Towards Confidentiality-by-Construction

Ina Schaefer[1](✉), Tobias Runge[1], Alexander Knüppel[1], Loek Cleophas[2,3], Derrick Kourie[3,4], and Bruce W. Watson[3,4]

[1] Software Engineering, TU Braunschweig, Braunschweig, Germany
{i.schaefer,tobias.runge,a.knueppel}@tu-bs.de
[2] Software Engineering Technology Group, TU Eindhoven, Eindhoven, The Netherlands
[3] Department of Information Science, Stellenbosch University, Stellenbosch, South Africa
{loek,derrick,bruce}@fastar.org
[4] Centre for Artificial Intelligence Research, Stellenbosch, South Africa

Abstract. Guaranteeing that information processed in computing systems remains confidential is vital for many software applications. To this end, language-based security mechanisms enforce fine-grained access control policies for program variables to prevent secret information from leaking through unauthorized access. However, approaches for language-based security by information flow control mostly work *post-hoc*, classifying programs into whether they comply with information flow policies or not after the program has been constructed. Means for constructing programs that satisfy given information flow control policies are still missing. Following the correctness-by-construction approach, we propose a development method for specifying information flow policies first and constructing programs satisfying these policies subsequently. We replace functional pre- and postcondition specifications with confidentiality properties and define rules to derive new confidentiality specifications for each refining program construct. We discuss possible extensions including initial ideas for tool support. Applying correctness-by-construction techniques to confidentiality properties constitutes a first step towards security-by-construction.

1 Introduction

Modern software applications often process confidential information, such as personal information, credit card numbers, health records etc. It is important to enforce that this confidential information is not leaked to unauthorised access. Language-based security mechanisms [19] allow fine-grained control over the confidential information and its influence on program execution in order to prevent such unwanted leakage. Information flow control approaches [19,20] model confidentiality by defining *security policies* which determine how secret information in a program may be used for computation and influence program execution. In a very simple security policy, the set of program variables is classified into *high* and *low variables*. Information may flow within the classes and from low to

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11244, pp. 502–515, 2018.
https://doi.org/10.1007/978-3-030-03418-4_30

high variables, but not from high to low variables. This captures the intuition that high information, i.e., confidential information, may not influence low, i.e., public information, or that it may not be observable or deducible from public information. However, approaches for language-based security by information flow control mostly work *post-hoc*. They classify programs into the ones which comply to information flow policies and the ones that do not, but do not provide means to construct programs that satisfy given information flow control policies.

Correctness-by-construction (CbC) [14] in contrast aims at developing programs in a way such that they satisfy their correctness specification by their design and development methodology. Classical CbC as proposed by Dijkstra [10] and others [11,16] aims at developing functional programs that are correct-by-construction. For this purpose, a number of refinement rules are proposed that allow refining an abstract specification into a concrete program that satisfies the given specification. CbC programs are guaranteed to be correct in the same sense as a proof of a mathematical theorem is guaranteed to be correct. CbC-based development tends to minimise post-hoc quality assurance costs and thereby reduce time to market [21].

In this paper, we propose to apply correctness-by-construction techniques to guarantee confidentiality properties that are expressed by information flow policies over programs leading to an approach for *confidentiality-by-construction (C14bC)*[1] as a first step towards *security-by-construction (SbC)*. We replace functional pre-/postcondition specifications—as traditionally used for classical functional correctness—with confidentiality specifications, expressing which variables contain secrets. Then we provide rules for each possible program construct to refine a program by introducing such a construct, and we derive a new information flow specification for the program statement, in the spirit of classical CbC. In order to allow assigning secret values to public variables, we incorporate means to explicitly declassify information [17,22]. Furthermore, we discuss extensions of C14bC as well as potential for tool support.

The remainder of this paper is structured as follows: In Sect. 2, we provide the background on classical CbC and language-based information-flow control. In Sect. 3, we describe our approach to confidentiality-by-construction. In Sect. 4, we present initial ideas for tool support. Section 5 provides an overview of related work, and Sect. 6 concludes the paper with a discussion of extensions of the presented approach.

2 Background

In this section, we provide the necessary background on classical CbC and information flow control policies as a basis for the approach proposed in this paper. In order to simplify the discussion and focus on the main ideas of C14bC, we restrict programs to procedural programs that can be expressed in the guarded command language [10].

[1] The numeronym C14bC abbreviates confidentiality as C14, as there are 14 letters after the first C.

2.1 Classical Functional CbC

CbC [14] is a formal approach which is used to develop code incrementally. CbC starts with an abstract program and its specification which is a Hoare triple consisting of a precondition, an abstract statement, and a postcondition. The program between the pre- and postconditions is specified in the *Guarded Command Language* (GCL) as proposed by Dijkstra [10]. The Hoare triple T should be read as a total correctness assertion, i.e., an assertion that if T's precondition holds and its abstract statement executes, then the execution will terminate and its postcondition will hold. The triple can be stepwise evolved to a concrete program by using refinement rules. The rules each replace an abstract statement by more concrete ones, cf. Fig. 1. By only using correctness-preserving refinement steps that are accurately applied, we know that the concrete program obtained by refinement is correct by construction.

