Efficient Self-composition for Weakest Precondition Calculi

Christoph Scheben and Peter H. Schmitt[⋆]

Karlsruhe Institute of Technology (KIT), Dept. of Informatics
Am Fasanengarten 5, 76131 Karlsruhe, Germany
http://www.key-project.org/DeduSec/

Abstract. This paper contributes to deductive verification of language based secure information flow. A popular approach in this area is *self-composition* in combination with off-the-shelf software verification systems to check for secure information flow. This approach is appealing, because (1) it is highly precise and (2) existing sophisticated software verification systems can be harnessed. On the other hand, self-composition is commonly considered to be inefficient.

We show how the efficiency of self-composition style reasoning can be increased. It is sufficient to consider programs only once, if the used verification technique is based on a weakest precondition calculus with an explicit heap model. Additionally, we show that in many cases the number of final symbolic states to be considered can be reduced considerably. Finally, we propose a comprehensive solution of the technical problem of applying software contracts within the self-composition approach. So far this problem had only been solved partially.

1 Introduction

In the last years, there has been an increasing interest, both in research and industry, in checking programs for unintended leakage of secret information. Language-based non-interference is one of the most prominent concepts promoted in this area and a number of theories and tools have been developed to support it. In Sect. 6 we will present a detailed summary of the different approaches and their relation to our contribution. The approach we follow is called *self-composition* as pioneered by [6,8]. To check that the high variables $\bar{h}$ in program α do not interfere with its low variables $\bar{\ell}$ a syntactic variation α' of α is considered by replacing every program variable v by a new primed version v'. Then it has to be proved that when program $\alpha;\alpha'$ is started in any state where the values of $\bar{\ell}$ and $\bar{\ell}'$ coincide it terminates in a state where the values of $\bar{\ell}$ and $\bar{\ell}'$ again coincide.

The advantages of this approach are its high degree of precision and the fact that off-the-shelf SMT-solvers or theorem provers can be harnessed. In our case

⋆ This work was supported by the German National Science Foundation (DFG) under project "Program-level Specification and Deductive Verification of Security Properties" within priority programme 1496 "Reliably Secure Software Systems – RS^3".

we used KeY, a software verification system for full sequential JAVA. On the other hand, disadvantages of the self-composition approach are that (1) naive implementations are quite inefficient and (2) it does not easily lend itself to modular verification.

The efficiency issue arises from two facts. Let n be the number of paths through a program α.

1. Analysis based on self-composition consider the same program at least twice. (Really naive analysis might consider the same program even $1 + n$ times.)
2. The self-composed program $\alpha; \alpha'$ has n^2 final symbolic states in which the low values have to be compared to each other.

As a first contribution we show that self-composition approaches based on weakest precondition calculi [9] need to consider α only once: we show in Theorem 1 that the problem can be rephrased in self-composition style such that the weakest precondition of $\alpha; \alpha'$ can be constructed from the weakest precondition of α, because α and α' do not interfere with each other and the weakest precondition of α' and ϕ' is the same as the one of α and ϕ except for the names of the program variables.

As a second contribution, inspired by the compositional reasoning of security type systems and specialized information flow calculi, we show that the number of final symbolic states to be considered can be reduced considerably if α is compositional with respect of information flow. In this case only $O(n)$ final symbolic states have to be considered. Depending on the structure of the program, this number can be reduced further up to $O(\log(n))$.

The latter approach relies on compositional / modular reasoning: If program α calls a block b, we (sometimes) do 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 modularization can also be applied to methods instead of blocks and is essential for the scalability of all deductive software verification approaches. With self-composition b is not only called in α, but b' is called in α'. This poses the technical problem of somehow synchronizing the calls of b and b' for contract application. This has already been pointed out in the paper by Naumann [16], who also gave hints to a possible solution. Dufay, Felty and Matwin [10] present a partial solution using ghost code, see Sect. 6.

As a third contribution we show how software contracts can be applied in self-composition proofs based on weakest precondition calculi. 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.

Structure. In the next two sections, we fix notation and recall the formalization of conditional non-interference. Based on this, Sect. 4 discusses two efficiency problems with self-composition and presents two orthogonal approaches to overcome these problems. Sect. 5 presents modular reasoning at the block level which the second approach relies on. Sect. 6 discusses related work and Sect. 7 concludes the paper.

2 Notation

Assertions like pre- and postconditions are formulated in typed first order logic. Among others, constant and function symbols are available for local program variables as well as instance and static fields. Terms t and formulas ϕ are inductively defined as usual. We use $\mathcal{M}$ to refer to interpretations of first order logic, and $t^{\mathcal{M}}$, $\phi^{\mathcal{M}}$ to denote the interpretation of term t and formula ϕ in $\mathcal{M}$. The data type heap is modeled by the theory of arrays [13,19]. The current heap of a program is given by an implicit program variable `heap`. A state is a mapping from program variables (including `heap`) to values of proper types. As a consequence of the theory of arrays the values of the local variables $\bar{\mathtt{x}}$ together with the value of `heap` completely determine the state of a program.

Let $\mathcal{M}$ be an interpretation and s a state. We denote by $\mathcal{M}^{\leftarrow s}$ the interpretation which coincides with $\mathcal{M}$ except for the interpretation of the program variables `heap` and $\bar{\mathtt{x}}$; these are interpreted according to s as $\mathtt{heap}^{\mathcal{M}^{\leftarrow s}} = s(\mathtt{heap})$ and $\bar{\mathtt{x}}^{\mathcal{M}^{\leftarrow s}} = s(\bar{\mathtt{x}})$. As usual, a formula is said to be universally valid iff it is true in every interpretation $\mathcal{M}$.

$\phi[x \leftarrow x']$ denotes the substitution of x by x' in ϕ. We use $\phi[x \leftarrow x', y \leftarrow y']$ as abbreviation for $(\phi[x \leftarrow x'])[y \leftarrow y']$. The weakest precondition [9] of a program α and a postcondition ϕ is denoted by $wp(\alpha, \phi)$. For simplicity we consider only terminating programs. Hence, $wp(\alpha, \phi)$ always exists.

In self-composition proofs any program variable $\mathtt{x}$ has a primed counterpart, denoted by $\mathtt{x}'$. Accordingly, α' denotes the program which is constructed from α by replacing all program variables by their primed counterpart. Similarly, ϕ' denotes the formula constructed from ϕ by replacing all program variables by their primed counterpart and the term t' denotes the counterpart of t.

Let α be a program and let s_1, s_2 be states. In the following, "α started in s_1 terminates in s_2" is denoted by $s_1 \overset{\alpha}{\leadsto} s_2$.

3 Formalizing Conditional Non-interference

We use the following quite general, passive attacker model. In our setting attackers may not only observe the values of program variables, but more generally the values of so called *observation expressions.* Observation expressions can be thought of as a generalization of side-effect free JAVA expressions:

Definition 1. *An* observation expression *can be:*

1. *A program variable (including method parameters).*
2. *$e.f$ for e an expression of type C and f a field declared in C.*
3. *$e[t]$ if e is an expression of array type, and t of integer type.*
4. *$op(e_1, \ldots, e_k)$ if op is a data type operation and e_i expressions of matching type.*
5. *The usual conditional operator b ? e_1 : e_2 (e_1, e_2 have to be of the same type).*

6. *The sequence definition operator $seq\{i\}(from, to, e)$. Its semantics is defined by*

$$(seq\{i\}(from, to, e))^{\mathcal{M}} \\ = \langle (e[i \leftarrow n])^{\mathcal{M}},\ (e[i \leftarrow n+1])^{\mathcal{M}},\ \ldots,\ (e[i \leftarrow m-1])^{\mathcal{M}} \rangle$$

if $from^{\mathcal{M}} = n < m = to^{\mathcal{M}}$, and $(seq\{i\}(from, to, e))^{\mathcal{M}} = \langle\rangle$ else.

