# Regression Verification for Java Using a Secure Information Flow Calculus

Bernhard Beckert
Karlsruhe Institute of Technology
beckert@kit.edu

Vladimir Klebanov
Karlsruhe Institute of Technology
klebanov@kit.edu

Mattias Ulbrich
Karlsruhe Institute of Technology
ulbrich@kit.edu

## ABSTRACT

Regression verification and checking for illicit information flow in programs are probably the two most prominent instances of so-called relational program reasoning. Regression verification is concerned with proving that two programs behave either equally or differently in a formally specified manner; information-flow checking aims to establish that the attacker cannot distinguish runs of a program that vary in a part of the initial state designated as secret. While the theoretical connections between the two problems are well understood, there are also subtle but significant pragmatical differences. This paper reports the results of an experiment to adapt a state-of-the-art deductive information-flow verification system for Java to the problem of regression verification.

## 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

Regression verification; program equivalence; secure information flow; formal methods

## 1. INTRODUCTION

### *Overview.*

Over the last years, there has been a growing interest in *relational* verification of programs, which reasons about the relation between the behaviour of two programs or program runs – instead of comparing a single program or program run to a more abstract specification. The main advantage of relational verification over standard functional verification is that there is no need to write and maintain complex specifications. The effort for relational verification mainly depends on the difference between the programs resp. program runs and not on the overall size and complexity of the program(s); one can thus exploit the fact that differences are often local and only affect a small portion of a program.

Relational verification can be used for various purposes. An example is *regression verification*, which requires to compare two different versions of a program for the same input. Another example is the verification of *secure information flow* (SIF) properties, which requires to compare runs of the same program for different inputs.

While the theoretical connection between different kinds of relational verification is well understood and relational verification methods for different purposes have much in common, there are also subtle but significant pragmatical differences. And, so far, relational verification tools are all dedicated to a particular purpose.

Our contribution described in this paper is an experiment where we make use of the similarities between different kinds of relational verification. We have adapted a state-of-the-art deductive information-flow verification system for Java to the problem of regression verification. This is done by reducing instances of regression verification problems to instances of information-flow verification.

It turns out that by exploiting the close connection between the two problems, one can successfully transform theoretical concepts, techniques and tools for information-flow verification into concepts, techniques and tools for regression verification. However, the experiment also shows that the pragmatics of how to use the technology remain different, and different heuristics and optimisations are needed for scalable and efficient tool use.

### *Regression verification.*

One of the main concerns during software evolution is to prevent the introduction of unwanted behavior, commonly known as *regressions,* when implementing new features, fixing defects, or during optimization. Undetected regressions can have severe consequences and incur high cost, in particular in late stages of development, or in software that is already deployed. Currently, the main quality assurance measure during software evolution is *regression testing* [1]. Regression testing uses a carefully crafted test suite to check that a modified version of a program is equivalent to the original one in relevant behavioral aspects.

Regression verification is a complementary approach that attempts to achieve the same goals with techniques from formal verification. This means establishing a formal proof of equivalence of two program versions. In its basic form, we are trying to prove that the two versions produce the same output for all inputs. In more sophisticated scenarios, we want to verify that the two versions are equivalent only on some inputs (*conditional equivalence*) or differ in a formally specified way (*relational equivalence*). Regression verification is an attractive additional instrument of software quality assurance and is not intended to replace testing. On the other hand, if regression verification is successful, it offers guaranteed coverage without requiring additional expenses to develop and maintain a test suite.

*Secure information flow.*

In complex software systems, one cannot easily tell how the input data is processed and flows onto output channels. At the same time, more and more data is brought into the reach of software systems and more and more data can potentially flow via dubious channels. The question that is raised for secure information flow (SIF) properties of a program is whether the output of a program depends on secret inputs to the program, or how much it depends. A typical example of an SIF property is confidentiality: An attacker must not gain information on secret data by inspection of the public outputs: the latter must be totally independent from the former. The survey paper [18] gives an introduction into the area of language-based SIF analyses.