To refine the program, GCL uses five different rules—one for each of its statements—as shown in Fig. 1. The skip statement (1) does not alter the program. The assignment statement (2) refines an abstract statement S to an assignment $\mathtt{x} := \mathtt{E}$. This refinement can only be used if the precondition $\mathtt{P}$ implies the postcondition $\mathtt{Q}$ where $\mathtt{x}$ is replaced by $\mathtt{E}$. A composition statement (3) splits an abstract statement $\mathtt{S}$ into two statements $\mathtt{S1}$ and $\mathtt{S2}$ with an intermediate condition $\mathtt{M}$ between both statements. In the selection statement (4), for simplicity and similarity to, e.g., Java, we use an if-else-construct while the classical formulation of GCL uses a more complex switch-like statement. If the guard is evaluated to true, the first statement is executed, else the second one is. The repetition statement (5) is similar. As long as the guard is evaluated to true, the statement is executed repeatedly. The repetition statement requires an *invariant* and a *variant*. The invariant specifies the effect of the loop and is true before and after every loop iteration. The variant shows the termination of the loop. It is a term which decreases monotonically and is bounded from below; here we choose zero without loss of generality. In this discussion, we omit the refinement rules that allow strengthening of postconditions and weakening of preconditions.

$$
\begin{array}{lll}
\{\mathtt{P}\}\ \mathtt{S}\ \{\mathtt{Q}\} & \textit{can be refined to} & \\
\textit{Skip}: & \{\mathtt{P}\}\ \textit{skip}\ \{\mathtt{Q}\}\ \textit{iff}\ \mathtt{P}\ \textit{implies}\ \mathtt{Q} & (1)\\
\textit{Assignment}: & \{\mathtt{P}\}\ x := E\ \{\mathtt{Q}\}\ \textit{iff}\ \mathtt{P}\ \textit{implies}\ \mathtt{Q}[\mathtt{x} := \mathtt{E}] & (2)\\
\textit{Composition}: & \{\mathtt{P}\}\ \mathtt{S1}\,;\,\mathtt{S2}\ \{\mathtt{Q}\}\ \textit{iff there is}\ \mathtt{M}\ \textit{s.t.}\{\mathtt{P}\}\ \mathtt{S1}\ \{\mathtt{M}\}\ \textit{and}\ \{\mathtt{M}\}\ \mathtt{S2}\ \{\mathtt{Q}\} & (3)\\
\textit{Selection}: & \{\mathtt{P}\}\ \textbf{if}\ \mathtt{G}\ \textbf{then}\ \mathtt{S1}\ \mathtt{else}\ \mathtt{S2}\ \textbf{fi}\ \{Q\}\ \textit{iff} & (4)\\
 & \{\mathtt{P} \wedge \mathtt{G}\}\ \mathtt{S1}\ \{\mathtt{Q}\}\ \textit{and}\ \{\mathtt{P} \wedge \neg\mathtt{G}\}\ \mathtt{S2}\ \{\mathtt{Q}\} & \\
\textit{Repetition}: & \{\mathtt{P}\}\ \textbf{do}\ \mathtt{G} \rightarrow\ \mathtt{S}\ \textbf{od}\ \{\mathtt{Q}\}\ \textit{iff there is invariant}\ \mathtt{I}\ \textit{and variant}\ \mathtt{V}\ \textit{s.t.} & (5)\\
 & (\mathtt{P}\ \textit{implies}\ \mathtt{I})\ \textit{and}\ (\mathtt{I} \wedge \neg\mathtt{G}\ \textit{implies}\ \mathtt{Q})\ \textit{and}\ \{\mathtt{I} \wedge \mathtt{G}\}\ \mathtt{S}\ \{\mathtt{I}\} & \\
 & \textit{and}\ \{\mathtt{I} \wedge \mathtt{G} \wedge \mathtt{V} = \mathtt{V}_0\}\ \mathtt{S}\ \{\mathtt{I} \wedge 0 \leq \mathtt{V} < \mathtt{V}_0\} &
\end{array}
$$

Fig. 1. Refinement Rules in CbC [14]

To give an example of how CbC-refinements work, we consider the abstract triple $\{\mathtt{x} > 0\}\ \mathtt{S}\ \{\mathtt{x} > 1\}$. An assignment refinement rule associated with line (2) of Fig. 1 indicates how this triple can be refined to $\{\mathtt{x} > 0\}\ \mathtt{x} := \mathtt{x} + 1\ \{\mathtt{x} > 1\}$, delivering a program that ensures the postcondition if the precondition holds.

2.2 Information Flow Control

Information flow control [19,20] can be used to establish confidentiality of program data. A security policy defines security domains for data and determines how information may flow between those domains. In this paper, we restrict ourselves to a simple security policy by only considering two security domains, secret and public, where information may flow from public to secret, but not the other way around. The program variables are subdivided into high (secret) and low (public) variables. The high variables contain information which must not flow to low variables. Information in a program can flow in two ways: first, there can be direct information flow in an assignment, e.g., $l = h$ assigns the confidential value of h to a low variable l; second, there can be indirect information flow through conditional statements where secret information is used in the guard of the statement. For example, the statement *if* $h == 0 \rightarrow l := 0$ *else* $l := 1$ reveals information about the variable h. If l is zero, we know that h is also zero.

