Chapter 13
Information Flow Analysis

Christoph Scheben and Simon Greiner

13.1 Introduction

Software systems are becoming increasingly trusted to handle sensitive information, though they have the potential to abuse this trust with serious consequences. in particular if they are connected to the internet. We allow web browsers to access our bank accounts, but also allow them to send usage reports to the browser's developers. A mainstream smartphone application has permissions to read our digital photo albums, contact lists and calendar, while at the same time it is free to use the phone's internet connection in every way possible. This chapter discusses how the KeY System can be used to address the increasingly important question of *information flow control*: does a program introduce information flows between resources in a way which is in violation of our security policy?

As a concrete example consider an electronic voting system: an important property of voting systems is the preservation of privacy of votes. Information on votes may not be published directly nor indirectly.

```
for (int i = 0; i < votes.lengh; i++) {
  publish(votes[i]);
}
```

Listing 13.1 Example for an explicit leak

In Listing 13.1, the secret value of a vote is directly written to the output channel `publish`. Therefore, this kind of information leak is called *explicit*. In Listing 13.2, the information is leaked indirectly via the control flow of the program. By observing the output, it is possible to decide whether the first vote was cast for candidate 0 or not. This is called an *implicit* leak. In complex programs, these leaks can be much more subtle.

W. Ahrendt et al. (Eds.): Deductive Software Verification, LNCS 10001, pp. 453–471, 2016
DOI: 10.1007/978-3-319-49812-6_13

```
if (votes[0] == candidates[0])
  publish("The␣result␣is␣");
  publish(calculateResult(votes, candidates));
} else {
  publish("The␣outcome␣is␣");
  publish(calculateResult(votes, candidates));
}
```

Listing 13.2 Example for an implicit leak

Information can also be leaked via *side channels*, such as execution time, power consumption, heat generation, and others. These kinds of information flow are not considered here. Instead, we focus on explicit and implicit leaks.

In order to verify a program for secure information flow, we need a general notion on what *secure information flow* means. Intuitively, a program has this property, if the observable output is not influenced by secret input, i.e., the observable output does not depend on the secret input. This is obviously the case, if for all program executions with the same nonsecret input, the public output is equal. Darvas et al. [2005] phrase this for a program α the following way: A program α has secure information flow if "Running two instances of α with equal low-security values and arbitrary high-security values, the resulting low-security values are equal, too." Here, low-security values are values which can be observed by potential attackers whereas all other values are called high-security values. This policy is called *noninterference* [Lampson, 1973, Denning, 1976, Cohen, 1977, Goguen and Meseguer, 1982].

For instance, let the observable output be the variable `l`, while all other variables are not observable. Then, the program `l = h + 1;` is insecure: Two runs of `l = h + 1;` with different values of `h` result in states with different values for `l`. If, on the other hand, neither `l` nor `h` are observable, the program has secure information flow. The program also has secure information flow, if `h` and `l` are observable. The program `l = 0; if (h) { l = 1; }` is insecure if solely the value of `l` is observable, because `l` has the value 0 if, and only if, `h` has the value `false`. The program `h = 0; if (l) { h = 1; }` on the other hand has secure information flow in thiscase. Indeed, `l` is not changed at all.

In the past, a variety of sophisticated information flow analysis techniques and tools have been developed. As in functional verification, the proposed techniques can be divided into lightweight (that is, automatic but approximate) and heavyweight (that is, semiautomatic but precise approaches.

Popular lightweight approaches are security type systems (a prominent example in this field is the Java Information Flow (JIF) system by Myers [1999]), the analysis of program dependence graphs for graph-theoretical reachability properties [Hammer et al., 2006], specialized approximate information flow calculi based on Hoare like logics [Amtoft et al., 2006, Scheben, 2014] and the usage of abstraction and ghost code for explicit tracking of dependencies [Bubel et al., 2009]. A popular heavyweight approach is to state information flow properties by self-composition [Barthe et al., 2004, Darvas et al., 2005] and use off-the-shelf software verification systems to check for them. An alternative is to formalize information flow properties in higher-order logic and use higher-order theorem provers for the verification of those properties, as presented for instance by Nanevski et al. [2011].

Lightweight approaches are usually efficient and scale well on large programs, but do not have the necessary precision to express and verify complex information flow-properties of programs with controlled release of information. An instance of programs with controlled release of information are electronic voting systems. In those systems, secrecy of votes is an important property which could not be proven by approximate approaches so far. Heavyweight approaches on the other hand were, until recently, applicable to artificially small examples only.

This chapter discusses deductive verification of complex information flow-properties of open programs with controlled release of information. This approach allows analysis of Java programs by comparing two symbolic executions of the program, a variation of self-composition [Scheben and Schmitt, 2012, Scheben, 2014]. The feasibility of the approach has been proven by a case study on a simplified electronic voting system (Chapter 18), carried out in cooperation with the research group of Prof. Ralf Küsters from the University of Trier. The approach has also been used in [Dörre and Klebanov, 2015] to analyze information flow in the Android pseudo-random number generator.

In the following section we give an intuitive understanding of an information flow specification and its relation to a possible attacker model. In Section 13.3 we formally define noninterference. In Section 13.4, JML specifications for information flow properties for Java programs are defined, and used in Section 13.5 to formalize noninterference in JavaDL to gain proof obligations for the KeY prover. Directly proving the resulting proof obligation with the KeY tool may not be feasible for realistic programs, we therefore also present optimizations of the proof process for information flow properties in KeY in this section. Finally, we conclude in Section 13.6 and point to alternative approaches for verification of information flow properties in KeY.

The presentation (including the introduction) is based on [Scheben and Schmitt, 2012, 2014, Scheben, 2014].

We assume the reader to be familiar with some topics presented in earlier chapters. For understanding the details of the following presentation, it might be helpful to read the chapters on JavaDL (Chapter 3), theories used in the KeY framework (Chapter 5), specifications in JML (Chapter 7), and modular specification and verification (Chapter 9) first.

13.2 Specification and the Attacker Model

Information flow is a property of a program, and thus can be analyzed and verified. In order to verify the flow of information in a program, we need a specification describing the intended flow of information. The examples in the introduction separate the input and output of a program into an observable and a secret part. The input is the state and the parameters before execution of the program, the output is the state after execution and possibly the return value. The specification describes the observable part of these states, and thus implicitly specifies the secret parts as everything else.

To describe the observable part of a state, we use *observation expressions*. In the simplest case, an observation expression is a list (or sequence) of program variables. The sequence $\langle \mathtt{x}, \mathtt{y} \rangle$ of program variables `x` and `y`, for instance, describes that `x` and `y` are observable.

Restricting observation expressions to program variables is often too coarse-grained. It may be necessary to specify that only parts of the information contained in a program variable or the aggregation of several variables is observable. Therefore, we allow arbitrary JavaDL terms or JML expressions to appear in observation expressions. To specify, for example that only the last bit of `x` and the sum of `y` and `z` is observable, the observation expression $\langle$ `x%2, (y + z)` $\rangle$ can be used.[1] In general, it is possible to combine two observations described by two observation expressions R_1 and R_2 of sequence type by concatenation. We denote their concatenation by $R_1;R_2$. Since any observation expression R can be embedded into a singleton sequence, we extend the concatenation of observation expressions to any type in the obvious way.