We denote the concatenation of two observation expressions R_1 and R_2 by $R_1; R_2$.

Attackers can observe the values of a set of (low) observation expressions in the initial and final state of a program run: for any expression an attacker can see the expression and the corresponding evaluation. An attacker can compare observed values as by using the == operator of Java. Additionally we assume that attackers know the program-code.

Let us describe this scenario a bit more formally. Let R be an observation expression an let $\mathcal{M}$ be any interpretation. If s is the initial or the final state of a program run, then attackers are able to observe the tuple $(R, R^{\mathcal{M}^{\leftarrow s}})$, where $R^{\mathcal{M}^{\leftarrow s}} = \langle e_1^{\mathcal{M}^{\leftarrow s}}, \ldots, e_k^{\mathcal{M}^{\leftarrow s}} \rangle$ if $R = \langle e_1, \ldots, e_k \rangle$. Thus, an attacker knows that $e_i^{\mathcal{M}^{\leftarrow s}}$ is the value of the expression e_i in state s (for $1 \leq i \leq k$) and they can compare any two values, $e_i^{\mathcal{M}^{\leftarrow s}} = e_j^{\mathcal{M}^{\leftarrow s'}}$, for any pair of initial or final states s and s'. Knowing the program-code is formalized by the assumption that attackers know which initial state of a program run relates to which final state.

Definition 2 (Agreement of states). *Let R be an observation expression and let $\mathcal{M}$ be an interpretation. Two states s, s' agree on R in $\mathcal{M}$, abbreviated by $agree^{\mathcal{M}}(R, s, s')$, iff $R^{\mathcal{M}^{\leftarrow s}} = R^{\mathcal{M}^{\leftarrow s'}}$.*

Thus two states agree on R if an attacker cannot distinguish them.

Definition 3. *Let R be an observation expression using the local variables $\bar{\mathtt{x}}$ and the variable* heap. *Let $\mathtt{heap}_2$, $\bar{\mathtt{x}}_2$ and $\mathtt{heap}'_2$, $\bar{\mathtt{x}}'_2$ be two copies of these program variables. We will use the following abbreviation*

$$obsEq(\bar{\mathtt{x}}_2, \mathtt{heap}_2, \bar{\mathtt{x}}'_2, \mathtt{heap}'_2, R) \\ \equiv R[\mathtt{heap} \leftarrow \mathtt{heap}_2, \bar{\mathtt{x}} \leftarrow \bar{\mathtt{x}}_2] = R[\mathtt{heap} \leftarrow \mathtt{heap}'_2, \bar{\mathtt{x}} \leftarrow \bar{\mathtt{x}}'_2]$$

We note that for any interpretation $\mathcal{M}$ we have

$$obsEq(\bar{\mathtt{x}}_2, \mathtt{heap}_2, \bar{\mathtt{x}}'_2, \mathtt{heap}'_2, R)^{\mathcal{M}} = tt \text{ iff } agree^{\mathcal{M}}(R, s_2, s'_2)$$

for $s_2(\bar{\mathtt{x}}) = \bar{\mathtt{x}}_2^{\mathcal{M}}$, $s_2(\mathtt{heap}) = \mathtt{heap}_2^{\mathcal{M}}$, $s'_2(\bar{\mathtt{x}}) = \bar{\mathtt{x}}_2'^{\mathcal{M}}$, $s'_2(\mathtt{heap}) = \mathtt{heap}_2'^{\mathcal{M}}$.

Definition 4 (Conditional Non-Interference). *Let α be a program with program variables* heap *and $\bar{\mathtt{x}}$, let R_1, R_2 be observation expressions and let ϕ be a formula. Further, let* heap *and $\bar{\mathtt{x}}$ be the only program variables occurring in R_1, R_2 and ϕ.*

Program α allows information to flow only from R_1 to R_2 when started in s_1 under condition ϕ, denoted by flow$(s_1, \alpha, R_1, R_2, \phi)$, iff for all interpretations $\mathcal{M}$ and all states s'_1, s_2, s'_2 such that $s_1 \overset{\alpha}{\rightsquigarrow} s_2$ and $s'_1 \overset{\alpha}{\rightsquigarrow} s'_2$ we have

$$\textit{if}\quad \phi^{\mathcal{M}^{\leftarrow s_1}} = tt,\ \phi^{\mathcal{M}^{\leftarrow s'_1}} = tt \textit{ and agree}^{\mathcal{M}}(R_1, s_1, s'_1) \quad \textit{then} \quad \textit{agree}^{\mathcal{M}}(R_2, s_2, s'_2).$$

flow(α, R_1, R_2, ϕ) denotes the case that flow$(s_1, \alpha, R_1, R_2, \phi)$ holds for all states s_1.

We think of R_1, R_2 as the publicly observable information of a state of the system. In the simplest case what goes into R_i is determined by explicit declarations of which program variables and which fields are considered low. In more sophisticated scenarios the R_i may be inferred from a multi-level security lattice (see for instance [20]). Usually we will have $R_1 = R_2$. But, there are other cases: to *declassify* an expression e_{decl}, for instance, one would choose $R_1 = R_2; e_{decl}$.

Lemma 1 (Compositionality of *flow*). *Let α_1, α_2 be programs and let $\alpha_1;\alpha_2$ be their sequential composition. If flow$(s_1, \alpha_1, R_1, R_2, \phi_1)$, flow$(s_2, \alpha_2, R_2, R_3, \phi_2)$ and $(\phi_1 \Rightarrow wp(\alpha_1, \phi_2))^{\mathcal{M}^{\leftarrow s_1}} = tt$ hold for all interpretations $\mathcal{M}$ and all states s_1, s_2, s_3 such that $s_1 \overset{\alpha_1}{\rightsquigarrow} s_2$ and $s_2 \overset{\alpha_2}{\rightsquigarrow} s_3$ then flow$(s_1, \alpha_1;\alpha_2, R_1, R_3, \phi_1)$ holds.*

Proof. Let s'_1, s'_2, s'_3 be a second set of states such that $s'_1 \overset{\alpha_1}{\rightsquigarrow} s'_2$, $s'_2 \overset{\alpha_2}{\rightsquigarrow} s'_3$, and $(\phi_1 \Rightarrow wp(\alpha_1, \phi_2))^{\mathcal{M}^{\leftarrow s'_1}} = tt$. Assume that the precondition ϕ_1 holds in s_1 and s_2. In other words, assume $\phi_1^{\mathcal{M}^{\leftarrow s_1}} = tt$ and $\phi_1^{\mathcal{M}^{\leftarrow s'_1}} = tt$. Additionally, assume agree$^{\mathcal{M}}(R_1, s_1, s'_1)$. By $(\phi_1 \Rightarrow wp(\alpha_1, \phi_2))^{\mathcal{M}^{\leftarrow s_1}} = tt$ and $(\phi_1 \Rightarrow wp(\alpha_1, \phi_2))^{\mathcal{M}^{\leftarrow s'_1}}$ $= tt$ we infer $\phi_2^{\mathcal{M}^{\leftarrow s_2}} = tt$ and $\phi_2^{\mathcal{M}^{\leftarrow s'_2}} = tt$. In other words, we infer that ϕ_2 holds in s_2 and s'_2. By flow$(s_1, \alpha_1, R_1, R_2, \phi_1)$ we get agree$^{\mathcal{M}}(R_2, s_2, s'_2)$. Further, by agree$^{\mathcal{M}}(R_2, s_2, s'_2)$, $\phi_2^{\mathcal{M}^{\leftarrow s_2}} = tt$, $\phi_2^{\mathcal{M}^{\leftarrow s'_2}} = tt$ and flow$(s_2, \alpha_2, R_2, R_3, \phi_2)$ we get agree$^{\mathcal{M}}(R_3, s_3, s'_3)$, as desired. □

4 Efficient Self-composition

The Problem. We illustrate the efficiency issues of self-composition approaches by an example. Consider the following program α:

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

Let `l` be a low variable and let `h` be a high variable. Then α has no information leak: the value of `l` after line 2 is the same as the initial value of `l`. Thus the value of `l` after line 3 depends only on the initial value of `l`. The control flow graph of α is sketched in Figure 1(a).