To discard programs which violate confidentiality as expressed by a security policy, a security type system can be introduced (cf. Fig. 2) according to [19]. The type system assigns every variable and expression a security type. $E : t$ means that expression E has security type t; in our case t can be either *high* or *low*. The type system uses a security context which is an environment variable tracking the current status of the program (high or low) to control implicit information flow. In a high context, no assignments to low variables may occur. The typing rules are depicted in Fig. 2. The rules define that an expression `exp` can always have a high type (1), but can only have a low type if no high variables occur in the expression (2). A skip is always typeable (3), and every expression can be assigned to a high variable (4). If we want to assign an expression to a low variable, the expression must be low (5). A composition of two statements keeps the same context (6). Rules (7) and (8) are used to ensure that if the guard has a high context, the statements are typable in a high context.

$$\vdash \mathtt{exp} : \mathtt{high} \qquad \frac{\mathtt{h} \notin \mathtt{Vars(exp)}}{\vdash \mathtt{exp} : \mathtt{low}} \qquad [\mathtt{ct}] \vdash \mathtt{skip} \qquad [\mathtt{ct}] \vdash \mathtt{h} = \mathtt{exp} \qquad (1\text{–}4)$$

$$\frac{\mathtt{exp} : \mathtt{low}}{[\mathtt{low}] \vdash \mathtt{l} = \mathtt{exp}} \qquad \frac{[\mathtt{ct}] \vdash \mathtt{S}_1 \quad [\mathtt{ct}] \vdash \mathtt{S}_2}{[\mathtt{ct}] \vdash \mathtt{S}_1; \mathtt{S}_2} \qquad \frac{\vdash \mathtt{exp} : \mathtt{ct} \quad [\mathtt{ct}] \vdash \mathtt{S}}{[\mathtt{ct}] \vdash \mathtt{while}\ \mathtt{exp}\ \mathtt{do}\ \mathtt{S}} \qquad (5\text{–}7)$$

$$\frac{\vdash \mathtt{exp} : \mathtt{ct} \quad [\mathtt{ct}] \vdash \mathtt{S}_1 \quad [\mathtt{ct}] \vdash \mathtt{S}_2}{[\mathtt{ct}] \vdash \mathtt{if}\ \mathtt{exp}\ \mathtt{then}\ \mathtt{S}_1\ \mathtt{else}\ \mathtt{S}_2} \qquad (8)$$

Fig. 2. Security Type System [19]

3 Confidentiality-by-Construction

In this section, we present the confidentiality-by-construction (C14bC) approach by providing a specification framework and refinement rules for the basic high-low security policy as described in Sect. 2.2. Essentially, the C14bC approach re-casts the typing rules of the security type system (cf. Fig. 2 in [19]) in a constructive fashion, thereby enabling the construction of programs that preserve the desired security policy *ab initio* rather than rejecting a program as non-compliant *ex post facto*.

3.1 C14bC Refinement Rules

In the following, we present refinement rules for all five statement types of the GCL to enforce the basic high-low information flow policy. In Fig. 3, we define the basic notation we use for defining the refinement rules.

A triple $\{\mathcal{H}^{pre}\}S\{\mathcal{H}^{post}\}[\eta]$ in C14bC defines the following: The set of high variables before execution of the statement `S` is captured in $\mathcal{H}^{pre}$, the set of high variables after execution of statement `S` is $\mathcal{H}^{post}$, the confidentiality level η classifies the confidentiality context for the execution of the statement `S`, which in our case can be either *high* or *low*. So a triple $\{\mathcal{H}^{pre}\}S\{\mathcal{H}^{post}\}[\eta]$ can be read as: if a program `S` is executed in a program state that satisfies $\mathcal{H}^{pre}$, i.e. where the variables in $\mathcal{H}^{pre}$ are classified as high, then the program will finish in a program state that satisfies $\mathcal{H}^{post}$, i.e., the high variables are contained in $\mathcal{H}^{post}$, while in confidentiality level η. Note that we are not concerned with termination here, so the triple can either refer to partial or total correctness.

The confidentiality level η is necessary to reason about implicit information flow in selection and repetition statements. If the if-condition or loop-guard contains high variables, the following program block is executed in a high-context, as its execution depends on the high variables contained in the if-condition or the loop-guard. Thus, at confidentiality level *high*, assignments to variables classified as *low* are forbidden, as this would implicitly reveal confidential information. For the considered high-low security policy, we additionally enforce the invariant $\mathcal{H}^{pre} \subseteq \mathcal{H}^{post}$ (i.e., variables can not be degraded to a lower confidentiality level). As the assignment statement creates explicit information flow, the assignment statement is the only statement where the set of high variables in the post-condition can be extended. We also assume the implicit frame condition that all program variables in *Vars* that are not classified as high are classified as low variables.

We now proceed by defining the C14bC refinement rules for the five possible GCL statements. Refining a triple in C14bC means that we refine an abstract statement S in a triple $\{\mathcal{H}^{pre}\}S\{\mathcal{H}^{post}\}[\eta]$ into a more concrete statement such that the more concrete statement satisfies the same specification w.r.t. the confidential variables in $\mathcal{H}^{pre}$ and $\mathcal{H}^{post}$. However, if the refinement is by a repetition or selection statement, the confidentiality level may change to reflect indirect information flow introduced. In our approach, once the confidentiality level has switched to high it will stay high for all subsequent refinements. The refinement

$Vars$	Set of program variables
$\mathcal{H}^{pre}, \mathcal{H}^{post} \subseteq Vars$	Sets of variables classified as *high*
S	Statement (from the GCL)
$x \in Vars$	Program variable
E, G	Expressions over the program variables in $Vars$
$Vars(E) \subseteq Vars$	Set of variables occurring in expression E
$\eta \in \{high, low\}$	Confidentiality level
$\{\mathcal{H}^{pre}\}S\{\mathcal{H}^{post}\}[\eta]$	C14bC triple

Fig. 3. Basic notions for C14bC

rules presented below are formulated in a way that the refinement property holds if the side conditions of the rules are satisfied.