This very flexible way of specification has two major advantages. For one, it allows us to express very precisely the information which actually may be seen by a possible attacker. In Chapter 18, we present the verification of information flow in an e-voting system. In this context, we specify that the result of an election may be observable, while other information, for example who voted for which candidate is not. Second, the approach allows a precise specification of method contracts and loop invariants, which is helpful when a modular analysis for realistic programs is necessary.

Typically in literature, information flow is used to verify the security of a program with respect to an attacker. The attacker is able to see the low part of the input and output of a program, which we call observations. It is counterintuitive to specify that an attacker is able to observe only the last bit of a parameter or only some elements of the heap but not others. Therefore we want to point out that the motivation behind our approach for specification is mainly driven by a precise specification of information flow, not by a realistic attacker model. Usually, an attacker is able to observe certain outputs of a program, for example the return value of a method or calls to logging methods. Observation expressions do not describe the ability of a realistic attacker, but the parts of inputs and outputs of a program which may influence each other. A program that has a specified information flow is secure against all attackers who are able to see only a subset of the information described by observation expressions.

Given the specification of observable parts of states, we can now give a formal definition of what it means for a program to have secure information flow.

[1] For a precise definition of observation expressions see [Scheben, 2014].

13.3 Formal Definition of Secure Information Flow

Intuitively, a program is noninterferent, i.e., it has secure information flow, if two runs of the program with equal low-security input have equal low-security output. Observation expressions describe the low part of states, while states are the input and output of programs. We can formally define what it means for states to have equal low-security values.

Definition 13.1 (Agreement of states). Let R be an observation expression.

Two states s and s' agree on R, abbreviated by $\mathrm{agree}(R, s, s')$, if and only if $eval_s(R) = eval_{s'}(R)$.

With the agreement of states we can define noninterference formally.

Definition 13.2 (Unconditional Noninterference). Let α be a program and R_1, R_2 observation expressions.

Program α allows information to flow only from R_1 to R_2, denoted by the predicate $\mathrm{flow}(\alpha, R_1, R_2)$, if and only if for all states s_1, s_1', s_2, s_2' such that $eval_{s_1}(\alpha) = \{s_2\}$ and $eval_{s_1'}(\alpha) = \{s_2'\}$, we have

$$\text{if } \mathrm{agree}(R_1, s_1, s_1') \quad \text{then} \quad \mathrm{agree}(R_2, s_2, s_2').$$

The observation expressions R_1 and R_2 describe the publicly available information of a pre- and a poststate of the system respectively. For all states which agree on the publicly available information, the states resulting from an execution of α agree on the part of the state described by R_2. Of course, this only holds if both runs of α actually terminate. If one run does not terminate, its poststate is undefined and therefore $\mathrm{agree}(R_2, s_2, s_2')$ is undefined. Therefore this notion of noninterference is *termination insensitive*.

In the simplest case, R_i expresses explicit declarations of program variables and fields which are considered low. In more sophisticated scenarios the R_i may be inferred from a multi-level security lattice (see for instance [Scheben, 2014]). Usually we will have $R_1 = R_2$. But, there are other cases: to declassify an expression e_{decl}, for instance, one would choose $R_1 = R_2; e_{decl}$.

As seen in Chapter 9, method contracts are useful in order to provide abstract knowledge about parts of a program, for example the states in which a method may be called. We would like to have a notion of noninterference which also respects knowledge about these states. In contract-based specifications, this condition is given by the precondition. The following definition of *conditional noninterference* allows us to use this knowledge.

Definition 13.3 (Conditional Noninterference). Let α be a program, R_1, R_2 observation expressions and ϕ a formula.

Program α allows information to flow only from R_1 to R_2 under condition ϕ, denoted by $\mathrm{flow}(\alpha, R_1, R_2, \phi)$, if and only if for all states s_1, s_1', s_2, s_2' such that $eval_{s_1}(\alpha) = \{s_2\}$ and $eval_{s_1'}(\alpha) = \{s_2'\}$ we have

$$\text{if} \quad s_1 \models \phi,\ s_1' \models \phi \text{ and } \mathrm{agree}(R_1, s_1, s_1') \quad \text{then} \quad \mathrm{agree}(R_2, s_2, s_2').$$

The idea behind this generalization is that in many cases a method in isolation has secure information flow only in case a precondition holds, for instance, if a parameter is not `null`. In such a situation, it is necessary to use the precondition within the information flow proof and show in a different proof that the precondition holds whenever the method is called. For details about modular specification and verification, please refer to Chapter 9.

Conditional noninterference enjoys the following compositionality property.

Lemma 13.4 (Compositionality of *flow*). *Let α_1, α_2 be programs, and $\alpha_1;\alpha_2$ their sequential composition. If flow$(\alpha_1, R_1, R_2, \phi_1)$, flow$(\alpha_2, R_2, R_3, \phi_2)$ and $s_1 \vDash (\phi_1 \rightarrow \langle\alpha_1\rangle\phi_2) = true$ hold for all states s_1, s_2, s_3 such that α_1 started in s_1 and terminates in s_2, and α_2 started in s_2 and terminates in s_3, then flow$(\alpha_1;\alpha_2, R_1, R_3, \phi_1)$ holds.*

Now that conditional noninterference has been defined formally, we show how it can be specified on program level with the help of JML. Finally, we present how noninterference can be verified using the KeY System.

13.4 Specifying Information Flow in JML

In Chapter 7 JML was introduced as a specification language, mainly for functional properties of Java programs. In this section, we want to show how JML can be extended to allow the specification of noninterference properties for Java programs. The presentation follows [Scheben and Schmitt, 2012] and [Scheben, 2014].

JML is built according to the *design by contract* (DBC) concept. To achieve a natural integration of information flow and functional specifications, the JML extension uses DBC for the specification of noninterference as well. Conditional noninterference with declassification is specified by information flow method contracts. Similar to functional method contracts, which specify the functional behavior of methods, information flow method contracts specify the information flow behavior of methods.

Information flow contracts augment functional JML contracts by **`determines`** clauses. Each **`determines`** clause defines a restriction on the information flow. The clause defines two lists of JML expressions, one expressing the observation expression for the poststate, the other list expressing the observation expression for the prestate. The **`determines`** clause

```
//@ determines l \by l;
void m();
```

specifies for the method `m()` that attackers may observe the value of the program variable `l` before and after the execution of `m()`.

It is possible to define different observation expressions for the pre- and the poststate of a method:

```
//@ determines l \by l, x;
void m();
```

specifies that the observation expression in the prestate of method m() contains the locations l and x, while in the poststate, it contains l only. This is useful for declassification as the method sum() in Figure 13.1 illustrates. The method calculates

```
class C {
  private int[] values;

  /*@ determines  \result  \by
    @     (\sum int i; 0 <= i && i < values.length; values[i]);
    @*/
  int sum() {
    int s = 0;
    for (int value : values) {
      s += value;
    }
    return s;
  }
}
```

Figure 13.1 Program declassifying the sum of an array

the sum of the entries of the array values and returns the result. Accordingly, the specification allows a declassification of the sum to the result.

