the tool is recommended for use when writing new C/C++ codebases. The critical sections of code can ensure they are isolated from unsupported language features and use CBMC's full toolbox to prove the correctness of the software.

```
> cbmc good_delete_element_good_main.c --function test

...

** Results:
good_delete_element.c function delete_element
[delete_element.assertion.1] line 22 iteration is inside the length of the
    array: SUCCESS
[delete_element.precondition_instance.1] line 26 0-index position is less than
     1-indexed size: SUCCESS
[delete_element.precondition_instance.2] line 26 Size is > 0: SUCCESS
[delete_element.precondition_instance.3] line 26 position is valid (> 0):
    SUCCESS
[delete_element.precondition_instance.4] line 26 values is a non-null array:
    SUCCESS
[delete_element.assertion.2] line 31 Value has been deleted: SUCCESS

good_delete_element.c function shift_down
[shift_down.postcondition.1] line 12 Iterated through the whole array: SUCCESS
[shift_down.postcondition.2] line 13 values hasn't changed: SUCCESS

good_delete_element_good_main.c function main
[main.precondition_instance.1] line 8 pointer to values array size is non-null
    : SUCCESS
[main.precondition_instance.2] line 8 values pointer is non-null: SUCCESS

** 0 of 10 failed (1 iterations)
VERIFICATION SUCCESSFUL
```

54 CBMC Assertion Main - Output

```c
#include "good_delete_element.c"

void test() {
    int size;
    int position;
    int* values;

    __CPROVER_assume(size > 0);
    __CPROVER_assume(size > position);
    __CPROVER_assume(position >= 0);
    __CPROVER_assume(values == NULL);

    shift_down(size, position, values);
}
```

55 CBMC Precondition Violated

```
> cbmc good_delete_element_bad_precondition_main.c --function test --unwind 1

...

** Results:
good_delete_element.c function delete_element
[delete_element.assertion.1] line 22 iteration is inside the length of the
    array: SUCCESS
[delete_element.precondition_instance.1] line 26 0-index position is less than
     1-indexed size: SUCCESS
[delete_element.precondition_instance.2] line 26 Size is > 0: SUCCESS
[delete_element.precondition_instance.3] line 26 position is valid (> 0):
    SUCCESS
[delete_element.precondition_instance.4] line 26 values is a non-null array:
    SUCCESS
[delete_element.assertion.2] line 29 Iterate through the whole array: SUCCESS
[delete_element.assertion.3] line 31 Value has been deleted: SUCCESS

good_delete_element.c function shift_down
[shift_down.postcondition.1] line 12 Iterated through the whole array: SUCCESS
[shift_down.postcondition.2] line 13 values hasn't changed: SUCCESS

good_delete_element_bad_precondition_main.c function test
[test.precondition_instance.1] line 13 0-index position is less than 1-indexed
     size: SUCCESS
[test.precondition_instance.2] line 13 Size is > 0: SUCCESS
[test.precondition_instance.3] line 13 position is valid (> 0): SUCCESS
[test.precondition_instance.4] line 13 values is a non-null array: FAILURE

** 1 of 13 failed (2 iterations)
VERIFICATION FAILED
```

56 CBMC Precondition Violation Output

```
#include <iostream>

void shift_down(const int pos, const int size, int* values) {
  __CPROVER_precondition(size > pos, "0-index position is less than 1-indexed size");
  __CPROVER_precondition(size > 0, "Size is > 0");
  __CPROVER_precondition(pos >= 0, "position is valid (> 0)");
  __CPROVER_precondition(values != NULL, "values is a non-null array");
  int i = pos;
  for (i = pos; i < size; ++i) {
    values[i] = values[i + 1];
  }
  __CPROVER_postcondition(i == size, "Iterated through the whole array");
  __CPROVER_postcondition(values !=  NULL, "values hasn't changed");
}
```

57 Shift Down function including iostream

```
> cbmc include_woes.cpp
CBMC version 5.49.0 (cbmc-5.49.0-37-g637f13816) 64-bit x86_64 linux
Parsing include_woes.cpp
--- begin invariant violation report ---
Invariant check failed
File: /tmp/cmbc-git/src/cpp/parse.cpp:7515 function: rStatement
Condition: false
Reason: Unimplemented
Backtrace:
Backtraces not supported
--- end invariant violation report ---
Aborted (core dumped)
```

58 CBMC iostream parsing trouble

Investment into CBMC should also be met with investment into understanding the JBMC side of the tool. But, just like in C/C++, some of the Java language features are unsupported and before JBMC is integrated into SVT's tool set, the full list of unsupported features must be understood first. Those unsupported features will again limit the amount of already-written source code where JBMC can be applied.

Table 3-7 CBMC Pros and cons

Pros	Cons
Both Java and C/C++ language support	C++11 templates are largely unsupported
Direct integration into source code and variables	Only supports older versions of C/C++ (<2011)
Output can be well-structured (JSON or XML)	C/C++ include statements are always evaluated and can get CBMC into parsing trouble
Support for both Static and Formal analysis	

3.4.4. Final Thoughts

Unfortunately, CBMC barely misses the mark of a gold standard for formal verification of C/C++ programs. With respect to plain C programs, CBMC would be worth investment with the caveat that it can only be used only in the specific versions of C and C compilers supported by CBMC. Some testing will have to be done to figure out what specific versions of which compilers are supported. However, other than providing that information to the developer, it is not recommended that SVT updates CBMC to work with newer versions of C/C++ due the complexity behind supporting a modern C++ parser – the CBMC developers have also indicated their lack of desire for this in issue 6375 on their github [29].

While the Java side of CBMC was left unevaluated, SVT should consider adding JBMC to its tool sets with a same C/C++ caveat and utilizing it for new software only. The language feature restrictions may prove to be too restrictive for use in already written software.

Summarily, CBMC and JBMC are strong tools for formal software verification due to their robustness and feature sets. Both tools align well with SVT's goals and unless another tool can offer a similar feature set while shoring up the language feature restrictions or newer language versions, CBMC should be a recommended tool for SVT's C/C++ and Java use cases.

Table 3-8 CBMC Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Fair	Neutral	Very Easy	Fair	Excellent	Excellent	Fair

3.5. Clang Tools

Clang is a compiler for the family of C languages (C, C++, C#, Cuda) with the intent of being a front end for LLVM. One of their main goals is to provide the infrastructure to develop source code analysis tools and source-to-source transformation tools for the C language family. They have a few tutorials and plenty of examples on their main website [6]. The compiler has reached critical mass and is used in large, commercial projects like Mozilla's Firefox and Google's Chrome web browsers. It is actively developed with dozens of commits on a daily basis that fix bugs and add new features.

3.5.1. Description

The Clang tool set is built with tool modularity and extensibility in mind. They developed multiple entry points for various tools depending on the level of control the developer wants to have over the analyzed source code. The tool set works at various levels of abstraction over the code's AST; some of them meant to integrate into IDEs, syntax checking, or perform source-to-source translations for automatic bug fixing. Each of those abstractions gets closer to the underlying code's representation and can be analyzed for a variety of common bug patterns.

