Precise and Automated Contract-based Reasoning for Verification and Certification of Information Flow Properties of Programs with Arrays

Torben Amtoft, John Hatcliff, and Edwin Rodríguez

SAnToS Laboratory
Kansas State University
{tamtoft,hatcliff,edwin}@cis.k-state.edu

Abstract. Embedded information assurance applications that are critical to national and international infrastructures, must often adhere to certification regimes that require information flow properties to be specified and verified. SPARK, a subset of Ada for engineering safety critical systems, is being used to develop multiple certified information assurance systems. While SPARK provides information flow annotations and associated automated checking mechanisms, industrial experience has revealed that these annotations are not precise enough to specify many desired information flow policies. One key problem is that arrays are treated as indivisible entities – flows that involve only particular locations of an array have to be abstracted into flows on the whole array. This has substantial practical impact since SPARK does not allow dynamic allocation of memory, and hence makes heavy use of arrays to implement complex data structures.
In this paper, we present a Hoare logic for information flow that enables precise compositional specification of information flow in programs with arrays, and automated deduction algorithms for checking and inferring contracts in an enhanced SPARK information flow contract language. We demonstrate the expressiveness of the enhanced contracts and effectiveness of the automated verification algorithm on realistic embedded applications.

1 Introduction

Secure information flow [14] has become a crucial property in the current landscape of security certification for highly sensitive systems. The MILS (Multiple Independent Levels of Security) architecture, an example of an approach supported and advocated by various defense and security agencies, uses information flow as a core concept and advocates a tight control of information flow channels, mostly by means of physical and/or logical separation [11].

Much effort has been spent on developing techniques to analyze information flow in computer programs [28] – leading to several languages such as Myers' JFlow [22], and FlowCaml [29], that include language-level specifications (often in the form of "security types") and automated checking mechanisms that establish that a program's information flow conforms to supplied specifications. SPARK, a safety-critical subset of Ada, is being used by various organizations, including Rockwell Collins [24] and the US National Security Agency (NSA) [8], to engineer information assurance systems including cryptographic controllers, network guards, and key management systems. SPARK provides automatically checked procedure annotations that specify information flows between procedure inputs and outputs. In the certification process, these annotations play a key role justifying conformance to information flow requirements and separation policies relevant to MILS development. However, experience in these

industrial/government development efforts has shown that the annotations of SPARK, as well as those of other language-based information flow specification frameworks, are not precise enough to specify many important information flow policies. In such situations, policy adherence arguments are often reduced to informal claims substantiated by manual inspections that are time-consuming, tedious, and error-prone.

Inability to specify desired information flow policies in realistic applications, using existing language annotation frameworks, often stems from two issues: a) Coarse treatment of information channels, where information flowing between two variables is regarded as creating a channel without regard to the conditions under which that channel is active; and b) Coarse treatment of structured data, such as arrays, where information can only be specified as flowing into/from an array as a whole, instead of its constituent cells. Our previous work [5] gives one approach for addressing the first issue by providing inference and checking of conditional information flow contracts, allowing the specification of conditions that determine when the information flow channels are active, using a precondition generation algorithm and an extension to the logic previously developed by Amtoft and Banerjee [2, 3]. This paper builds on this earlier work to address the second problem: precise information flow analysis for arrays.

Support for precise reasoning about information flow in arrays is especially important in resource-bounded embedded high-assurance security applications, because storage for data structures such as buffers, rule tables, *etc*., must often be statically allocated and accessed via offset calculations. [1].

Motivated by the need to guarantee analyzability and conformance to resource bounds, SPARK does not include pointers and heap-based data. Thus, complex data structures must be implemented in terms of arrays whose size is fixed at compile time.

Although the need for a more fine grained information flow analysis for arrays might not seem so apparent, it turns out that the absence of such functionality may seriously hinder industrial certification efforts. Our interactions with industry partners, such as our colleagues at the Advanced Technology Center at Rockwell Collins, have revealed that when working with safety-critical language subsets, such as SPARK/Ada [7], which restrict dynamic allocation of memory, arrays play an important role in creating efficient memory landscapes within applications. Under these circumstances, during some of their certification projects (such as the certification of the AAMP7 separation kernel) they have found the need for fine grained reasoning of information flow on arrays, representing regions within the arrays as abstract functions of the array indices.