In the self-composition approach a copy α' of α is constructed by replacing all program variables by renamed ones. We decided to rename `l` to `l2` and `h` to `h2`. This leads to the following self-composed program $\alpha; \alpha'$:

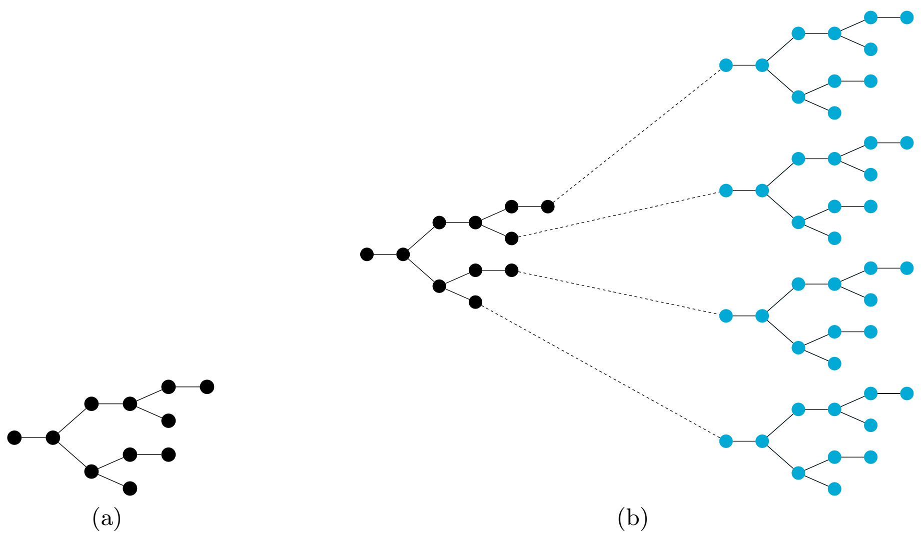

Fig. 1. Sketch of the control flow graphs of (a) the original program and (b) the self-composed program

```
l = l + h;
if (h != 0) { l = l - h; }
if (l > 0) { l--; }
l2 = l2 + h2;
if (h2 != 0) { l2 = l2 - h2; }
if (l2 > 0) { l2--; }
```

The control flow graph of $\alpha; \alpha'$ is sketched in Figure 1(b).

h does not interfere with l in α, iff $\alpha; \alpha'$ started in any state with $\mathtt{l} = \mathtt{l2}$ terminates in a state where $\mathtt{l} = \mathtt{l2}$ holds. Hence, in the self-composition approach essentially the outcome of any path through α has to be compared to the outcome of any path through α'. If n is the number of paths through α, this results in $O(n^2)$ comparisons of the low variables. In contrast, specialized information flow calculi, which consider α only once, have to check only the outcome of the n paths through α. This is one reason why self-composition often is considered to be inefficient. The other reason is that the computation of a weakest precondition for $\alpha; \alpha'$ is at least twice as costly as the calculation of a weakest precondition for α.

Reducing the Cost for the Weakest Precondition Calculation. We tackle the second problem first by showing that it is possible to show non-interference in self-composition style with the help of only one weakest preconditions calculation on α.

Let $\mathtt{heap}_2$ and $\bar{\mathtt{x}}_2$ be a set of fresh program variables. $wp(\alpha, \mathtt{heap} = \mathtt{heap}_2 \wedge \bar{\mathtt{x}} = \bar{\mathtt{x}}_2)$ characterizes the initial state s such that α started in s terminates in the state described by $\mathtt{heap}_2$ and $\bar{\mathtt{x}}_2$. Further we observe that $\mathit{wp}(\alpha', \phi') = \mathit{wp}(\alpha, \phi)'$ holds. Therefore, $\mathit{wp}(\alpha', \mathtt{heap}' = \mathtt{heap}_2' \wedge \bar{\mathtt{x}}' = \bar{\mathtt{x}}_2')$ can be constructed from $\mathit{wp}(\alpha, \mathtt{heap} = \mathtt{heap}_2 \wedge \bar{\mathtt{x}} = \bar{\mathtt{x}}_2)$ by the renaming of $\mathtt{heap}$, $\bar{\mathtt{x}}$, $\mathtt{heap}_2$ and $\bar{\mathtt{x}}_2$ to $\mathtt{heap}'$, $\bar{\mathtt{x}}'$, $\mathtt{heap}_2'$ and $\bar{\mathtt{x}}_2'$, respectively.

During the construction of $\mathit{wp}(\alpha, \mathtt{heap} = \mathtt{heap}_2 \wedge \bar{\mathtt{x}} = \bar{\mathtt{x}}_2)$ fresh (skolem) symbols might be introduced (see Sect. 5). Let c' be a fresh (primed) symbol for any fresh symbol c introduced during the construction of $\mathit{wp}(\alpha, \mathtt{heap} = \mathtt{heap}_2 \wedge \bar{\mathtt{x}} = \bar{\mathtt{x}}_2)$ such that c' does not occur in $\mathit{wp}(\alpha, \mathtt{heap} = \mathtt{heap}_2 \wedge \bar{\mathtt{x}} = \bar{\mathtt{x}}_2)$. Let $\mathit{wp}'(\alpha', \mathtt{heap}' = \mathtt{heap}_2' \wedge \bar{\mathtt{x}}' = \bar{\mathtt{x}}_2')$ denote the formula which results from $\mathit{wp}(\alpha', \mathtt{heap}' = \mathtt{heap}_2' \wedge \bar{\mathtt{x}}' = \bar{\mathtt{x}}_2')$ by renaming all fresh symbols to their primed counterparts. Given these weakest preconditions, non-interference can be proved as follows:

Theorem 1. *Let α be a program with program variables* $\mathtt{heap}$ *and* $\bar{\mathtt{x}}$*, let R_1, R_2 be observation expressions and let ϕ be a formula. Let* $\mathtt{heap}$ *and* $\bar{\mathtt{x}}$ *be the only program variables occurring in R_1, R_2 and ϕ. Let further* $\mathtt{heap}'$ *and* $\bar{\mathtt{x}}'$, $\mathtt{heap}_2$ *and* $\bar{\mathtt{x}}_2$ *and* $\mathtt{heap}_2'$ *and* $\bar{\mathtt{x}}_2'$ *be three copies of the program variables of α; let α' and ϕ' be the primed counterparts to α and ϕ, respectively.*