The most powerful aspects of the Clang tool set are the source-to-source tools that allow developers to write code that can generate source code directly. Though performing source-to-source translations can be rather complicated, Clang has a series of tools already built in with Clang-Tidy that shows the power of these types of transformations.

Clang does support interfacing with the Clang tooling through other languages besides the C-family languages; as an example, there are Python bindings.

3.5.2. SVT Applicability

At a minimum, SVT could develop an environment that makes it easy to use the vast number of Clang tools already built and maintained. The sheer number of static analysis features Clang provides shows the momentum Clang has as a useful static analyzer for the C-family language ecosystem.

If SVT wanted to invest into Clang further, SVT could use the Clang LibTooling interface to develop software that aims at augmenting source code for C-family languages to perform some formal verification on top of static analysis. SVT could transform the source with the help of one of the formal verifiers (2) and bind source statements to statements the formal verifier could parse and understand.

The only knock against Clang is that it does not support the other languages SVT is interested in. The narrow language options might make investing in a formal verification tool using Clang's LibTooling for C-family languages too specialized for SVT to pursue.

3.5.3. Final Thoughts

Ultimately, for SVT efforts towards the C-family languages, Clang needs to be included in the SVT supplied tool set. Clang's feature set is rich and support is ongoing through a large open-source effort that Clang's tools should not be ignored. Since the tool is being picked up by behemoths in the software engineering market, SVT can all but guarantee that Clang will continue to see active development and new features as time goes on.

Table 3-9 Clang Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Excellent	Very Easy	Easy	Excellent	Excellent	Excellent	Poor

3.6. Frama-C

Frama-C is a collaborative framework that contains several static and dynamic analysis tools to verify code written in C. These tools are provided as plug-ins, and they can interact to check security, verify requirements, and improve upon each analysis. Frama-C is free and open-source, and can be found at [11].

3.6.1. Description

Frama-C is made up of two parts: the kernel and plug-ins. The kernel is a core set of features that provides key services to end users and developers. It is based on a modified version of C Intermediate language (CIL), a front end for C that parses programs into a normalized representation. Frama-C extends CIL to support ANSI/ISO C Specification Language (ACSL) to

produce an AST that can be shared among analyzers. The kernel provides basic services such as program parsing that build an AST of the analyzed program, specialized services such as ACSL annotations for code analyses, and general-purpose libraries. Additional services provided by the kernel include: ensuring messages, source code, and annotations are uniformly displayed, handling parameters and command line options homogeneously, consistency mechanisms control the collaboration between analyzers, and a serialization mechanism allows a user to save analysis results and reload them later. These all help to set the groundwork for plug-ins to operate on a common representation of the source code.

Plug-ins are a set of analyzers that are dynamically linked against the kernel that can manipulate the AST and add ACSL annotations. Frama-C has four core plug-ins: Eva focuses on detecting undefined behaviors or vulnerabilities, WP aims to prove functional properties, E-ACSL checks properties at runtime and PathCrawler generates test cases. There are also specialized plug-ins to perform more complex analysis of programs, and developers can continually add plug-ins as they are created.

These analyzers can work together in two different ways:

1. **Sequentially** Sequentially, in which they use the results of an analyzer as input to another one by using the plug-in database stored by the kernel. This allows them to change analysis results and perform more complex operations.
2. **In Parallel** In parallel the plug-ins generate program annotations using ACSL as a collaborative language to combine partial analysis results into a full program verification. Partial results are integrated by the kernel to provide a consolidated status of the validity of all ACSL properties. The kernel uses the information provided by all analyzers to automatically compute the validity status of each program property and ensure the correctness of the entire verification process.

Installation of Frama-C is available through opam, the OCaml package manager. It is developed mainly in Linux, can be tested in macOS or using WSL. Once installed, analysis can be run on programs using the command line or the Frama-C GUI. When using the command-line, the structure *Frama-c -<plugin> <sourcefile>* is used.

3.6.2. Examples

Listing 59 shows the output of running Frama-C on a simple C example, delete_element.c, using the Eva plugin. Frama-C computes the initial state, and then the final states for each function in the program. An out of bounds warning is generated for this function.

The second example, shown in Listing 60, is the output of running Frama-C on the same C program, now using the WP plug-in. This plug-in requires the user to specify rules and assign variables to prove them, otherwise it assigns them to 'everything' and the result is unknown. Without manually adding assertions, the RTE plug-in can be used which will generate all the necessary assertions to be verified. This example proves 8 of the 23 goals scheduled, and users can add assertions to try and improve this ratio.

```
[kernel] Parsing delete_element.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva] Computing initial state
[eva] Initial state computed
[eva:initial-state] Values of globals at initialization

[eva] using specification for function printf_va_1
[eva] using specification for function printf_va_2
[eva] delete_element.c:26: starting to merge loop iterations
[eva] delete_element.c:12: starting to merge loop iterations
[eva] delete_element.c:5: starting to merge loop iterations
[eva:alarm] delete_element.c:6: Warning:
    out of bounds write. assert \valid(values + i);
[eva] using specification for function printf_va_3
[eva] using specification for function printf_va_4
[eva] delete_element.c:29: starting to merge loop iterations
[eva] using specification for function printf_va_5
[eva] done for function main
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function shift_down:
    i $\in$ [0..9]
    values[0] $\in$ [0..9]
        [1..8] $\in$ [1..9]
        [9] $\in$ {9}
[eva:final-states] Values at end of function delete_element:
    i $\in$ [0..2147483647]
    size $\in$ [0..10]
    values[0] $\in$ [0..9]
        [1..8] $\in$ [1..9]
        [9] $\in$ {9}
[eva:final-states] Values at end of function main:
    i $\in$ [0..10]
    size $\in$ [0..10]
    values[0] $\in$ [0..9]
        [1..8] $\in$ [1..9]
        [9] $\in$ {9}
    __retres $\in$ {0}
    S___fc_stdout[0..1] $\in$ [--..--]
```