Information flow is a relational property of a program. To define this property, it is typically assumed that the program has inputs and outputs that are either secret (also called "high") or public (also called "low"). The goal of an attacker is to learn the secret input by observing the public output. The attacker knows the source code of the program and can control the public input. The degree of success that the attacker can attain is the information flow of a program. In particular, a program has no (illicit) information flow, if its public input alone completely *determines* its public output. Thus, for any given public input, nothing can be discerned about the secret input, as the public output is always constant.[1]

*The KeY Tool and its logic.*

We present the translation from regression verification to secure information flow properties for the specification and verification framework within the KeY system [5]. The core of KeY is a theorem prover for a program logic combining a variety of automated reasoning techniques that also allows for interactive reasoning. KeY's underlying program logic is *Dynamic Logic* (DL) [11], a first order multi-modal logic. DL extends first order logic (FOL) with two families of modal operators: $\langle P \rangle$ ("diamond") and $[P]$ ("box") where $P$ is a program fragment. The formula $\langle P \rangle \phi$ expresses that the program $P$ terminates in a state in which $\phi$ holds, while $[P]\phi$ does not demand termination. If $\phi$ is a FOL formula, $\langle P \rangle \phi$ corresponds to the weakest precondition of $P$ w.r.t. $\phi$; the DL formula $\phi \rightarrow [P]\psi$ corresponds to $\{\phi\}\, P\, \{\psi\}$ in Hoare logic [13]. The program fragment inside the modal operators are code pieces in the Java programming language. KeY fully supports the JavaCard language and most of the language constructs available in full sequential Java.

KeY allows the verification of Java programs against a formal specification in the Java Modeling Language (JML), a behavioural interface specification language for Java [15]. JML specifications define formal *functional* requirements (like method pre- and postconditions or class invariants). Loops that occur in program fragments can be handled in KeY by unwinding or by abstraction (in which case a JML *loop invariant* must be provided by the specifier).

[1]A full definition of information flow in terms of the input-output function's equivalence kernel can be found, e.g., in [14].

*Related work.*

Various methods and tools both for regression verification and SIF verification have been presented in the literature. But most focus on their particular application for relational verification, and they do not consider a cross-fertilisation.

For regression verification, Godlin and Strichman [9] present an approach for automatic general-purpose regression verification. The technique is implemented in the RVT tool and supports a subset of ANSI C. Verdoolaege et al. [22] have developed an automatic approach to prove equivalence of static affine programs. It is implemented in the isa tool for the static affine subset of ANSI C. Hawblitzel et al. [12] have put forth the idea of mutual function summaries. This concept is implemented in the equivalence checker SymDiff, where the user supplies the mutual summary, and the verification conditions are discharged by Boogie. The BCVerifier tool of Welsch and Poetzsch-Heffter allows to prove the backwards compatibility of Java class libraries [23]. Felsing et al. [8] present a method for proving the equivalence of two related imperative integer programs, implemented in the ReVe tool.

Several tools and approaches exist in the literature for checking information-flow properties. Security type systems are one of the most popular approaches. A prominent example in this field is the JIF system [16]. Type system approaches are efficient, but sometimes also quite imprecise. A further approach is checking the dependence graph of a program for graph-theoretical reachability properties [10]. Though this technique is substantially different from type system approaches, it is efficient and sometimes quite imprecise, too. Further approaches use abstraction and ghost code for explicit tracking of dependencies [6]. The most popular approach in logic based information flow analysis is stating secure information flow with the help of self-composition [3, 4, 7] and using off-the-shelf software verification systems to check for it, as we do. Finally, secure information flow can be formalized in higher-order logic, and higher-order theorem provers like Coq can be used for checking secure information flow [17].

A general purpose relational verification calculus is presented by Barthe et al. [2]. The calculus is based on pure program transformation; it offers rules to merge two programs into a single product program.