A contract may contain several **determines** clauses. This is useful if a program run is observed by different parties with different abilities. For instance, there might be a party which may observe the unrestricted information stored in the field unrestricted and another party which may observe the information in unrestricted as well as the restricted information stored in the field restricted. Both parties may not access the information in secret1 and secret2. This situation can be specified naturally with the help of two **determines** clauses as shown in Figure 13.2.

```
class C {
  private int unrestricted, restricted, secret1;

  /*@ determines unrestricted              \by unrestricted;
    @ determines unrestricted, restricted  \by unrestricted,
    @                                          restricted;
    @*/
  void m(int secret2) {
    unrestricted++;
    restricted = restricted + unrestricted;
    secret1 = secret1 * (restricted + secret2);
  }
}
```

Figure 13.2 Program with multiple information flow contracts

The semantics of the **`determines`** clauses is defined with the help of conditional noninterference (see Definition 13.3): Let R_{post} be defined as the concatenation of the expressions behind the **`determines`** keyword. Let R_{pre} be defined as the concatenation of the expressions behind the **`\by`** keyword and the expressions behind an optional **`\declassifies`** keyword. Let further ϕ_{pre} be the precondition of the contract defined as usual by **`requires`**-clauses and class invariants. A method `m` fulfills a **`determines`** clause if and only if flow$(\mathtt{m}, R_{pre}, R_{post}, \phi_{pre})$ is valid.

Similar to method contracts, we extend JML loop invariants by **`determines`** clauses. We omit a detailed presentation for loop invariants here, the interested reader may refer to [Scheben, 2014] for a complete discussion.

13.5 Information Flow Verification with KeY

We have a formalization of information flow and a specification method as an extension of JML. In this section we explain how these two parts can be translated into JavaDL, providing us with a proof obligation which can naturally be verified in KeY. Since performing proofs in KeY efficiently depends, among others, on the number of branches a proof has, we also introduce an optimization which neither limits expressiveness nor precision, but reduces the number of branches an information flow proof consists of.

When considering information flow in object-oriented languages like Java, some special cases arise when it comes to object creation. KeY makes the assumption that the identity of an object created by calling a constructor is nondeterministic. This means, for one, it is not guaranteed that two runs of a program with the same initial heap generate the same object. And second, it is not possible to judge the order of creation for two new objects. We do not discuss this special issue here, but refer the interested reader to the related work [Beckert et al., 2014, Scheben, 2014]. The implementation in the KeY system however does consider this.

First, we define the JavaDL equivalent for the semantic *agree* predicate.

Definition 13.5 (Observation Equivalence). The formulas $\bar{x}_1, \bar{x}_2$ and the heaps h_1, h_2 are *observationally equivalent* with respect to observation expression R, written $obsEq(\bar{x}_1, h_1, \bar{x}_2, h_2, R)$, iff $\{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}R \doteq \{\mathtt{heap} := h_2 \mid\mid \bar{\mathtt{x}} := \bar{x}_2\}R$ evaluates to $true$.

Observational equivalence and the *agree* predicate are indeed equivalent.

Lemma 13.6. *Let s_1, s_2 be two states described by the formulas $\bar{x}_1$, h_1 and $\bar{x}_2$, h_2, respectively. Let R be an observation expression.*

The formula $obsEq(\bar{x}_1, h_1, \bar{x}_2, h_2, R)$ is valid if and only if $agree(R, s_1, s_2)$ holds.

Now we are ready to formulate conditional noninterference (Definition 13.3) in JavaDL.

Lemma 13.7. *Let α be a program with local variables $\bar{\mathtt{x}}$ of types $\bar{X}$, let R_1, R_2 be observation expressions and let ϕ be a formula.*

The formula

$$\begin{aligned}\Psi_{\alpha,\bar{\mathtt{x}},R_1,R_2,\phi} \equiv \forall Heap\, h_1,h_1',h_2,h_2'\ \forall \bar{X}\, \bar{x}_1,\bar{x}_1',\bar{x}_2,\bar{x}_2' \\ \{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}(\phi \wedge \langle\alpha\rangle(\mathtt{heap} \doteq h_2 \wedge \bar{\mathtt{x}} \doteq \bar{x}_2)) \wedge \\ \{\mathtt{heap} := h_1' \mid\mid \bar{\mathtt{x}} := \bar{x}_1'\}(\phi \wedge \langle\alpha\rangle(\mathtt{heap} \doteq h_2' \wedge \bar{\mathtt{x}} \doteq \bar{x}_2')) \\ \rightarrow \big(obsEq(\bar{x}_1,h_1,\bar{x}_1',h_1',R_1) \rightarrow obsEq(\bar{x}_2,h_2,\bar{x}_2',h_2',R_2)\big)\end{aligned}$$

is valid if and only if flow(α,R_1,R_2,ϕ) *holds.*

The formula shown in Lemma 13.7 is a direct formalization of information flow in JavaDL. This direct formalization expresses the intended property very precisely, however, containing two modalities and requiring two symbolic program executions comes at a price during verification. In the following we show some inefficiencies of this approach and introduce some optimization of the proof process which takes these inefficiencies into consideration and allows proving noninterference for larger programs.

13.5.1 Efficient Double Symbolic Execution

We use the example in Figure 13.3 to show several points for improvement when performing noninterference proofs.

The first point becomes obvious, when we have a closer look at the symbolic execution of the program. In the proof obligation as defined in Lemma 13.7 the program, which is executed first only differs in the name of the heap variable in the update and some renaming of parameters and return values from the second execution. Nevertheless, all rules necessary for symbolic execution are applied twice, once for each modality containing the program. Especially for larger programs and more complicated programs, this additional effort can become relevant. We can reduce the costs of calculating the weakest precondition by performing this calculation only once and then reuse the result for the noninterference proof.

Second, the poststate of one program execution is compared to all possible post-states of the second program execution. If the program has n possible execution paths, the symbolic execution yields n branches. Combining both program executions

```
/*@ public normal_behavior
  @ determines l \by l;
  @*/
public void m() {
  l = l+h;
  if (h!=0) {l = l-h;}
  if (l>0) {l--;}
}
```

Figure 13.3 Example of a secure program

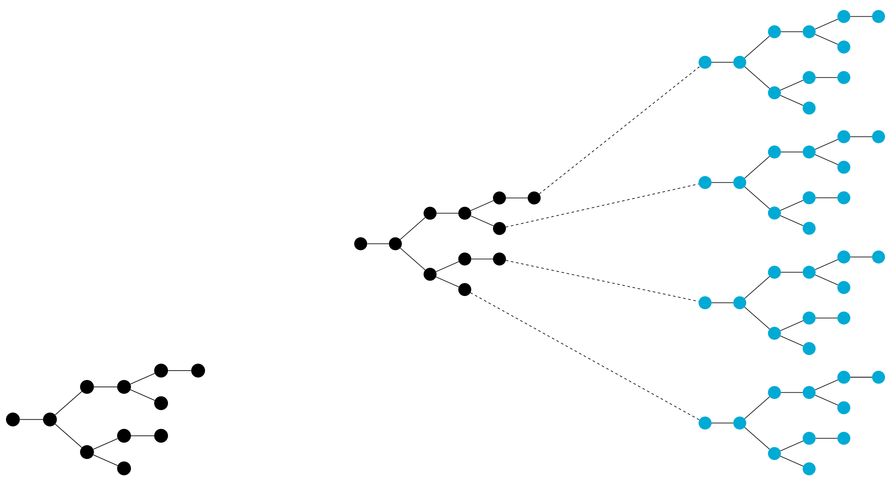
Figure 13.4 Sketch of the control flow graphs of (a) the original program and (b) the program with double symbolic execution

results in $O(n^2)$ branches for which the observation expressions have to be compared. In contrast, specialized information flow calculi, which consider the program only once, have to check only the outcome of the n paths through the program.