59 Example Frama-C Output using Eva plugin

```
[kernel] Parsing delete_element.c (with preprocessing)
[rte] annotating function delete_element
[rte] annotating function main
[rte] annotating function shift_down
[wp] delete_element.c:12: Warning:
    Missing assigns clause (assigns 'everything' instead)
[wp] delete_element.c:29: Warning:
    Missing assigns clause (assigns 'everything' instead)
[wp] delete_element.c:26: Warning:
    Missing assigns clause (assigns 'everything' instead)
[wp] delete_element.c:5: Warning:
    Missing assigns clause (assigns 'everything' instead)
[wp] 23 goals scheduled
[wp] [Alt-Ergo] Goal typed_delete_element_assert_rte_mem_access_3 : Unknown (95ms)
[wp] [Alt-Ergo] Goal typed_delete_element_assert_rte_mem_access_2 : Unknown (90ms)
[wp] [Alt-Ergo] Goal typed_delete_element_assert_rte_mem_access : Unknown (82ms)
[wp] [Alt-Ergo] Goal typed_delete_element_assert_rte_signed_overflow_2 : Unknown (86ms)
[wp] [Alt-Ergo] Goal typed_main_assert_rte_index_bound_2 : Unknown (272ms)
[wp] [Alt-Ergo] Goal typed_main_assert_rte_index_bound : Unknown (Qed:16ms) (267ms)
[wp] [Alt-Ergo] Goal typed_main_call_printf_va_1_pre : Unknown (126ms)
[wp] [Alt-Ergo] Goal typed_main_assert_rte_index_bound_3 : Unknown (592ms)
[wp] [Alt-Ergo] Goal typed_main_assert_rte_index_bound_4 : Unknown (624ms)
[wp] [Alt-Ergo] Goal typed_main_call_printf_va_2_pre : Unknown (371ms)
[wp] [Alt-Ergo] Goal typed_main_call_printf_va_3_pre : Unknown (362ms)
[wp] [Alt-Ergo] Goal typed_shift_down_assert_rte_mem_access : Unknown (Qed:16ms) (76ms)
[wp] [Alt-Ergo] Goal typed_shift_down_assert_rte_mem_access_2 : Unknown (127ms)
[wp] [Alt-Ergo] Goal typed_main_call_printf_va_4_pre : Unknown (783ms)
[wp] [Alt-Ergo] Goal typed_main_call_printf_va_5_pre : Unknown (Qed:16ms) (699ms)
[wp] Proved goals:    8 / 23
    Qed:             3
    Alt-Ergo:        5  (16ms) (34) (unknown: 15)
```

60 Example Frama-C Output using WP plugin

Table 3-10 Frama-C Pros and cons

Pros	Cons
Well Documented	Support for languages other than C
Free and open source	Installation requires many dependencies
Actively maintained	Significant level of user interaction needed
Includes static & formal analysis	
Integration of tools	

3.6.3. SVT Applicability

Frama-C contains plug-ins that do formal verification, static analysis, and test generation. This makes it a good framework for SVT to study because it has the same goals of verifying the source code of existing software. The main drawback is that it has been developed for C programs and use on other languages SVT is interested in is limited. Integrating Frama-C with SVT might be difficult because it is specific to the C language and requires user interaction to specify the plug-ins and how to run them. However, it could be a good framework to study because it utilizes a common language, ACSL, that all plug-ins use to collaborate. Similarly, SVT might want to find a common intermediate language for formal verification and static analysis tools to work on.

3.6.4. Final Thoughts

Frama-C is a powerful platform for analyzing and verifying the source code of software written in C. It combines many static and dynamic analysis tools and creates an environment for them to build upon the results computed by each tool. The usage of the E-ASCL as an annotation based system for defining contracts for functions can provide a blueprint for future implementations in other languages. However, it is specialized for the C language and might be difficult to integrate with SVT.

Table 3-11 Frama-C Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Excellent	Neutral	Easy	Very Good	Excellent	Excellent	Poor

3.7. SonarQube

SonarQube is an open-source quality management tool that uses static and dynamic analysis tools to evaluate source code. It is developed by Sonar Source for continuous inspection of code quality to provide a detailed report of bugs, code smells, vulnerabilities, and code duplications. It has

support for 29 languages through built-in rulesets and can also be extended with various plugins. Documentation and more information about SonarQube can be found at [35]. The SonarQube GitHub is actively maintained at [36].

3.7.1. Description

SonarQube can integrate with existing workflows to enable continuous code inspection across project branches and pull requests.

Analyzing code starts with installing and configuring the SonarQube scanner that can either run on the build or as part of the CI pipeline. The scanner performs a scan whenever build process is triggered. SonarQube evaluates code against a set of rules that are called quality profiles, which can be uniquely configured or set to global defaults. Severity levels show how significant the rule that is broken is, and fixes are provided for each issue.

SonarQube grades code by a set of criteria called quality gates which can also be configured uniquely or set to the global defaults. SonarQube translates these non-descript metrics and statistics about the code into business values including risk and technical debt. This helps create a common standard for coding best practices.

The SonarQube architecture includes a server, a scanner, and the source code. The scanner runs on the source code, which contains a property file that helps the scanner understand what language it is. After reading, the scanner travels to the server to retrieve the rules and then runs those rules on the source code and generates a report which is sent to the database. This report from the database is shown to the web dashboard on the server.

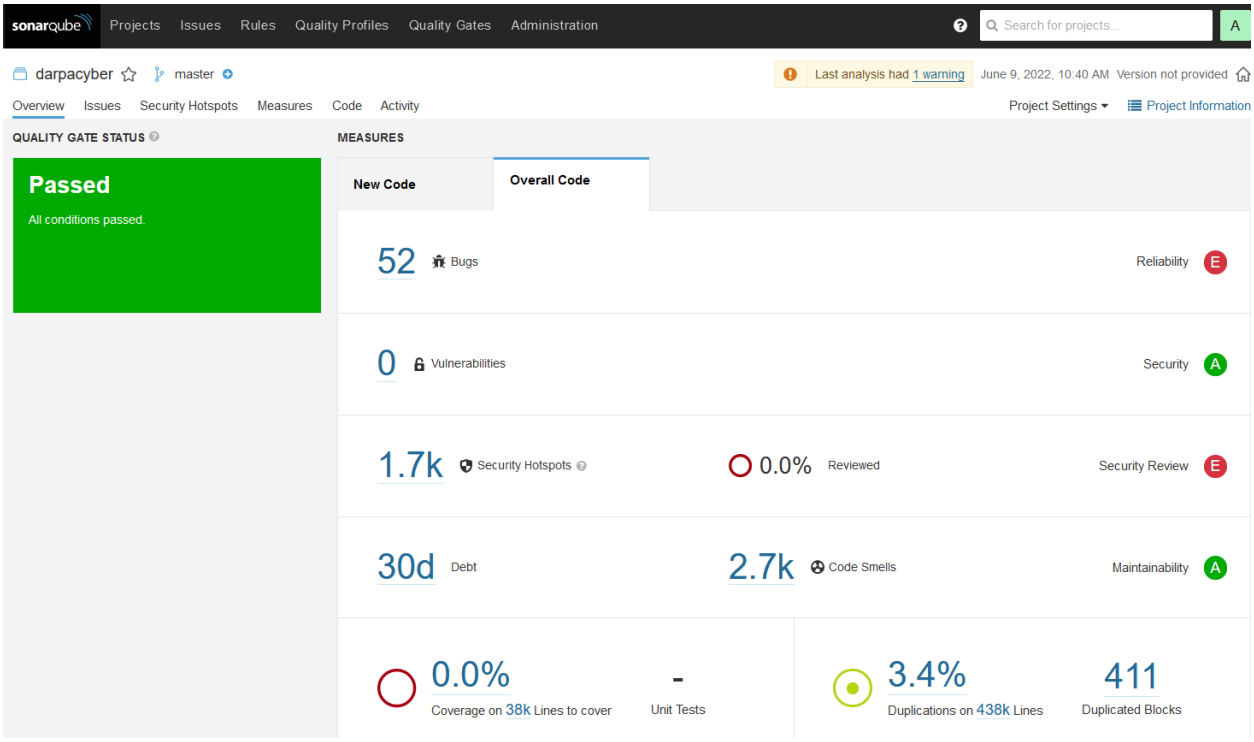
SonarQube is developed in Java, and a runtime environment needs to be installed from SonarSource. It is powered by plug-ins to extend functionality. Installation can be done from a zip file or a docker image. SonarQube is started from the command line, and once an instance is up and running a user can login to access the web dashboard. The scanner can be configured and run from the command line in the project directory, and the results are shown at the webpage. There is also an option to integrate with GitLab, Azure DevOps, and other platforms to automatically scan anytime a build process runs.