*Structure of the paper.*

Sect. 2 formally defines SIF (Sect. 2.2) and regression verification (Sect. 2.3) and how the latter can be reduced to the former (Sect. 2.4). In Sect. 3 the reduction technique is lifted to the Java language. We report on our experiences with reusing the SIF calculus in Sect. 4.

# 2. FORMAL FOUNDATIONS

## 2.1 Programs and states

A *program state* is a logical structure assigning values to program variables and reachable memory locations. We refer to the set of all possible states for a given program as $S$. Every syntactically valid program $P$ describes a *state transition relation* $\rho_P \subseteq S \times S$ on program states. If the program $P$ started in state $s$ terminates in state $s'$, then (and only then) $(s, s') \in \rho_P$ (we call such a tuple an *execution of* $P$). Relation $\rho_p$ is fixed by the semantics of the programming language. We only consider deterministic and terminating programs $P$. This means that all state transition relations $\rho_P$ are actually total functions: for every initial state, there is exactly one final state.

The restriction to deterministic programs is natural as Java is a deterministic language. The analysis of SIF properties presented in the following will compare the poststates of terminating program runs. We therefore assume that all investigated methods are terminating by design. For the verification of SIF properties of reactive systems, other trace-based approaches may be better suited. The termination is left to be shown using some other technique (e.g. functional KeY).

## 2.2 SIF by self-composition

It has already been mentioned that for SIF properties, a distinction of secret and public data must exist. In many language-based approaches, from all program variables $V$ a subset $L \subseteq V$ can be marked as "low" (constituting the public part of the state). Two states $s_1, s_2 \in S$ are called $L$-equivalent (we write $s_1 \approx_L s_2$) if they evaluate all variables in $L$ equally ($l^{s_1} = l^{s_2}$ for all $l \in L$). To be secure, information must not flow from $V \setminus L$ to $L$:

Definition 1 (Secure information flow). *Let $P$ be a program, and $L$ be a set of variables. The program $P$ has secure information flow w.r.t. to the (low) variables $L$ if, for any two executions $(s_1, t_1), (s_2, t_2) \in \rho_P$ of $P$, the assumption $s_1 \approx_L s_2$ implies $t_1 \approx_L t_2$.*

This definition says that a secure program started in $L$-equivalent states always terminates in $L$-equivalent states, not leaking any information from $V \setminus L$ to $L$. The following proof obligation encodes absence of illicit information flow as a DL formula:

$$(\bigwedge_{l \in L} l_{@1} = l_{@2}) \to [P]_{@1}[P]_{@2}(\bigwedge_{l \in L} l_{@1} = l_{@2}) \qquad (1)$$

This formula uses two modalities $[P]$ to evaluate two executions of $P$, a technique called *self-composition* [7]. The two iterations must operate on different copies $V_1$ and $V_2$ of the same variable set $V$. We use shorthand notation like $l_{@1}$ for a term denoting the value of $l$ in state $s_1$ (first copy), and $[P]_{@1}$ for symbolic execution of the program on $V_1$.

## 2.3 Regression verification by composition

Semantically regression verification is less involved than SIF. In its basic form, we require that the two programs $P_1$ and $P_2$ behave completely equivalently:

Definition 2 (Perfect program equivalence). *Two programs $P_1$ and $P_2$ are equivalent if their state transition relations $\rho_{P_1}$ and $\rho_{P_2}$ are the same.*

The proof obligation for regression verification

$$(\bigwedge_{v \in V} v_{@1} = v_{@2}) \to [P_1]_{@1}[P_2]_{@2}(\bigwedge_{v \in V} v_{@1} = v_{@2}) \qquad (2)$$

is thus similar to (1) with the difference that (i) two different programs and (ii) that all variable in $V$ (not only $L$) are considered.