Let again α be the program as shown in Figure 13.3. The control flow graph of α is sketched in Figure 13.4(a). After combining both executions we have to perform a proof on the proof tree according to Figure 13.4(b).

In the following sections, we introduce optimizations, first regarding the calculation of the weakest precondition. This is followed by a discussion how the number of comparisons can be reduced. Finally, we show how block contracts can be used to further increase scalability and present how these optimizations can be used in the KeY system. The following argumentations are based on Dynamic Logic. Readers which are more familiar with weakest precondition calculi might prefer the presentation in [Scheben and Schmitt, 2014].

13.5.1.1 Reducing the Cost for the Weakest Precondition Calculation

First, we show that it is possible to prove noninterference in our setting with the help of only one symbolic execution of α.

Lemma 13.8. *Let* `heap` *and* $\bar{\mathtt{x}}$ *be the program variables of* α *and let* h_1, $\bar{x}_1$, h_2, $\bar{x}_2$, h'_1, $\bar{x}'_1$, h'_2 *and* $\bar{x}'_2$ *be variables of appropriate type.*

There exist formulas ψ *and* ψ' *without modalities, which replace* $\{\mathtt{heap} := h_1 \mathbin{||} \bar{\mathtt{x}} := \bar{x}_1\}\langle\alpha\rangle(\mathtt{heap} \doteq h_2 \land \bar{\mathtt{x}} \doteq \bar{x}_2)$ *and* $\{\mathtt{heap} := h'_1 \mathbin{||} \bar{\mathtt{x}} := \bar{x}'_1\}\langle\alpha\rangle(\mathtt{heap} \doteq h'_2 \land \bar{\mathtt{x}} \doteq \bar{x}'_2)$ *in Lemma 13.7.*

The formulas ψ *and* ψ' *can be calculated with a single symbolic execution of* α*.*

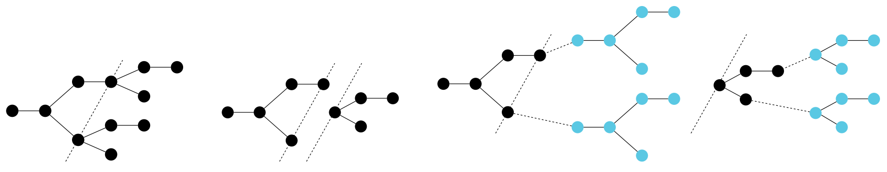

Figure 13.5 Reducing the verification overhead by compositional reasoning

Proof. Let $\mathscr{K}$ be a Kripke structure, s a state and β a variable assignment. The main step is finding a formula ψ—by symbolic execution of α—such that $(\mathscr{K},s,\beta) \vDash \{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}\langle\alpha\rangle(\mathtt{heap} \doteq h_2 \wedge \bar{\mathtt{x}} \doteq \bar{x}_2)$ implies $(\mathscr{K}_{ext},s,\beta) \vDash \psi$ for an extension $\mathscr{K}_{ext}$ of $\mathscr{K}$ by new Skolem symbols. (We need to consider extensions of $\mathscr{K}$, because the symbolic execution of α might introduce new Skolem symbols.) Note that the application of the JavaDL calculus—which contains all necessary rules for the symbolic execution of α—on $\{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}\langle\alpha\rangle(\mathtt{heap} \doteq h_2 \wedge \bar{\mathtt{x}} \doteq \bar{x}_2)$ does not deliver the desired implication: it approximates $\{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}\langle\alpha\rangle(\mathtt{heap} \doteq h_2 \wedge \bar{\mathtt{x}} \doteq \bar{x}_2)$ in the wrong direction. We have to take an indirection.

Intuitively, the formula $\{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}\langle\alpha\rangle(\mathtt{heap} \doteq h_2 \wedge \bar{\mathtt{x}} \doteq \bar{x}_2)$ is valid in $(\mathscr{K},s,\beta)$ if α started in state $s_1 : \mathtt{heap} \mapsto h_1^\beta, \bar{\mathtt{x}} \mapsto \bar{x}_1^\beta$ terminates in state $s_2 : \mathtt{heap} \mapsto h_2^\beta, \bar{\mathtt{x}} \mapsto \bar{x}_2^\beta$. We calculate a formula ψ_{not} which is *at most* true if α started in $s_1 : \mathtt{heap} \mapsto h_1^\beta, \bar{\mathtt{x}} \mapsto \bar{x}_1^\beta$ does *not* terminate in $s_2 : \mathtt{heap} \mapsto h_2^\beta, \bar{\mathtt{x}} \mapsto \bar{x}_2^\beta$. Then $\psi = \neg\psi_{not}$ is *at least* true if α started in s_1 terminates in s_2. We obtain ψ_{not} by symbolic execution of $\{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}\langle\alpha\rangle(\mathtt{heap} \not\doteq h_2 \vee \bar{\mathtt{x}} \not\doteq \bar{x}_2)$: application of the JavaDL calculus on the sequent $\Longrightarrow \{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}\langle\alpha\rangle(\mathtt{heap} \not\doteq h_2 \vee \bar{\mathtt{x}} \not\doteq \bar{x}_2)$ results in a set of sequents F_{seq}, where each $f_{seq} \in F_{seq}$ does not contain modalities any more. Let F be the set of meaning formulas for F_{seq}. We set $\psi_{not} = \bigwedge_{f \in F} f$.

Given ψ, we observe that we obtain a formula ψ' such that $(\mathscr{K},s,\beta) \vDash \{\mathtt{heap} := h'_1 \mid\mid \bar{\mathtt{x}} := \bar{x}'_1\}\langle\alpha\rangle(\mathtt{heap} \doteq h'_2 \wedge \bar{\mathtt{x}} \doteq \bar{x}'_2)$ implies $(\mathscr{K},s,\beta) \vDash \psi'$ by a simple renaming of the variables $h_1, \bar{x}_1, h_2, \bar{x}_2$ to $h'_1, \bar{x}'_1, h'_2, \bar{x}'_2$ and by the renaming of the new Skolem symbols $\bar{c}$ to new primed Skolem symbols $\bar{c}'$. The thus obtained formulas ψ and ψ' can be used to replace $\{\mathtt{heap} := h_1 \mid\mid \bar{\mathtt{x}} := \bar{x}_1\}\langle\alpha\rangle(\mathtt{heap} \doteq h_2 \wedge \bar{\mathtt{x}} \doteq \bar{x}_2)$ and $\{\mathtt{heap} := h'_1 \mid\mid \bar{\mathtt{x}} := \bar{x}'_1\}\langle\alpha\rangle(\mathtt{heap} \doteq h'_2 \wedge \bar{\mathtt{x}} \doteq \bar{x}'_2)$ in Lemma 13.7. Their calculation involves only one symbolic execution.