The statement `skip` applies the identity function to the current state in a program. In compliance with our information flow policy, any statement regardless of the current confidentiality level can be refined to statement `skip` without changing the set of high variables.

Rule 1 (Skip)
$\{\mathcal{H}^{pre}\}\ \mathtt{S}\ \{\mathcal{H}^{post}\}[\eta]$ *is refinable to* $\{\mathcal{H}^{pre}\}\ \mathtt{skip}\ \{\mathcal{H}^{post}\}[\eta]$.

Assignments represent typical direct information flow where information flows directly from one location to another. Refining a statement `S` to the assignment `x := E` is possible in the following cases: (a) if the confidentiality level is *high* or the expression `E` comprises high variables, then the assigned variable `x` has to be a high variable after the execution of the assignment, i.e., $\mathtt{x} \in \mathcal{H}^{post}$; or (b) if the confidentiality level is *low* or the expression `E` comprises only low variables, then the set of high variables remains unchanged. For instance, $\{\mathtt{h}\}\ \mathtt{l} := \mathtt{h} * 2\ \{\mathtt{h}\}$ does not comply with our policy, since variable `l` must be *high* after the assignment (i.e., $\{\mathtt{h}\}\ \mathtt{l} := \mathtt{h} * 2\ \{\mathtt{h}, \mathtt{l}\}$).

Rule 2 (Assignment)
$\{\mathcal{H}^{pre}\}\ \mathtt{S}\ \{\mathcal{H}^{post}\}[\eta]$ *is refinable to* $\{\mathcal{H}^{pre}\}\ \mathtt{x} := \mathtt{E}\ \{\mathcal{H}^{post}\}[\eta]$ *iff*
$(\eta = high$ *or* $Vars(\mathtt{E}) \cap \mathcal{H}^{pre} \neq \emptyset)$ *implies* $\mathcal{H}^{post} = \mathcal{H}^{pre} \cup \{\mathtt{x}\}$.

In a composition statement, the two single statements are executed sequentially. Therefore, there has to be an intermediate condition denoting the high variables and the confidentiality level after the execution of the first statement which then serves as precondition specification for the second statement. The composition statement `S1;S2` itself does not change the current confidentiality level for its composed statements `S1` and `S2`.

Rule 3 (Composition)
$\{\mathcal{H}^{pre}\}\ \mathtt{S}\ \{\mathcal{H}^{pre}\}[\eta]$ *is refinable to* $\{\mathcal{H}^{pre}\}\ \mathtt{S1;S2}\ \{\mathcal{H}^{post}\}[\eta]$
if there exists $\mathcal{H}' \subseteq Vars$ *such that*
$\{\mathcal{H}^{\mathtt{pre}}\}\ \mathtt{S1}\ \{\mathcal{H}'\}[\eta]$ *and* $\{\mathcal{H}'\}\ \mathtt{S2}\ \{\mathcal{H}^{\mathtt{post}}\}[\eta]$ *and* $\mathcal{H}^{pre} \subseteq \mathcal{H}^{'} \subseteq \mathcal{H}^{post}$

The selection statement may give rise to implicit information flow if the if-guard contains high variables such that the confidentiality level has to be adapted to prevent insecure implicit information flow. Hence, the selection statement determines the confidentiality level of its sub-statements S1 and S2. We distinguish two cases. If the confidentiality level of the statement to be refined is *high* or the guard of the selection statement contains high variables, we have to set the confidentiality level of the sub-statements to *high*. In the other case, the confidentiality level of the sub-statements is *low*. To give an example, the selection statement $\{\mathtt{h}\}\,\mathbf{if}\ \mathtt{h} == \mathtt{1} \rightarrow \mathtt{l} := \mathtt{1}\ \mathbf{else}\ \mathtt{skip}\ \mathbf{fi}\,\{\mathtt{h}\}$ does not comply with our security policy because the guard comprises a high variable, and therefore the confidentiality level is set to high for the sub-statement, such that the assignment to a low variable is forbidden or the variable l has to become high as well.

Rule 4 (Selection)
$\{\mathcal{H}^{pre}\}\,\mathtt{S}\,\{\mathcal{H}^{pre}\}[\eta]$ *is refinable to* $\{\mathcal{H}^{\mathtt{pre}}\}\,\mathbf{if}\ \mathtt{G} \rightarrow \mathtt{S1}\ \mathbf{else}\ \mathtt{S2}\ \mathbf{fi}\,\{\mathcal{H}^{\mathtt{post}}\}[\eta]$ *if*

(i) $(\eta = \mathtt{high}\ \textit{or}\ \mathit{Vars}(G) \cap \mathcal{H}^{pre} \neq \emptyset)$
implies $\{\mathcal{H}^{\mathtt{pre}}\}\ \mathtt{S1}\ \{\mathcal{H}^{\mathtt{post}}\}[\mathit{high}] \wedge \{\mathcal{H}^{\mathtt{pre}}\}\ \mathtt{S2}\ \{\mathcal{H}^{\mathtt{post}}\}[\mathit{high}]$
(ii) $(\eta = \mathtt{low}\ \textit{and}\ \mathit{Vars}(G) \cap \mathcal{H}^{pre} = \emptyset)$
implies $\{\mathcal{H}^{\mathtt{pre}}\}\ \mathtt{S1}\ \{\mathcal{H}^{\mathtt{post}}\}[\mathit{low}] \wedge \{\mathcal{H}^{\mathtt{pre}}\}\ \mathtt{S2}\ \{\mathcal{H}^{\mathtt{post}}\}[\mathit{low}]$