3.7.2. Examples

The example in Figure 3-2 shows the SonarQube GUI after running the scanner using the default settings on the DARPA cyber challenge repository. While the entire repository passed, there are still issues that need to be resolved. A programmer can see the issues in different categories on display along with an estimate of how much time it would take to resolve everything. In this case, SonarQube estimates 30 days of technical debt to resolve all possible issues.

SonarQube assigns each issue a severity of info, minor, major, critical, or blocker. As programmers work to resolve these issues, they can indicate whether the issue was a false positive. An example of the display for the bugs is shown in Figure 3-3. SonarQube also explains

Figure 3-2 SonarQube Example



why it is an issue, and what category it falls into (OWASP Top 10, CWE, etc.). This helps a programmer understand why SonarQube detected the issue and how to resolve it.

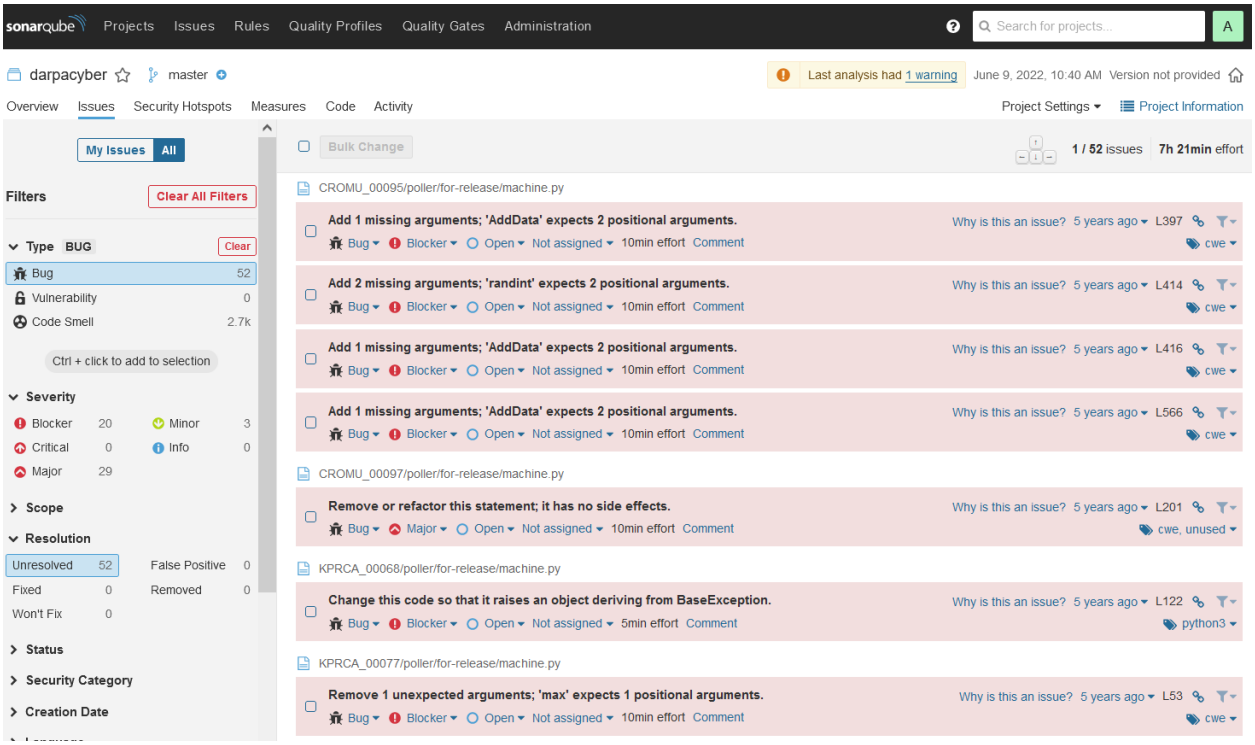
3.7.3. SVT Applicability

SonarQube has the ability to analyze all of the languages that SVT is interested in. It supports these languages through built-in rulesets, and these rulesets can be modified for any given project. To integrate with SVT, we would have to set up a SonarQube server to run on the software project being tested and extract the results that get stored in the database on this server. SonarQube supports many options for databases including MYSQL, H2, PS, and Oracle which makes it flexible for integration. The coverage report is stored as a .xml file and can be converted as we need to compare with other tools and build up the analysis. The main drawback is the level of interaction needed to specify the properties file and create a ruleset for each project. However, the language support and analysis provided would be useful for SVT to help create a standard for coding best practices.

3.7.4. Final Thoughts

SonarQube is a widely used tool for static code analysis and creating common coding standards within organizations. It can be integrated into the development process for continuous analysis to detect, report, and help resolve errors. It supports over 25 languages, including all the ones that

Figure 3-3 SonarQube Issues Display Example



SVT is interested. SonarQube appears to be the most common tool used in industry for static code analysis, so it is well documented and actively maintained. SVT could utilize SonarQube as one of its static analysis tools.

Table 3-13 SonarQube Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Excellent	Easy	Very Easy	Very Good	Excellent	Excellent	Excellent

Table 3-12 SonarQube Pros and cons

Pros	Cons
Language support	Significant level of user interaction needed
Well documented	
Actively maintained	
Integration of tools	

4. TEST GENERATION

Testing can come in many forms: Unit Tests, Component Tests, Integration Tests, Smoke Tests, Acceptance Tests, and Soak Tests just to name a few. Normally, tests are developed by software engineers in order to satisfy the requirements of a specific task or feature implementation. Coverage tools exist to assist in ensuring complete test coverage, yet these tools aren't always excellent at ensuring that specific functionality is stressed. Because developers can miss these edge cases, it may be possible to more effectively create test cases that stress a given set of software automatically using static software analysis methods. These test generation tools scan possible entry points and then create inputs that attempt to exhaust all possible use case scenarios. The goal is to ensure that the software functions correctly using these generated tests. The following sections describe various forms of test generation tools, from property based testing generators to automatic static analysis based test generators.