Let $\Psi_{\alpha,\bar{\mathtt{x}},R_1,R_2,\phi}$ be defined by

$$\begin{aligned}\Psi_{\alpha,\bar{\mathtt{x}},R_1,R_2,\phi} \equiv \quad & (\phi \wedge \mathit{wp}(\alpha, (\mathtt{heap} = \mathtt{heap}_2 \wedge \bar{\mathtt{x}} = \bar{\mathtt{x}}_2)))\\ & \wedge (\phi' \wedge \mathit{wp}'(\alpha', (\mathtt{heap}' = \mathtt{heap}_2' \wedge \bar{\mathtt{x}}' = \bar{\mathtt{x}}_2')))\\ & \wedge \mathit{obsEq}(\bar{\mathtt{x}}, \mathtt{heap}, \bar{\mathtt{x}}', \mathtt{heap}', R_1)\\ & \Rightarrow \mathit{obsEq}(\bar{\mathtt{x}}_2, \mathtt{heap}_2, \bar{\mathtt{x}}_2', \mathtt{heap}_2', R_2)\end{aligned}$$

The formula $\Psi_{\alpha,\bar{\mathtt{x}},R_1,R_2,\phi}$ is universally valid iff $\mathit{flow}(\alpha, R_1, R_2, \phi)$ holds.

Proof.
"$\Rightarrow$": Let $\Psi_{\alpha,\bar{\mathtt{x}},R_1,R_2,\phi}$ be universally valid. We have to show $\mathrm{flow}(\alpha, R_1, R_2, \phi)$. Consider an arbitrary structure $\mathcal{M}$ and let s_1, s_2, s_1', s_2' be the states given by $s_i(\bar{x}) = \bar{x}_i^{\mathcal{M}}$, $s_i(\mathtt{heap}) = \mathtt{heap}_i^{\mathcal{M}}$, $s_i'(\bar{x}) = (\bar{x}_i')^{\mathcal{M}}$, $s_i'(\mathtt{heap}) = (\mathtt{heap}_i')^{\mathcal{M}}$. According to Definition 4, we have to show that $s_1 \overset{\alpha}{\rightsquigarrow} s_2$, $s_1' \overset{\alpha}{\rightsquigarrow} s_2'$, $\phi^{\mathcal{M}\leftarrow s_1} = \mathit{tt}$, $\phi^{\mathcal{M}\leftarrow s_1'} = \mathit{tt}$ and $\mathrm{agree}^{\mathcal{M}}(R_1, s_1, s_1')$ imply $\mathrm{agree}^{\mathcal{M}}(R_2, s_2, s_2')$.

Assume $s_1 \overset{\alpha}{\rightsquigarrow} s_2$, $s_1' \overset{\alpha}{\rightsquigarrow} s_2'$, $\phi^{\mathcal{M}\leftarrow s_1} = \mathit{tt}$, $\phi^{\mathcal{M}\leftarrow s_1'} = \mathit{tt}$ and $\mathrm{agree}^{\mathcal{M}}(R_1, s_1, s_1')$. Then there exists a structure $\mathcal{M}'$ such that (1) $\mathcal{M}'$ differs from $\mathcal{M}$ only in the interpretation of the fresh symbols and (2) the formulas $\mathit{wp}(\alpha, (\mathtt{heap} = \mathtt{heap}_2 \wedge \bar{\mathtt{x}} = \bar{\mathtt{x}}_2))$ and $\mathit{wp}'(\alpha', (\mathtt{heap}' = \mathtt{heap}_2' \wedge \bar{\mathtt{x}}' = \bar{\mathtt{x}}_2'))$ hold in $\mathcal{M}'$. Because ϕ and ϕ' do not contain fresh variables, $\phi^{\mathcal{M}\leftarrow s_1} = \phi^{\mathcal{M}'\leftarrow s_1} = \mathit{tt}$ and $\phi^{\mathcal{M}\leftarrow s_1'} = \phi^{\mathcal{M}'\leftarrow s_1'} = \mathit{tt}$. Therefore, the first two lines of $\Psi_{\alpha,\bar{\mathtt{x}},R_1,R_2,\phi}$ are valid in $\mathcal{M}'$.

Further we get by the remark to Definition 3 that line 3 evaluates to true iff $\mathrm{agree}^{\mathcal{M}'}(R_1, s_1, s_2)$ holds. Because $\mathit{obsEq}(\bar{\mathtt{x}}, \mathtt{heap}, \bar{\mathtt{x}}', \mathtt{heap}', R_1)$ does not contain fresh variables, this is the case iff $\mathrm{agree}^{\mathcal{M}}(R_1, s_1, s_2)$ holds. Thus, the formula $\mathit{obsEq}(\bar{\mathtt{x}}, \mathtt{heap}, \bar{\mathtt{x}}', \mathtt{heap}', R_1)$ is valid in $\mathcal{M}'$. Now we get by the universal validity of $\Psi_{\alpha,\bar{\mathtt{x}},R_1,R_2,\phi}$ that line 4 has to hold in $\mathcal{M}'$, too. Again, by the remark to Definition 3 $\mathrm{agree}^{\mathcal{M}'}(R_2, s_2, s_2')$ holds and because $\mathit{obsEq}(\bar{\mathtt{x}}_2, \mathtt{heap}_2, \bar{\mathtt{x}}_2', \mathtt{heap}_2', R_2)$ does not contain fresh variables $\mathrm{agree}^{\mathcal{M}}(R_2, s_2, s_2')$ holds, too.

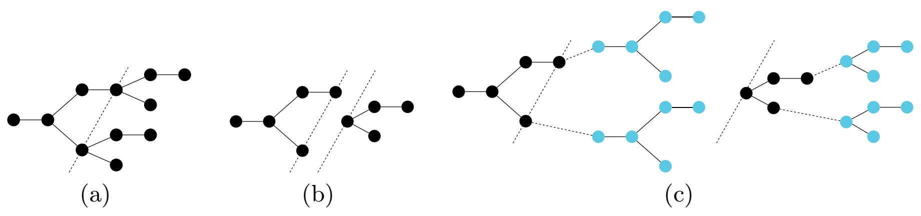

Fig. 2. Reducing the verification overhead by compositional reasoning

"$\Leftarrow$": Let $\text{flow}(\alpha, R_1, R_2, \phi)$ hold. We have to show that $\Psi_{\alpha,\bar{\mathtt{x}},R_1,R_2,\phi}$ is universally valid. Again, consider an arbitrary structure $\mathcal{M}$ and let s_1, s_2, s'_1, s'_2 be the states given by $s_i(\bar{x}) = \bar{x}_i^{\mathcal{M}}$, $s_i(\mathtt{heap}) = \mathtt{heap}_i^{\mathcal{M}}$, $s'_i(\bar{x}) = (\bar{x}'_i)^{\mathcal{M}}$, $s'_i(\mathtt{heap}) = (\mathtt{heap}'_i)^{\mathcal{M}}$. We have to show that $\Psi_{\alpha,\bar{\mathtt{x}},R_1,R_2,\phi}$ is valid in $\mathcal{M}$.

