Cartesian Hoare Logic for Verifying k-Safety Properties *

Marcelo Sousa
University of Oxford, UK
marcelo.sousa@cs.ox.ac.uk

Isil Dillig
The University of Texas at Austin, USA
isil@cs.utexas.edu

Abstract

Unlike safety properties which require the absence of a "bad" program trace, k-safety properties stipulate the absence of a "bad" interaction between k traces. Examples of k-safety properties include transitivity, associativity, anti-symmetry, and monotonicity. This paper presents a sound and relatively complete calculus, called *Cartesian Hoare Logic (CHL)*, for verifying k-safety properties. Our program logic is designed with automation and scalability in mind, allowing us to formulate a verification algorithm that automates reasoning in CHL. We have implemented our verification algorithm in a fully automated tool called DESCARTES, which can be used to analyze any k-safety property of Java programs. We have used DESCARTES to analyze user-defined relational operators and demonstrate that DESCARTES is effective at verifying (or finding violations of) multiple k-safety properties.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Relational hoare logic; safety hyper-properties; product programs; automated verification

1. Introduction

Following the success of Hoare logic as a formal system to verify program correctness, many tools can prove safety properties expressed as pre- and post-conditions. That is, given a Hoare triple $\{\Phi\}\ S\ \{\Psi\}$, program verifiers establish that terminating executions of S on inputs satisfying Φ produce outputs consistent with Ψ. Hence, a valid Hoare triple states which input-output pairs are feasible in a *single*, but arbitrary, execution of S.

However, some important functional correctness properties require reasoning about the relationship between *multiple* program executions. As a simple example, consider determinism, which requires $\forall x, y.\ x = y \Rightarrow f(x) = f(y)$. This property is *not* a standard safety property because it cannot be violated by any individual execution trace. Instead, determinism is a so-called *2-safety hyperproperty* [10] since we need two execution traces to establish its violation. In general, a *k-safety (hyper-)property* requires reasoning about relationships between k different execution traces.

Even though there has been some work on verifying 2-safety properties in the context of secure information flow [1, 3, 23] and security protocols of probabilistic systems [2, 4], we are not aware of any general-purpose program logics that allow verifying k-safety properties for arbitrary values of k. Furthermore, even for $k = 2$, existing tools do not support a high degree of automation.

This paper argues that many important functional correctness properties are k-safety properties for $k \geq 2$, and we present *Cartesian Hoare Logic (CHL)* for verifying general k-safety specifications. [1] Our new program logic for k-safety is sound, relatively complete, and has been designed to be easy to automate: We have built a verification tool called DESCARTES that fully automates reasoning in CHL and successfully used it to analyze k-safety requirements of user-defined relational operators in Java programs.

Motivating Example. To motivate the need for verifying k-safety properties, consider the widely used `Comparator` interface in Java. By implementing the `compare` method of this interface, programmers can define an ordering between objects of a given type. Unfortunately, writing a correct `compare` method is notoriously hard, as any valid comparator must satisfy three different k-safety properties:

- **P1**: $\forall x, y.\ \mathit{sgn}(\mathit{compare}(x, y)) = -\mathit{sgn}(\mathit{compare}(y, x))$
- **P2**: $\forall x, y, z.\ (\mathit{compare}(x, y) > 0 \wedge \mathit{compare}(y, z) > 0) \Rightarrow \mathit{compare}(x, z) > 0$

* This work was supported in part by NSF Award #1453386 and AFRL Awards # 8750-14-2-0270 and 8750-15-2-0096.

[1] We call our calculus *Cartesian Hoare Logic* because it allows reasoning about the cartesian product of sets of input-output pairs from different runs.

- **P3**: $\forall x, y, z.\ (\mathit{compare}(x, y) = 0 \Rightarrow (\mathit{sgn}(\mathit{compare}(x, z)) = \mathit{sgn}(\mathit{compare}(y, z))))$

Among these properties, P1 is a 2-safety property, while P2 and P3 are 3-safety properties. For instance, property P2 corresponds to transitivity, which can only be violated by a collection of three different runs of `compare`: Specifically, two runs where `compare(a,b)` and `compare(b,c)` both return positive values, and a third run `compare(a,c)` that returns zero or a negative value.

To demonstrate that implementing comparators can be hard, consider the `compare` method shown in Figure 1, which is taken verbatim from a Stackoverflow post. This method is used for sorting poker hands according to their strength and represents hands as strings of 13 characters. In particular, the occurrence of number n at position k of the string indicates that the hand contains n cards of type k. Unfortunately, this method has a logical error when one of the hands is a full house and the other one has 3 cards of a kind; it therefore ends up violating properties P2 and P3. As a result, programs using this comparator either suffer from run-time exceptions or lose elements inserted into collections.

Cartesian Hoare Logic. As illustrated by this example, analyzing k-safety properties is extremely relevant for ensuring software correctness; yet there are no general purpose automated tools for analyzing k-safety properties. This paper aims to rectify this situation by proposing a general framework for specifying and verifying general k-safety properties.

In our approach, k-safety properties are specified using *Cartesian Hoare triples* of the form $\|\Phi\|\ S\ \|\Psi\|$, where Φ and Ψ are first-order formulas that relate k different program runs. For instance, we can express property P2 from above using the following Cartesian Hoare triple:

$$\begin{gathered}\|y_1 = x_2 \wedge x_1 = x_3 \wedge y_2 = y_3\| \\ \texttt{compare}(x, y)\{...\} \\ \|\mathit{ret}_1 > 0 \wedge \mathit{ret}_2 > 0 \Rightarrow \mathit{ret}_3 > 0\|\end{gathered}$$

Here, a variable named v_i refers to the value of program variable v in the i'th program execution. Hence, the precondition Φ states that we are only interested in triples of executions (π_1, π_2, π_3) where (i) the value of y in π_1 is equal to the value of x in π_2, (ii) the values of x in π_1 and π_3 are the same, and (iii) the values of y are the same in π_2 and π_3. The postcondition Ψ says that, if `compare` returns a positive value in both π_1 and π_2, then it should also be positive in π_3.

To verify such Cartesian Hoare triples, our new program logic reasons about the Cartesian product of sets of input-output pairs from k program runs. Specifically, if Σ represents the set of all input-output pairs of code S, then the judgment $\vdash \|\Phi\|\ S\ \|\Psi\|$ is derivable in our calculus iff every k-tuple in Σ^k that satisfies Φ also satisfies Ψ.

The key idea underlying our approach is to reduce the verification of a Cartesian Hoare triple to a *standard* Hoare triple that can be easily verified. Specifically, our calculus derives judgments of the form $\langle\Phi\rangle\ (S_1 \circledast \ldots \circledast S_k)\ \langle\Psi\rangle$ where $S_1 \circledast \ldots \circledast S_k$ represents a set of programs that simultaneously execute $S_1, \ldots, S_k$. While every pair of programs $P, P' \in (S_1 \circledast \ldots \circledast S_k)$ are semantically equivalent, our calculus exploits the fact that some of these programs are much easier to verify than others. Furthermore, and perhaps more crucially, our approach does not explicitly construct the set of programs in $(S_1 \circledast \ldots \circledast S_k)$ but reasons directly about the provability of Ψ on tuples of runs that satisfy Φ.

Contributions. To summarize, this paper makes the following key contributions:

- We argue that many natural functional correctness properties are k-safety properties, and we propose Cartesian Hoare triples for specifying them (Section 3).
- We present a sound and complete proof system called *Cartesian Hoare Logic (CHL)* for proving valid Cartesian Hoare triples (Section 4).
- We describe a practical algorithm based on CHL for automatically proving k-safety properties (Section 5).
- We use our techniques to analyze user-defined relational operators in Java and show that our tool can successfully verify (or find bugs in) such methods (Section 7).

2. Language and Preliminaries

Figure 2 presents a simple imperative language that we use for formalizing Cartesian Hoare Logic. In this language, atomic statements A include skip (denoted as $\natural$), assignments ($x := e$), array writes $x[e] := e$, and assume statements written as $\surd c$. Statements also include composition $S; S$ and conditionals $S_1 \oplus_c S_2$, which execute S_1 if c evaluates to true and S_2 otherwise. In our language, loops have the syntax $[(c_1, S_1, S_1'), \ldots, (c_n, S_n, S_n')]^*$ which is short-hand for the more convential loop construct:

```
while(true) {
  if(c_1) then S_1 else { S_1'; break; }
  ...
  if(c_n) then S_n  else { S_n'; break; }
}
```

In this paper, we choose to formalize loops with breaks rather than the simpler loop construct $\mathrm{while}(c)\ \mathrm{do}\ S$ (i.e., $[(c, S, \natural)]^*$) because break statements are ubiquitous in real programs and present a challenge for verifying k-safety.

We assume an operational semantics that is specified using judgments of the form $\sigma, e \Downarrow \mathrm{int}$ and $\sigma, S \Downarrow \sigma'$, where e is a program expression, S is a statement, and σ, σ' are valuations (stores) mapping program variables to their values. Since the operational semantics of this language is standard, we refer the interested reader to the extended version of the paper.