4.1. jqwik

jqwik is an alternative test engine for the JUnit 5 platform. That means that you can use it either stand-alone or combine it with any other JUnit 5 engine. The main purpose of jqwik is to bring PBT to the JVM. [17].

4.1.1. Description

Property-Based Testing tries to combine the intuitiveness of Microtests with the effectiveness of randomized, generated test data. A property is supposed to describe a generic invariant or post condition of your code, given some precondition. By using a few annotations jqwik tries to make it as simple as possible for programmers to write and run Properties. [17].

4.1.2. Examples

Listing 61 demonstrates how a property can be used to test functionality in code. These examples fail for the following reasons:

- 2's complement of the max negative number when converted to a positive number comes back around to itself and is negative Listing 62
- If a string is empty then the resultant length is that of the first string which is not greater than itself Listing 63

```
class PropertyBasedTests {

  @Property
  boolean absoluteValueOfAllNumbersIsPositive(@ForAll int anInteger) {
    return Math.abs(anInteger) >= 0;
  }

  @Property
  void lengthOfConcatenatedStringIsGreaterThanLengthOfEach(
      @ForAll String string1, @ForAll String string2) {
    String conc = string1 + string2;
    Assertions.assertThat(conc.length()).isGreaterThan(string1.length());
    Assertions.assertThat(conc.length()).isGreaterThan(string2.length());
  }
}
```

61 Java PDT Example

```
                                ├──────────────────jqwik──────────────────
tries = 103                     | # of calls to property
checks = 103                    | # of not rejected calls
generation = RANDOMIZED         | parameters are randomly generated
after-failure = PREVIOUS_SEED   | use the previous seed
when-fixed-seed = ALLOW         | fixing the random seed is allowed
edge-cases#mode = MIXIN         | edge cases are mixed in
edge-cases#total = 9            | # of all combined edge cases
edge-cases#tried = 4            | # of edge cases tried in current run
seed = 6967448925491521030      | random seed to reproduce generated values

Sample
------
  anInteger: -2147483648
```

62 jqwik Example Test Failure - 2's compliement

```
                              ├──────────────────jqwik──────────────────
tries = 5                     | # of calls to property
checks = 5                    | # of not rejected calls
generation = RANDOMIZED       | parameters are randomly generated
after-failure = PREVIOUS_SEED | use the previous seed
when-fixed-seed = ALLOW       | fixing the random seed is allowed
edge-cases#mode = MIXIN       | edge cases are mixed in
edge-cases#total = 9          | # of all combined edge cases
edge-cases#tried = 0          | # of edge cases tried in current run
seed = -4015032654156306469   | random seed to reproduce generated values

Shrunk Sample (1 steps)
-----------------------
  string1: ""
  string2: ""

Original Sample
---------------
  string1: "This is a test string test"
  string2: ""

  Original Error
  --------------
  java.lang.AssertionError:
    Expecting actual:
      28
    to be greater than:
      28
```

63 jqwik Example Test Failure - String Concatenation

4.1.3. SVT Applicability

jqwik allows developers to annotate their Java methods and classes with properties to define conditions that must be upheld during test generation. On its own, this requires a fair bit of developer interaction. In order for the developer to implement useful properties the developer must have an intuition or understanding of what the software is doing. This makes the process of annotating the software manual and somewhat intensive and error prone. That said, the contractual definition of the properties ensures that the functions are well documented and truly well defined in their use cases. Directly, jqwik is not particularly useful for inclusion in SVT, though it may be possible to consider a more language agnostic property based test generation system for future study.

Table 4-1 jqwik Pros and Cons

Pros	Cons
Good way to introduce PBT	Requires in-depth knowledge of system
Easy to install with good documentation	Requires source code modification
Developer interaction producing better code	Creates tests that must be run manually via CI/CD
	Only works in Java

4.1.4. Final Thoughts

Jqwik does implement powerful property based testing tools, as it allows developers to annotate functions and classes with property descriptors which are then validated and upheld through generated test cases. However, the coverage of these tests is suspect in certain instances, as it still requires in-depth developer knowledge of the contract a specific function or class is required to uphold. Property based testing has potential, as it should be possible to generate test cases from similar annotations used to apply formal verification conditions.

Table 4-2 jqwik Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Excellent	Very Easy	Easy	Good	Good	Good	Poor

4.2. Pynguin

Pynguin is an open-source Python library designed to automatically generate unit test cases for Python modules. The code can be found at [34].

4.2.1. Description

Given a provided Python module, Pynguin attempts to automatically generate unit tests that maximize code coverage [25, 24]. It is inspired by the EvoSuite [12] and Randoop [32] test generation tools for Java. To accomplish the automated test generation task, Pynguin utilizes a feedback-directed random approach. To construct a test case, Pynguin first selects a function in the provided module to test and then fulfills the arguments of the involved function(s) in a recursive, random fashion. For test input generation, a search optimization algorithm is used to try and maximize the line or branch coverage of the function to be tested; a variety of well-established search algorithms are implemented.

Pynguin utilizes the Python module `coverage.py` to compute code coverage, which is used for the search-based optimization that attempts to achieve 100% coverage. Pynguin continues executing the optimization process until a desired stopping criterion is achieved (e.g., full line coverage, maximum time exceeded, etc.) [24]. One issue with this approach is that Pynguin does not recognize the ability of `coverage.py` to skip lines during the code coverage computation, which can adversely affect the computation and resulting optimization.

4.2.2. Examples

The following is an example of a single generated test case by Pynguin for a simple Python function that returns the maximum of two integer inputs. Figure 4-1 shows the implementation of the Python function to be tested, while Figure 4-2 shows a single test case generated by Pynguin.

```
1 def max(a: int, b: int) -> int:
2     if (a > b):
3         return a
4     return b
```

Figure 4-1 Python function to determine maximum of two integers

Interestingly, the generated test case does in fact determine a valid and useful test case for the maximum function. However, Pynguin also generates a lot of unnecessary code for this very simple function to test (i.e., the continued call to the `main` function). It is unclear how much unnecessary code would be generated for more complex codebases. Additionally, although the test case is valid, Pynguin is incapable of verifying code (i.e., determining an implementation is correct) as it has no knowledge of the function/module to be tested.

Unfortunately, Pynguin does not provide functionality to ignore certain segments or functions in the module, which affects the code coverage calculation and can thereby unnecessarily affect the stopping criterion. This can result in Pynguin continuing to execute the test case generation much longer than necessary.

```
def test_case_0():
    int_0 = -872
    var_0 = module_0.main()
    var_1 = module_0.main()
    int_1 = 203
    var_2 = module_0.main()
    int_2 = module_0.max(int_0, int_1)
    assert int_2 == 203
    var_3 = module_0.main()
    var_4 = module_0.main()
```

Figure 4-2 Pynguin generated test case for maximum function

4.2.3. SVT Applicability

In the context of SVT, Pynguin has limited applicability. Although Pynguin is fairly easy-to-use and can automatically generate valid test cases, it does not provide any indication of the correctness of an implementation. Additionally, it is unclear how useful the generated test cases are in relation to test cases that would be implemented by an experienced developer. Finally, the limitation to a single language also limits its applicability.