Assume that the first three lines of $\Psi_{\alpha,\bar{\mathtt{x}},R_1,R_2,\phi}$ are valid in $\mathcal{M}$ (otherwise we are already done). We have to show that $\mathit{obsEq}(\bar{\mathtt{x}}_2, \mathtt{heap}_2, \bar{\mathtt{x}}'_2, \mathtt{heap}'_2, R_2)$ is valid in $\mathcal{M}$, too. As before, we get by the validity of the first three lines that $s_1 \overset{\alpha}{\rightsquigarrow} s_2$, $s'_1 \overset{\alpha}{\rightsquigarrow} s'_2$, $\phi^{\mathcal{M}^{\leftarrow s_1}} = tt$, $\phi^{\mathcal{M}^{\leftarrow s'_1}} = tt$ and $\text{agree}^{\mathcal{M}}(R_1, s_1, s'_1)$ hold. Therefore we get by $\text{flow}(\alpha, R_1, R_2, \phi)$ that $\text{agree}^{\mathcal{M}}(R_2, s_2, s'_2)$ holds, too. As before, this implies that $\mathit{obsEq}(\bar{\mathtt{x}}_2, \mathtt{heap}_2, \bar{\mathtt{x}}'_2, \mathtt{heap}'_2, R_2)$ holds in $\mathcal{M}$. $\square$

Altogether we have shown that it is possible to prove non-interference in self-composition style with the help of only one weakest precondition calculation on α.

Note 1. Because $\mathit{wp}(\alpha, (\mathtt{heap} = \mathtt{heap}_2 \wedge \bar{\mathtt{x}} = \bar{\mathtt{x}}_2))$ occurs on the left hand side of an implication, it may *not* be approximated in the usual manner by a formula ψ such that $\psi \Rightarrow \mathit{wp}(\alpha, (\mathtt{heap} = \mathtt{heap}_2 \wedge \bar{\mathtt{x}} = \bar{\mathtt{x}}_2))$ holds. Instead, $\mathit{wp}(\alpha, (\mathtt{heap} = \mathtt{heap}_2 \wedge \bar{\mathtt{x}} = \bar{\mathtt{x}}_2)) \Rightarrow \psi$ needs to hold. Because we consider deterministic programs, the usual wp-calculus can still be used to calculate ψ in the following manner: instead of calculating a condition under which the state s_2 given by $\mathtt{heap}_2$ and $\bar{\mathtt{x}}_2$ is definitely reached we have to calculate a condition ψ_{not} under which s_2 is definitely *not* reached. ψ is then the negation of ψ_{not}. In other words, $\mathit{wp}(\alpha, (\mathtt{heap} = \mathtt{heap}_2 \wedge \bar{\mathtt{x}} = \bar{\mathtt{x}}_2))$ and $\mathit{wp}'(\alpha', (\mathtt{heap}' = \mathtt{heap}'_2 \wedge \bar{\mathtt{x}}' = \bar{\mathtt{x}}'_2))$ in Theorem 1 have to be replaced by $\neg \mathit{wlp}(\alpha, (\mathtt{heap} \neq \mathtt{heap}_2 \vee \bar{\mathtt{x}} \neq \bar{\mathtt{x}}_2))$ and $\neg \mathit{wlp}'(\alpha', (\mathtt{heap}' \neq \mathtt{heap}'_2 \vee \bar{\mathtt{x}}' \neq \bar{\mathtt{x}}'_2))$, respectively, if approximations are involved. The intuition behind this replacement is that ψ_{not} characterizes a set S_{not} of initial states such that α started in any $s \in S_{\text{not}}$ does not terminate in s_2 and, thus, ψ characterizes a set S of initial states such that if there is an initial state s_1 such that α started in s_1 terminates in s_2, then s_1 is an element of S.

Reducing the Number of Comparisons. The second problem, reducing the number of comparisons, can be tackled with the help of compositional reasoning, if the structure of the program allows for it. Reconsider the initial example:

```
1  l = l + h;
2  if (h != 0) { l = l - h; }
3  // - - - - - - - - - - - - - - -
4  if (l > 0) { l--; }
```

As discussed above, the first part, lines 1 and 2, and the second part, line 4, are non-interfering on their own. Therefore, by Lemma 1, the complete program is non-interfering. As illustrated in Fig. 2, checking the two parts independently from each other results in less verification effort: the control flow graph of each self-composed part on its own contains only four paths. Thus, altogether only eight comparisons have to be made to prove non-interference of the complete program. Checking the complete program at once would require (about) 12 comparisons.[1] We summarize the above observation in the following lemma.

Lemma 2. *Let α be a program with m branching statements.*

If α can be divided into m non-interfering blocks with at most one branching statement per block, then non-interference of α can be shown with the help of self-composition with $3m$ comparisons.

Proof. Using symmetry, for any block at most 3 paths have to be considered. Hence, for m blocks $3m$ comparisons are sufficient.

Because a program with m branching statements has at least $n = m + 1$ paths, Lemma 2 shows that the verification effort of self-composition approaches can be reduced from $O(n^2)$ comparisons to $O(n)$, 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 non-interfering, it is necessary to use the fact that b_1 is non-interfering in the proof of b_2. Unfortunately, the self-composition approach does not easily lend itself to such compositional / modular verification. In the next section the problem of compositional / modular reasoning will be discussed.

5 Modular Self-composition with Contracts

In the context of functional verification, modularity is achieved through method contracts. We want to extend this approach to the verification of information flow properties. We define *information flow contracts* on the basis of [20]:

[1] 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.

Definition 5 (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* $\mathcal{C}_{\mathtt{b},\bar{\mathtt{x}}::\bar{A}} = (\mathit{Pre}, R_1, R_2)$*, 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 post-state.*

A flow contract $\mathcal{C}_{\mathtt{b},\bar{\mathtt{x}}::\bar{A}} = (\mathit{Pre}, R_1, R_2)$ *is valid iff for all states s the predicate* $\mathit{flow}(s, \mathtt{b}, R_1, R_2, \mathit{Pre})$ *is valid.*

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

Example 1. Consider the example `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 $\mathcal{C}_{b_1,\bar{\mathtt{x}}::\bar{A}} = \mathcal{C}_{b_2,\bar{\mathtt{x}}::\bar{A}} = (\mathit{true}, \mathtt{l}, \mathtt{l})$ be flow contracts for b_1 and b_2. To prove $\mathcal{C}_{b_2,\bar{\mathtt{x}}::\bar{A}}$ by self-composition,

$$\mathit{wp}(\texttt{if (l>0) \{l++; } b_1\texttt{\}; if (l}'\texttt{>0) \{l}'\texttt{++; } b_1'\texttt{\}},\ \mathtt{l} = \mathtt{l}') \tag{1}$$

has to be computed. Application of the *wp*-calculus yields:

$$\begin{aligned}\equiv \quad & \big(\mathtt{l} > 0 \Rightarrow \mathit{wp}(b_1, (\mathtt{l}' > 0 \Rightarrow \mathit{wp}(b_1',\ \mathtt{l} = \mathtt{l}')[\mathtt{l}' \leftarrow \mathtt{l}' + 1]))[\mathtt{l} \leftarrow \mathtt{l} + 1]\big) \\ & \wedge \big(\mathtt{l} \leq 0 \Rightarrow (\mathtt{l}' > 0 \Rightarrow \mathit{wp}(b_1',\ \mathtt{l} = \mathtt{l}')[\mathtt{l}' \leftarrow \mathtt{l}' + 1])\big) \\ & \wedge \big(\mathtt{l} > 0 \Rightarrow \mathit{wp}(b_1, (\mathtt{l}' \leq 0 \Rightarrow \mathtt{l} = \mathtt{l}'))[\mathtt{l} \leftarrow \mathtt{l} + 1]\big) \\ & \wedge \big(\mathtt{l} \leq 0 \Rightarrow (\mathtt{l}' \leq 0 \Rightarrow \mathtt{l} = \mathtt{l}')\big)\end{aligned} \tag{2}$$

If $\mathtt{l} = \mathtt{l}'$ is valid, then the last three lines of (2) are obviously fulfilled. To see that also the first line is fulfilled, $\mathcal{C}_{b_1,\bar{\mathtt{x}}::\bar{A}}$ needs to be used to remove the remaining *wp*'s—but it is not obvious how this can be done, because the *wp*'s are nested. A similar problem occurs if Theorem 1 is used to prove $\mathcal{C}_{b_2,\bar{\mathtt{x}}::\bar{A}}$.