To obtain notions of (a) conditional and (b) relational equivalence Def. 2 can be modified accordingly by (a) allowing a condition under which equality must hold or (b) replacing the requirement that states are the "same" by another relation. We will not look into that here further.

## 2.4 Regression verification as information flow

This section reports how regression verification (or program equivalence) can be reduced to the problem of verifying the absence of illicit information flow. Both problems are relational in the sense that they are concerned with indistinguishability of program executions: we pick two arbitrary state transitions $(s_{1/2}, t_{1/2})$ in the following. The differences between the two problems are as follows:

- For regression verification, two different programs $P_1$ and $P_2$ are compared, but the runs share the same initial state ($s_1 = s_2$). Indistinguishability is defined as equality of final states ($t_1 = t_2$).
- For SIF, the state transitions originate from the same program $P$, but the initial states coincide only in their non-secret part ($s_1 \approx_L s_2$), as well as the final states are only supposed to be indistinguishable in their non-secret part ($t_1 \approx_L t_2$).

To reduce the former problem to the latter, we synthesize a new program $Q$ as follows:

$$Q \triangleq \texttt{if(h) \{ } P_1 \texttt{ \} else \{ } P_2 \texttt{ \}} \qquad (3)$$

where `h` is a fresh boolean variable, which decides which of $P_1$ or $P_2$ is to be executed. If `h` is a secret that the attacker cannot learn by choosing the input for $Q$ and observing its output (both times not including `h` obviously), then the two program are equivalent.

Theorem 1. *Let two code blocks $P_1$ and $P_2$ in an imperative while language over the same variables be given and $h$ be a variable which does not occur in $P_1$ and $P_2$. Then: $Q$ as defined in (3) is non-interferent w.r.t. all variables but $h$ if and only if $P_1$ and $P_2$ are equivalent.*

Termination is an issue with relational verification in general. What is the status of the property if one of the two executions terminates and the other does not? Is Information leaked? Are programs equivalent. In functional verification, termination is often ignored (partial correctness). We will do likewise here and leave the orthogonal problem of mutual termination unconsidered.

# 3. LIFTING TO JAVA

After the considerations in the last section that operated on simplified languages (only program variables), we shall now lift the specification and verification conditions to Java. This means in particular that states incorporate heaps now.

## 3.1 Specifying information flow with JML*

KeY already supports a language for specifying SIF as part of its JML* extension of JML. This extension was originally published in [20] though we refer the interested reader to the more up-to-date information source [19].

```
//@ determines \result \by l1, l2;
int a(int h, int l1, int l2) { ... }
```

(a) No flow from `h` to `\result`

```
//@ determines \result \by \nothing;
int b(int h) { ... }
```

(b) No flow from `h` to `\result`

```
int x, y;
//@ determines x+y \by \itself;
void c(int h) { ... }
```

(c) No flow from `h` to `x+y`.

**Figure 1: Examples of specifying information flow in JML***

The main instrument for specifying absence of illicit information flow with JML* is a *determines clause* that can be attached to method declarations. A determines clause reads as follows:

$$\texttt{//@ determines } o_1, \ldots, o_m \texttt{ \textbackslash by } i_1, \ldots, i_n; \qquad (4)$$

in which the list of JML expressions $o_1, \ldots, o_m$ specifies the public output values and the expression list $i_1, \ldots, i_m$ the public input. The KeY concept of information flow is more flexible than the scenario outlined in Sect. 2 as it is possible to list *expressions* (rather than program variables) that take the role of the public ("low") input and output.