This paper presents a novel approach for automated contract-based reasoning about information flow within arrays – targeted to applications that require high assurance and certification. The specific contributions of this work are as follows:

- A language-independent Hoare-like logic for secure information flow that can be used to reason precisely about information flow between array components,
- An extension of the SPARK information flow contract language (with semantics provided by the Hoare logic) that supports specification of information flow policies about array components,
- An algorithm for automatically checking and inferring enhanced SPARK contracts against code,

[1] Domain requirements in these applications are similar to those in avionics. In fact, certification to the high-level evaluation levels (levels 6/7) of the Common Criteria security certification framework is more stringent than the highest assurance requirements of the avionics certification standard DO-178B (which is roughly equivalent to Level 5 of the Common Criteria)

– A novel approach for computing universally-quantified information flow properties for arrays,
– The study of an information assurance application that shows the importance of precise information flow analysis for arrays, based on the MILS Message Router specification given in [26], and
– An empirical evaluation of the performance and verification effectiveness of our approach against a collection of SPARK programs.

The logical/algorithmic foundations of our work are language independent, and could be applied to array-based data structures in other languages (see Section 6 for other contexts/languages to which the information flow logic used here has been applied). However, our implementation in the context of SPARK is especially relevant because SPARK is the only commercially supported framework that we know of for specifying and checking information flows. Indeed, this work has been inspired by challenge problems provided by our industrial collaborators at Rockwell Collins who are using SPARK on multiple information assurance development projects.

2 Information Flow Contracts in SPARK

SPARK is a safety critical subset of Ada [15] developed and supported by Praxis High Integrity Systems that provides (a) an annotation language for writing both functional and information flow software contracts, and (b) automated static analyses and semi-automated proof assistants for proving absence of run-time exceptions, and conformance of code to contracts. SPARK has been used to build a number of high-assurance systems; Praxis HIS is currently using it to implement the next generation of the UK air traffic control system.

```
procedure SimpleScalars(A : in Integer; B : in out Integer)
--# global in C, out Result;
--# derives B from *, A, C
--#      and Result from A;
is
  Temp : Integer;
begin
  Temp := A - 1;
  if C > 5 then
     B := B + Temp;
  else
     B := 5;
  end if;
  Result := Temp + 2;
end SimpleScalars;
```

Fig. 1. Illustrating SPARK Information Flow Contracts

Figure 1 shows a simple SPARK/Ada procedure with SPARK annotations. SPARK demands that all procedures explicitly declare all the global variables that they read and/or write. This is done in a `global` annotation, which is basically comprised of a list of variables prefixed by a modifier that indicates whether the variables are read (`in`), written (`out`), or both (`in out`). Parameters to the procedures must also be annotated with `in` and `out` modifiers, depending on their mode of utilization within the procedure, *i.e.*, whether they are read, write, or read-write variables. In addition, all `out` variables (*i.e.*all variables that are modified by the procedures) must declare a `derives` clause. A

```
procedure SinglePositionAssign
(Flag : in Int; Value : in Types.Flagvalue)
--# global in out Flags;
--# derives Flags from *, Flag, Value;
is
begin
  Flags(Flag) := Value;
end SinglePositionAssign;

procedure Scrub_Cache (cache : in out Sensor_Cache_Type)
--# derives cache from *;
is
 begin
    for I in Sensor_Ids loop
       cache(I) := 0;
    end loop;
end Scrub_Cache;

procedure Copy_Keys (inkeys : in Key_Table_Type,
                     outkeys : in out Key_Table_Type)
--# derives outkeys from *, inkeys;
is
 begin
    for I in Key_Table_Entries loop
       outkeys(I) := inkeys(I);
    end loop;
end Scrub_Cache;
```

(a)

```
procedure SinglePositionAssign
(Flag : in Int; Value : in Types.Flagvalue)
--# global out Flags(Flag);
--# derives Flags(Flag) from Value;
is
begin
  Flags(Flag) := Value;
end SinglePositionAssign;

procedure Scrub_Cache (cache : out Sensor_Cache_Type)
--# derives for all J in Sensor_Ids => (cache(J) from {});
is
 begin
    for I in Sensor_Ids loop
       cache(I) := 0;
    end loop;
end Scrub_Cache;

procedure Copy_Keys (inkeys : in Key_Table_Type,
                     outkeys : out Key_Table_Type)
--# derives for all J in Key_Table_Entries
--#                 => (outkeys(J) from inkeys(J));
is
 begin
    for I in Key_Table_Entries loop
       outkeys(I) := inkeys(I);
    end loop;
end Copy_Keys;
```

(b)

Fig. 2. (a) Limitations of SPARK annotations and (b) proposed enhancements.