The main idea of the solution is a coordinated delay of the application of flow contracts. The solution is compatible with the optimizations of Section 4 and additionally allows the combination of flow contracts with functional contracts.

Let b be a block with the functional contract $\mathcal{F}_{\mathtt{b},\bar{\mathtt{x}}::\bar{A}} = (\mathit{Pre}, \mathit{Post}, \mathit{Mod})$ consisting of: (1) a formula *Pre* representing the precondition; (2) a formula *Post* representing the postcondition; and (3) a term *Mod* representing the modifies clause for b. We introduce the formula

$$\mathit{Pre} \wedge (\mathit{Post} \Rightarrow \phi)[\mathit{Subst}_{anon}] \tag{3}$$

Here, $\mathit{Subst}_{anon} = (\mathtt{heap} \leftarrow \mathit{anon}\{\mathtt{heap}, \mathit{Mod}, h\}, \bar{\mathtt{x}} \leftarrow \bar{x}')$ is an anonymising substitution 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 $\mathtt{heap}_{pre} = \mathtt{heap}$ and $\bar{\mathtt{x}}_{pre} = \bar{\mathtt{x}}$ 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 post-condition. Additionally, we require that *Pre* and *Post* entail a formula which expresses that the heap is wellformed. For the sake of simplicity we do not handle exceptions here.

If b fulfills the contract $\mathcal{F}_{\mathtt{b},\bar{\mathtt{x}}::\bar{A}} = (\mathit{Pre}, \mathit{Post}, \mathit{Mod})$, then formula (3) approximates $wp(b,\ \phi)$ in the following sense:

Lemma 3

$$\mathit{Pre} \wedge (\mathit{Post} \Rightarrow \phi)[\mathit{Subst}_{anon}] \quad \Rightarrow \quad wp(b,\ \phi)$$

is valid in any interpretation $\mathcal{M}$.

Proof. See for example [12].

We introduce a new two-state predicate $C_{\mathtt{b}}(\bar{x}, h, \bar{x}', h')$ with the intended meaning that b started in state $\langle \bar{\mathtt{x}}, \mathtt{heap}\rangle \mapsto \langle \bar{x}, h\rangle$ terminates in state $\langle \bar{\mathtt{x}}, \mathtt{heap}\rangle \mapsto \langle \bar{x}', h'\rangle$. This predicate can be integrated into the approximation (3) of $wp(\mathtt{b}, \phi)$ as follows:

$$\begin{array}{l} \mathit{Pre} \wedge \big(\quad C_{\mathtt{b}}(\bar{\mathtt{x}}, \mathtt{heap}, \bar{x}', h') \\ \qquad \wedge (\mathtt{heap} = h' \wedge \bar{\mathtt{x}} = \bar{x}')[\mathit{Subst}_{anon}] \\ \qquad \Rightarrow (\mathit{Post} \Rightarrow \phi)[\mathit{Subst}_{anon}] \\ \quad \big) \end{array} \tag{4}$$

where h' of type *Heap* and $\bar{x}'$ of types $\bar{A}$ are fresh function symbols. By Lemma 4 below, formula (4) implies $wp(\mathtt{b}, \phi)$ and therefore is also a correct approximation of $wp(\mathtt{b}, \phi)$. The introduction of $C_{\mathtt{b}}(\bar{x}, h, \bar{x}', h')$ (by approximating $wp(\mathtt{b}, \phi)$ by (4)) allows us to store the initial and the final state of b for a delayed application of information flow contracts: as we show in Theorem 2 below, if two predicates $C_{\mathtt{b}}(\bar{x}_1, h_1, \bar{x}'_1, h'_1)$ and $C_{\mathtt{b}}(\bar{x}_2, h_2, \bar{x}'_2, h'_2)$ are true in a structure $\mathcal{M}$, then they can be approximated by an instantiation of a flow contract $\mathcal{C}_{\mathtt{b},\bar{\mathtt{x}}::\bar{A}} = (\mathit{Pre}, R_1, R_2)$ for b by

$$\begin{array}{c} \mathit{Pre}[\mathtt{heap} \leftarrow h_1, \bar{\mathtt{x}} \leftarrow \bar{x}_1] \wedge \mathit{Pre}[\mathtt{heap} \leftarrow h_2, \bar{\mathtt{x}} \leftarrow \bar{x}_2] \\ \Rightarrow \big(\mathit{obsEq}(\bar{x}_1, h_1, \bar{x}'_1, h'_1, R_1) \Rightarrow \mathit{obsEq}(\bar{x}_2, h_2, \bar{x}'_2, h'_2, R_2)\big)\ . \end{array} \tag{5}$$

Example 2. Let $\mathcal{F}_{b_1,\bar{\mathtt{x}}::\bar{A}} = (\mathit{true}, \mathit{true}, \mathit{allLocs})$ be the trivial functional contract for b_1. Applied on our example, the first line of (2) can be simplified as follows. First $wp(b'_1,\ \mathtt{l} = \mathtt{l}')[\mathtt{l}' \leftarrow \mathtt{l}' + 1]$ can be approximated by (4) by

$$\begin{array}{l} (\quad C_{b_1}(\mathtt{l}', \mathtt{heap}', \ell', h') \\ \quad \wedge (\mathtt{heap}' = h' \wedge \mathtt{l}' = \ell')[\mathtt{heap}' \leftarrow h'_{anon}, \mathtt{l}' \leftarrow \ell'_{anon}] \\ \quad \Rightarrow (\mathtt{l} = \mathtt{l}')[\mathtt{heap}' \leftarrow h'_{anon}, \mathtt{l}' \leftarrow \ell'_{anon}] \\ )[\mathtt{l}' \leftarrow \mathtt{l}' + 1] \end{array} \tag{6}$$

$$\equiv\ C_{b_1}(\mathtt{l}' + 1, \mathtt{heap}', \ell', h') \wedge h'_{anon} = h' \wedge \ell'_{anon} = \ell' \Rightarrow \mathtt{l} = \ell'_{anon} \tag{7}$$

Similarly, $wp(b_1, (\mathtt{l}' > 0 \Rightarrow \phi'))[\mathtt{l} \leftarrow \mathtt{l} + 1]$ can be approximated by

$$\mathtt{l}' > 0 \wedge C_{b_1}(\mathtt{l} + 1, \mathtt{heap}, \ell, h) \wedge h_{anon} = h \wedge \ell_{anon} = \ell \Rightarrow \phi'_{anon} \tag{8}$$

with $\phi'_{anon} = \phi'[\mathtt{heap} \leftarrow h_{anon}, \mathtt{l} \leftarrow \ell_{anon}]$. Therefore (2) can be approximated by

$$\begin{array}{l} \mathtt{l} > 0 \Rightarrow (\quad \mathtt{l}' > 0 \wedge C_{b_1}(\mathtt{l}+1, \mathtt{heap}, \ell, h) \wedge h_{anon} = h \wedge \ell_{anon} = \ell \\ \qquad \Rightarrow (\quad C_{b_1}(\mathtt{l}'+1, \mathtt{heap}', \ell', h') \\ \qquad\quad \wedge h'_{anon} = h' \wedge \ell'_{anon} = \ell' \\ \qquad\quad \Rightarrow \ell_{anon} = \ell'_{anon} \\ \qquad) \\ \quad) \end{array} \tag{9}$$

$$\begin{array}{ll} \equiv & \mathtt{l} > 0 \wedge C_{b_1}(\mathtt{l}+1, \mathtt{heap}, \ell, h) \wedge h_{anon} = h \wedge \ell_{anon} = \ell \\ & \wedge\, \mathtt{l}' > 0 \wedge C_{b_1}(\mathtt{l}'+1, \mathtt{heap}', \ell', h') \wedge h'_{anon} = h' \wedge \ell'_{anon} = \ell' \\ & \Rightarrow \ell_{anon} = \ell'_{anon} \end{array} \tag{10}$$