The full correctness proof of the approach can by found in [Scheben, 2014].

13.5.1.2 Reducing the Number of Comparisons

The second problem, the quadratic growth of the number of necessary comparisons in the number of program paths, can be tackled with the help of compositional reasoning if the structure of the program allows for it. Reconsider the initial example:

```
l = l + h;
if (h != 0) { l = l - h; }
-------------------------------
if (l > 0) { l--; }
```

As discussed above, the first part above the dashed line, and the second part below the line, are noninterferent on their own. Therefore, by Lemma 13.4 on compositionality of *flow*, the complete program is noninterferent. As illustrated in Figure 13.5, checking the two parts independently from each other results in less verification effort: When splitting the control flow graph of the entire program along the dashed line (Figure 13.5(a)), each subprogram has only two paths as shown in (b). When symbolically executing both of the subprograms twice, we gain four paths each (c and d).

Thus, altogether only eight comparisons of post states have to be made to prove noninterference of the complete program. Checking the complete program at once would require (about) 12 comparisons.[2] We summarize the above observation in the following lemma.

Lemma 13.9. *Let α be a program with m branching statements.*

If α can be divided into m noninterferent blocks with at most one branching statement per block, then noninterference of α can be shown with the help of double symbolic execution with $3m$ *comparisons.*

Since a program with m branching statements has at least $n = m+1$ paths, Lemma 13.9 shows that the verification effort of double symbolic execution approaches can be reduced from $O(n^2)$ to $O(n)$ comparisons, if the program under consideration is compositional with respect to information flow. In the best case, a program with m branching statements has $\Omega(2^m)$ paths. In this case, the verification effort reduces to $O(\log(n))$ comparisons, if the program under consideration is compositional with respect to information flow.

Unfortunately, the separation is not always as nice as in the example above. Consider for instance the following program:

```
if (l > 0) { if (l % 2 == 1) { l--; } }
```

The program can be divided into blocks

$b_1 =$ `if (l % 2 == 1) { l--; }`

and

$b_2 =$ `if (l > 0) {` b_1 `}`.

To conclude that b_2 is noninterferent, it is necessary to use the fact that b_1 is noninterferent in the proof of b_2. Unfortunately, the double execution approach does not easily lend itself to such compositional verification. In the next section, the problem of compositional reasoning will be discussed.

[2] By symmetry, the number of comparisons can be reduced further in both cases: in the first case $2 \cdot (2+1) = 6$ comparisons are sufficient, in the second case $4+3+2+1 = 10$ comparisons are enough.

13.5.1.3 Compositional Double Symbolic Execution

If a program α calls a block b, one (sometimes) does not want to look at its code but rather use a software contract for b, a contract that had previously been established by looking only at the code of b. This kind of compositionality can also be applied to methods instead of blocks and is essential for the scalability of all deductive software verification approaches. With double symbolic execution, the block b is not only called in the first execution of α, but also in the second execution. This poses the technical problem of somehow synchronizing the first and second call of b for contract application.

In this paragraph, we show how software contracts can be applied in proofs using double symbolic execution. An important feature of our approach is the seamless integration of information flow and functional reasoning allowing us to take advantage of the precision of functional contracts also for information flow contracts, if necessary.

In the context of functional verification, compositionality is achieved through method contracts. We extend this approach to the verification of information flow properties. We define *information flow contracts* as a tuple of a precondition and observation expressions for the pre- and the poststate.

Definition 13.10 (Information Flow Contract). An information flow contract (in short: flow contract) to a block (or method) $\mathtt{b}$ with local variables $\bar{\mathtt{x}} := (\mathtt{x}_1, \ldots, \mathtt{x}_n)$ of types $\bar{A} := (A_1, \ldots, A_n)$ is a tuple $\mathscr{C}_{\mathtt{b},\bar{\mathtt{x}}::\bar{A}} = (pre, R_1, R_2)$, where (1) pre is a formula which represents a precondition and (2) R_1, R_2 are observation expressions which represent the low expressions in the pre- and poststate.

A flow contract $\mathscr{C}_{\mathtt{b},\bar{\mathtt{x}}::\bar{A}} = (pre, R_1, R_2)$ is valid if and only if the predicate $\mathit{flow}(\mathtt{b}, R_1, R_2, pre)$ is valid.

The difficulty in the application of flow contracts arises from the fact that flow contracts refer to two invocations of a block $\mathtt{b}$ in different contexts.

Example 13.11. Consider

```java
if (l>0) { l++; if (l%2 == 1) {l--;} }
```

again, with blocks $b_1 =$ `if (l%2 == 1) {l--;}` and $b_2 =$ `if (l>0) {l++;` b_1`}`. Let $\mathscr{C}_{b_1,\bar{\mathtt{x}}::\bar{A}} = \mathscr{C}_{b_2,\bar{\mathtt{x}}::\bar{A}} = (true, \mathtt{l}, \mathtt{l})$ be flow contracts for b_1 and b_2. To prove $\mathscr{C}_{b_2,\bar{\mathtt{x}}::\bar{A}}$ by double symbolic execution,

$$\mathtt{l} \doteq \mathtt{l}' \rightarrow (\langle \mathtt{if\ (l>0)\ \{l{+}{+};}\ b_1\mathtt{\};\ if\ (l'>0)\ \{l'{+}{+};}\ b_1\mathtt{\}} \rangle \mathtt{l} \doteq \mathtt{l}')$$

has to be shown. (For presentation purposes, we ignore the heap in this example.) Symbolic execution of the program, as far as possible, yields:

$$
\dfrac{
\begin{array}{l}
\mathtt{l} \doteq \mathtt{l}', \mathtt{l} > 0 \\
\Longrightarrow \{\mathtt{l} := \mathtt{l}+1\} \\
\quad \langle b_1; \\
\quad\; \mathtt{if\,(l'>0)\,\{} \\
\quad\quad \mathtt{l'++;} \\
\quad\quad b_1 \\
\quad\; \mathtt{\}} \\
\quad \rangle \mathtt{l} \doteq \mathtt{l}'
\end{array}
\quad (13.1)
\qquad
\text{apply-Equality + close}\ \dfrac{*}{\begin{array}{l}\mathtt{l} \doteq \mathtt{l}' \\ \mathtt{l}' > 0 \\ \Longrightarrow \mathtt{l} > 0, \\ \{\mathtt{l}' := \mathtt{l}'+1\} \\ \quad \langle b_1' \rangle \mathtt{l} \doteq \mathtt{l}'\end{array}}
\qquad
\text{close}\ \dfrac{*}{\begin{array}{l}\mathtt{l} \doteq \mathtt{l}' \\ \Longrightarrow \mathtt{l} > 0, \\ \mathtt{l}' > 0, \\ \mathtt{l} \doteq \mathtt{l}'\end{array}}
}{
\dfrac{\vdots\ \textit{symbolic execution}}{\Longrightarrow \mathtt{l} \doteq \mathtt{l}' \rightarrow (\langle \mathtt{if\ (l>0)\ \{l++;}\ b_1\mathtt{\};\ if\ (l'>0)\ \{l'++;}\ b_1'\mathtt{\}}\rangle \mathtt{l} \doteq \mathtt{l}')}
}
$$

To close branch (13.1), $\mathscr{C}_{b_1,\bar{\mathtt{x}}::\bar{A}}$ needs to be used—but it is not obvious how this can be done, because $\mathscr{C}_{b_1,\bar{\mathtt{x}}::\bar{A}}$ refers to the invocation of b_1 in the first and the second execution at the same time. A similar problem occurs if $\mathscr{C}_{b_2,\bar{\mathtt{x}}::\bar{A}}$ is proved with the help of the optimizations discussed above.