`derives` clause what other variables were used to derive the final value of a particular variable (including potentially itself). In Fig. 1, the `derives` clause states that `B` is derived from itself (`*`), `A` and `C`. SPARK also provides other annotation mechanism to specify pre- and post-conditions, but for this discussion we will focus in those directly related to information flow analysis (which also happen to be the ones made mandatory by SPARK).

Figure 2 (a) shows a collection of very simple procedures with SPARK information flow annotations. SPARK demands that all procedures explicitly declare all the global variables that they read and/or write. As illustrated in the `SinglePositionAssign` procedure, this is done via a `global` annotation that lists global variables with each variable prefixed by a modifier that indicates the *mode* of the variable, *i.e.*, whether the variable is read (`in`), written (`out`), or both (`in out`). Parameters to the procedures must also be annotated with `in` and `out` modifiers indicating their mode. In addition, all `out` variables (*i.e.*all variables that are modified by the procedures) must declare a `derives` clause. A `derives` clause for `out` variable `X` specifies the `in` parameters/globals whose initial values were used to derive the final value of variable `X`. In `SinglePositionAssign`, the `derives` clause states that the out variable `Flags` is derived from itself (`*`), `Flag` and `Value`. Stated in terms of conventional notions of program dependence/slicing, the dependence notion captured by a derives clauses is comprised of both data and control dependence, and the `in` variables on the right-hand side of the derives clause are those that would appear in expressions occurring in a conventional static backwards slice on the final definition(s) of the `out` variable. SPARK also provides other annotation mechanisms to specify pre- and postconditions, but for this discussion we will focus on those directly related to information flow analysis.

As the reader may already have noticed, the `derives` clause provides a natural framework for information flow specifications: for each variable modified in a procedure, the `derives` clause states all possible sources of information. The `in`/`out` mode declarations and derives clauses form an information flow *contract* for a procedure. The `out` mode declarations specify a *frame condition e.g.*, analogous, to JML's [20] `modifies` clause in that they specify the pieces of the program state can be modified by the procedure.

One can view these annotations as providing a *contracts* for information flow. , where the `out` variables provide the postcondition (indicating which variables are modified by the procedure), and the corresponding list of `in` variables associated to each `out` variable in the `derives` clause gives the precondition (indicating which variables are used in the computation of the output variables). Following the same approach used in conventional contract-based reasoning, these contracts enable a compositional information flow specification/verification framework.

While the semantics of existing SPARK contracts, as presented in Figure 2 (a), can be captured using conventional slicing and data/control-dependence, we have developed a more powerful and flexible theory of information flow contracts backed by a Hoare-style logic, and a precondition generation algorithm [5] that is able to provide additional analysis precision and contract expressiveness not found in conventional static-analysis-based approaches. Moreover, in the context of embedded applications and languages like SPARK, which eschew complicated language features, we have been able to achieve this power while maintaining a very high degree of automation and low computational costs. In our previous work [5], we demonstrated how this logical framework could support extensions to SPARK contracts that allow developers to specify that information flows from inputs to an output only under certain conditions, *i.e.*, *conditional information flow*. This provides the ability to state information flow policies that are typical of *network guard applications*, where a message on an input port may flow to a certain output in one state, but may flow to a different output in another state.

In this paper, we overcome other limitations of conventional dependence/information flow frameworks by adding additional capabilities to the logic, and associated automated deduction algorithms that enable precise reasoning about array-based data structures. Figure 2 (a) presents a series of micro-examples that illustrate the deficiencies of current SPARK annotations for arrays, and Fig. 2 (b) shows our proposed enhancements. These examples are concise representations of common idioms that occur in the embedded information assurance applications of our industrial partners.

Procedure `SinglePositionAssign` assigns a value to a particular index position (the value of `Flag`) in the array `Flags`. However, the SPARK information flow contract states that (a) the whole array is modified (*i.e.*, `global out flags`), and (b) the new value of the array is derived from its old value, the `Value` parameter, and the `Flag` index parameter. This is an over-approximation of the true frame-condition and information flow, but the contract cannot be made more precise in the current SPARK annotation language. To remedy this, Figure 2 (b) illustrates that our enhanced language provides the ability to specify properties of particular array cells. The `global out` declaration now indicates that the only array cell modified is `Flags(Flag)` (which currently is a disallowed `global` expression in SPARK) while the contents of other cells remain unchanged. The enhanced `derives` indicates that the modified cell derives its value only from the parameter `Value`. To support this more precise reasoning, the underlying analysis algorithm must be able to reason symbolically about array index values.