The considerations for the confidentiality level of the repetition statement are similar as for the selection statement. If the confidentiality level of the statement to be refined is *high* or the loop-guard comprises high variables, the confidentiality level of the loop body is set to *high*. If the confidentiality level of the refined statement is *low* and the guard excludes high variables, the confidentiality level of the loop body is set to *low*. In this way, we can prevent insecure implicit information flow for loops (the same as for selection statements). As an example, consider the following repetition statement: $\{\mathtt{h}\}\,\mathbf{do}\ \mathtt{h} > \mathtt{0} \rightarrow \mathtt{l} := \mathtt{l} + \mathtt{1}; \mathtt{h} := \mathtt{h} - \mathtt{1}\ \mathbf{od}\,\{\mathtt{h}\}$. From the value of the low variable l, an attacker can infer the value of the high variable h, therefore the confidentiality level is *high* and the assignment $\mathtt{l} := \mathtt{l} + \mathtt{1}$ is either forbidden or the variable l has to be included in the high variables.

Rule 5 (Repetition)
$\{\mathcal{H}^{pre}\}\,\mathtt{S}\,\{\mathcal{H}^{pre}\}[\eta]$ *is refinable to* $\{\mathcal{H}^{\mathtt{pre}}\}\,\mathbf{do}\ \mathtt{G} \rightarrow \mathtt{S1}\ \mathbf{od}\,\{\mathcal{H}^{\mathtt{post}}\}[\eta]$ *if*

(i) $(\eta = \mathit{high}\ \textit{or}\ \mathit{Vars}(G) \cap \mathcal{H}^{pre} \neq \emptyset)$ *implies* $\{\mathcal{H}^{\mathtt{pre}}\}\ \mathtt{S1}\ \{\mathcal{H}^{\mathtt{post}}\}[\mathit{high}]$.
(ii) $(\eta = \mathit{low}\ \textit{and}\ \mathit{Vars}(G) \cap \mathcal{H}^{pre} = \emptyset)$ *implies* $\{\mathcal{H}^{\mathtt{pre}}\}\ \mathtt{S1}\ \{\mathcal{H}^{\mathtt{post}}\}[\mathit{low}]$.

3.2 Declassification

According to the high-low security policy considered in this paper, we are not allowed to assign an expression comprising high variables to a low variable. However, in order to develop meaningful applications, it may sometimes be necessary to allow some information flow from high values to low values. This, however, needs to be made explicit and might need some kind of declassification [17,22] such that the initial secret is not directly deducible. In order to allow declassification in the above sense, we extend our refinement rules with a specific rule for declassification. We introduce a function to declassify a high expression, so that the assignment of the declassified expression to a low variable is valid. The declassification function on an expression should be used with care and not lead to a leak in confidentiality. The concrete declassification operation performed depends on the application context. In general, it is assumed that declassification removes some secret information such that the secret is not (easily) deducible from the declassified data. For example, if a password is encrypted (and thus declassified), the encrypted value can be assigned to a low variable. By application of a declassification operator to the expression used in an if-condition or a loop-guard before executing the selection or repetition statement, respectively, also implicit information flow can be avoided.

The declassification operation modifies the assignment rule of the C14bC framework. In the case of an assignment, if the assigned expression comprises high variables or the confidentiality level of the specification is high, the assigned variable has to be contained in the set of high variables of the post-condition $((\eta = \mathit{high}$ or $\mathtt{vars}(E) \cap \mathcal{H}^{pre} \neq \emptyset)$ implies $x \in \mathcal{H}^{post})$. With declassification, we alter the condition to $(\eta = \mathit{high}$ or $(\neg\mathtt{isDecl}(\mathtt{E})$ and $\mathit{Vars}(E) \cap \mathcal{H}^{pre} \neq \emptyset))$ implies $x \in \mathcal{H}^{post}$. The predicate $\mathtt{isDecl}(\mathtt{E})$ checks if the expression is declassified. Only if we do not declassify the expression and the expression comprises high variables, the assigned variable `x` has to be a high variable in the post-condition as well. These considerations give rise to a modified assignment rule for declassification. This refinement rule only makes sense if it is applied at a low confidentiality level, if the assigned variable is a low variable and if the declassified expression indeed contains high variables.

Rule 6 (Declassification Assignment)
$\{\mathcal{H}^{pre}\}\ \mathtt{S}\,\{\mathcal{H}^{post}\}[\mathtt{low}]$ *is refinable to* $\{\mathcal{H}^{pre}\}\ \mathtt{x} = \mathtt{declassify(E)}\ \{\mathcal{H}^{post}\}[\mathtt{low}]$ *iff* $\mathtt{x} \notin \mathcal{H}^{pre}$ *and* $\mathit{Vars}(\mathtt{E}) \cap \mathcal{H}^{pre} \neq \emptyset)$*.*

3.3 Example

In Listing 1, we show an example for C14bC. The program checks if the user wants to pay, and if this is the case, a valid credit card number is required. In the end, the masked credit card number is passed to an output variable.