The main idea of the solution is a coordinated delay of the application of flow contracts. The solution is compatible with the optimizations discussed above and additionally allows the combination of flow contracts with functional contracts.

Let b be a block with the functional contract $\mathscr{F}_{\mathtt{b},\bar{\mathtt{x}}::\bar{A}} = (\mathit{pre}, \mathit{post}, \mathit{Mod})$ where (1) the formula *pre* represents the precondition; (2) the formula *post* represents the postcondition; and (3) the term *Mod* represents the assignable clause for b. In functional verification, block contracts are applied by the rule useBlockContract, introduced by Wacker [2012]. The rule is an adaptation of the rule useMethodContract from Section 9.4.3 for blocks. For presentation purposes, we consider a simplified version of the rule only:

$$
\text{useBlockContract}\ \dfrac{\begin{array}{rr}\mathit{pre} & \Gamma \Longrightarrow \{u\}\mathit{pre}, \Delta \\ \mathit{post} & \Gamma \Longrightarrow \{u; u_{anon}\}(\mathit{post} \rightarrow [\pi\ \omega]\phi), \Delta\end{array}}{\Gamma \Longrightarrow \{u\}[\pi\ \mathtt{b};\ \omega]\phi, \Delta}
$$

Here, u is an arbitrary update; $u_{anon} = (\mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathit{Mod}, h), \bar{\mathtt{x}} := \bar{x}')$ is an anonymizing update setting the locations of *Mod* (which might be modified by b) and the local variables which might be modified to unknown values; h of type *Heap* and $\bar{x}'$ of appropriate types are fresh symbols. We require *pre* to entail equations $\mathtt{heap}_{pre} \doteq \mathtt{heap}$ and $\bar{\mathtt{x}}_{pre} \doteq \bar{\mathtt{x}}$ which store the values of the program variables of the initial state in program variables $\mathtt{heap}_{pre}$ and $\bar{\mathtt{x}}_{pre}$ such that the initial values can be referred to in the postcondition. Additionally, we require that *pre* and *post* entail a formula which expresses that the heap is well-formed.

The plan is to use an extended version of the rule useBlockContract during symbolic execution—in many cases for the trivial functional contract $\mathscr{F}_{b,\bar{\mathtt{x}}::\bar{A}} = (\mathit{true}, \mathit{true}, \mathit{allLocs})$—which adds some extra information to the sequent allowing a delayed application of information flow contracts. The extra information is encapsulated in a new two-state predicate $C_{\mathtt{b}}(\bar{x}, h, \bar{x}', h')$ with the intended meaning that b started in state $s_1 : \mathtt{heap} \mapsto h, \bar{\mathtt{x}} \mapsto \bar{x}$ and terminates in state $s_2 : \mathtt{heap} \mapsto h', \bar{\mathtt{x}} \mapsto \bar{x}'$. This predicate can be integrated into the rule useBlockContract as follows:

useBlockContract2

$$\frac{\begin{array}{ll} \mathit{pre} & \Gamma \Longrightarrow \{u\}\mathit{pre}, \Delta \\ \mathit{post} & \Gamma, \{u\}C_{\mathtt{b}}(\bar{\mathtt{x}}, \mathtt{heap}, \bar{x}', h'), \{u; u_{anon}\}(\mathtt{heap} \doteq h' \wedge \bar{\mathtt{x}} \doteq \bar{x}') \\ & \qquad \Longrightarrow \{u; u_{anon}\}(\mathit{post} \rightarrow [\pi\ \omega]\phi), \Delta \end{array}}{\Gamma \Longrightarrow \{u\}[\pi\ \mathtt{b};\ \omega]\phi, \Delta}$$

where h' and $\bar{x}'$ are fresh function symbols. By [Scheben, 2014], useBlockContract2 is sound. The introduction of $C_{\mathtt{b}}(\bar{x}, h, \bar{x}', h')$ to the post branch allows us to store the initial and the final state of b for a delayed application of information flow contracts: the two predicates $C_{\mathtt{b}}(\bar{x}_1, h_1, \bar{x}'_1, h'_1)$ and $C_{\mathtt{b}}(\bar{x}_2, h_2, \bar{x}'_2, h'_2)$ appearing on the antecedent of a sequent can be approximated by an instantiation of a flow contract $\mathscr{C}_{\mathtt{b},\bar{\mathtt{x}}::\bar{A}} = (\mathit{pre}, R_1, R_2)$ for b by

$$\begin{array}{l} \{\mathtt{heap} := h_1 \parallel \bar{\mathtt{x}} := \bar{x}_1\}\mathit{pre} \wedge \{\mathtt{heap} := h_2 \parallel \bar{\mathtt{x}} := \bar{x}_2\}\mathit{pre} \\ \rightarrow \big(\mathit{obsEq}(\bar{x}_1, h_1, \bar{x}'_1, h'_1, R_1) \rightarrow \mathit{obsEq}(\bar{x}_2, h_2, \bar{x}'_2, h'_2, R_2)\big) \ . \end{array}$$

This approximation is applied by the rule useFlowContract:

useFlowContract

$$\frac{\begin{array}{l} \Gamma, C_{\mathtt{b}}(\bar{x}_1, h_1, \bar{x}'_1, h'_1), C_{\mathtt{b}}(\bar{x}_2, h_2, \bar{x}'_2, h'_2), \\ \qquad \{\mathtt{heap} := h_1 \parallel \bar{\mathtt{x}} := \bar{x}_1\}\mathit{pre} \wedge \{\mathtt{heap} := h_2 \parallel \bar{\mathtt{x}} := \bar{x}_2\}\mathit{pre} \\ \quad \rightarrow \big(\mathit{obsEq}(\bar{x}_1, h_1, \bar{x}'_1, h'_1, R_1) \rightarrow \mathit{obsEq}(\bar{x}_2, h_2, \bar{x}'_2, h'_2, R_2)\big) \\ \Longrightarrow \Delta \end{array}}{\Gamma, C_{\mathtt{b}}(\bar{x}_1, h_1, \bar{x}'_1, h'_1), C_{\mathtt{b}}(\bar{x}_2, h_2, \bar{x}'_2, h'_2) \Longrightarrow \Delta}$$

Formally, a flow contract $C_{\mathtt{b}}(\bar{x}, h, \bar{x}', h')$ is valid in a Kripke structure $\mathscr{K}$ and a state s if and only if

$$\{\bar{\mathtt{x}} := \bar{x} \parallel \mathtt{heap} := h\}\langle \mathtt{b} \rangle(\mathtt{heap} \doteq h' \wedge \bar{\mathtt{x}} \doteq \bar{x}')$$

is valid in $(\mathscr{K}, s)$. Note that the usage of the rule useBlockContract2 during symbolic execution allows the application of arbitrary functional contracts in addition to flow contracts. This allows for taking advantage of the precision of functional contracts within information flow proofs, if necessary. The default, however, is using the trivial functional contract $\mathscr{F}_{b,\bar{\mathtt{x}}::\bar{A}} = (\mathit{true}, \mathit{true}, \mathit{allLocs})$ as in the presented example. The soundness proof for the above approach can be found in [Scheben, 2014].