Example 1. *The specification shown in Figure 1(a) says that the return value of the method* `a` *is completely determined by the method parameters* `l1` *and* `l2`*. This means that the method parameter* `h` *has no influence on it. There is no information flow from* `h` *to the return value of* `f`*.*

*In the specification for* `b` *in Figure 1(b) there are no public input values (hence "*`\by \nothing`*"), yet the result is public and must not depend oh* `h` *–* `b` *must return a constant value regardless of the input state.*

*In Figure 1(c) the expression* `x+y` *makes up the public output and input*[2]*. This means that the individual values of* `x` *and* `y` *after the execution of* `c` *may very well depend on* `h`*, but their sum must be the same regardless of the value of* `h`*.*

The definition of information flow security for a method with a determines clause is an adaptation of Def. 1 to the situation of Java. The notion of $L$-equivalence gives way to two equivalences $\approx_i, \approx_o$ on the input and output states based on the expressions in the determines clause.

Definition 3. *(*Secure information flow for `determines` clauses*) If, for a list $x_1, \ldots, x_n$ of expressions, every expression $x_i$ evaluates equally in two states $s_1, s_2 \in S$, we call $s_1$ and $s_2$ equal w.r.t. $x_1, \ldots, x_n$ and write $s_1 \approx_x s_2$.*

*A Java method* `m` *with its* `determines` *clause according to* (4) *has secure information flow with respect to that determines clause if, for any two executions $(s_1, t_1), (s_2, t_2) \in \rho_P$ of $P$, the assumption $s_1 \approx_i s_2$ implies $t_1 \approx_o t_2$.*

In correspondence to (1) for SIF in a simple while language, the SIF proof obligation for Java program `m` is

$$\Big(\bigwedge_{k=1}^{n} i_{k@1} = i_{k@2}\Big) \to [P]_{@1}[P]_{@2}\Big(\bigwedge_{k=1}^{m} o_{k@1} = o_{k@2}\Big) \ . \qquad (5)$$

[2]the notation `\by \itself` is an abbreviation for repeating the same expressions

(5) captures the essential parts of the proof obligation, the formula created within the KeY calculus is more complicated as the heap constructs call for more encoding. Full details of the calculus can be found in [21].

## 3.2 Weaving Java programs

In this section we report how the program composition from (3) can be extended such that two Java programs can be woven into a single program. Showing non-interference on the combined program entails equivalence for the two original programs.

For the reasoning within the information flow calculus that deals with the flow of one program, the two revisions of the program must be merged into one single program. Here, by "Java program," we understand a collection of Java classes. Like in (3), a synthetic variable `h` is employed to distinguish the control flow for the two original programs. It determines the semantics of the woven program: By choosing `h` to be true, the semantics of the original revision is assumed, otherwise the program behaves undistinguishably from the second revision.

The programs are woven in a method-by-method fashion. Since we want to compare two related revisions of the same program and not two unrelated different programs, a syntactic resemblance between the class collections can be assumed. We assume that the change in the program only concerns the code within method bodies. In particular, that means that all method signatures and return types are left untouched in the course of the evolution step. It is possible to extend the presented approach also to the case that method signatures are modified. That makes the translation more technically involved but does not contribute to the lessons to be learnt from the paper; that is why we leave such changes aside here.

In its initial version, the weaving of two revisions of a Java method happens by combining both method bodies into one. To this end, every method receives an additional synthetic boolean argument `h`. Within every combined method, an artificial case distinction is added to distinguish between the behaviours of the two programs. The semantics now depends on the global variable `h`, which decides about the path and thus the program version to be executed.

Example 2. *A Java program contains a method computing the Gaussian triangular sum up to n. During a code revision, a reviewer requires that this algorithm should be implemented differently and changes the implementation, modifying the range of the control variable* `x`*. The two versions (before and after the revision) of the method are the following:*

```
int triangle(int n){            int triangle(int n){
  int x = 0;                      int x = 1;
  int sum = 0;                    int sum = 0;
  while(x < n) {                  while(x <= n) {
    sum += x + 1;                   sum += x;
    x++;                            x++;
  }                               }
  return sum;                     return sum;
}                               }
```

*A proof of equivalence of the two revisions is desired. Hence the programs are woven into one which is then checked for non-interference:*

```
int triangle(boolean h, int n) {
```

```
if(h) {
  int x = 0;
  int sum = 0;
  while(x < n) {
    sum += x + 1;
    x++;
  }
  return sum;
}
```

```
else {
  int x = 1;
  int sum = 0;
  while(x <= n) {
    sum += x;
    x++;
  }
  return sum;
}
}
```

By applying such a combining step to every method within the program revisions $P_1$ and $P_2$, we receive a program $Q$ that contains all possible executions of $P_1$ and $P_2$. The value of the parameter `h` decides which implementation is chosen.