Definition 1. (Entailment) *Let σ be a valuation and φ a formula. We write $\sigma \models \varphi$ iff $\sigma, \varphi \Downarrow \mathrm{true}$.*

```
public int compare(String c1, String c2) {
    if (c1.indexOf('4') != -1 || c2.indexOf('4') != -1) {  // Four of a kind
        if (c1.indexOf('4') == c2.indexOf('4')) {
            for (int i = 12; i >= 0; i--) {
                if (c1.charAt(i) != '0' && c1.charAt(i) != '4') {
                    if (c2.charAt(i) != '0' && c2.charAt(i) != '4') return 0;
                    return 1;  }
                if (c2.charAt(i) != '0' && c2.charAt(i) != '4') return -1; } }
        return c1.indexOf('4') - c2.indexOf('4'); }
    int tripleCount1 = StringFunctions.countOccurrencesOf(c1, "3");
    int tripleCount2 = StringFunctions.countOccurrencesOf(c2, "3");
    if (tripleCount1 > 1 || (tripleCount1 == 1 && c1.indexOf('2') != -1) ||
       tripleCount2 > 1 || (tripleCount2 == 1 && c2.indexOf('2') != -1)) {  // Full house
        int higherTriple = c1.lastIndexOf('3');
        if (higherTriple == c2.lastIndexOf('3')) {
            for (int i = 12; i >= 0; i--) {
                if (i == higherTriple) continue;
                if (c1.charAt(i) == '2' || c1.charAt(i) == '3') {
                    if (c2.charAt(i) == '2' || c2.charAt(i) == '3') return 0;
                    return 1;  }
                if (c2.charAt(i) == '2' || c2.charAt(i) == '3') return -1; }  }
        return higherTriple - c2.lastIndexOf('3'); }
    return 0; }
```

Figure 1. A (buggy) comparator for sorting poker hands

$$
\begin{array}{lcl}
\text{Statement } S & := & A \mid S; S \mid S \oplus_c S \mid \\
 & & [(c_1, S_1, S_1'), \ldots, (c_n, S_n, S_n')]^* \\
\text{Atom } A & := & \natural \mid x := e \mid x[e] := e \mid \sqrt{c} \\
\text{Expr } e & := & \text{int} \mid x \mid x[e] \mid e \ \text{op}_a \ e \\
\text{Cond } c & := & \top \mid \bot \mid \star \mid e \ \text{op}_l \ e \mid \neg c \mid c \wedge c \mid c \vee c
\end{array}
$$

Figure 2. Language for formalization. op_a, op_l denote arithmetic and logical operators, and $\star$ represents non-deterministic choice.

Definition 2. *(***Semantic Equivalence***) Two statements S_1 and S_2 are semantically equivalent, written $S_1 \equiv S_2$, if for all valuations σ, we have $\sigma, S_1 \Downarrow \sigma'$ iff $\sigma, S_2 \Downarrow \sigma'$.*

3. Validity of Cartesian Hoare Triples

We now introduce Cartesian Hoare triples and provide use cases from real-world programming idioms to motivate the relevance and general applicability of k-safety properties.

Definition 3. (Hoare Triple) *Given statement S, a Hoare triple $\{\Phi\}\ S\ \{\Psi\}$ is valid if for all pairs of valuations (σ, σ') satisfying $\sigma \models \Phi$ and $\sigma, S \Downarrow \sigma'$, we have $\sigma' \models \Psi$.*

Cartesian Hoare triples generalize standard Hoare triples by allowing us to relate different program executions:

Definition 4. (Cartesian Hoare Triple) *Let S be a statement over variables $\vec{x}$, and let Φ, Ψ be formulas over variables $\vec{x}_1, \ldots, \vec{x}_k$. Then, the Cartesian Hoare triple $\|\Phi\|\ S\ \|\Psi\|$ is valid, written $\models \|\Phi\|\ S\ \|\Psi\|$, if for every set of valuation pairs $\{(\sigma_1, \sigma_1'), \ldots, (\sigma_k, \sigma_k')\}$ satisfying*

$$\Big(\biguplus_{1 \le i \le k} \sigma_i[\vec{x}_i/\vec{x}]\Big) \models \Phi \ \text{ and } \ \forall i \in [1,k].\ \sigma_i, S \Downarrow \sigma_i'$$

we also have: $\quad \Big(\biguplus_{1 \le i \le k} \sigma_i'[\vec{x}_i/\vec{x}]\Big) \models \Psi$

Intuitively, the validity of $\|\Phi\|\ S\ \|\Psi\|$ means that, if we run S on k inputs whose relationship is given by Φ, then the resulting outputs must respect Ψ. Hence, unlike a standard Hoare triple for which a counterexample consists of a single execution, a counterexample for a Cartesian Hoare triple includes k different executions. In the rest of the paper, we refer to the value k as the *arity* of the Cartesian Hoare triple.

Example 1. *Figure 3 shows some familiar properties and their corresponding specification. Among these Cartesian Hoare triples, transitivity and homomorphism correspond to 3-safety properties, and associativity is a 4-safety property. The remaining Cartesian Hoare triples, such as monotocity, injectivity, and idempotence, have arity 2.*

Example 2. *Figure 4 shows some Cartesian Hoare triples and indicates whether they are valid. For instance, consider the first two rows of Figure 4. For both examples, the precondition tells us to consider a pair of executions π_1, π_2 where the values of x and y are swapped (i.e., the value of x in π_1 is the value of y in π_2 and vice versa). Since the statement of interest is $z := x - y$, the value of z in π_1 will be the additive inverse of the value of z in π_2. Hence, the second Cartesian Hoare triple is valid, but the first one is not.*

3.1 Realistic Use Cases for Cartesian Hoare Triples

We now highlight some scenarios where k-safety properties are relevant in modern programming. Since it is well-known that security properties such as non-interference are 2-safety properties [3, 23], we do not include them in this discussion.

Equality. A ubiquitious programming practice is to implement a custom equality operator for a given type, for example, by overriding the `equals` method in Java. A key property of `equals` is that it must be an equivalence relation (i.e., reflexive, symmetric, and transitive). While reflexivity can be expressed as a standard Hoare triple, symmetry and transitivity are 2 and 3-safety properties respectively. Furthermore, `equals` must satisfy another 2-safety property called *consistency*, which requires multiple invocations of `x.equals(y)` to *consistently return true or false*.

Property	Specification
Monotonicity	$\|x_2 \leq x_1\|\ \mathrm{f(x)}\ \|ret_2 \leq ret_1\|$
Determinism	$\|\vec{x}_1 = \vec{x}_2\|\ \mathrm{f}(\vec{x})\ \|ret_1 = ret_2\|$
Injectivity	$\|\vec{x}_1 \neq \vec{x}_2\|\ \mathrm{f}(\vec{x})\ \|ret_1 \neq ret_2\|$
Symmetric relation	$\|x_1 = y_2 \wedge y_1 = x_2\|\ \mathrm{R(x,y)}\ \|ret_1 = ret_2\|$
Anti-symmetric relation	$\|x_1 = y_2 \wedge y_1 = x_2 \wedge x_1 \neq y_1\|\ \mathrm{R(x,y)}\ \|ret_1 = \mathit{false} \vee ret_2 = \mathit{false}\|$
Asymmetric relation	$\|x_1 = y_2 \wedge y_1 = x_2\|\ \mathrm{R(x,y)}\ \|ret_1 = \mathit{true} \Rightarrow ret_2 = \mathit{false}\|$
Transitive relation	$\|y_1 = x_2 \wedge x_1 = x_3 \wedge y_2 = y_3\|\ \mathrm{R(x,y)}\ \|(ret_1 = \mathit{true} \wedge ret_2 = \mathit{true}) \Rightarrow ret_3 = \mathit{true}\|$
Total relation	$\|x_1 = y_2 \wedge y_1 = x_2\|\ \mathrm{R(x,y)}\ \|ret_1 = \mathit{true} \vee ret_2 = \mathit{true}\|$
Associativity	$\|y_1 = x_2 \wedge y_2 = y_3 \wedge x_1 = x_4\|\ \mathrm{f(x,y)}\ \|(x_3 = ret_1 \wedge y_4 = ret_2) \Rightarrow ret_3 = ret_4\|$
Homomorphism	$\|x_3 = x_1 \cdot x_2\|\ \mathrm{f(x)}\ \|ret_3 = ret_1 \cdot ret_2\|$
Idempotence	$\|\mathit{true}\|\ \mathrm{f(x)}\ \|ret_1 = x_2 \Rightarrow ret_1 = ret_2\|$

Figure 3. Example k-safety properties and their specification

Cartesian Hoare Triple	Valid?
$\|x_1 = y_2 \wedge x_2 = y_1\|\ z := x - y\ \|z_1 = z_2\|$	✗
$\|x_1 = y_2 \wedge x_2 = y_1\|\ z := x - y\ \|z_1 = -z_2\|$	✓
$\|x_1 = x_2 \wedge y_1 = y_2\|\ z := 1 \oplus_{x>y} z := 0\ \|z_1 = z_2\|$	✓
$\|x_1 = x_2 \wedge y_1 = y_2\|\ z := 1 \oplus_{\star} z := 0\ \|z_1 = z_2\|$	✗
$\|x_1 = x_2\|\ [(\mathrm{true}, x := x + 1, \flat)]^*\ \|x_1 \neq x_2\|$	✓
$\|x_1 = x_2\|\ [(x < 10, x := x + 1, \flat)]^*\ \|x_1 \neq x_2\|$	✗

Figure 4. Example Cartesian Hoare triples

Comparators. Another common programming pattern involves writing a custom ordering operator, for example by implementing the `Comparator` interface in Java or `operator<=` in C++. Such comparators are required to define a total order – i.e., they must be reflexive, anti-symmetric, transitive, and total. Programming errors in comparators are quite common and can have a variety of serious consequences. For instance, such bugs can cause collections to unexpectedly lose elements or not be properly sorted.

Map/Reduce. Map-reduce is a widely-used paradigm for writing parallel code [13, 24]. Here, the user-defined `reduce` function must be associative and stateless. Furthermore, if `reduce` is commutative and associative, then it can be used as a `combiner`, a "mini-reducer" that can be executed between the `map` and `reduce` phases to optimize bandwidth. In concurrent data structures that support `reduce` operations (e.g., the Java `ConcurrentHashMap`), the user-defined function must also be commutative and associative.