Table 4-3 Pynguin Pros and cons

Pros	Cons
Generates test cases automatically	Only applicable to Python code
Provides simple command-line interface for ease-of-use	Cannot ignore lines for code coverage computation
Supports output formats for popular Python unit tests frameworks	Generates unnecessary test code
	Needs type information for best results
	Requires Python 3.8 or above

4.2.4. Final Thoughts

Although Pynguin is an interesting tool that is capable of generating valid test cases for Python modules, it is unclear how practical Pynguin would actually be for automatically verifying complex, real-world codebases. The automatically generated tests are fairly contrived in their implementation. Additionally, Pynguin itself requires a lot of additional setup to ensure that function parameters and return values are represented correctly. The idea of statically analyzing software and then generating interesting and useful test cases is a tempting proposition, but Pynguin's implementation fails to inspire confidence in the generated solutions.

Table 4-4 Pynguin Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Good	Easy	Neutral	Fair	Fair	Fair	Poor

```
1 java.util.Collections
2 java.util.TreeSet
```

64 File Describing Classes Tested by Randoop

4.3. Randoop

Randoop is a Java utility that will automatically test publicly facing functions and create unit tests for those functions. In addition to the unit tests, regression tests can also be created to verify that new implementations maintain backwards compatability.

4.3.1. Description

Randoop has many operational modes for testing, which include: individual methods in a class, all methods in a class, a list of classes, all the classes in a specific jar, and all jars inside a directory. Randoop looks at the available functions in a class and then attempts to break them and reports any successful attempts to the user. Randoop can take a failure case and then attempt to determine if there are similar error cases or if the first case can be narrowed down in scope to find the root cause. A regression test can be generated that can be used to determine if any changes in code revert to a past error case as well as making sure that new versions are free of previous errors. While Randoop does create JUnit tests, the tests aren't meant to replace unit tests that a developer would typically create and are meant to supplement the test coverage that already exists.

4.3.2. Examples

The first example showcases Randoop's capability at generating tests for the JRE classes: *Collections* and *TreeSet*. Listings 64 and 65 showcase the setup and the command utilized to execute Randoop's test generation capabilities. Listing 66 showcases the output of running that command. From the output we can see that Randoop generated around 2348 different sets of test inputs during the 60 second timeframe given to generate tests for the classes, and also that there were no failing input test cases generated. Additionally multiple regression tests files were generated, all of which contained JUnit test cases that could be run to ensure compatibility with the current API definition.

```
1 java -classpath ${RANDOOP_JAR} randoop.main.Main gentests --classlist=classesTested.txt --time-limit=60
```

65 randoop Java Collections Command

```
Randoop for Java version 4.2.6.

Will try to generate tests for 2 classes.
PUBLIC MEMBERS=113
Explorer = ForwardGenerator(steps: 0, null steps: 0, num_sequences_generated: 0;
    allSequences: 0, regresson seqs: 0, error seqs: 0=0=0, invalid seqs: 0, subsumed_sequences: 0,
        num_failed_output_test: 0;
    runtimePrimitivesSeen:38)

Progress update: steps=1, test inputs generated=0, failing inputs=0       (2022-05-18T22:08:16.805599Z
        11.2M used)
Progress update: steps=1000, test inputs generated=729, failing inputs=0       (2022-05-18T22
    :08:36.023727Z       77.5M used)
Progress update: steps=2000, test inputs generated=1434, failing inputs=0       (2022-05-18T22
    :08:53.162839Z       202M used)
Progress update: steps=3000, test inputs generated=2066, failing inputs=0       (2022-05-18T22
    :09:09.212944Z       150M used)
Progress update: steps=3434, test inputs generated=2348, failing inputs=0       (2022-05-18T22
    :09:16.817681Z       173M used)
Progress update: steps=3434, test inputs generated=2348, failing inputs=0       (2022-05-18T22
    :09:16.835068Z       190M used)
Normal method executions: 5357984
Exceptional method executions: 412

Average method execution time (normal termination):       0.000241
Average method execution time (exceptional termination): 0.0496
Approximate memory usage 190M
Explorer = ForwardGenerator(steps: 3434, null steps: 1086, num_sequences_generated: 2348;
    allSequences: 2348, regresson seqs: 1725, error seqs: 0=0=0, invalid seqs: 0, subsumed_sequences: 0,
        num_failed_output_test: 623;
    runtimePrimitivesSeen:44)

No error-revealing tests to output.

About to look for failing assertions in 1058 regression sequences.

Regression test output:
Regression test count: 1058
Writing regression JUnit tests...
Created file /Users/jdfost/code/RegressionTest0.java
Created file /Users/jdfost/code/RegressionTest1.java
Created file /Users/jdfost/code/RegressionTest2.java
Created file /Users/jdfost/code/RegressionTest.java
Wrote regression JUnit tests.
About to look for flaky methods.

Invalid tests generated: 0
```

66 randoop Java Collections Results

```
1 java -classpath $GMS_CODE/*:${RANDOOP_JAR} randoop.main.Main gentests --testjar=$GMS_CODE/bridged-data-
      source-interval-simulator-LATEST.jar --time-limit=60
```

67 randoop SNL Project Command

Listings 67 and 68 detail the results of running Randoop on Sandia project code. In this example, code from the GMS project bridge data simulator is utilized. This data set contains a single class with multiple functions that simulate data inputs for the GMS system. Running Randoop over the supplied jar generates only 13 test data inputs with roughly 44 total path executions. In this example again no invalid tests were generated. Ideally this would imply that there is a certain amount of satisfiability with regards to the implementation of the software. However, the generated regression scenarios are obtuse and provide minimum confidence in their actual capability to stress test multiple input values.

4.3.3. SVT Applicability

Randoop functions well as a test generation tool for Java applications. However, it is difficult to assert that the quality of the tests generated ensure that the software is validated and works as intended. Also, unlike property based testing or static analysis tools, it becomes difficult to understand when exactly an error can occur and what should truly be considered a failing test scenario. Automatically generating tests for code coverage purposes can assist in guiding the developer to understand what exactly certain test case scenarios need to be considered, but is poor at ensuring software is functioning correctly. This is especially so for Randoop because the generated test cases are difficult to read and understand: more time may be spent deciphering the generated tests to see if they are correct than it would take to simply write the tests manually. Because of this, Randoop is not recommended for inclusion in SVT.