To construct the program, we start with an abstract program $\{\mathcal{H}^{\mathtt{pre}}\}\ \mathtt{S}\ \{\mathcal{H}^{\mathtt{post}}\}[\mathit{low}]$ where both sets $(\mathcal{H}^{pre}, \mathcal{H}^{post})$ are empty. The refinement steps are shown in Fig. 4. Note that we only add program variables to the set of high variables in the postcondition if that is required by the refinement rules in order to keep track of where information flow actually occurs. In this sense, we are treating the refinement rules rather like transformation rules. Of course, the variables added to the set of high variables in the postcondition need to be added to the postconditions up the refinement hierarchy as well in order to establish a proper refinement relationship. A way to allow expressing information flow policies without having to refer to concrete variables in the program during refinement would be to introduce ghost variables [2] for the set of high variables in the pre- and postcondition whose value is an symbolic expression that can be dynamically updated. We leave this to future work.

```
 1  boolean low paymentAction := true;
 2  if (paymentAction) {
 3    int high creditCard := getNumber();
 4    while (!valid(creditCard)) {
 5      creditCard = getNumber();
 6    }
 7    String low output := declassify(mask(creditCard);
 8  } else {
 9    skip
10  }
```

Listing 1. C14bC example for credit card payment

The first statement, on line 1 of the listing, is an assignment. The assignment is introduced by using the composition and the assignment rule (ref. 1 and 2). The composition splits the program into the first statement and the rest. In line 1, a constant is assigned to a low variable which stays low. This is possible without problems at any confidentiality level. By introducing the selection statement of lines 2–10 (ref. 3), the confidentiality level stays low because the guard does not comprise a high variable and the level was low before. The skip statement of the else branch in line 9 is introduced by refinement 4. The sets of variables and the confidentiality level are unaffected. The assignment in line 3 is introduced by refinement 5 and 6. A composition statement is needed to split the program. In line 3, we assign a value to a variable `creditCard`. We assume that `getNumber` is a high expression. We have to ensure that `creditCard` is in the post set of high variables. This propagates up to the composition statement. The variable `creditCard` (`cC` in the high variable sets in Listing 1) is added: $\{\}$ `TS1` $\{$`cC`$\}[low] \wedge \{$`cC`$\}$ `TS2` $\{$`cC`$\}[low]$. The repetition statement of lines 4–6 is introduced by using refinement 7 and 8. A composition statement is needed, so the assignment in line 7 can be created. The repetition statement changes the confidentiality level. We have a guard which comprises a high variable, so the level is raised to high for all sub-statements. For the assignment in line 5 (ref. 9), the variable `creditCard` has to be in $\mathcal{H}^{post}$ which is the case. The assignment in line 7 (ref. 10) is inside the scope of the selection statement, but outside the scope of the repetition statement. It has the low confidentiality level of the selection statement and the intermediate composition statements. Here, we assign a high to a low variable. This violates our assignment Rule 2. With declassification in Rule 6, we allow this assignment since the credit card number is masked.

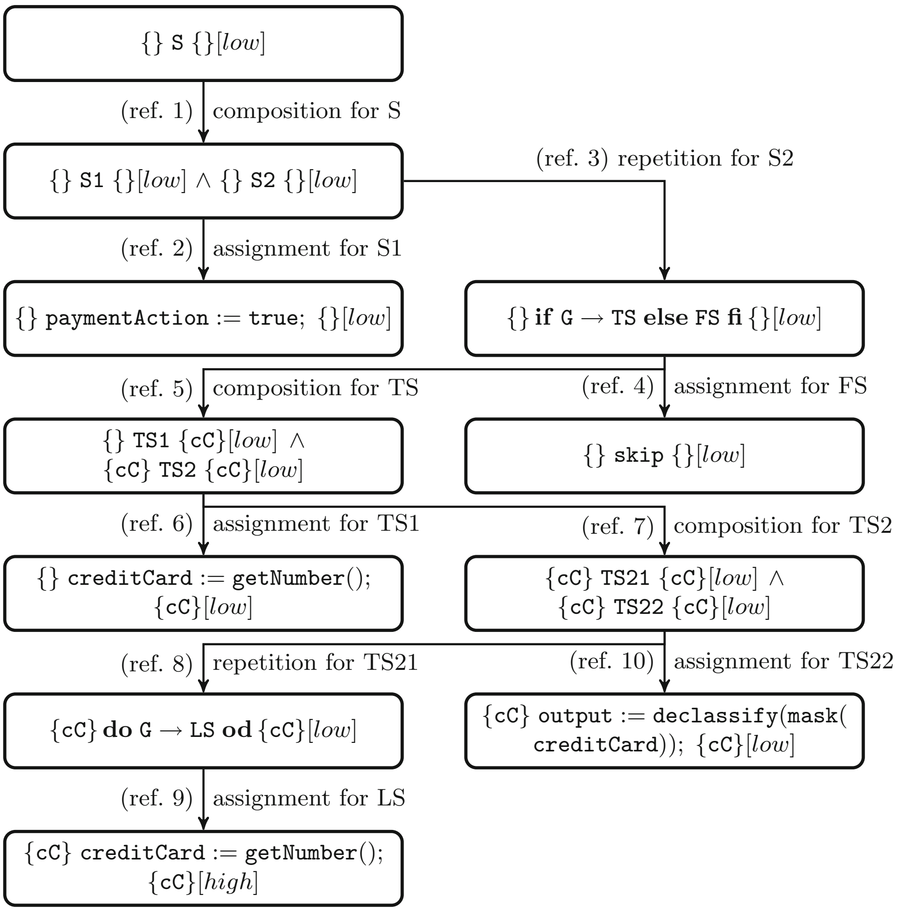
{} S {}[low]
(ref. 1) composition for S
{} S1 {}[low] ∧ {} S2 {}[low]
(ref. 3) repetition for S2
(ref. 2) assignment for S1
{} paymentAction := true; {}[low]
{} if G → TS else FS fi {}[low]
(ref. 5) composition for TS
(ref. 4) assignment for FS
{} TS1 {cC}[low] ∧ {cC} TS2 {cC}[low]
{} skip {}[low]
(ref. 6) assignment for TS1
(ref. 7) composition for TS2
{} creditCard := getNumber(); {cC}[low]
{cC} TS21 {cC}[low] ∧ {cC} TS22 {cC}[low]
(ref. 8) repetition for TS21
(ref. 10) assignment for TS22
{cC} do G → LS od {cC}[low]
{cC} output := declassify(mask(creditCard)); {cC}[low]
(ref. 9) assignment for LS
{cC} creditCard := getNumber(); {cC}[high]