`Scrub_Cache` in Fig. 2 (a) presents a code idiom often used when initializing an array or scrubbing the contents of a message buffer; all positions of the array are initialized to a constant value. The SPARK annotations required for this example exhibit several forms of imprecision. First, the `cache` array parameter must be declared with mode `in` even though no array element value is read during execution of the procedure. Second, the information flow specification captured in the `derives` clause is the antithesis of what we desire: it states that the final value of `cache` depends on the initial value of `cache`, whereas we desire a specification that captures the fact that the final

value of `cache` *does not* depend on the initial value of `cache`, *i.e.*, all values in the input `cache` have been erased.

This imprecision stems from the fact that on each iteration of the loop, the entire array is treated as a single entity in the information flow analysis: the updated value of the array depends on a constant value at position `I` and on its previous value at all positions other than `I`. Since flow from constants is not indicated in SPARK contracts, the information flow analysis indicates that the new value of the array depends on the old value at every iteration. There is no way to indicate that the loop has carried out an exhaustive processing of each position of the array in which the old value at each position is overwritten with a new value not based on the array's previous contents. Figure 2 (b) illustrates that we address this problem by extending the specification language with a notion of universal quantification (using syntax based on SPARK's universal quantification allowed in assertions) to specify schematically the information flow for each array cell. We also add the capability to indicate that the source of the information flow is some constant (represented by $\{\}$). Together, these additions allow us to formalize the higher level security policy: the array contents are indeed scrubbed – `cache`'s final value does not depend in any way on its initial value, nor does information from any other piece of the program state flow into it.

To support this more precise reasoning, the underlying analysis algorithm must be able to perform a logical *universal generalization* step to introduce the quantified flow specification. In general, this is quite difficult to do, but we have found that loops that manipulate arrays often follow a structure that admits an automated solution. When an automated solution is not possible, the developer may supply a information flow loop invariant (which are simpler than functional invariants) that enables the rest of the checking to be completed automatically.

The `Copy_Keys` example of Fig. 2 (a) illustrates a common idiom in which the contents of a table are copied, or where a portion of a database is moved from a central database to a copy for a client. In essence, this creates multiple channels of information flow – one channel for each index position of the arrays. In such cases, one often seeks to verify a separation policy that states that information flow between the different channels is not confused or merged. The SPARK `derives` clause for `Copy_Keys` simply states that information flows from the `inkeys` array to the `outkeys` array and cannot capture the separation property that information only flows between corresponding entries of the arrays. Fig. 2 (b) illustrates that, using the universal quantification introduced in the previous paragraph, one formalizes the policy that information only flows between entries at the same index position. Notice also that using this extra power we could specify flow between different regions of the array, by having the quantified variables take values from more restricted ranges of the possible index values.

3 Syntax and Semantics Background

We now present the foundations of our approach using a simple imperative language that can be considered an "idealized core language" for SPARK. Since SPARK omits constructs that are difficult to reason about, such as dynamically allocated data, pointers, and exceptions, its semantics is very close to that of this language.

Figure 3, presents the syntax of the simple imperative language. For commands, procedures are parameterless to simplify our exposition; our implementation supports procedures with parameters (there are no conceptual challenges in this extended functionality). In **for** loops, following similar restrictions in SPARK, we require that the index variable q is not modified by S, and does not occur anywhere except in S. Arrays

Expressions:

arithmetic

$A ::= x \mid u \mid c \mid A \;\mathsf{op}\; A \mid H[A]$

array

$H ::= h \mid Z \mid H\{A : A\}$

boolean

$\phi, B ::= A \;\mathsf{bop}\; A \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi$

Commands:

$S ::= \mathbf{skip} \mid S\,;S \mid x := A \mid \mathbf{assert}(\phi)$
$\mid \mathbf{call}\; p$
$\mid \mathbf{if}\; B\; \mathbf{then}\; S\; \mathbf{else}\; S$
$\mid \mathbf{for}\; q \leftarrow 1\; \mathbf{to}\; y\; \mathbf{do}\; S$
$\mid \mathbf{while}\; B\; \mathbf{do}\; S\; \mathbf{od}$
$\mid h := \mathbf{new} \mid h[A] := A$

Fig. 3. Syntax of a simple imperative language.

are restricted to a single dimension with integer contents. Array assignment has two forms: $h := \mathbf{new}$ creates an array with all elements set to 0, and $h[A_0] := A_1$ assigns the integer value of A_1 to array h at the index position given by A_0. For convenience of presentation, we omit some SPARK features such as records and package structure since these do not present conceptual challenges.