4. Cartesian Hoare Logic

We now present our program logic (CHL) for verifying Cartesian Hoare triples. The key idea underlying CHL is to verify $\|\Phi\|\ S\ \|\Psi\|$ by proving judgements of the form $\langle\Phi\rangle\ (S_1 \circledast \ldots \circledast S_k)\ \langle\Psi\rangle$ where each S_i corresponds to a different execution of S. Here, the notation $S_1 \circledast \ldots \circledast S_k$ represents the *set* of all programs that are semantically equivalent to simultaneously running $S_1, \ldots, S_k$. However, rather than explicitly constructing this set –or even *any* program in this set– we only reason about the *provability* of Ψ on sets of runs that jointly satisfy Φ. Hence, unlike previous approaches for proving equivalence or non-interference [3, 5, 26], our technique does not construct a *product program* that is subsequently fed to an off-the-shelf verifier. Instead, we combine the verification task with the construction of only the *relevant* parts of the product program.

To motivate the advantages of our approach over explicit product construction, consider the following specification:

$$\|x_1 > 0 \wedge x_2 \leq 0\|\ S_1 \oplus_{x>0} S_2\ \|y_1 = -y_2\|$$

where S_1 and S_2 are *very large* code fragments. Now, let S_{ij} represent $S_j[\vec{x}_i/\vec{x}]$, and let $S_{ij,kl}$ be a program that executes S_{ij} and S_{kl} in lockstep. One way to verify our Cartesian Hoare triple is to construct the following *product program* and feed it to an off-the-shelf verifier:

$$P:\ (S_{11,21} \oplus_{x_2>0} S_{11,22}) \oplus_{x_1>0} (S_{12,21} \oplus_{x_2>0} S_{12,22})$$

However, this strategy is quite suboptimal: Since the precondition is $x_1 > 0 \wedge x_2 \leq 0$, we are only interested in a small subset of P, which only includes $S_{11,22}$. In contrast, by combining semantic reasoning with construction of the relevant Hoare triples, our approach can avoid constructing and reasoning about redundant parts of the product program.

4.1 Core Cartesian Hoare Logic

With this intuition in mind, we now explain the core subset of Cartesian Hoare Logic shown in Figure 5. Since our calculus derives judgments of the form $\langle\Phi\rangle\ S_1 \circledast \ldots \circledast S_n\ \langle\Psi\rangle$, we first start by defining the *product space* of two programs S_1 and S_2, which we denote as $S_1 \circledast S_2$:

Definition 5. (Product space) *Let S_1 and S_2 be two statements such that* $\mathrm{vars}(S_1) \cap \mathrm{vars}(S_2) = \emptyset$. *Then,*

$$[\![S_1 \circledast S_2]\!] = \left\{ S \;\middle|\; \begin{array}{c} \forall \sigma_1, \sigma_2, \sigma_1', \sigma_2'.\ (\sigma_1, S_1 \Downarrow \sigma_1' \wedge \sigma_2, S_2 \Downarrow \sigma_2') \\ \Leftrightarrow (\sigma_1 \uplus \sigma_2, S \Downarrow \sigma_1' \uplus \sigma_2') \end{array} \right\}$$

Intuitively, $S_1 \circledast S_2$ represents the set of programs that are semantically equivalent to the simultaneous execution of S_1 and S_2. We generalize this notion of product space

to arbitrary values of k and consider *product terms* χ of the form $S_1 \circledast \ldots \circledast S_k$ where:

$$\begin{array}{rcl} [\![S]\!] & = & \{S\} \\ [\![S \circledast \chi]\!] & = & [\![S \circledast S']\!] \text{ where } S' \in [\![\chi]\!] \end{array}$$

Since we want to reason about different runs of the *same* program, we also define the *self-product space* as follows:

Definition 6. (Self product space)

$$\begin{array}{rcl} \circledast S^1 & \equiv & S[\vec{x}_1/\vec{x}] \\ \circledast S^n & \equiv & (\circledast S^{n-1}) \circledast S[\vec{x}_n/\vec{x}] \end{array}$$

In other words, $\circledast S^n$ represents the product space of n different α-renamed copies of S.

At a high level, the calculus rules shown in Figure 5 derive judgments of the form $\vdash \langle\Phi\rangle\ \chi\ \langle\Psi\rangle$ indicating the provability of the standard Hoare triple $\{\Phi\}\ P\ \{\Psi\}$ for some program $P \in [\![\chi]\!]$. Since all programs in $[\![\chi]\!]$ are semantically equivalent, this means that the Hoare triple $\{\Phi\}\ P'\ \{\Psi\}$ is valid for any $P' \in [\![\chi]\!]$. However, since some programs are easier to verify than other semantically equivalent variants, our calculus enables the verification of a Cartesian Hoare triple $\|\Phi\|\ S\ \|\Psi\|$ by considering different variants in the self-product space $\circledast S^n$.

We now explain the CHL rules in more detail and highlight how these proof rules allow us to achieve a good combination of flexibility, automatability, and scalability.

Expand. The first rule of Figure 5 reduces the verification of an n-ary Cartesian Hoare triple $\|\Phi\|\ S\ \|\Psi\|$ to the provability of $\langle\Phi\rangle\ \circledast S^n\ \langle\Psi\rangle$. Since each program $P \in \circledast S^n$ is semantically equivalent to the sequential execution of n α-renamed copies of S, the derivability of $\langle\Phi\rangle\ \circledast S^n\ \langle\Psi\rangle$ implies the validity of $\|\Phi\|\ S\ \|\Psi\|$.

Lift. The Lift rule allows us to prove $\langle\Phi\rangle\ \circledast S^n\ \langle\Psi\rangle$ in the degenerate case where $n = 1$. In this case, we resort to standard Hoare logic to prove the validity of $\{\Phi\}\ S\ \{\Psi\}$.

Skip intro. The next two rules, labeled $\flat$-intro, allow us to introduce no-ops after statements and at the end of other product terms. These rules are useful for eliminating redundancy in our calculus.

Skip elim. The $\flat$-elim rule, which is the analog of the $\flat$-intro 2 rule, eliminates skip statements from product terms. Note that the elimination analog of the $\flat$-intro 1 rule is unnecessary because it is derivable using the other rules.

Associativity. The Assoc rule exploits the associativity of operator $\circledast$: To prove $\langle\Phi\rangle\ (\chi_1 \circledast \chi_2) \circledast \chi_3\ \langle\Psi\rangle$, it suffices to show $\langle\Phi\rangle\ \chi_1 \circledast (\chi_2 \circledast \chi_3)\ \langle\Psi\rangle$. The associativity rule is very important for the flexibility of our calculus because, together with the commutativity rule, it allows us to consider different interleavings between k program executions.

Commutativity. The Comm rule states the commutativity of $\circledast$. Specifically, it says that a proof of $\langle\Phi\rangle\ \chi_1 \circledast \chi_2\ \langle\Psi\rangle$ also constitutes a proof of $\langle\Phi\rangle\ \chi_2 \circledast \chi_1\ \langle\Psi\rangle$.

$$\text{(Expand)}\quad \frac{\vdash\ \langle\Phi\rangle\ \circledast S^n\ \langle\Psi\rangle}{\vdash\ \|\Phi\|\ S\ \|\Psi\|}$$

$$\text{(Lift)}\quad \frac{\vdash\ \{\Phi\}\ S\ \{\Psi\}}{\vdash\ \langle\Phi\rangle\ S\ \langle\Psi\rangle}$$

$$(\flat\text{−intro 1})\quad \frac{\vdash\ \langle\Phi\rangle\ S; \flat \circledast \chi\ \langle\Psi\rangle}{\vdash\ \langle\Phi\rangle\ S \circledast \chi\ \langle\Psi\rangle}$$

$$(\flat\text{−intro 2})\quad \frac{\vdash\ \langle\Phi\rangle\ \chi \circledast \flat\ \langle\Psi\rangle}{\vdash\ \langle\Phi\rangle\ \chi\ \langle\Psi\rangle}$$

$$(\flat\text{−elim})\quad \frac{\vdash\ \langle\Phi\rangle\ \chi\ \langle\Psi\rangle}{\vdash\ \langle\Phi\rangle\ \chi \circledast \flat\ \langle\Psi\rangle}$$

$$\text{(Assoc)}\quad \frac{\vdash\ \langle\Phi\rangle\ \chi_1 \circledast (\chi_2 \circledast \chi_3)\ \langle\Psi\rangle}{\vdash\ \langle\Phi\rangle\ (\chi_1 \circledast \chi_2) \circledast \chi_3\ \langle\Psi\rangle}$$

$$\text{(Comm)}\quad \frac{\vdash\ \langle\Phi\rangle\ \chi_2 \circledast \chi_1\ \langle\Psi\rangle}{\vdash\ \langle\Phi\rangle\ \chi_1 \circledast \chi_2\ \langle\Psi\rangle}$$

$$\text{(Step)}\quad \frac{\vdash\ \{\Phi\}\ S_1\ \{\Phi'\}\ \vdash\ \langle\Phi'\rangle\ S_2 \circledast \chi\ \langle\Psi\rangle}{\vdash\ \langle\Phi\rangle\ S_1; S_2 \circledast \chi\ \langle\Psi\rangle}$$

$$\text{(Havoc)}\quad \frac{V = \text{vars}(\chi)}{\vdash\ \langle\Phi\rangle\ \chi\ \langle\exists V.\Phi\rangle}$$

$$\text{(Consq)}\quad \frac{\Phi \Rightarrow \Phi' \quad \Psi' \Rightarrow \Psi \quad \vdash\ \langle\Phi'\rangle\ \chi\ \langle\Psi'\rangle}{\vdash\ \langle\Phi\rangle\ \chi\ \langle\Psi\rangle}$$

$$\text{(Seq)}\quad \frac{\begin{array}{c}\vdash\ \langle\Phi\rangle\ (S_1 \circledast \ldots \circledast S_n)\ \langle\Phi'\rangle \\ \vdash\ \langle\Phi'\rangle\ (R_1 \circledast \ldots \circledast R_n) \circledast \chi\ \langle\Psi\rangle\end{array}}{\vdash\ \langle\Phi\rangle\ (S_1; R_1 \circledast \ldots \circledast S_n; R_n) \circledast \chi\ \langle\Psi\rangle}$$

$$\text{(If)}\quad \frac{\begin{array}{c}\vdash\ \langle\Phi \wedge c\rangle\ S_1; S \circledast \chi\ \langle\Psi\rangle \\ \vdash\ \langle\Phi \wedge \neg c\rangle\ S_2; S \circledast \chi\ \langle\Psi\rangle\end{array}}{\vdash\ \langle\Phi\rangle\ (S_1 \oplus_c S_2; S) \circledast \chi\ \langle\Psi\rangle}$$

$$\text{(Loop)}\quad \frac{\begin{array}{c}\Phi \Rightarrow \mathbb{I} \\ \vdash\ \{\mathbb{I}\}\ [(c_1, S_1, \flat), \ldots, (c_n, S_n, \flat)]^*\ \{\mathbb{I}\} \\ \vdash\ \langle \text{post}_i(\mathbb{I}) \wedge \neg c_i\rangle\ S_i' \circledast \chi\ \langle\Psi\rangle \quad (\forall i \in [1, n])\end{array}}{\vdash\ \langle\Phi\rangle\ [(c_1, S_1, S_1'), \ldots, (c_n, S_n, S_n')]^* \circledast \chi\ \langle\Psi\rangle}$$