Application of $\mathcal{C}_{b_1,\bar{\mathtt{x}}::\bar{A}}$ by Theorem 2 yields

$$\begin{array}{ll} \equiv & \mathtt{l} > 0 \wedge h_{anon} = h \wedge \ell_{anon} = \ell \\ & \wedge\, \mathtt{l}' > 0 \wedge h'_{anon} = h' \wedge \ell'_{anon} = \ell' \\ & \wedge\, (\mathtt{l}+1 = \mathtt{l}'+1 \Rightarrow \ell = \ell') \\ & \Rightarrow \ell_{anon} = \ell'_{anon} \end{array} \tag{11}$$

which is obviously true if $\mathtt{l} = \mathtt{l}'$.

Formally, $C_{\mathtt{b}}(\bar{x}, h, \bar{x}', h')$ is valid in structure $\mathcal{M}$ iff

$$wp(\mathtt{b},\ \mathtt{heap} = h' \wedge \bar{\mathtt{x}} = \bar{x}')[\bar{\mathtt{x}} \leftarrow \bar{x}, \mathtt{heap} \leftarrow h]$$

is valid in $\mathcal{M}$. In the following we show that the above approach is sound.

Lemma 4. *Let* b *be a block which fulfills the functional contract* $\mathcal{F}_{\mathtt{b},\bar{\mathtt{x}}::\bar{A}} = (Pre, Post, Mod)$.
$wp(b,\ \phi)$ is valid if (4) is valid.

Proof. Because of Lemma 3 it suffices to show that (4) is valid iff (3) is valid.
If (3) is valid then by simple propositional logic also (4) is valid. So, we assume that (4) is valid and set out to show that (3) is true in an arbitrary structure $\mathcal{M}$. By assumption *Pre* is true in $\mathcal{M}$. We assume $Post[Subst_{anon}]$ is true in $\mathcal{M}$ with the aim to show that $\phi[Subst_{anon}]$ is also true in $\mathcal{M}$. Since the new constant symbols h' and $\bar{x}'$ do not occur in $Post[Subst_{anon}]$ we find a structure $\mathcal{M}'$ that differs from $\mathcal{M}$ only in the interpretation of these symbols such that in $\mathcal{M}'$ both $Post[Subst_{anon}]$ and $C_{\mathtt{b}}(\bar{\mathtt{x}}, \mathtt{heap}, \bar{x}', h') \wedge (\mathtt{heap} = h' \wedge \bar{\mathtt{x}} = \bar{x}')[Subst_{anon}]$ are true. This may be achieved by choosing $\mathcal{M}'$ such that the state s_2 presented by $(h'^{\mathcal{M}'}, \bar{x}'^{\mathcal{M}'})$ is the final state of b when started in the state s_1 presented by $(\mathtt{heap}^{\mathcal{M}}, \bar{\mathtt{x}}^{\mathcal{M}})$. By validity of (4) we obtain that $\phi[Subst_{anon}]$ is true in $\mathcal{M}'$. Since $\phi[Subst_{anon}]$ does likewise not contain the new symbols it is also true in the orignal structure $\mathcal{M}$. □

Theorem 2. *Let* $\mathtt{b}$ *be a block fulfilling the flow contract* $\mathcal{C}_{\mathtt{b},\bar{\mathtt{x}}::\bar{A}} = (Pre, R_1, R_2)$. *(5) is valid if* $C_{\mathtt{b}}(\bar{x}_1, h_1, \bar{x}'_1, h'_1)$ *and* $C_{\mathtt{b}}(\bar{x}_2, h_2, \bar{x}'_2, h'_2)$ *are valid.*

Proof. We need to show, that under the given assumptions the implication (5) is true in any first-order structure $\mathcal{M}$. So we assume that the left-hand side of (5) is true in $\mathcal{M}$, i.e. $Pre[\mathtt{heap} \leftarrow h_1, \bar{\mathtt{x}} \leftarrow \bar{x}_1]^{\mathcal{M}} = tt$ and $Pre[\mathtt{heap} \leftarrow h_2, \bar{\mathtt{x}} \leftarrow \bar{x}_2]^{\mathcal{M}} = tt$. By assumption $C_{\mathtt{b}}(\bar{x}_1, h_1, \bar{x}'_1, h'_1)$ and $C_{\mathtt{b}}(\bar{x}_2, h_2, \bar{x}'_2, h'_2)$ are true in $\mathcal{M}$, which by definition says $wp(\mathtt{b},\ \mathtt{heap} = h'_1 \wedge \bar{\mathtt{x}} = \bar{x}'_1)[\bar{\mathtt{x}} \leftarrow \bar{x}_1, \mathtt{heap} \leftarrow h_1]^{\mathcal{M}} = tt$ and $wp(\mathtt{b},\ \mathtt{heap} = h'_2 \wedge \bar{\mathtt{x}} = \bar{x}'_2)[\bar{\mathtt{x}} \leftarrow \bar{x}_2, \mathtt{heap} \leftarrow h_2]^{\mathcal{M}} = tt$. The assumption that the flow contract $\mathcal{C}_{\mathtt{b},\bar{\mathtt{x}}::\bar{A}} = (Pre, R_1, R_2)$ is fulfilled implies via Theorem 1 that $\Psi_{b,\bar{\mathtt{x}},R_1,R_2,Pre}$ is true in $\mathcal{M}$. Inspection of this formula shows that in the present situation it implies that $obsEq(\bar{x}_1, h_1, \bar{x}'_1, h'_1, R_1) \Rightarrow obsEq(\bar{x}_2, h_2, \bar{x}'_2, h'_2, R_2)$ is valid in $\mathcal{M}$, as desired. □

6 Related Work

The most popular approaches to check for non-interference of programs are approximative methods like security type systems (a prominent example in this field is the JIF-System [14]), the analyses of the dependence graph of a program for graph-theoretical reachability properties [11], specialized approximative information flow calculi based on Hoare like logics [1] and the usage of abstraction and ghost code for explicit tracking of dependencies [17,7,21]. These approaches are efficient, but do not have the precision of self-composition nor do they allow for as fine-grained specifications as they are possible with the help of observation expressions (Section 3). Nanevski, Banerjee and Garg [15] formalise information flow properties in a higher-order logic and use Coq for the verification of those properties. This approach seems to be extremely expressive, but comes with the price of more and more complex interactions with the proof system.