Example 13.12. Let $\mathscr{F}_{b_1,\bar{\mathtt{x}}::\bar{A}} = (\mathit{true}, \mathit{true}, \mathit{allLocs})$ be the trivial functional contract for b_1. Applied on the example from above, (13.1) can be simplified as shown in Figure 13.6. For presentation purposes, all heap symbols have been removed from the example. Therefore, C_{b_1} takes only two parameters and *obsEq* only three parameters. Adding the heap results in essentially the same proof but with more complex formulas.

The proof uses the following abbreviations of rule names:

$$\textsf{close}\ \frac{\ast}{\begin{array}{l}\mathtt{l}\doteq\mathtt{l}',\ \mathtt{l}>0,\ C_{b_1}(\mathtt{l}+1,\ell),\ \ell_{anon}\doteq\ell,\\ \mathtt{l}'>0,\ C_{b_1}(\mathtt{l}'+1,\ell'),\ \ell'_{anon}\doteq\ell',\\ \ell_{anon}\doteq\ell'_{anon}\\ \Longrightarrow \ell_{anon}\doteq\ell'_{anon}\end{array}}$$

$$\textsf{eq + simp}\ \frac{\cdots}{\begin{array}{l}\mathtt{l}\doteq\mathtt{l}',\ \mathtt{l}>0,\ C_{b_1}(\mathtt{l}+1,\ell),\ \ell_{anon}\doteq\ell,\\ \mathtt{l}'>0,\ C_{b_1}(\mathtt{l}'+1,\ell'),\ \ell'_{anon}\doteq\ell',\\ \mathtt{l}+1\doteq\mathtt{l}'+1\rightarrow\ell\doteq\ell'\\ \Longrightarrow \ell_{anon}\doteq\ell'_{anon}\end{array}}$$

$$\textsf{obsEq}\ \frac{\cdots}{\begin{array}{l}\mathtt{l}\doteq\mathtt{l}',\ \mathtt{l}>0,\ C_{b_1}(\mathtt{l}+1,\ell),\ \ell_{anon}\doteq\ell,\\ \mathtt{l}'>0,\ C_{b_1}(\mathtt{l}'+1,\ell'),\ \ell'_{anon}\doteq\ell',\\ \quad obsEq(\mathtt{l}+1,\mathtt{l}'+1,\mathtt{l}\doteq\mathtt{l}')\\ \ \rightarrow obsEq(\ell,\ell',\mathtt{l}\doteq\mathtt{l}')\\ \Longrightarrow \ell_{anon}\doteq\ell'_{anon}\end{array}}$$

$$\textsf{uFC}\ \frac{\cdots}{\begin{array}{l}\mathtt{l}\doteq\mathtt{l}',\ \mathtt{l}>0,\ C_{b_1}(\mathtt{l}+1,\ell),\ \ell_{anon}\doteq\ell,\\ \mathtt{l}'>0,\ C_{b_1}(\mathtt{l}'+1,\ell'),\ \ell'_{anon}\doteq\ell'\\ \Longrightarrow \ell_{anon}\doteq\ell'_{anon}\end{array}}$$

$$\textsf{simp}\ \frac{\cdots}{\begin{array}{l}\mathtt{l}\doteq\mathtt{l}',\ \mathtt{l}>0,\ C_{b_1}(\mathtt{l}+1,\ell),\ \ell_{anon}\doteq\ell,\\ \mathtt{l}'>0,\\ \{\mathtt{l}:=\ell_{anon}\}\{\mathtt{l}':=\mathtt{l}'+1\}C_{b_1}(\mathtt{l}',\ell'),\\ \{\mathtt{l}:=\ell_{anon}\}\{\mathtt{l}':=\mathtt{l}'+1\}\{\mathtt{l}:=\ell'_{anon}\}\mathtt{l}\doteq\ell'\\ \Longrightarrow \{\mathtt{l}:=\ell_{anon}\}\{\mathtt{l}':=\mathtt{l}'+1\}\{\mathtt{l}:=\ell'_{anon}\}\mathtt{l}\doteq\mathtt{l}'\end{array}}$$

$$\textsf{uBC2}\ \frac{\cdots}{\begin{array}{l}\mathtt{l}\doteq\mathtt{l}',\ \mathtt{l}>0,\ C_{b_1}(\mathtt{l}+1,\ell),\ \ell_{anon}\doteq\ell,\\ \mathtt{l}'>0\\ \Longrightarrow \{\mathtt{l}:=\ell_{anon}\}\{\mathtt{l}':=\mathtt{l}'+1\}\langle b'_1\rangle\mathtt{l}\doteq\mathtt{l}'\end{array}}$$

$$++\ \frac{\cdots}{\begin{array}{l}\mathtt{l}\doteq\mathtt{l}',\ \mathtt{l}>0,\ C_{b_1}(\mathtt{l}+1,\ell),\ \ell_{anon}\doteq\ell,\\ \mathtt{l}'>0\\ \Longrightarrow \{\mathtt{l}:=\ell_{anon}\}\langle\mathtt{l'++;}\ b'_1\rangle\mathtt{l}\doteq\mathtt{l}'\end{array}}$$

$$\textsf{simp}\ \frac{\cdots}{\begin{array}{l}\mathtt{l}\doteq\mathtt{l}',\ \mathtt{l}>0,\ C_{b_1}(\mathtt{l}+1,\ell),\ \ell_{anon}\doteq\ell,\\ \{\mathtt{l}:=\ell_{anon}\}\mathtt{l}'>0\\ \Longrightarrow \{\mathtt{l}:=\ell_{anon}\}\langle\mathtt{l'++;}\ b'_1\rangle\mathtt{l}\doteq\mathtt{l}'\end{array}}$$

$$\textsf{close}\ \frac{\ast}{\begin{array}{l}\mathtt{l}\doteq\mathtt{l}',\\ \mathtt{l}>0,\\ C_{b_1}(\mathtt{l}+1,\ell),\\ \ell_{anon}\doteq\ell\\ \Longrightarrow \mathtt{l}>0,\\ \ell_{anon}\doteq\mathtt{l}'\end{array}}$$

$$\textsf{eq}\ \frac{\cdots}{\begin{array}{l}\mathtt{l}\doteq\mathtt{l}',\\ \mathtt{l}>0,\\ C_{b_1}(\mathtt{l}+1,\ell),\\ \ell_{anon}\doteq\ell\\ \Longrightarrow \mathtt{l}'>0,\\ \ell_{anon}\doteq\mathtt{l}'\end{array}}$$