Fig. 4. Refinement steps for the credit card payment example

4 Tool Support

In order to make C14bC applicable to larger programs, we need to provide tool support. Currently, we are developing tool support for classical functional CbC by providing an IDE-like development environment for deriving programs in a CbC-based fashion in a textual and graphical manner[2].

This tool support can be easily extended with the above ideas to cover C14bC. From an analysis and verification point of view, C14bC specifications are easier to check and analyse than functional CbC specifications. For functional CbC specifications, we need a way to verify functional Hoare triples over assignments and establish variants and invariants over repetition statements. For this task, we can use a program verification tool, such as the KeY prover [2]. For C14bC

[2] https://github.com/TUBS-ISF/CorC.

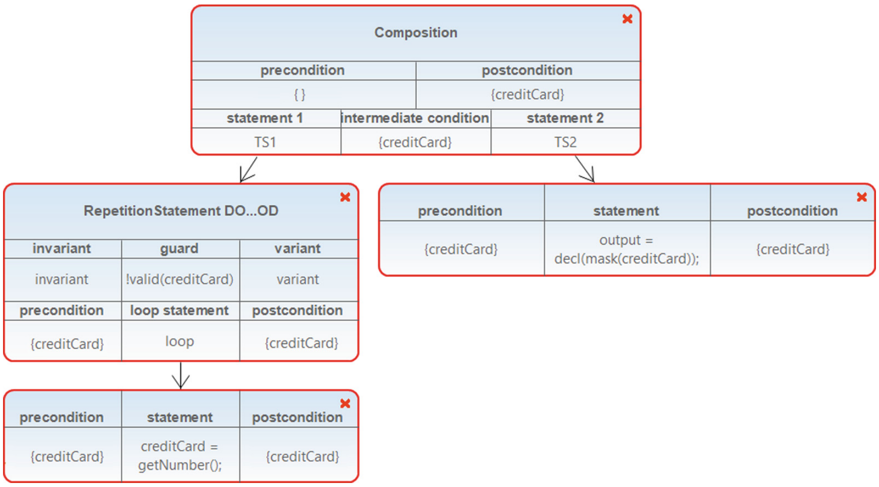

Fig. 5. Graphical representation in the C14bC editor

specifications, as presented in this paper, it is sufficient to be able to statically analyse the variables contained in expressions and to reason about their usage in the program while maintaining their classification as high and low program variables. Of course, if we apply a richer programming language with side effects, we need more sophisticated static program analysis techniques to reason about the respective information flow.

In Fig. 5, we show what the editor could look like. The lines 4–7 from the example in Listing 1 are shown. We have a composition statement at the top, containing the high variable `creditCard` in the intermediate- and postcondition. The statement `TS1` is refined to a repetition statement, and the other statement is refined to an assignment. The assignment contains the assignment `output` := `decl`(`mask`(`creditCard`)). In the repetition statement, invariant, guard and variant (needed for functional CbC) can be specified. The pre- and postcondition contain again the high variable `creditCard`. The inner loop statement is refined to an assignment.

5 Related Work

The CbC approach to software construction was pioneered by Dijkstra, Hoare and others and based on weakest precondition semantics [10,11,16]. Kourie and Watson [14] propose a light-weight version of this approach. In [21], we have proposed a combination of CbC and post-hoc verification in order to obtain the best of both worlds. The approach should not be confused with other concepts that carry the same name, such as the correctness-by-construction (CbyC) promoted by Hall and Chapman [13]. Their CbyC is a software development process where formal modeling techniques and analyses are used for different development phases, in order to detect and remove any defects that do occur as early as possible after introduction [8]. Another approach to correctness-by-construction is the Event-B framework [1] where automata-based system specifications are refined by provably correct transformation steps until an implementable program is obtained [15].

Language-based information flow security is a broad field in the literature; for a survey or earlier work consult [19]. The main approaches essentially rely on static or dynamic program analysis [18], such as taint analysis [7], or security type systems [20] etc. Some approaches combine information flow control with logic-based or Hoare-style program logics. An early effort was made by Andrews and Reitman [6], who proposed a compile-time certification technique for information policies with multiple security levels based on a Hoare-style semantics. Their work also covers programs beyond the sequential ones we cover, i.e., involving parallelism and semaphores. However, their approach is a post-hoc one, unlike our by-construction approach. Amtoft and Banerjee [4] formulated compositional intraprocedural analyses of conditional information flow, which served as the basis for a formulation of Hoare-style contracts for conditional information flow for SPARK Ada [5].