Figure 5. Core Cartesian Hoare Logic (CHL)

Step. The Step rule allows us to decompose the verification of $\langle\Phi\rangle\ S_1; S_2 \circledast \chi\ \langle\Psi\rangle$ in the following way: First, we find an auxiliary assertion Φ' and prove the validity of $\{\Phi\}\ S_1\ \{\Phi'\}$. If we can also establish the validity of $\langle\Phi'\rangle\ S_2 \circledast \chi\ \langle\Psi\rangle$, we obtain a proof of $\langle\Phi\rangle\ S_1; S_2 \circledast \chi\ \langle\Psi\rangle$. This rule turns out to be particularly useful when we can execute S_2 and χ in lockstep, but not S_1 and χ.

Havoc. This rule allows us to prove $\langle\Phi\rangle\ \chi\ \langle\exists V.\Phi\rangle$ for any χ, where V denotes free variables in χ. Note that by, existentially quantifying V, we effectively assume that χ invalidates all facts we know about variables V. However, facts involving variables that are not modified by χ are still valid. The main purpose of the Havoc rule is to prune irrelevant parts of the state space by combining it with the Consq rule.

Consequence. The Consq rule is the direct analog of the standard consequence rule in Hoare logic. As expected, this rule states that we can prove the validity of $\langle\Phi\rangle\ \chi\ \langle\Psi\rangle$ by proving $\langle\Phi'\rangle\ \chi\ \langle\Psi'\rangle$ where $\Phi \Rightarrow \Phi'$ and $\Psi' \Rightarrow \Psi$.

Example 3. *We now illustrate why the Havoc and Consq rules are very useful. Suppose we would like to prove:*

$$\langle\text{true}\rangle\ x_1 := \text{false} \circledast S_2 \circledast S_3\ \langle(x_1 \wedge x_2) \Rightarrow x_3\rangle$$

where S_2, S_3 are statements over x_2, x_3. After using Step to prove $\{\text{true}\}\ x_1 := \text{false}\ \{\neg x_1\}$, we end up with proof obligation $\langle\neg x_1\rangle\ S_2 \circledast S_3\ \langle x_1 \wedge x_2 \Rightarrow x_3\rangle$. Since S_2, S_3 do not contain variable x_1, we can use Havoc to obtain a proof of $\langle\neg x_1\rangle\ S_2 \circledast S_3\ \langle\neg x_1\rangle$. Now, since we have $\neg x_1 \Rightarrow (x_1 \wedge x_2 \Rightarrow x_3)$, the consequence rule allows us to obtain the desired proof. Observe that, even though S_2 and S_3 are potentially very large program fragments, we were able to complete the proof without reasoning about $S_2 \circledast S_3$ at all.

Sequence. The Seq rule allows us to execute statements from different runs in lockstep. In particular, it tells us that we can prove $\langle\Phi\rangle\ (S_1; R_1\ \circledast \ldots \circledast\ S_n; R_n) \circledast \chi\ \langle\Psi\rangle$ by first showing $\langle\Phi\rangle\ S_1\ \circledast \ldots \circledast\ S_n\ \langle\Phi'\rangle$ and then separately constructing a proof of $\langle\Phi'\rangle\ R_1\ \circledast \ldots \circledast\ R_n \circledast \chi\ \langle\Psi\rangle$.

If. The If rule allows us to "embed" product term χ inside each branch of $S_1 \oplus_c S_2$. As the next example illustrates, this rule can be useful for simplifying the verification task, particularly when combined with Havoc and Consq.

Example 4. *Suppose we want to prove:*

$$\|\varphi_1\|\ y := 1 \oplus_{x>0} y := 0\ \|\varphi_2\|$$

where φ_1 is $x_1 = -x_2 \wedge x_1 \neq 0$ and φ_2 is $y_1 + y_2 = 1$. After applying the Expand and If rules (combined with $\flat$ elim and intro), we end up with the following two proof obligations:

$$(1)\quad \langle\varphi_1 \wedge x_1 > 0\rangle\ y_1 := 1 \circledast (y_2 := 1 \oplus_{x_2>0} y_2 := 0)\ \langle\varphi_2\rangle$$
$$(2)\quad \langle\varphi_1 \wedge x_1 \leq 0\rangle\ y_1 := 0 \circledast (y_2 := 1 \oplus_{x_2>0} y_2 := 0)\ \langle\varphi_2\rangle$$

Let's only focus on (1), since (2) is similar. Using first Comm and then If, we generate the following proof obligations:

$$(1a)\quad \langle\varphi_1 \wedge x_1 > 0 \wedge x_2 > 0\rangle\ y_2 := 1 \circledast y_1 := 1\ \langle\varphi_2\rangle$$
$$(1b)\quad \langle\varphi_1 \wedge x_1 > 0 \wedge x_2 \leq 0\rangle\ y_2 := 0 \circledast y_1 := 1\ \langle\varphi_2\rangle$$

Since the precondition of (1a) is unsatisfiable, we can immediately discharge it using Havoc and Consq. We can also discharge (1b) by computing its postcondition $y_2 = 0 \wedge y_1 = 1$, which can be obtained using Step, $\flat$-intro, elim, and Lift.

Note that we could even verify this property using a path-insensitive analysis (e.g., the polyhedra [11] or octagon [17] abstract domains) using the above proof strategy. In particular, conceptually embedding the second program inside the then and else branches of the first program greatly simplifies the proof. While we could also prove this simple property using self composition, the proof would require a path-sensitive analysis.

Loop. At a conceptual level, the Loop rule allows us to reason about $L = [(c_1, S_1, S_1'), \ldots, (c_n, S_n, S_n')]^* \circledast \chi$ as though χ was "embedded" inside each exit (break) point of L. In particular, this rule first reasons about the first $k-1$ iterations of the loop (i.e., all except the last one) and then considers $L_k \circledast \chi$ where L_k represents the computation performed during the last iteration.

Let us now consider this rule in more detail. The first two lines of the premise state that $\mathbb{I}$ is an inductive invariant of the first $k-1$ iterations of L, where k denotes the number of iterations of L. Hence, we know that $\mathbb{I}$ holds at the end of the $k-1$'th iteration. Now, we essentially perform a case analysis: When the loop terminates, one of the c_i's must be false, but we don't know which one, so we try to prove Ψ for each of the n possibilities. Specifically, suppose L terminated because c_i was the first condition to evaluate to false during the k'th iteration. In this case, we need to symbolically execute $S_1, \ldots, S_{i-1}$ before we can reason about $S_i' \circledast \chi$. For this purpose, we define $\text{post}_i(\mathbb{I})$ as follows:

$$\begin{array}{lcl} \text{post}_1(\mathbb{I}) & = & \mathbb{I} \\ \text{post}_i(\mathbb{I}) & = & \text{post}(\text{post}_{i-1}(\mathbb{I}) \wedge c_{i-1}, S_{i-1}) \end{array}$$

where $\text{post}(\phi, S)$ denotes a post-condition of S with respect to ϕ. Hence, $\text{post}_i(\mathbb{I})$ represents facts that hold right before executing (c_i, S_i, S_i'). Now, if the loop terminates due to condition c_i, we need to prove Ψ holds after executing $S_i' \circledast \chi$ assuming $\text{post}_i(\mathbb{I}) \wedge \neg c_i$. Hence, if we can establish $\langle\text{post}_i(\mathbb{I}) \wedge \neg c_i\rangle\ S_i' \circledast \chi\ \langle\Psi\rangle$ for all $i \in [1, n]$, this also gives us a proof of $\langle\Phi\rangle\ [(c_1, S_1, S_1'), \ldots, (c_n, S_n, S_n')]^*\ \circledast\ \chi\ \langle\Psi\rangle$.