Expressions include arithmetic, boolean, and array expressions. Boolean expressions are also used as assertions. We use x to range over integer (scalar) variables, h to range over array variables, u to range over universally quantified variables, c to range over integer constants, op to range over arithmetic operators in $\{+, \times, \mathsf{mod}, \ldots\}$, and bop to range over comparison operators in $\{=, <, \ldots\}$.

To enable convenient reasoning about individual array elements, in particular the computation of preconditions, we follow Gries [18] and allow, in intermediate forms of assertions manipulated by the automated reasoning engine, the construct $H\{A_0 : A'\}$, which represents the value of array H except that index A_0 now has value A'. We also use Z to denote an initial array as created by the command $h := \mathbf{new}$. We require the original form of programs (command) submitted for verification to be *pure* in the sense that they do not contain these additional array constructs. Thus, in a pure entitiy, all array accesses are of the form $h[A]$ with h a variable. Similarly, universal variables u are only used in specifications; we require that *programs* submitted for verification do not contain universal variables.

The use of programmer assertions is optional, but often helps to improve the precision of our analysis. We refer to the assertions of Fig. 3 as *1-assertions* since they represent predicates on a single program state; they can be contrasted with *2-assertions* that we introduce later for reasoning about information flow in terms of a *pair* of program states. For an expression E, we write $\mathrm{fv}(E)$ for the variables occurring free in E (this included scalar, array, and universal variables), and write $E[A/x]$ for the result of substituting in E all occurrences of x by A (similarly, for other variables and syntactic domains).

Fig. 4 gives the definition of language's semantics. In the expression semantics, we model an array as a mapping ($a \in \mathit{Array}$) from integers to values, where a value ($v \in \mathit{Val}$) is an integer n. To keep the exposition simple, we shall ignore bounds and range checks, and assume that an array reference $a(n)$ is always well-defined. A *store* $s \in \mathit{Store}$ (we shall also use σ to range over stores) maps scalar and universal variables to values, and array variables to arrays; we write $\mathit{dom}(s)$ for the domain of s and write $[s \mid x \mapsto v]$ for the store that is like s except that it maps x into v, write $[s \mid h \mapsto a]$ for the store that is like s except that it maps h into a, and write $[s \mid h(n) \mapsto v]$ for $[s \mid h \mapsto [h \mid n \mapsto v]]$. We write $s \models \phi$ for $[\![\phi]\!]_s = \mathsf{True}$. We define ϕ and ϕ' to be

Expressions:

$$[\![x]\!]_s = s(x) \quad \text{similarly for } u \qquad [\![h]\!]_s = s(h)$$
$$[\![H[A]]\!]_s = [\![H]\!]_s([\![A]\!]_s) \qquad [\![Z]\!]_s = \lambda n.0$$
$$[\![H\{A_0 \,:\, A\}]\!]_s = [[\![H]\!]_s \mid [\![A_0]\!]_s \mapsto [\![A]\!]_s]$$

Commands:

$$s\ [\![\mathbf{skip}]\!]\ s' \text{ iff } s' = s$$

$$s\ [\![x := A]\!]\ s' \text{ iff } \exists v : v = [\![A]\!]_s \text{ and } s' = [s \mid x \mapsto v]$$

$$s\ [\![S_1\,;S_2]\!]\ s' \text{ iff } \exists s'' : s\ [\![S_1]\!]\ s'' \text{ and } s''\ [\![S_2]\!]\ s'$$

$$s\ [\![\mathbf{assert}(\phi)]\!]\ s' \text{ iff } s \models \phi \text{ and } s' = s$$

$$s\ [\![\mathbf{call}\ p]\!]\ s' \text{ iff } s\ \mathcal{P}(p)\ s'$$

$$s\ [\![\mathbf{if}\ \phi\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2]\!]\ s' \text{ iff } ([\![\phi]\!]_s = \mathsf{True} \text{ and } s\ [\![S_1]\!]\ s') \text{ or } ([\![\phi]\!]_s = \mathsf{False} \text{ and } s\ [\![S_2]\!]\ s')$$