However, a basic single program evolution step is usually local and touches only on a very limited part of the code base. Most of the existing code is retained as it was before the modification. The weaving procedure described so far would thus reduplicate much code without necessity.

One can do better and reuse shared code by pulling the case distinction further into the method body. Apart from the obvious effect that it shortens the resulting program, this also eases the reasoning about security properties since more code is shared and can be subject to the SIF verification techniques. In the extreme, if a method is not touched at all by the revision, the case distinction can be dropped altogether and non-interference needs not be proved since the woven method makes no reference to the variable `h` and its result cannot depend on it.

Example 2 (cont'd). *Instead of the simple weaving shown above, the two implementation of the triangular loop can also be combined as follows yielding a semantically equivalent method:*

```
int triangle(boolean h, int n) {
  int x = h ? 0 : 1;
  int sum = 0;
  while(h ? x < n : x <= n) {
    sum += h ? x+1 : x;
    x++;
  }
  return sum;
}
```

*The case distinctions have been moved inside the code block as far as possible to make the code changes as local as possible.*

It is not only code sharing that can be achieved during weaving. Partial loop unwinding, method inlining and clever rearrangement plays also an important role in the process of weaving programs. We shall not elaborate on this matter here, but propose to follow the ideas presented in an earlier work [8]. A more general account of how two programs can be woven into one is by Barthe et al. [2].

## 3.3 Relational annotations

With the woven program, proving equivalence now means proving absence of illicit information flow. However, the SIF property to be verified still needs to be defined: The result of a method shall not depend on the decision variable `h` but it may, of course, depend on the other method parameters. This means that information may flow from all heap locations and the method parameters into all heap locations and the method result. As a JML* SIF method contract this is annotated as follows:

`//@determines \result,\heap \by \heap,` $p_1, p_2, ..., p_n$`;`

indicating that the result value and all values on the heap depend on the values of the heap and the parameters $p_1, ..., p_n$ of the method. The only "high" part of the state here is the decision variable `h` upon which the result is not to depend.

With the SIF method specification added to the source code, we could proceed with the verification process. However, the program's loops need special attention. In functional verification, a loop invariant (with other annotations) guides the verification engine into proving a program with loops correct. Likewise we need a relational SIF loop annotation. An annotation similar to the ones for methods can be attached to loop statements in JML* to indicate how information flows from loop iteration to loop iteration. The clause `determines` $c_1, \ldots, c_n$ for a loop lists (JML) expressions $c_i$ that are independent of the secret in any loop iteration – and by induction principle throughout the entire loop. In our case of regression verification that means these expressions have the same values in the program states related to $P_1$ and $P_2$ in every loop iteration: $\bigwedge_{i=1..n} c_i^{s1} = c_i^{s2}$. They thus serve as *coupling invariants* between the two programs. Their shape is fixed to a conjunction of equalities between the same terms evaluated in both states. Often, a functional relationship between the program states exists but cannot be expressed as equality of the same expression but of different terms $t_1$ and $t_2$. In the SIF framework of JML*, we can express an equality between different terms by using the ternary if-then-else operator and the decision variable `h`. The expression `h ?` $t_1$ `:` $t_2$ evaluates to $t_1$ in the first execution and $t_2$ in the second.