Example 5. *Consider the following loop L:*

$$[(i < n, \flat, \flat), (a[i] \geq b[i], \flat, r{:=}{-1}), (a[i] \leq b[i], i{+}{+}, r{:=}1)]^*$$

which often arises when implementing a lexicographic order. Suppose we want to prove the following 3-safety property:

$$\|a_1 = a_3 \wedge b_1 = a_2 \wedge b_2 = b_3 \wedge n_1 = n_2 \wedge n_2 = n_3\|$$
$$r := 0; i := 0; L$$
$$\|r_1 \leq 0 \wedge r_2 \leq 0 \Rightarrow r_3 \leq 0\|$$

We can verify this Cartesian Hoare triple using our Loop rule and loop invariant $\forall j.\ 0 \leq j < i \Rightarrow a[j] = b[j]$. However, this invariant is not *sufficient if we try to verify this code using self-composition.*

4.2 Cartesian Loop Logic

The core calculus we have considered so far is sound and relatively complete, but it does not allow us to execute loops from different runs in lockstep. Since lockstep execution can greatly simplify the verification task (e.g., by requiring simpler invariants), we have augmented our calculus with additional rules for reasoning about loops. These proof rules, which we refer to as "Cartesian Loop Logic" are summarized in Figure 6 and explained next.

$$\text{(Transform}-\text{single)}\qquad \frac{}{[(c,S,S')]^* \rightsquigarrow [(c,S,\flat)]^*;\surd\neg c;S'}$$

$$\text{(Transform}-\text{multi)}\qquad \frac{[R]^* \rightsquigarrow [(c',S',\flat)]^*;S'' \quad c'' = c_1 \wedge \mathrm{wp}(c',S_1)}{[(c_1,S_1,S_1'),R]^* \rightsquigarrow [(c'',S_1;S',\flat)]^*;\ (\surd c_1;S_1;S'') \oplus_\star (\surd\neg c_1;S_1')}$$

$$\text{(Flatten)}\qquad \frac{[L]^* \rightsquigarrow [(c,B,\flat)]^*;R \quad \vdash \langle\Phi\rangle\ [(c,B,\flat)]^*;R \circledast \chi\ \langle\Psi\rangle}{\vdash \langle\Phi\rangle\ [L]^* \circledast \chi\ \langle\Psi\rangle}$$

$$\text{(Fusion 1)}\qquad \frac{\begin{array}{c}\vdash \langle \mathbb{I} \wedge \bigwedge_{1\le i\le n} c_i\rangle\ S_1 \circledast \ldots \circledast S_n\ \langle\mathbb{I}\rangle \\ \Phi \Rightarrow \mathbb{I} \\ (\mathbb{I} \wedge \neg \bigwedge_{1\le i\le n} c_i) \Rightarrow \bigwedge_{1\le i\le n} \neg c_i\end{array}}{\vdash \langle\Phi\rangle\ [(c_1,S_1,\flat)]^* \circledast \ldots \circledast [(c_n,S_n,\flat)]^*\ \langle \mathbb{I} \wedge \neg \bigwedge_{1\le i\le n} c_i\rangle}$$

$$\text{(Fusion 2)}\qquad \frac{\begin{array}{c}\vdash \langle \mathbb{I} \wedge \bigwedge_{1\le i\le n} c_i\rangle\ S_1 \circledast \ldots \circledast S_n\ \langle\mathbb{I}\rangle \\ \vdash \langle \mathbb{I} \wedge \neg c_1\rangle\ [(c_2,S_2,\flat)]^* \circledast \ldots \circledast [(c_n,S_n,\flat)]^*\ \langle\Psi\rangle \\ \vdash \langle \mathbb{I} \wedge \neg c_2\rangle\ [(c_1,S_1,\flat)]^* \circledast [(c_3,S_3,\flat)]^* \circledast \ldots \circledast [(c_n,S_n,\flat)]^*\ \langle\Psi\rangle \\ \ldots \\ \vdash \langle \mathbb{I} \wedge \neg c_n\rangle\ [(c_1,S_1,\flat)]^* \circledast \ldots \circledast [(c_{n-1},S_{n-1},\flat)]^*\ \langle\Psi\rangle\end{array}}{\vdash \langle\Phi\rangle\ [(c_1,S_1,\flat)]^* \circledast \ldots \circledast [(c_n,S_n,\flat)]^*\ \langle\Psi\rangle}\ (n \ge 2., \Phi \Rightarrow \mathbb{I})$$

Figure 6. Cartesian Loop Logic (CLL)

Transform. The first two rules, labeled Transform-single and Transform-multi, allow us to "rewrite" complicated loops L containing break statements into code snippets of the form `while(C) do {S}; S'`. These rules derive judgments of the form $L \rightsquigarrow S$ stating that any Hoare triple that is valid for S is also valid for L. In particular, while L and S may *not* be semantically equivalent, our transformation $L \rightsquigarrow S$ guarantees that any valid Hoare triple $\{\Phi\}\ S\ \{\Psi\}$ implies the validity of $\{\Phi\}\ L\ \{\Psi\}$.

Let us first consider the Transform-single rule where the loop contains a single break statement. This rule simply "rewrites" the loop $[(c,S,S')^*]$ to $[(c,S,\flat)^*;\surd\neg c;S']$. In other words, it says that we can replace S' with a skip statement and simply execute S' once the loop terminates.

The Transform-multi rule handles loops containing multiple break points. In particular, suppose we have a loop L of the form $[(c_1,S_1,S_1'),R]^*$ and suppose that $R^* \rightsquigarrow [(c',S',\flat)]^*;S''$. This means that R is semantically equivalent to $(c',S',\flat)$ in all iterations of L except the last one. Hence, the loop $L' : [(c_1,S_1,\flat),(c',S',\flat)]^*$ is also equivalent to L under all valuations in which L executes at least once more. Furthermore, if $\mathrm{wp}(c',S_1)$ denotes the weakest precondition of c' with respect to S_1, then L' (and therefore L) is also equivalent to

$$L'' :\ [(c_1 \wedge \mathrm{wp}(c',S_1),S_1;S',\flat)]^*$$

under all valuations in which L executes one or more times.

Now, let's consider the last iteration of L. There are two possibilities: Either we broke out of the loop because c_1 evaluated to false or some other condition c_i in R evaluated to false. In the former case, we can assume $\neg c_1$ and we need to execute S_1'. In the latter case, we can assume c_1, and we still need to execute S_1 as well as the "part" of R that is executed in the last iteration, which is given by S''. Hence, we can soundly reason about the last iteration of L using $(\surd c_1;S_1;S'') \oplus_\star (\surd\neg c_1;S_1')$. [2]

Example 6. *Consider again loop L from Example 5. Using Transform rules of Figure 6, we have $L \rightsquigarrow S$ where S is:*

$$[(i<n \wedge a[i]=b[i], i{+}{+},\flat)]^*;$$
$$(\surd c_1;(\surd c_2;\surd\neg c_3;r{:=}\,1 \oplus_\star \surd\neg c_2;r{:=}\,-1)) \oplus_\star \surd\neg c_1$$

Here, c_1,c_2,c_3 represent $i<n$, $a[i] \ge b[i]$, and $a[i] \le b[i]$.

Flatten. The Flatten rule uses the previously discussed Transform rules to simplify the proof of $\langle\Phi\rangle\ [L]^*\ \langle\Psi\rangle$. In particular, it first "rewrites" $[L]^*$ as S and then proves the validity of $\langle\Phi\rangle\ S \circledast \chi\ \langle\Psi\rangle$. This proof rule is sound since $\{\phi\}\ S\ \{\varphi\}$ always implies the validity of $\{\phi\}\ [L]^*\ \{\varphi\}$ for any ϕ,φ whenever $[L]^* \rightsquigarrow S$. As we will see shortly, the Flatten rule can make proofs significantly easier, particularly when combined with the Fusion rules explained below.

Fusion 1. This rule effectively allows us to perform lock-step execution for loops $L_1,\ldots,L_n$ that always execute the *same* number of times. Observe that this rule requires each L_i to contain a single break point; hence, if this requirement is violated, we must first apply the Flatten rule.

Let us now consider this rule in more detail. Recall that $S_1 \circledast \ldots \circledast S_n$ is the set of programs that are semantically equivalent to $S_1;\ldots;S_n$. Hence, the first two lines of the premise effectively state that $\mathbb{I}$ is an inductive invariant of:

$$L : [(c_1 \wedge \ldots \wedge c_n,\ S_1;\ldots;S_n,\ \flat)]^*$$

Furthermore, the third line of the premise tells us that every c_i will evaluate to false when L terminates, which, in turn,

[2] The careful reader may wonder whether the last iteration of the loop could be accounted for using $S_1;S'' \oplus_{c_1} S_1'$. This reasoning would not be sound since c_1 may be modified in $S_2,\ldots,S_n$.

indicates that all L_i's always execute the same number of times. Thus, $\mathbb{I} \wedge \neg \bigwedge_i c_i$ is a valid post-condition of $L_1 \circledast \ldots \circledast L_n$ under precondition Φ.

Example 7. *Consider loop L from Example 5, and suppose we want to verify the following Cartesian Hoare triple:*

$$\|\Phi\| \ (r := 0; i := 0; L) \ \|\Psi\|$$

where Φ is $a_1 = b_2 \wedge b_1 = a_2 \wedge a_1 \neq b_1$ and Ψ is $r_1 < 0 \Rightarrow r_2 \geq 0$. After applying Seq, we generate the following proof obligation:

$$\langle \Phi \wedge r_1 = r_2 = 0 \wedge i_1 = i_2 = 0 \rangle \ L_1 \circledast L_2 \ \langle \Psi \rangle$$

We then apply Flatten to both L_1 and L_2, which yields the following proof obligation:

$$\langle \Phi \wedge r_1 = r_2 = 0 \wedge i_1 = i_2 = 0 \rangle \ L_1'; S_1' \circledast L_2'; S_2' \ \langle \Psi \rangle$$

where L_1', L_2' are α-renamed versions of the loop obtained in Example 6 and S_1, S_2 are α-renamed versions of S from Example 6. We now apply Seq again, grouping together L_1 and L_2, and then use Fusion to obtain a postcondition Φ':

$$\langle \Phi \wedge r_1 = r_2 = 0 \wedge i_1 = i_2 = 0 \rangle \ L_1' \circledast L_2' \ \langle \Phi' \rangle$$

Using the loop invariant $\Phi \wedge i_1 = i_2$ and the Fusion and Consq rules, we can obtain the post-condition $\Phi' = (\Phi \wedge i_1 = i_2)$. This finally leaves us with the proof obligation $\langle \Phi' \rangle \ S_1' \circledast S_2' \ \langle \Psi \rangle$, which is easy to discharge using If and Havoc: Note that the post-condition can only be violated in the branches where r_1 and r_2 are both assigned to -1. However since $r_1 := -1$ and $r_2 := -1$ only execute when $a_1[i_1] < b_1[i_1]$ and $a_2[i_2] < b_2[i_2]$ respectively, the precondition $\Phi \wedge i_1 = i_2$ implies $\neg(a_1[i_1] < b_1[i_1] \wedge a_2[i_2] < b_2[i_2])$. Hence, the original Cartesian Hoare triple is valid.