$$s\ [\![\mathbf{for}\ q \leftarrow 1\ \mathbf{to}\ y\ \mathbf{do}\ S]\!]\ s' \text{ iff } \exists n \geq 1 : n = s(y) \text{ and } \forall i \in \{0 \ldots n\}\ \exists s_i : s_0 = s \text{ and } s' = [s_n \mid q \mapsto n+1] \text{ and } \forall j \in \{1 \ldots n\} :\ [s_{j-1} \mid q \mapsto j]\ [\![S]\!]\ s_j$$

$$s\ [\![\mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}]\!]\ s' \text{ iff } \exists i \geq 0 : s\ f_i\ s' \text{ where } f_i \text{ is inductively defined by:}$$
$$s\ f_0\ s' \text{ iff } [\![B]\!]_s = \mathsf{False} \text{ and } s' = s$$
$$s\ f_{i+1}\ s' \text{ iff } \exists s'' : ([\![B]\!]_s = \mathsf{True} \text{ and } s\ [\![S]\!]\ s'' \text{ and } s''\ f_i\ s')$$

$$s\ [\![h[A_0] := A]\!]\ s' \text{ iff } \exists n, v : n = [\![A_0]\!]_s,\ v = [\![A]\!]_s \text{ and } s' = [s \mid h(n) \mapsto v]$$

$$s\ [\![h := \mathbf{new}]\!]\ s' \text{ iff } s' = [s \mid h \mapsto \lambda n.0]$$

Fig. 4. Semantics of the Simple Programming Language (excerpts).

1-equivalent, written $\phi \equiv_1 \phi'$, if for all s it holds that $s \models \phi$ iff $s \models \phi'$. Similarly, we write $\phi \rhd_1 \phi'$ if whenever $s \models \phi$ then also $s \models \phi'$.

In the definition of the **call** command, we assume a global procedure environment $\mathcal{P}$ that for each p returns a relation between input and output stores (we expect that if $s\ \mathcal{P}(p)\ s'$ then, with S the body of p, we have $s\ [\![S]\!]\ s'$). For some S and s, there may not exist any s' such that $s\ [\![S]\!]\ s'$; this can happen if a **while** loop does not terminate, a **for** loop has a non-positive upper bound, or an **assert** fails.

4 Information Flow Contracts for Arrays

4.1 Capturing non-interference in a compositional logic

Example 1.
```
procedure p
  begin x := a+1;  h(7) := z; y := b * 2; end p;
```

For the example procedure above, there are two "channels" of information flow associated with x and y: (1) from a to x, and (2) from b to y (we ignore the additional flow for h at the moment). Using SPARK to specify these flows, we would write:

```
derives x from a & y from b;
```

Following Goguen and Meseguer [16], we may express the *non-interference* of the assignment to y with channel (1) via the following semantic property: for any pair of states s_1 and s_2, if $s_1(a) = s_2(a)$ then $s_1'(x) = s_2'(x)$ where s_1', s_2' are the states that result from executing the procedure body on s_1 and s_2, respectively. We desire to state such non-interference properties (which would provide a semantic foundation for `derives`

contracts), using program level assertions. However, the non-interference property requires reasoning about *two* states at method pre/postcondition (cf. s_1 and s_2). Thus, it cannot be stated using traditional assertions, because such assertions are interpreted in terms of *one* state at a particular program point.

The innovation of the logic developed in [1, 2] lies in the introduction of a novel *agreement assertion* $x\ltimes$ that is satisfied by a *pair* of states, s_1 and s_2, if $s_1(x) = s_2(x)$. Using this assertion, the non-interference property above is phrased $\{\mathsf{a}\ltimes\}\ S\ \{\mathsf{x}\ltimes\}$. In general, triples are of the form $\{x_1\ltimes, \ldots, x_n\ltimes\}\ P\ \{y_1\ltimes, \ldots, y_m\ltimes\}$ which is interpreted as follows: *given two runs of P that initially agree on variables $x_1 \ldots x_n$, at the end of both runs, they agree on variables $y_1 \ldots y_m$*. Such a specification says that the variables y_j may depend *only* on the variables x_i, and not on any other variables. In situations as above where we want to reason about multiple separated channels of information flow simultaneously (*e.g.*, a to x and b to y), we would *not* write $\{a\ltimes, b\ltimes\}\ S\ \{x\ltimes, y\ltimes\}$ since this would imply that y may depend on a and x depend on b. Instead, *channel-indexed agreement assertions* would be used to distinguish the separate channels for x and y: $\{a\ltimes_x, b\ltimes_y\}\ S\ \{x\ltimes_x, y\ltimes_y\}$. This is equivalent to requiring both $\{a\ltimes\}\ S\ \{x\ltimes\}$ and $\{b\ltimes\}\ S\ \{y\ltimes\}$ to hold in the unindexed version of the logic. Our implementation uses the indexed assertions to deal with multiple channels, but to simplify the formalization, in this document we shall assume we deal with one channel at a time.