The first paper to reformulated information flow properties as a deductive verification problem in a program logic was [9]. Hähnle et al. [12] show how a type system ensuring confidentiality can be embedded into a form of dynamic logic. The setting is once again one focused on post-hoc verification, with an eye to using the KeY theorem prover. This paper, to the best of our knowledge, is the first to propose constructive CbC style reasoning for information flow properties by means of a refinement-based approach such that by a sequence of small, incremental refinement steps, a program is obtained that preserves security policies by construction.

6 Conclusion and Future Work

The presented approach on C14bC can be seen as a first step in the direction of security-by-construction. In this paper, we only focus on confidentiality and base our considerations on a simple programming language. There are several directions to extend this work:

- Besides evaluation of the practical applicability and scalability on larger scale case examples, we also have to formally verify the correctness and completeness of our C14bC construction approach. This includes formulating the ideas in a formal refinement framework and proving the soundness against the corresponding type system-based approaches. Furthermore, we should investigate the benefits of C14bC when combined with post-hoc verification and analysis approaches, similar to the work presented in [21].
- We can extend the programming language constructs that are considered for deriving a program. In order to achieve modularity of our approach, we can integrate a refinement rule that introduces a method call and thus allow modular refinement of the program into several methods. However, in that, we

have to be careful about side effects and appropriate frame conditions. Furthermore, we can lift the presented approach to object-oriented programs, following information flow control approaches for object-oriented languages [3].
- By considering security policies in information flow as specifications for integrity (rather than confidentiality), we can also provide an approach for integrity- or trust-by-construction. To this end, we label program variables as trusted and untrusted (in contrast to high and low) and only allow information to flow from trusted to untrusted, but not vice versa, as this would allow untrusted information to influence trusted information. In essence, this is the same set-up as the high-/low security policy considered in this paper, i.e., the presented framework directly lends itself to trust-by-construction.
- Additionally, we can make the presented C14bC approach generic with respect to the information flow policy. In the paper, we have only focused on a very simply high/low security policy with the possibility to declassify data. However, for practical applications, it might be necessary to introduce several security layers and more fine-grained security policies. To this end, the refinement rules presented in this work need to be generalised with respect to the security policies they can operate on.
- In this paper, C14bC is considered in isolation. We focused on confidentiality specifications only, for ease of presentation. However, we can of course combine functional CbC with C14bC in order to derive a functionally correct program that also complies to the desired security policy. Technically, this is a combination of the classical functional pre-/post-conditions and refinement rules with the C14bC pre/post-conditions and refinement rules laid out in this paper.

Acknowledgements. The authors would like to thank the anonymous reviewers for valuable comments and suggestions for improvements and future work.

References

1. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, New York (2010)
2. Ahrendt, W., Beckert, B., Hähnle, R., Schmitt, P.H., Ulbric, M. (eds.): Deductive Software Verification The KeY Book From Theory to Practice. LNCS, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
3. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: POPL, pp. 91–102 (2006)
4. Amtoft, T., Banerjee, A.: Information flow analysis in logical form. In: SAS, pp. 100–115 (2004)
5. 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. (eds.): FM 2008. LNCS, vol. 5014, pp. 229–245. Springer, Boston (2008)
6. Andrews, G.R., Reitman, R.P.: An axiomatic approach to information flow in programs. ACM Trans. Program. Lang. Syst. **2**(1), 56–76 (1980)

7. Arzt, S., et al.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: PLDI, pp. 259–269 (2014)
8. Chapman, R.: Correctness by construction: a manifesto for high integrity software. In: Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software, SCS 2005, vol. 55, pp. 43–46 (2006)
9. 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). https://doi.org/10.1007/978-3-540-32004-3_20
10. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Englewood Cliffs (1976)
11. Gries, D.: The Science of Programming. Springer, New York (1987). https://doi.org/10.1007/978-1-4612-5983-1
12. Hähnle, R., Pan, J., Rümmer, P., Walter, D.: Integration of a security type system into a program logic. Theor. Comput. Sci. **402**(2–3), 172–189 (2008)
13. Hall, A., Chapman, R.: Correctness by construction: developing a commercial secure system. IEEE Softw. **19**(1), 18–25 (2002)
14. Kourie, D.G., Watson, B.W.: The Correctness-By-Construction Approach to Programming. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27919-5
15. Méry, D., Monahan, R.: Transforming event B models into verified C# implementations. In: First International Workshop on Verification and Program Transformation, VPT 2013, Saint Petersburg, Russia, pp. 57–73, 12–13 July 2013 (2013)
16. Morgan, C.: Programming from Specifications, 2nd edn. Prentice Hall, New York (1994)
17. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. **9**(4), 410–442 (2000)
18. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-662-03811-6
19. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003)
20. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2/3), 167–188 (1996)
21. Watson, B.W., Kourie, D.G., Schaefer, I., Cleophas, L.: Correctness-by-construction and post-hoc verification: a marriage of convenience? In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 730–748. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_52
22. Zdancewic, S., Myers, A.C.: Robust declassification. In: 14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11–13 June 2001, pp. 15–23, Cape Breton, Nova Scotia, Canada (2001)