Observe that the Flatten and Fusion rules allow us to verify the desired property using the simple loop invariant $i_1 = i_2$ without requiring any quantified array invariants.

Fusion 2. The Fusion 2 rule is similar to Fusion 1 and allows us to perform partial lockstep execution when we cannot prove that loops $L_1, \ldots, L_k$ execute the same number of times. Conceptually, this rule "executes" the loop bodies $S_1, \ldots, S_n$ in lockstep until one of the continuation conditions c_i becomes false. Then, the rule proceeds with a case analysis: Assuming c_i is the first condition to evaluate to false, we potentially still need to execute loops $L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n$. Furthermore, when we execute these loops, we can assume the loop invariant $\mathbb{I}$ of the first loop as well as condition $\neg c_i$ (since we are assuming that c_i is the first condition to evaluate to false). Hence, lines 3-5 in the premise verify the following proof obligation for each i:

$$\langle \mathbb{I} \wedge \neg c_i \rangle \ L_1 \circledast \ldots \circledast L_{i-1} \circledast L_{i+1} \circledast \ldots \circledast L_n \ \langle \Psi \rangle$$

Theorem 1 (Soundness)**.** *If $\vdash \|\Phi\| \ S \ \|\Psi\|$, then $\models \|\Phi\| \ S \ \|\Psi\|$.*

Theorem 2 (Relative Completeness)**.** *Given an oracle for deciding the validity of any standard Hoare triple, if $\models \|\Phi\| \ S \ \|\Psi\|$, then we also have $\vdash \|\Phi\| \ S \ \|\Psi\|$.*

The proofs of both theorems are provided in the extended version of the paper.

```
1:  procedure K-VERIFY(Φ, χ, Ψ)
2:      Input: Precondition Φ, term χ, postcondition Ψ
3:      Output: Boolean indicating whether property holds
4:      let V := Vars(χ) in
5:      if ((∃V.Φ) ⇒ Ψ) then return true;        ▷ Havoc and Consq
6:      match χ with
7:      | S:  return VERIFY(Φ, S, Ψ);                          ▷ Lift
8:      | ♭ ⊛ χ':  return K-VERIFY(Φ, χ', Ψ)                  ▷ ♭-elim
9:      | [L1]*; S1 ⊛ ... ⊛ [Ln]*; Sn:                        ▷ all loops
10:             return K-VERIFYLOOP(Φ, χ, Ψ)
11:     | [L]*; S ⊛ χ'
12:             return K-VERIFY(Φ, χ' ⊛ [L]*; S)              ▷ Comm
13:     | A; S ⊛ χ':  let Φ' := post(A, Φ) in
14:                   return K-VERIFY(Φ', S ⊛ χ', Ψ)          ▷ Step
15:     | S1 ⊕c S2; S ⊛ χ':
16:             let r := K-VERIFY(Φ ∧ c, S1; S ⊛ χ', Ψ) in
17:             if(r != true) return false;                   ▷ If
18:             return K-VERIFY(Φ ∧ ¬c, S2; S ⊛ χ', Ψ )
19:     | S ⊛ χ':  return K-VERIFY(Φ, S; ♭ ⊛ χ', Ψ)           ▷ ♭-intro
20:     | (χ1 ⊛ χ2) ⊛ χ3:
21:             return K-VERIFY(Φ, χ1 ⊛ (χ2 ⊛ χ3), Ψ)        ▷ Assoc
```

Figure 7. Verification algorithm for k-safety

5. Verification Algorithm Based on CHL

We now describe how to incorporate the CHL proof rules into a useful verification algorithm. At a high level, there are three important ideas: First, since the Havoc and Consequence rules can greatly simplify the verification task, our algorithm applies these rules at every opportunity. Second, we always prefer the If rule over other rules like Seq or Step, as this strategy avoids reasoning about parts of $\circledast S^n$ that contradict the desired precondition. Third, we try to maximize opportunities for lockstep execution of loops. For example, we use Associativity and Commutativity in a way that tries to maximize possible applications of the Fusion rules.

Our verification algorithm is presented in Figure 7: K-VERIFY takes as input pre- and post-conditions Φ, Ψ, product term χ, and returns true iff we can prove $\langle \Phi \rangle \ \chi \ \langle \Psi \rangle$. As a first step, we check if $\langle \Phi \rangle \ \chi \ \langle \Psi \rangle$ can be verified using only the Havoc and Consequence rules (lines 4–5).

Next, the algorithm performs pattern matching on χ in lines 6–21. In the base case, χ is a single program S, so we simply invoke a procedure VERIFY for proving the standard Hoare triple $\{\Phi\} \ S \ \{\Psi\}$ (line 7). The next case $\flat \circledast \chi'$ at line 8 is equally simple; in this case, we proceed by applying $\flat$-elimination to verify $\langle \Phi \rangle \ \chi' \ \langle \Psi \rangle$.

The next two cases concern terms χ that start with a loop. In particular, the pattern at line 9 checks if χ is of the form $[L_1]^*; S_1 \circledast \ldots \circledast [L_n]^*; S_n$ (i.e., all programs start with a loop). In this case, we invoke the helper function K-VERIFYLOOP which chooses between the proof tactics from Figure 6 and which we consider in more detail later. On the other hand, if the first program starts with a loop (but not all the remaining ones), we then use Commutativity (line 12) with the hope that we will eventually find matching loops in all programs (i.e., match line 9 in a future recursive call).

```
1:  procedure K-VERIFYLOOP(Φ, χ, Ψ)
2:      match χ with
3:      | [(c1, S1 ♭)]*; R1 ⊛ ... ⊛ [(cn, Sn ♭)]*; Rn:
4:          let 𝕀 := FINDINV(Φ, ⋀i ci, S1, ..., Sn) in
5:          let Φ' := 𝕀 ∧ ¬ ⋀i ci in
6:          if(Φ' ⇒ ⋀i ¬ci) then                          ▷ Fuse 1, Seq
7:              return K-VERIFY(Φ', R1 ⊛ ... ⊛ Rn, Ψ);
8:          else
9:              forach(i ∈ [1,n])                          ▷ Fuse 2, Seq
10:                 let χi := REMOVEITH(χ, i) in
11:                 if(!K-VERIFY(Φ', χi, Ψ)) return false;
12:             return true;
13:     | [L]*; R ⊛ χ':
14:         if([L]* ⇝ [(c, B, ♭)]*; R' ) then              ▷ Flatten
15:             return K-VERIFY(Φ, χ' ⊛ [(c, B, ♭)]*; R'; R, Ψ);
16:         else
17:             let 𝕀 := LOOPINV([L]*) in                  ▷ Loop
18:             foreach (i ∈ [1,n])
19:                 let Φ' := post_i(𝕀) ∧ ¬ci;
20:                 if(!K-VERIFY(Φ', Si'; R ⊛ χ'))
21:                     then return false;
22:             return true;
```

Figure 8. Helper procedure for loops

Continuing with line 13, we next pattern match on terms of the form $A; S \circledast \chi'$ where the first program starts with an atom A. Since we can compute the strongest postcondition for atomic statements in an exact way, we apply the Step rule and generate a new proof obligation $\langle \mathrm{post}(A, \Phi) \rangle\ S \circledast \chi'\ \langle \Psi \rangle$ where $\mathrm{post}(A, \Phi)$ denotes the (strongest) postcondition of A with respect to Φ.

Next, consider the case where χ starts with a program whose first statement is $S_1 \oplus_c S_2$. In this case, we apply the If rule rather than the Step rule, as this strategy has two advantages: First, as we saw in Example 4, this rule often simplifies the verification task. Second, since one of $\Phi \wedge c$ or $\Phi \wedge \neg c$ may be unsatisfiable, we might be able to easily discharge the new proof obligations in the recursive calls to K-VERIFY using the Consequence rule.

The next case at line 19 applies to product terms of the form $S \circledast \chi'$. Note that we only match this case if S consists of a single loop or conditional. Since we would like to apply the appropriate rules for conditionals and loops, we use $\flat$ intro and then make a recursive call — this strategy ensures that we will match one of the cases at lines 9, 11, or 15 in the next recursive call.

The final case at line 20 exploits associativity: If the first term of χ is not a single program, but rather another product term of the form $\chi_1 \circledast \chi_2$, we apply associativity. This tactic guarantees that χ will eventually be of the form $S \circledast \chi'$ in one of the future recursive calls.

Let us now turn our attention to the auxiliary procedure K-VERIFYLOOP shown in Figure 8. The interface of this function is the same as that of K-VERIFY, but it is only invoked on terms of the form $[L_1]^*; S_1 \circledast \ldots \circledast [L_n]^*; S_n$ (i.e., all programs start with a loop). The goal of K-VERIFYLOOP is to determine which loop-related proof rule to use.

In the case analysis of lines 2-21, we first check whether all terms in χ start with loops of the form $[c, S, \flat]^*$, and if so, we apply one of the Fusion rules from Figure 6. Towards this goal, we invoke a function called FINDINV which returns a formula $\mathbb{I}$ with the following properties:

$$(1)\quad \Phi \Rightarrow \mathbb{I}$$
$$(2)\quad \vdash \langle \mathbb{I} \wedge \textstyle\bigwedge_i c_i \rangle\ S_1 \circledast \ldots \circledast S_n\ \langle \mathbb{I} \rangle$$

Continuing with lines 6-12, we next check whether it is legal to apply the simpler Fusion 1 rule or whether it is necessary to use the more general Fusion 2 rule. If it is the case that $(\mathbb{I} \wedge \neg \bigwedge_i c_i) \Rightarrow \bigwedge_i \neg c_i$, all loops execute the same number of times, so we can apply Fusion 1.