Notice how non-interference is captured as a special case by this type of specifications: given the conventional lattice of security levels HIGH $>$ LOW, then the non-interference property is equivalent to demanding that if $z_1 \ldots z_n$ are the LOW variables then $\{z_1\ltimes, \ldots, z_n\ltimes\}\ P\ \{z_1\ltimes, \ldots, z_n\ltimes\}$,

The logic developed in [2] was designed to verify specifications of the following form: *given two runs of P that initially agree on variables $x_1 \ldots x_n$, the runs agree on variables $y_1 \ldots y_m$ at the end of the runs*. This includes non-interference as a special case, as can be seen by letting $x_1 \ldots x_n$, and $y_1 \ldots y_m$, be the variables of one partition. We may express such a specification, which makes the "end-to-end" (input to output) aspect of verifying confidentiality explicit, in Hoare-logic style as $\{x_1\ltimes, \ldots, x_n\ltimes\}\ P\ \{y_1\ltimes, \ldots, y_m\ltimes\}$, where the *agreement assertion* $x\ltimes$ is satisfied by a *pair* of states, s_1 and s_2, if $s_1(x) = s_2(x)$.

One advantage of this logical approach over traditional data/control-flow based approaches to reasoning about information flow and program dependencies, is that the assertion primitive can be enhanced to reason about additional properties of the state – leading to greater precision and flexibility. For example, to capture conditional information flow, we use *conditional* agreement assertions $\phi \Rightarrow E\ltimes$, also called *2-assertions*, introduced by Banerjee and the first author. Such assertions are satisfied by a pair of stores if either at least one of them does not satisfy ϕ, or they agree on the value of E: $s\ \&\ s_1 \models \phi \Rightarrow E\ltimes$ iff whenever $s \models \phi$ and $s_1 \models \phi$ then $[\![E]\!]_s = [\![E]\!]_{s_1}$.

We use $\theta \in \mathbf{2Assert}$ to range over 2-assertions. For $\theta = (\phi \Rightarrow E\ltimes)$, we call ϕ the antecedent of θ and write $\phi = ant(\theta)$, and we call E the consequent of θ and write $E = con(\theta)$. We often write $E\ltimes$ for *true* $\Rightarrow E\ltimes$. We use $\Theta \in \mathcal{P}(\mathbf{2Assert})$ to range over sets of 2-assertions (where we often write θ for the singleton set $\{\theta\}$), with conjunction implicit. Thus, $s\&s_1 \models \Theta$ iff $\forall \theta \in \Theta : s\&s_1 \models \theta$.

For the semantics of command triples, we write $\{\Theta\}S\{\Theta'\}$ iff for all s, s', σ, σ', if $s\ [\![S]\!]\ s'$ and $\sigma\ [\![S]\!]\ \sigma'$, and also $s\&\sigma \models \Theta$, then $s'\&\sigma' \models \Theta'$. Similarly for procedures, we write $\{\Theta\}p\{\Theta'\}$ iff for all s, s', σ, σ', if $s\ \mathcal{P}(p)\ s'$ and $\sigma\ \mathcal{P}(p)\ \sigma'$, and also $s\&\sigma \models \Theta$, then $s'\&\sigma' \models \Theta'$.

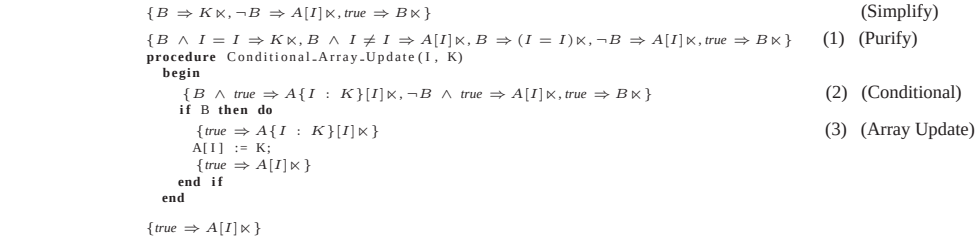

Fig. 5. A simple conditional array update procedure showing the logical derivation of the procedure's information flow precondition.