Table 4-5 Randoop Pros and Cons

Pros	Cons
Creates tests for code automatically, reducing developer work	Setup to scan large projects complicated
Generates regression tests for future proofing an API	Generated tests are difficult to decipher
Simple to execute over singular files	Minimal confidence in the generated test's capability to validate the software

4.3.4. Final Thoughts

Randoop is a tool that attempts to automatically generate unit tests in a smart manner by analyzing function prototypes and by using static parsing techniques. However, the tests that

```
1  Randoop for Java version 4.2.6.
2  WARNING: sun.reflect.Reflection.getCallerClass is not supported. This will impact performance.
3
4  Will try to generate tests for 1 classes.
5  PUBLIC MEMBERS=8
6  Explorer = ForwardGenerator(steps: 0, null steps: 0, num_sequences_generated: 0;
7      allSequences: 0, regresson seqs: 0, error seqs: 0=0=0, invalid seqs: 0, subsumed_sequences: 0,
           num_failed_output_test: 0;
8      runtimePrimitivesSeen:38)
9
10 Progress update: steps=1, test inputs generated=0, failing inputs=0       (2022-05-18T22:40:34.471969Z
       23.9M used)
11 Progress update: steps=1000, test inputs generated=13, failing inputs=0       (2022-05-18T22:40:35.295052
       Z       22.6M used)
12 Progress update: steps=2000, test inputs generated=13, failing inputs=0       (2022-05-18T22:40:35.340118
       Z       32.5M used)
13 Progress update: steps=3000, test inputs generated=13, failing inputs=0       (2022-05-18T22:40:35.377678
       Z       43.0M used)
14 Progress update: steps=5230000, test inputs generated=13, failing inputs=0       (2022-05-18T22
       :41:34.452056Z       147M used)
15 Progress update: steps=5231000, test inputs generated=13, failing inputs=0       (2022-05-18T22
       :41:34.464398Z       156M used)
16 Progress update: steps=5231545, test inputs generated=13, failing inputs=0       (2022-05-18T22
       :41:34.471045Z       160M used)
17 Normal method executions: 35
18 Exceptional method executions: 9
19
20 Average method execution time (normal termination):       0.0332
21 Average method execution time (exceptional termination): 0.138
22 Approximate memory usage 160M
23 Explorer = ForwardGenerator(steps: 5231545, null steps: 5231532, num_sequences_generated: 13;
24     allSequences: 13, regresson seqs: 6, error seqs: 0=0=0, invalid seqs: 0, subsumed_sequences: 0,
           num_failed_output_test: 7;
25     runtimePrimitivesSeen:38)
26
27 No error-revealing tests to output.
28
29 About to look for failing assertions in 5 regression sequences.
30
31 Regression test output:
32 Regression test count: 5
33 Writing regression JUnit tests...
34 Created file /Users/jdfost/code/RegressionTest0.java
35 Created file /Users/jdfost/code/RegressionTest.java
36 Wrote regression JUnit tests.
37 About to look for flaky methods.
```

68 randoop SNL Project Results

Randoop generates are difficult to read and understand, and do little to assuage a developer's understanding that a given piece of software is functionally correct and without error. From the perspective of automated test generation, a common problem appears to be creation of a minimal set of unique and interesting test cases to ensure full code coverage. If automated testing tools could create readable, understandable, and functionally minimal sets of test cases to optimize code coverage then that would be more useful than a randomized value approach. This would require more introspection into the existing software, however, which may be incredibly difficult to perform without hints or properties describing the parameters. If anything it may be a good idea to look at automatic test generation from a different perspective. Instead of generating tests from existing code, it may be useful to generate tests pre-emptively by supplying an API with conditions or properties that the API should uphold. This would allow developers to create API definitions with generated test cases, facilitating test driven development.

Table 4-6 Randoop Summarization.

Documentation	Installation	Learning Curve	Usability	Robustness	Validation	Languages
Very Good	Neutral	Easy	Good	Good	Good	Poor

5. SUMMARY

Throughout the course of this report, we have explored a multitude of different formal verification, static analysis, and test generation tools. In the course of these analyses, we have crafted simple, yet usable examples showcasing how each tool could be used. We have also analyzed each tool with respect to its utilization in SVT proper and created a general summarization table for each tool which can be found at the end of each section. Through our general analyses we have come across a variety of software verification solutions.

The majority of the formal verification solutions revolve around the creation of functional based mathematical implementations. These solutions are implemented in a tool specific language, usually of a functional type (SML, OCaml, Isar, etc.), and have no direct mapping to existing software. In these cases, it is on the onus of the developer to ensure an implementation matches the functional proof of correctness. There are also certain solutions that attempt to generate code from a written formal proof. Isabelle/HOL and TLA+ have some functionality to generate Haskell/Scala and Python code from proofs. Additionally, there are some tools that take the opposite route and generate formal language specific proofs from existing software in a specific language. Verifiable C and AutoCorres are two such tools. Then, there are some annotation or injection based tools, such as Z3, which attempt to formally verify conditions about software using custom parsers and compilers. All of these methods, while powerful, require a lot of work on the part of the user to implement correctly.

The static analysis solutions, in general, are much more similar with regards to their implementation details. The majority of these solutions take an input software dataset and generate an IR upon which the static analysis software then operates. These tools parse the IR and can intelligently determine when arrays can go out of bounds, when values may overflow, and null pointers can be dereferenced. The main differences in the tools are with regards to how the reporting is generated, and the accuracy/capability of the tool at detecting software issues. Certain static analysis tools border on implementing formal verification type capabilities through the usage of annotations. These tools include Frama-C, Vercors, and CBMC. Overall, each static analysis tool is robust and performs well at detecting simple software issues. Using the more complex features to verify software contracts, however, takes as much effort as utilizing a formal verification tool.

Test generation tools are fairly similar to their static analysis counterparts in that they parse code and then generate test cases in an intelligent way. Some tools attempt to automatically parse software without any additional help, whereas other tools require the user to create properties to describe various conditions which a given component should uphold. In general, the property based testing tools are more robust than the automatic test generation tools. It is also generally fairly straightforward to create properties for functions and components, but does require knowledge of the contract for a given section of code.

In terms of validating software, static analysis tools are very useful at finding simple bugs, and are already widely employed by existing projects in CI/CD pipelines. Test generation tools fill a valid niche, but in general static analysis tools accomplish solving the majority of the issues that automated test generation can solve. With regards to verifying a function actually succeeds at

fulfilling a contract, however, formal verification is required. In this report we have explored a wide variety of formal verification tools, but they all have similar issues revolving around usability, complexity, and understandability. There are some tools that are easier to use than others, and with better documentation than others, but they all have similar pitfalls. It may be difficult to create, but there exist the potential to create a tool to improve the usability of formal method contract writing.