Otherwise, we apply the more general Fusion 2 rule. In particular, the recursive calls at line 11 correspond to discharging the proof obligations in the premises of the Fusion 2 rule from Figure 6. Here, the call to the auxiliary procedure to REMOVEITH returns the following formula:

$$[(c_1, S_1, \flat)]^*; R_1 \circledast \ldots \circledast [(c_{i-1}, S_{i-1}, \flat)]^*; R_{i-1} \circledast [(c_{i+1}, S_{i+1}, \flat)]^*; R_{i+1} \circledast \ldots \circledast [(c_n, S_n, \flat)]^*; R_n$$

Observe that we return true if and only if we can discharge all proof obligations for every $i \in [1, n]$.

Now consider the final case at line 13 of the algorithm. In this case, we know that not all loops are of the form $[c, S, \flat]^*$, so it may not be possible to apply the Fusion rules. Hence, we first apply the Transform rules from Figure 6 to rewrite the first loop in the form $[(c, B, \flat)]^*$ If this rewrite is possible, we then apply the Flatten rule at line 15; otherwise, we resort to the Loop rule from Figure 5. [3] Observe that recursive calls to K-VERIFY at line 20 correspond to the proof obligations in the premise of the Loop rule from Figure 5.

Theorem 3 (Termination)**.** K-VERIFY*(Φ, χ, Ψ) terminates for every Φ, χ, Ψ.*

6. Implementation

We implemented the verification algorithm from Section 5 in a tool called DESCARTES, written in Haskell. DESCARTES uses the Haskell *language-java* package to parse Java source code and the Z3 SMT solver [12] for determining satisfiability. Our implementation models object fields with uninterpreted functions and uses axioms for library methods implemented by the Java SDK.

The main verification procedure in DESCARTES receives a Cartesian Hoare triple that is composed of a method in the IR format, the arity of the specification, and the pre- and post-conditions. It then generates k alpha-renamed methods and applies the K-VERIFY algorithm from Section 5. If the specification is not respected, DESCARTES outputs the Z3 model that can used to construct a counterexample.

[3] Recall that the Transform rules from Figure 6 require computing weakest preconditions, which may be difficult to do in the presence of nested loops.

As a design choice, our loop invariant generator (i.e., the implementation of FINDINV) is decoupled from the main verification method since we aim to experiment with several loop invariant generation techniques in the future. Currently, our implementation of FINDINV follows standard guess-and check methodology [14, 20]:

1. A *guess* procedure guesses a candidate invariant I. Currently, we generate candidate invariants by instantiating a set of built-in templates with program variables and constants which include (i) equalities and inequalities between pairs of variables and (ii) quantified invariants of the form $\forall j.\bullet \leq j < \bullet \Rightarrow \bullet[\bullet]\ op\ \bullet\ [\bullet]$ where $\bullet$ indicates an unknown expression to be instantiated with variables and constants.
2. A *check* procedure verifies if candidate invariant I is inductive. This is performed by discharging two verification conditions (VCs):
   - The pre-condition implies I;
   - The cartesian hoare triple given by Eq (2) in Section 6 is valid. The validity of this cartesian hoare triple is checked by a recursive call to K-VERIFY.

7. Experimental Evaluation

To evaluate our approach, we used DESCARTES to verify user-defined relational operators in Java programs, including `compare`, `compareTo`, and `equals`. We believe this application domain is a good testbed for our approach for multiple reasons: First, as mentioned in Section 3, these relational operators must obey multiple k-safety properties, allowing us to consider five different k-safety properties in our evaluation. Second, since comparators and `equals` methods are ubiquitous in Java, we can test our approach on a variety of different applications. Third, comparators are notoriously tricky to get right; in fact, web forums like Stackoverflow abound with questions from programmers who are trying to debug their comparator implementations.

The goal of our experimental evaluation is to explore the following questions: (1) Is DESCARTES able to successfully uncover k-safety property violations in buggy programs? (2) Is DESCARTES practical? (i.e., how long does verification take, and how many false positives does it report?) (3) How does DESCARTES compare with competing approaches such as self-composition [1] and product construction [3, 5]?

To answer these three questions, we performed three sets of experiments. In the first experiment, we consider examples of buggy comparators from Stackoverflow as well as their repaired versions. In the second experiment, we use DESCARTES to analyze `equals`, `compare`, and `compareTo` methods that we found in industrial-strength Java projects from Github. In our third experiment, we compare DESCARTES with the self-composition and product construction approaches and report our findings. In what follows, we describe these experiments in more detail.

	P1		P2		P3	
Benchmark	V	$t(s)$	V	$t(s)$	V	$t(s)$
ARRAYINT	✓	0.14	✓	0.16	✗	0.27
ARRAYINT†	✓	0.19	✓	0.41	✓	0.38
CATBPOS	✗	0.33	✗	8.35	✗	2.59
CHROMOSOME	✓	0.15	✗	0.19	✗	0.34
CHROMOSOME†	✓	0.36	✓	4.26	✓	3.90
COLITEM	✗	0.09	✗	0.15	✗	0.17
COLITEM†	✓	0.07	✓	0.17	✓	0.17
CONTACT	✓	0.11	✗	1.14	✗	1.85
CONTAINER-V1	✗	0.04	✗	0.04	✗	0.04
CONTAINER-V2	✗	0.05	✗	0.04	✗	0.05
CONTAINER†	✓	0.27	✓	3.44	✓	1.28
DATAPOINT	✗	0.26	✗	0.89	✗	0.51
FILEITEM	✓	0.03	✓	0.03	✗	0.09
FILEITEM†	✓	0.05	✓	0.06	✓	0.08
ISOSPRITE-V1	✗	0.06	✗	0.04	✗	0.07
ISOSPRITE-V2	✗	0.40	✗	1.82	✓	0.18
MATCH	✗	0.05	✓	0.04	✗	0.06
MATCH†	✓	0.05	✓	0.05	✓	0.06
NAME	✗	0.15	✓	0.38	✓	0.27
NAME†	✓	0.16	✓	0.40	✓	0.50
NODE	✓	0.03	✓	0.03	✗	0.09
NODE†	✓	0.04	✓	0.06	✓	0.07
NZBFILE	✗	0.14	✓	0.28	✓	0.13
NZBFILE†	✓	0.25	✓	0.48	✓	0.99
POKERHAND	✓	0.55	✗	0.71	✗	2.34
POKERHAND†	✓	0.61	✓	0.78	✓	1.61
SOLUTION	✓	0.19	✓	0.52	✗	0.68
SOLUTION†	✓	0.42	✓	1.33	✓	1.24
TEXTPOS	✓	0.10	✗	0.25	✗	0.26
TEXTPOS†	✓	0.11	✓	0.48	✓	0.19
TIME	✗	0.10	✓	0.36	✓	0.02
TIME†	✓	0.08	✓	0.35	✓	0.22
WORD	✗	0.84	✗	6.19	✓	0.07
WORD†	✓	0.24	✓	0.52	✓	0.28

Figure 9. Evaluation on *Stackoverflow* examples.

7.1 Buggy Examples from Stackoverflow

We manually curated a set of 34 comparator examples from on-line forums such as *Stackoverflow*. In many online discussions that we looked at, a developer posts a buggy comparator –often with quite tricky program logic– and asks other forum users for help with fixing his/her code. Hence, we were often able to extract a buggy comparator together with its repaired version from each *Stackoverflow* post. Given this collection of benchmarks containing both buggy and correct programs, we used DESCARTES to check whether each comparator obeys properties P1-P3 required by the Java `Comparator` and `Comparable` interfaces.

The results of our evaluation are presented in Figure 9, and the benchmarks are provided under supplemental materials. For a buggy comparator called COMPARE, the benchmark labeled COMPARE† describes its repaired version. For the column labeled V, ✓ indicates that DESCARTES was able to verify the property, and ✗ means that DESCARTES reported a violation of the property. For a few benchmarks, (e.g., CATBPOS), the post did not contain the correct version of the comparator and the problem did not seem to have a simple

fix; hence, Figure 9 does not include the repaired versions of a few benchmarks.

From Figure 9, we see that DESCARTES automatically verifies all correct comparators and does not report any false alarms. Furthermore, for each buggy program, DESCARTES pinpoints the violated property and provides a counterexample. Finally, the running times of the tool are quite reasonable: On average, DESCARTES takes 1.21 seconds per benchmark to analyze all relevant k-safety properties.

7.2 Relational Operators from Github Projects

In our second experiment, we evaluated our approach on `equals`, `compare`, and `compareTo` methods assembled from top-ranked Java projects on Github, such as *Apache Storm*, *libGDX*, *Android* and *Netty*. Towards this goal, we wrote a script to collect relational operators satisfying certain complexity criteria, such as containing loops or at least 20 lines of code and 4 conditionals. Our script also filters out comparators containing features that are currently not supported by our tool (e.g., bitwise operators, reflection).

In total, we ran DESCARTES over 2000 LOC from 62 relational operators collected from real-world Java applications (provided under supplemental materials). Specifically, our benchmarks contain 28 comparators and 34 `equals` methods. In terms of running time, DESCARTES takes an average of 9.14 seconds to analyze all relevant k-safety properties of the relational operator. Furthermore, the maximum verification time across all benchmarks is 55.27 seconds. Among the analyzed methods, DESCARTES was able to automatically verify all properties in 52 of the 62 benchmarks. Upon manual inspection of the remaining 10 relational operators, we discovered that DESCARTES reported 5 false positives, owing to weak loop invariants. However, the five remaining methods turned out to be indeed buggy — they violate at least one of the three required k-safety properties.