Example 2 (cont'd). *The SIF contract for the above method* `triangle` *is*

```
//@ determines \result, \heap \by \heap, n;
```

*and a SIF loop annotation which is sufficient to imply the method contract is the following:*

```
//@ determines n, sum, h ? x+1 : x \by \itself;
```

*The latter implies that every iteration of both loops establishes* $n_{@1} = n_{@2}$ *and* $x_{@1} + 1 = x_{@2}$.

## 3.4 Object creation

Two programs are usually considered equivalent if their state transition functions are identical. If all program variables are of primitive datatypes, the comparison by identity is a sensible requirement. For object references the situation is different: The actual object identity (i.e., the memory location at which an object resides) is not of relevance since that can never be investigated by a Java program.[3] In the Java programming language, pointer arithmetic is not supported and references cannot compared other than by the `==` operator. Thus the actual memory location of an object is irrelevant, it is relevant how it *compares* to other references. It is therefore sufficient to relax the equivalence requirement for object-oriented routines to termination in *isomorphic* states. Two states are isomorphic if there exists a permutation (automorphism) of the object identities such that updating one state and all references to objects within it with the permutation yields the other state. Details about

[3]Assuming that identity revealing methods like `Object.hashCode()` are not allowed for the moment.

defining a theory for isomorphisms can be found in an earlier work [4].

The SIF extension to JML* provides possibilities to explicitly state the isomorphism under which non-interference is guaranteed. KeY supports the proof of such relaxed verification conditions.

EXAMPLE 3. *The following two methods are not identical since they do not produce the same results – but they are equivalent up to object isomorphism.*

```
class C {                    class C {
  C x, y;                       C x, y;
  void m() {                    void m() {
     x = new C();                y = new C();
     y = new C();                x = new C();
}}                           }}
```

# 4. LESSONS LEARNT

We have manually applied the transformation described in Sect. 3 to reduce Java regression verification problems to equivalent SIF problems, annotated the resulting code and proved them using KeY. The set of tested programs contains classes with one or two methods containing loops, recursion and/or object creation. Some proofs required manual interaction.

The following observations could be made:

1. *It works conceptually.* It is possible to specify and verify equivalence using an SIF calculus. Despite the fact that the shape of coupling invariants is limited (only conjunctions of equations, cf. Sect. 3.3), we experienced that relational properties always could be expressed within this framework (potentially using if-then-else expressions).

2. *It works practically for small examples.* Smaller examples (like the ones in this paper) can be proved using KeY's SIF calculus. The proof space grows rather large for relational proofs and automatic verification may need a minute or so even for very small examples. In case of a failed verification (e.g., due to a missing annotation), analysis of unclosed proof goals was difficult because it is hard to tell logical entities from the two program executions apart.

3. *The pragmatics for regression verification and SIF are different.* When it comes to exception handling, we experienced a noticeable difference between equivalence checking and SIF: For equivalence, behaviour should be retained also in cases of abnormal termination. For SIF (as it is handled in KeY), the exceptional case is usually excluded by functional preconditions. One is only interested in information flow in intended method usage. This required us to annotate more functional specifications and loop invariants (dealing with exceptions) than would actually be necessary. That is not a limitation of the approach – but the calculus has been trimmed towards its typical use case to make it more effective.

As a conclusion one can say that reducing one relational proof obligation (namely regression verification) to another relation proof obligation (namely SIF) is possible but that, for pragmatic reasons, a calculus that is tailored for the particular use case has advantages over one tailored for another use case.

# 5. REFERENCES


[1] AMMANN, P., AND OFFUTT, J. *Introduction to Software Testing*, first ed. Cambridge University Press, New York, NY, USA, 2008.

[2] BARTHE, G., CRESPO, J. M., AND KUNZ, C. Relational verification using product programs. In *FM 2011* (2011), vol. 6664 of *LNCS*, Springer, pp. 200–214.

[3] BARTHE, G., D'ARGENIO, P. R., AND REZK, T. Secure information flow by self-composition. CSFW '04, IEEE CS, pp. 100–115.