Almost all so far mentioned approaches check for unconditional information flow. There are only few approaches which study conditional information flow and in particular information flow contracts. One of the first contributions on conditional information flow was by Amtoft and Banerjee [2]. They developed a Hoare logic for compositional *intra*procedural analyses of conditional information flow. This approach was the basis for a contribution on software contracts for conditional information flow for SPARK Ada [4]. The latter approach works on a relatively simple while-language including method calls. The handling of arrays was added in a later contribution [3]. Object orientation is not supported. One advantage of our approach is that information flow and functional contracts can be combined easily. This results in arbitrary precision whereas [4] introduces fixed over-approximations.

Finally self-composition [6,8] is a popular approach to state non-interference and use off-the-shelf software verification systems to check for it, as we do. The approach has been applied to full-fledged programming languages like Java.

To the best of our knowledge there are only very few contributions aiming at an improvement of the efficiency of the self-composition approach. A very recent approach by Phan [18] uses bounded symbolic execution (symbolic execution without inductive invariants) and a formulation of (non-conditional) non-interference based on symbolic traces which is quite near in spirit to the one which we pioneered in [20] and which we reformulated for the *wp*-calculus in Theorem 1. Phan found that with this formulation it is sufficient to symbolically execute a program only once. Independently of [18], we found that the same holds if the *wp*-calculus or Dynamic Logic is used (Section 4). Therefore, our approach is not restricted to bounded programs. Additionally, we showed how the approach can be used to check for conditional non-interference and with more fine grained specifications. Barthe, Crespo and Kunz [5] build product programs to increase the level of automation in relational reasoning, which can also be used for information flow verification, but their focus is mainly on increasing the degree of automation and less on increasing efficiency.

Compositional / modular self-composition reasoning is also studied rarely: A contribution by Naumann [16] duplicates each variable, field, parameter and method body in the Java source code and uses standard JML method contracts to state non-interference with the help of the duplicates. The contracts are verified with the help of ESC/Java2. This approach has the drawback that there is no obvious translation of JML annotations from the non-duplicated source to the duplicated source: an object invariant `invariant (\sum Object o;; 1) < 10;` for instance might evaluate differently in the duplicated code than in the non-duplicated one. The paper mentions vague how modularity on the method level could be achieved, but thorough investigation is left for future work. Another contribution by Dufay, Felty and Matwin [10] introduces new JML-keywords which directly define relations between the program variables of two self-composed executions. In particular two keywords to distinguish the variables of the two runs are defined. The approach uses ghost code to store the return value and the values of parameters of the first run in order to use those values during the application of non-interference contracts in the second run. As the authors mention themselves, the approach is limited in case arrays are involved in method invocations. We do not see how even more complex data structures or equivalently complex heap manipulations can be tracked with ghost code. Hence, the proposed usage of ghost code seems to be a serious limitation of the approach. Resolving such limitations is mentioned as an aim of future work. Our approach on compositional reasoning overcomes such limitations: it does not use additional ghost code and is not limited by its usage.

7 Conclusions and Future Work

We presented two optimizations of self-composition style reasoning for weakest precondition calculi with explicit heap model which overcome two of the main efficiency issues with self-composition reasoning. Firstly we showed in Theorem 1 how self-composition can be rephrased such that it is sufficient to consider a program α only once in the weakest precondition calculation. The weakest precondition for α' can be extracted from the one of α by the renaming of program variables. Secondly we showed how the number of final states to be considered by a self-composed program can be reduced considerably by compositional information flow reasoning.

For the second optimization, compositional self-composition reasoning is essential. We presented an approach how weakest precondition calculi can be extended such that they can be used to construct fully modular and feasible self-composition proofs. The approach can be extended to information flow loop invariants. The main obstacle in the application of information flow loop invariants compared to flow contracts is that it has to be taken care that the self-composed programs execute the loop body equally often. An important feature of our approach is that (1) approximations are involved only at points where modular information flow reasoning is applied and (2) that our verification technique can get arbitrarily precise in those cases by the usage of preconditions and sufficiently strong functional contracts, if necessary. Further, our approach does not suffer from limitations of other approaches, like the ones of [10].

The presented approaches can easily be adopted to Dynamic Logic and other Hoare like logics. We implemented them (including information flow loop invariants) in the KeY-system on the basis of Java Dynamic Logic. Our implementation can handle the full subset of Java which can be handled by the non-extended KeY-system. This subset explicitly covers exceptions, object creation and static initialisation. It mainly does not cover concurrency, floating point arithmetic and generics. The implementation has been tested on several smaller case-studies. The tool itself as well as examples can be found on our web-side (`http://www.key-project.org/DeduSec/`).

References

1. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: Proceedings POPL, pp. 91–102. ACM (2006)
2. Amtoft, T., Banerjee, A.: Verification condition generation for conditional information flow. In: Proceedings of the 2007 ACM Workshop on Formal Methods in Security Engineering, FMSE 2007, pp. 2–11. ACM, New York (2007)
3. Amtoft, T., Hatcliff, J., Rodríguez, E.: Precise and automated contract-based reasoning for verification and certification of information flow properties of programs with arrays. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 43–63. Springer, Heidelberg (2010)
4. Amtoft, T., Hatcliff, J., Rodríguez, E., Robby, Hoag, J., Greve, D.A.: Specification and checking of software contracts for conditional information flow. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 229–245. Springer, Heidelberg (2008)
5. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011)
6. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: Proceedings of the 17th IEEE Workshop on Computer Security Foundations, CSFW 2004, pp. 100–115. IEEE CS, Washington (2004)

7. Bubel, R., Hähnle, R., Weiß, B.: Abstract interpretation of symbolic execution with explicit state updates. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 247–277. Springer, Heidelberg (2009)
8. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer, Heidelberg (2005)
9. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453–457 (1975)
10. Dufay, G., Felty, A., Matwin, S.: Privacy-sensitive information flow with JML. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 116–130. Springer, Heidelberg (2005)
11. Hammer, C., Krinke, J., Snelting, G.: Information flow control for Java based on path conditions in dependence graphs. In: IEEE International Symposium on Secure Software Engineering (ISSSE 2006), pp. 87–96. IEEE (March 2006)
12. Hoare, C.A.R.: Procedures and parameters: An axiomatic approach. In: Semantics of Algorithmic Languages. Lecture Notes in Mathematics, vol. 188, pp. 102–116. Springer (1971)
13. McCarthy, J.: Towards a mathematical science of computation. In: Information Processing, pp. 21–28 (1962)
14. Myers, A.C.: JFlow: Practical mostly-static information flow control. In: POPL, pp. 228–241 (1999)
15. Nanevski, A., Banerjee, A., Garg, D.: Verification of information flow and access control policies with dependent types. In: 2011 IEEE Symposium on Security and Privacy (SP), pp. 165–179 (May 2011)
16. Naumann, D.A.: From coupling relations to mated invariants for checking information flow. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 279–296. Springer, Heidelberg (2006)
17. Pan, J.: A theorem proving approach to analysis of secure information flow using data abstraction. Master's thesis, Dept. of Computer Science and Engineering, Chalmers U. of Technology (2005)
18. Phan, Q.-S.: Self-composition by symbolic execution. In: Imperial College Computing Student Workshop (ICCSW 2013), pp. 95–102, Schloss Dagstuhl (2013)
19. Ranise, S., Tinelli, C.: The SMT-LIB standard: Version 1.2. Tr, U. of Iowa (2006)
20. Scheben, C., Schmitt, P.H.: Verification of information flow properties of Java programs without approximations. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS 2011. LNCS, vol. 7421, pp. 232–249. Springer, Heidelberg (2012)
21. van Delft, B.: Abstraction, objects and information flow analysis. Master's thesis, Institute for Computing and Information Science, Radboud Uni Nijmegen (2011)