In summary, DESCARTES was able to verify complex real-world Java relational operators with a very low false positive rate of 8%. Furthermore, DESCARTES was able to uncover five real bugs in widely-used and well-tested Java projects. We believe this experiment demonstrates that reasoning about k-safety properties is difficult for programmers and that tools like DESCARTES can help programmers uncover non-trivial logical errors in their code.

7.3 Comparison with Self-Composition and Product

In our last experiment, we compare DESCARTES with two competing approaches, namely the self-composition [1] and product construction [3, 5] methods, which are the only existing techniques for verifying k-safety properties. Recall that both of these approaches explicitly construct a new program that is subsequently fed to an off-the-shelf verifier. To compare our technique with these approaches, we use the *non-buggy* benchmarks considered in Sections 7.1 and 7.2.

Self-composition. Given Cartesian Hoare triple $\|\Phi\|\ S\ \|\Psi\|$ of arity k, we emulate the self-composition approach by generating k-alpha-renamed copies of S and then verifying the standard Hoare triple $\{\Phi\}S_1;\ldots;S_k\{\Psi\}$. However, for the benchmarks from Sections 7.1 and 7.2, this strategy fails to verify more than half (52%) of the examples. Furthermore, for the examples where self composition is able to prove the relevant k-safety properties, it is $\sim 20\times$ slower compared to DESCARTES. We believe this result demonstrates that self-composition is not a viable strategy for verifying k-safety properties in many programs.

Product programs. As mentioned earlier, another strategy for verifying k-safety is to explicitly construct a so-called product program and use an off-the-shelf verifier. Unfortunately, there is no existing tool for product construction in Java, and there are several possible heuristics one can use for constructing the product program. Furthermore, previous work on sound product construction [3] only considers loops of the form `while(C) do S`, making it difficult to apply this technique to our benchmarks, almost all of which contain break or return statements inside loops. [4]

Hence, even though a completely reliable comparison with the product construction approach is not feasible, we tried to emulate this approach by considering two different strategies for product construction:

- ***S1:*** We eliminated break statements in loops in the standard way by introducing additional boolean variables.[5]
- ***S2:*** We eliminated break statements in loops by using our Transform rules from Figure 6.

In both strategies, our product construction tries to maximize the use of the Fusion rules from Figure 6, since this rule appears to be critical for successful verification.

Product construction with S1. In terms of the overall outcome, this product construction strategy yielded results very similar to self-composition. In particular, we were only able to verify half of the benchmarks (50.7% to be exact), and analysis time was a lot longer (143 seconds/benchmark for product construction vs. 8 seconds for DESCARTES). While it may be possible to decrease the false positive rate of this approach using a more sophisticated invariant generator, this result demonstrates that our approach is effective even when used with a relatively simple invariant generation technique.

Product construction with S2. When using strategy *S2* together with the Fusion rules of Figure 6, we were able to prove 85% of the correct benchmarks, but verification became significantly slower. In particular, DESCARTES is,

[4] The product construction described in [5] can, in principle, handle arbitrary control flow. However, the generated product program is only sound under certain restricted conditions, which must be verified using non-trivial verification conditions.

[5] In this approach, `while(true){S; if(b) break;}` is modeled as `f=true; while(f){S; if(b) f=false;}`

on average, $\sim 21\times$ faster compared to the explicit product construction approach, and, in several cases, three orders of magnitude faster. However, since the Transform rule is a contribution of this paper, it is entirely unclear whether previous product construction approaches can verify the 71 correct benchmarks we consider here.

8. Related Work

The term *2-safety property* was originally introduced by Terauchi and Aiken [23]. They define a 2-safety property to be "a property which can be refuted by observing two finite traces" and show that *non-interference* [16, 19] is a 2-safety property. Clarkson and Schneider generalize this notion to k*-safety hyperproperties* and specify such properties using second-order formulas over traces [10]. However, they do not consider the problem of automatically verifying k-safety.

Self-composition. Barthe et al. propose a method called *self-composition* for verifying secure information flow [1]. In essence, this method reduces 2-safety to standard safety by sequentially composing two alpha-renamed copies of the same program. Self-composition is, in theory, also applicable for verifying k-safety since we can create k alpha-renamed copies of the program and use an off-the-shelf verifier. While theoretically complete, self-composition does not work well in practice (see Section 7 and [23]).

Product programs. Our work is most closely related to a line of work on *product programs*, which have been used in the context of translation validation [18], proving non-interference [3, 5] and program optimizations [21]. Given two programs A, B (with disjoint variables), these approaches explicitly construct a new program, called the *product* of A and B, that is semantically equivalent to $A; B$ but is somehow easier to verify. While there are several different techniques for creating product programs, the key idea is to execute A and B in lockstep whenever possible, with the goal of making verification easier. Unlike these approaches, we do not explicitly create a full product program that is subsequently fed to an off-the-shelf verifier. In contrast, our approach combines semantic reasoning with the construction of Hoare triples that are relevant for verifying the desired k-safety property.

Relational program logics. Our work is also closely related to a line of work on relational program logics [6, 25]. Benton originally proposed *Relational Hoare Logic (RHL)* for verifying program transformations performed by optimizing compilers [6]. While Benton's RHL provides a general framework for relational correctness proofs, it can only be used when two programs always follow the same control-flow path, which is not the case for different executions of the same program. Several extensions of Benton's RHL have been devised by Barthe et al. to verify security properties of probabilistic systems, where the closest to our work is *pRHL* [2]. While pRHL is a relational logic specialized for probabilistic systems, our core CHL calculus is a generalization of pRHL for proving general k-safety properties for $k > 2$, modulo the rules dedicated to probabilistic reasoning. Specifically, CHL allows the exploration of any arbitrary interleaving of k programs and does not require any form of structural similarity between programs. It is unclear how to use prior work on relational program logics for proving k-safety properties for $k > 2$.

Another key difference of this work from previous techniques is the focus on automation. Specifically, unlike prior work that uses relational hoare logics, our verification procedure is fully automated and does not rely on interactive theorem provers to discharge proof obligations. Another common limitation of previous relational logics (e.g., RHL and pRHL) is that they do not yield satisfactory algorithmic solutions for reasoning about asynchronous loops that execute different numbers of times. In contrast, we propose a specialized calculus to reason about both synchronous and asynchronous loops and show that our approach can be successfully automated. Furthermore, our approach can reason about loops that contain multiple exit points, whereas previous work only considers loops of the form $\mathit{while}(C)\ \mathit{do}\ S$.

Bug finding. In this work, we focused on *verifying* the absence of k-safety property violations. While our technique is sound, it may have false positives. A complementary research direction is to develop automated bug finding techniques for *finding* violations of k-safety properties. One possible approach for automated bug detection is to construct a harness program using self-composition and then use off-the-shelf bug detection tools, such as dynamic symbolic execution [8, 15] or bounded model checking [7, 9]. Based on our experience with the CATG tool [22] for dynamic symbolic execution, this naive approach based on self-composition does not seem to scale for examples that contain input-dependent loops. For example, we were not able to successfully use dynamic symbolic execution to detect the bug in the comparator from Figure 1 since CATG times-out. We believe a promising direction for future research could be to explore complementary bug finding techniques for detecting violations of k-safety properties.

9. Conclusion

We have proposed *Cartesian Hoare Logic (CHL)* for proving k-safety properties and presented a verification algorithm that fully automates reasoning in CHL. Our approach can handle arbitrary values of k, does not require different runs to follow similar paths, and combines the verification task with the exploration of k simultaneous executions. Most importantly, our approach supports full automation and does not require the use of interactive theorem provers. Our evaluation shows that DESCARTES is practical and gives good results when verifying five different k-safety properties of relational operators in Java programs. Our comparison also demonstrates the advantages of DESCARTES over self-

composition and explicit product construction approaches, both in terms of precision as well as running time.

Acknowledgments

The authors would like to thank Valentin Wustholz and Kostas Ferles for their help with the CATG tool and Doug Lea, Hongseok Yang, Thomas Dillig, Yu Feng, Navid Yaghmazadeh, Vijay D'Silva, Ruben Martins, and the anonymous reviewers for their helpful feedback.

References

[1] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 100–114. IEEE, 2004.

[2] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 90–101. ACM, 2009.

[3] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM 2011: Formal Methods*, pages 200–214. Springer, 2011.

[4] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–110. ACM, 2012.

[5] G. Barthe, J. M. Crespo, and C. Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In *Logical Foundations of Computer Science*, pages 29–43. Springer, 2013.

[6] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN Notices*, volume 39, pages 14–25. ACM, 2004.

[7] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[8] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[9] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[10] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *Computer Security Foundations Symposium, 2008. CSF'08. IEEE 21st*, pages 51–65. IEEE, 2008.

[11] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.

[12] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[13] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1): 107–113, 2008.

[14] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *FME 2001: Formal Methods for Increasing Software Productivity*, pages 500–517. Springer, 2001.

[15] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[16] J. A. Goguen and J. Meseguer. *Security policies and security models*. IEEE, 1982.

[17] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[18] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166. Springer, 1998.

[19] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[20] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *Programming Languages and Systems*, pages 574–592. Springer, 2013.

[21] M. Sousa, I. Dillig, D. Vytiniotis, T. Dillig, and C. Gkantsidis. Consolidation of queries with user-defined functions. *PLDI*, 49(6):554–564, 2014.

[22] H. Tanno, X. Zhang, T. Hoshino, and K. Sen. Tesma and catg: Automated test generation tools for models of enterprise applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, pages 717–720. IEEE Press, 2015.

[23] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Static Analysis Symposium (SAS)*. Springer, 2005.

[24] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[25] H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1):308–334, 2007.

[26] A. Zaks and A. Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *FM 2008: Formal Methods*, pages 35–51. Springer, 2008.