[4] BECKERT, B., BRUNS, D., KLEBANOV, V., SCHEBEN, C., SCHMITT, P. H., AND ULBRICH, M. Information flow in object-oriented software. In *LOPSTR 2013, Revised Selected Papers* (2014), vol. 8901 of *LNCS*, Springer, pp. 19–37.

[5] BECKERT, B., HÄHNLE, R., AND SCHMITT, P. H., Eds. *Verification of Object-Oriented Software: The KeY Approach*, vol. 4334 of *LNCS*. Springer, 2007.

[6] BUBEL, R., HÄHNLE, R., AND WEISS, B. Abstract interpretation of symbolic execution with explicit state updates. In *FMCO* (2008), pp. 247–277.

[7] DARVAS, Á., HÄHNLE, R., AND SANDS, D. A theorem proving approach to analysis of secure information flow. In *SPC 2005* (2005), vol. 3450 of *LNCS*, Springer, pp. 193–209.

[8] FELSING, D., GREBING, S., KLEBANOV, V., RÜMMER, P., AND ULBRICH, M. Automating regression verification. In *ASE 2014* (2014), ACM, pp. 349–360.

[9] GODLIN, B., AND STRICHMAN, O. Regression verification: proving the equivalence of similar programs. *JSTVR 23*, 3 (2013), 241–258.

[10] HAMMER, C., KRINKE, J., AND SNELTING, G. Information flow control for Java based on path conditions in dependence graphs. In *ISSSE* (March 2006), IEEE, pp. 87–96.

[11] HAREL, D., KOZEN, D., AND TIURYN, J. *Dynamic Logic*. MIT Press, 2000.

[12] HAWBLITZEL, C., KAWAGUCHI, M., LAHIRI, S. K., AND REBÊLO, H. Towards modularly comparing programs using automated theorem provers. In *CADE-24* (2013), vol. 7898 of *LNCS*, Springer, pp. 282–299.

[13] HOARE, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (Oct. 1969), 576–580.

[14] KLEBANOV, V. Precise quantitative information flow analysis – a symbolic approach. *Theoretical Computer Science 538*, 0 (2014), 124–139.

[15] LEAVENS, G. T., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D., MÜLLER, P., KINIRY, J., CHALIN, P., ZIMMERMAN, D. M., AND DIETL, W. JML Reference Manual, 2008.

[16] MYERS, A. C. JFlow: Practical mostly-static information flow control. In *POPL* (1999), pp. 228–241.

[17] NANEVSKI, A., BANERJEE, A., AND GARG, D. Verification of information flow and access control policies with dependent types. In *SP* (2011), pp. 165 –179.

[18] SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *IEEE J. Sel. Areas Commun. 21*, 1 (2003), 5–19.

[19] SCHEBEN, C. *Program-level Specification and Deductive Verification of Security Properties*. PhD thesis, Karlsruhe Institute of Technology, 2014.

[20] SCHEBEN, C., AND SCHMITT, P. H. Verification of information flow properties of Java programs without approximations. In *FoVeOOS 2011, Revised Selected Papers* (2011), vol. 7421 of *LNCS*, Springer, pp. 232–249.

[21] SCHEBEN, C., AND SCHMITT, P. H. Efficient self-composition for weakest precondition calculi. In *FM 2014* (2014), vol. 8442 of *LNCS*, Springer, pp. 579–594.

[22] VERDOOLAEGE, S., JANSSENS, G., AND BRUYNOOGHE, M. Equivalence checking of static affine programs using widening to handle recurrences. In *CAV 2009* (2009), vol. 5643 of *LNCS*, Springer, pp. 599–613.

[23] WELSCH, Y., AND POETZSCH-HEFFTER, A. Verifying backwards compatibility of object-oriented libraries using Boogie. In *FTfJP 14* (2012), FTfJP '12, ACM, pp. 35–41.