$$\textsf{simp}\ \frac{\cdots}{\begin{array}{l}\mathtt{l}\doteq\mathtt{l}',\\ \mathtt{l}>0,\\ C_{b_1}(\mathtt{l}+1,\ell),\\ \ell_{anon}\doteq\ell\\ \Longrightarrow \{\mathtt{l}:=\ell_{anon}\}\\ \quad \mathtt{l}'>0,\\ \{\mathtt{l}:=\ell_{anon}\}\\ \quad \mathtt{l}\doteq\mathtt{l}'\end{array}}$$

$$\textsf{if}\ \frac{\cdots\qquad\cdots}{\begin{array}{l}\mathtt{l}\doteq\mathtt{l}',\ \mathtt{l}>0,\ C_{b_1}(\mathtt{l}+1,\ell),\ \ell_{anon}\doteq\ell\\ \Longrightarrow \{\mathtt{l}:=\ell_{anon}\}\langle\mathtt{if\ (l'>0)\ \{l'++;}\ b'_1\mathtt{\}}\rangle\mathtt{l}\doteq\mathtt{l}'\end{array}}$$

$$\textsf{simp}\ \frac{\cdots}{\begin{array}{l}\mathtt{l}\doteq\mathtt{l}',\ \mathtt{l}>0,\ \{\mathtt{l}:=\mathtt{l}+1\}C_{b_1}(\mathtt{l},\ell),\ \{\mathtt{l}:=\mathtt{l}+1\}\{\mathtt{l}:=\ell_{anon}\}\mathtt{l}\doteq\ell\\ \Longrightarrow \{\mathtt{l}:=\mathtt{l}+1\}\{\mathtt{l}:=\ell_{anon}\}\langle\mathtt{if\ (l'>0)\ \{l'++;}\ b'_1\mathtt{\}}\rangle\mathtt{l}\doteq\mathtt{l}'\end{array}}$$

$$\textsf{uBC2}\ \frac{\cdots}{\begin{array}{l}\mathtt{l}\doteq\mathtt{l}',\ \mathtt{l}>0\\ \Longrightarrow \{\mathtt{l}:=\mathtt{l}+1\}\langle b_1\mathtt{;\ if\ (l'>0)\ \{l'++;}\ b'_1\mathtt{\}}\rangle\mathtt{l}\doteq\mathtt{l}'\end{array}}$$

Figure 13.6 Proof tree of Example 13.12.

Abbreviation	Full name	Abbreviation	Full name
uBC2	useBlockContract2	eq	applyEquality
uFC	useFlowContract	if	conditional
obsEq	replaces $obsEq(\cdot)$ by its definition (Lemma 13.6)	simp	combination of all update simplification rules
++	plusPlus	close	close
eq + simp	repeated application of the rules eq and simp		

Firstly, the symbolic execution is continued by the rule useBlockContract2 and (after several simplifications) by the rule conditional. The conditional rule splits the proof into two branches. The right branch, which represents the case that the condition $\mathtt{l}' > 0$ evaluates to false, can be closed after further simplifications and the application of equalities. On the other branch, the remaining program is executed symbolically by the rule plusPlus and another application of useBlockContract2, now on the block b_1'. After some further simplifications, we are in the position to apply the flow contract for b_1: the antecedent of the sequent contains the two predicates $C_{b_1}(\mathtt{l}+1,\ell)$ and $C_{b_1}(\mathtt{l}'+1,\ell')$ on which the rule useFlowContract can be applied. With the help of the guarantees from the flow contract for b_1, the proof closes after some final simplifications.

13.5.2 Using Efficient Double Symbolic Execution in KeY

In this section, we show how efficient double execution proofs can be performed in the KeY system. Readers not familiar with the KeY system may find it helpful to read Chapter 15 on using the KeY prover first.

Efficient double execution is implemented in KeY with the help of strategy macros. The simplest way to use the optimizations is by application of the macro **Full Information Flow Auto Pilot**. It can be selected by highlighting an arbitrary term, left-clicking, choosing the menu item **Strategy macros** and then **Full Information Flow Auto Pilot**. KeY should be able to prove most of the information flow examples delivered with KeY (under **examples** $\rightarrow$ **firstTouch** $\rightarrow$ **InformationFlow**) automatically this way.

On complicated examples, the auto pilot might fail. In this case, we can gain better control of the proof by application of the following steps.

As discussed in Section 13.5.1, double execution considers the same program twice, but it suffices to calculate only one weakest precondition. Therefore we start a side-proof for the weakest precondition calculation. This is done as follows:

1. We highlight an arbitrary term and left-click. We then choose the menu item **Strategy macros** and in the upcoming menu the item **Auxiliary Computation Auto Pilot**. A side proof opens and KeY tries to automatically calculate the weakest precondition.
   KeY succeeded in the calculation if the open goals of the side proof do not contain modalities any more. If a goal still contains a modality, then one can either simply try to increase the number of auto-mode steps or one can remove the modalities by interactive steps.

If one of the open goals contains an information flow proof obligation from a block contract or from an information flow loop invariant, then this goal has to be closed by going through steps (1) to (3) again before continuing with step (2).

2. We choose an open goal, highlight an arbitrary term in the sequent and left-click. We choose the menu item **Strategy macros** and in the upcoming menu the item **Finish auxiliary computation**. The side-proof closes and a new taclet (rule) is introduced to the main proof. The new taclet is able to replace the double execution term (the shortest term which contains both modalities) by two instantiations of the calculated formula.
3. On simple examples, it suffices to activate the auto mode by choosing the menu item **Continue** from the menu **Proof**. On more complex examples it is helpful to run the strategy macro **self-composition state expansion with inf flow contracts** first. The latter macro applies the new rule and afterwards tries to systematically apply information flow contracts.

The macro **Full Information Flow Auto Pilot** applies steps (1)–(3) automatically.

13.6 Summary and Conclusion

We have presented how information flow properties can be specified in JML and that KeY can analyze whether Java programs satisfy the specification. The approach implemented in the KeY prover allows for a very precise specification of information flows which is important especially in a real-world object-oriented programming language. Information flow is represented in JavaDL directly by the semantic meaning of the property: We directly compare two executions of a program only differing in the secret input. While this allows for precise reasoning with KeY, the pairwise comparison of all execution paths leads to quadratic growth of proof obligations. Therefore, we also show how the proof process can be optimized such that verifying real-world programs becomes feasible.

The approach as presented here was applied for the verification of a simplified e-voting case study. Experiences of this work can be found in Chapter 18.

Another approach for precise information flow analysis has been developed by Klebanov [2014]. The approach is based on symbolic execution in KeY, combined with an external quantifier elimination procedure and a model counting procedure. The method and tool chain not only identify information leaks in programs but quantify them using a number of information-theoretical metrics.

Very popular enforcement methods for information flow properties in the literature are type systems. These approaches are usually less precise than the approach presented here, however only a single execution of a program has to be considered during analysis. So-called *dependent types* allow to further improve precision of type-based analysis. Here, dependencies between variables and partial and aggregated information is tracked during symbolic execution. Using theorem provers for the analysis of programs with dependent types, it is possible to track the semantics of information during a program run. An extension for KeY supporting a type-based reasoning of information flow in programs can be found in [Bubel et al., 2009, van Delft and Bubel, 2015].