6. MOVING FORWARD

All the tools analyzed above have caveats on usability with regards to the languages and the versions of those languages they support. For example, many of the C++ based tools simply don't work with C++ versions greater than 11. This is simply unacceptable; formal verification and static analysis cannot be relegated to old languages and software systems as newer languages include performance, readability, and security features that should be prioritized for inclusion in modern software systems. Something that few, if any, verification tools utilize is an abstract software representation. Utilizing an IR instead of a direct software compiler/interpreter has the potential to open up many possibilities with regards to agnostically verifying software.

Of all the methods and systems analyzed, we believe that the usage of an annotation based system is one of the more robust, usable, and descriptive ways to describe a software contract. Annotations provide developers with the capability to link the function contract with the software it lives with, connecting documentation, contract, and source code in a single unit. Additionally, the utilization of an annotation based system has the potential to be language agnostic with regards to the actual properties being validated, as a function contract in one language should hold in another language.

For any future work undertaken to improve the landscape of software verification, we would recommend creating a tool that utilizes an IR combined with an annotation based system to verify software. The creation of such a tool would give developers powerful abilities to verify properties or contracts of function level software components in a language agnostic way using a single typed interface.

REFERENCES

[1] Lennart Appel, Andrew W. Beringer and Qinxiang Cao. *Software Foundations Volume 5: Verifiable C*. 2020.

[2] Lennart Appel, Andrew W. with Beringer, Qinxiang Cao, and Josiah Dodds. Verifiable c: Applying the verified software toolchain to c programs. Technical report, Princeton University, 05 2022.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[4] Cambridge, Bell Labs. Hol. `https://hol-theorem-prover.org/`, 2022. Last accessed on 2022-08-09.

[5] Carnegie Mellon, Daniel Kröning. Cbmc homepage. `http://www.cprover.org/cbmc/`, 2022. Last accessed on 2022-08-09.

[6] Clang Compiler. Clang compiler. `https://clang.llvm.org/`, 2022. Last accessed on 2022-08-09.

[7] David Basin, Cas Cremers, Jannik Dreier, Simon Meier, Ralf Sasse, Benedikt Schmidt. Tamarin prover. `https://tamarin-prover.github.io/`, 2022. Last accessed on 2022-08-09.

[8] David Greenaway, Japheth Lim, June Andronick, Rohan Ben Jacob Rao. Autocorres home page. `https://trustworthy.systems/projects/TS/autocorres/`, 2022. Last accessed on 2022-08-09.

[9] Facebook Infer. About infer. `https://fbinfer.com/docs/about-Infer`, 2022. Last accessed on 2022-05-31.

[10] Facebook Infer. Separation logic and bi-abduction. `https://fbinfer.com/docs/separation-logic-and-bi-abduction`, 2022. Last accessed on 2022-05-31.

[11] Frama C. Frama c. `https://git.frama-c.com/pub/frama-c`, 2022. Last accessed on 2022-08-09.

[12] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[13] Georges Gonthier. Formal proof - the four color theorem. *Notices of the American Mathematical Society*, 55:1382–1393, 2008.

[14] Isabelle. Isabelle. `https://isabelle.in.tum.de/`, 2022. Last accessed on 2022-08-09.

[15] Lawrence C. Paulson Jasmin Blanchette, Martin Descharnais. *Hammering Away, A User's Guild to Sledgehammer for Isabelle/HOL.*

[16] Jeremy Avigad, Leonardo de Moura, Soonho Kong and Sebastian Ullrich. *Theorem Proving in Lean 4*.

[17] jqwik. jqwik. https://menloservice.sandia.gov/https://jqwik.net/, 2022. Last accessed on 2022-08-09.

[18] K Framework. K framework. https://kframework.org/, 2022. Last accessed on 2022-08-09.

[19] Lean. Lean: About. https://leanprover.github.io/about/, 2022. Last accessed on 2022-08-18.

[20] Lean. Lean: Frequently asked questions. https://github.com/leanprover/lean/blob/master/doc/faq.md, 2022. Last accessed on 2022-08-18.

[21] Lean. Lean: Tutorial. https://leanprover.github.io/introduction_to_lean/, 2022. Last accessed on 2022-08-18.

[22] Leslie Lamport. Tla+ home page. https://lamport.azurewebsites.net/tla/tla.html, 2022. Last accessed on 2022-08-09.

[23] LLVM. Llvm compiler infrastructure. https://llvm.org/docs/LangRef.html, 2022. Last accessed on 2022-08-09.

[24] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. *CoRR*, abs/2202.05218, 2022.

[25] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. Automated unit test generation for python. In *Proceedings of the 12th Symposium on Search-based Software Engineering (SSBSE 2020, Bari, Italy, October 7–8)*, volume 12420 of *Lecture Notes in Computer Science*, pages 9–24. Springer, 2020.

[26] Martin Brain, Peter Schrammel. Cprover manual. http://cprover.diffblue.com/cbmc-architecture.html, 2022. Last accessed on 2022-08-09.

[27] Konrad Slind Michael Norrish. *HOL Kananaskis-13 Help Documents*.

[28] Konrad Slind Michael Norrish. *The HOL System REFERENCE*.

[29] Michael Tautschnig, Peter Schrammel. Cbmc github url. https://github.com/diffblue/cbmc, 2022. Last accessed on 2022-08-09.

[30] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig. Code-level model checking in the software development workflow. In IEEE, editor, *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2020.

[31] Nikolaj Bjorner, Leonardo de Moura, Lev Nachmanson, Christoph Wintersteiger. Programming z3. http://theory.stanford.edu/~nikolaj/programmingz3.html, 2022. Last accessed on 2022-08-09.

[32] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, page 815–816, New York, NY, USA, 2007. Association for Computing Machinery.

[33] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015.

[34] S. Lukasczyk, G. Fraser. Pynguin. https://menloservice.sandia.gov/https://www.pynguin.eu/, 2022. Last accessed on 2022-08-09.

[35] SonarQube. Sonarqube. https://www.sonarqube.org/, 2022. Last accessed on 2022-08-09.

[36] SonarQube. Sonarqube. https://github.com/SonarSource/sonarqube, 2022. Last accessed on 2022-08-09.

[37] Terence Parr. Antlr. https://www.antlr.org/, 2022. Last accessed on 2022-08-09.

[38] VerCors. Vercors. https://vercors.ewi.utwente.nl/wiki, 2022. Last accessed on 2022-08-09.

[39] Viper. Viper. https://www.pm.inf.ethz.ch/research/viper.html, 2022. Last accessed on 2022-08-09.

[40] z3 git. z3 git. https://github.com/Z3Prover/z3, 2022. Last accessed on 2022-08-09.

APPENDIX A. SANDIA DEVELOPMENT

Formal software verification is not a completely unknown topic here at Sandia. There are other divisions at Sandia that have a focus on the validation of software that we pulled knowledge from on the creation of this report. These divisions include:

- Division 08740, contact Jon Aytac
- Division 05550, contact Kirk Landin

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Tu-Thach Quach	9364	tong@sandia.gov
Nick Davis	9364	nadavi@andia.gov
Technical Library	1911	sanddocs@sandia.gov

Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.