We define $\Theta \rhd_2 \Theta'$, pronounced "Θ 2-implies Θ'", to hold iff for all s, s_1: whenever $s\&s_1 \models \Theta$ then also $s\&s_1 \models \Theta'$. In development terms, when $\Theta \rhd_2 \Theta'$ holds we can think of Θ as a *refinement* of of Θ', and Θ' an *abstraction* of Θ. Intuitively, Θ requires agreement in more cases than Θ' (Θ is a strengthening of Θ'). For example, $\{x\ltimes\ ,\ y\ltimes\}$ refines $x\ltimes$ by adding an (unconditional) agreement requirement on y, and $y < 10 \Rightarrow x\ltimes$ refines $y < 7 \Rightarrow x\ltimes$ by weakening the antecedent of a 2-assertion so that agreement on x is required for more values of y.

Figure 5 shows an example derivation in our logic, with the result of applying each rule interspersed with each line of code, and the name of the corresponding rule. We will return to this example in later discussions, but it is worth looking at assertion (3), which shows the workings of the new array update rule. In this case, since $A[I]$ has been updated with K, the precondition is obtained by substituting $A\{I\ :\ K\}$ (which, as explained before, is the same as A, except that location I is mapped to K) for A in the postcondition. Thanks to this construct, we can keep track of modifications at the level of each array location. However, since the idiom $H\{A_0\ :\ A'\}$ does not exist in the target assertion language (*i.e.*SPARK annotation language, and certainly most others), we remove such expressions at procedure boundaries using a strengthening procedure (Purify) presented later.

The information flow contract is given by the dependence relation between the variables in the postcondition and the variables obtained in the precondition. In this case, we have that the final value of $A[I]$ depends on the initial value of K whenever B holds, and on its own initial value otherwise, and of course it depends on the initial value of B due to control dependence.

4.2 Representing SPARK Information Flow Contracts

A SPARK information flow contract for a procedure p has a representation in our core language that includes (1) assertions specifying frame conditions, (2) 2-assertions specifying information flow properties.

Frame conditions are represented by a set of assertions that state which variables *may* be modified by the body of p (e.g, SPARK `out` variables). Specifically, mod^p_w holds when variable w may be modified by p. If $s\ \mathcal{P}(p)\ s'$ and mod^p_w does *not* hold then $s(w) = s'(w)$. Also, for each array variable h there exists mod^p_h which is a *quantified assertion* of the form $\forall u.\phi_h$ such that for all n, if $h(n)$ may be modified then $\phi_h[n/u]$. That is, if $s\ \mathcal{P}(p)\ s'$ and $[\![\phi_h]\!]_{[s|u\rightarrow n]}$ does *not* hold then $[\![h]\!]_s(n) = [\![h]\!]_{s'}(n)$.

In Example 1 above, both `x` and `h(7)` are modified by the procedure, therefore we have: $\mathrm{mod}^p_x \equiv true$, and mod^q_h is given by $u = 7$.

Contract information flow properties for a procedure p are represented by agreement preconditions for each variable/array-location modified by p. These preconditions indi-

cates the input variables and array locations upon which the modified variables depend. That is,

- For each w such that $\mathrm{mod}^p_w \equiv_1$ *false* does not hold, there is a precondition $2PC^p_w$ such that $\{2PC^p_w\}p\{w\ltimes\}$.
- For each h such that $\mathrm{mod}^p_h \equiv_1$ *false* does not hold, there is a precondition $2PC^p_{h[\_]}$ which is a *quantified set of 2-assertions* of the form $\forall u.\Theta$. Here we demand that $\{\Theta\}p\{h[u]\ltimes\}$.

In Example 1 above, we would have

$$2PC^p_{h[\_]} = \{u = 7 \Rightarrow z\ltimes,\ u \neq 7 \Rightarrow h[u]\ltimes\}$$

4.3 Computing Precondions

Figure 6, selected parts of which will be explained later, presents a rule-based precondition generation algorithm inductively defined over the language syntax. The definition uses rules of the form $\{\Theta\} \Longleftarrow S\ \{\Theta'\}$ to specify that, given command S and postcondition Θ', the algorithm computes precondition Θ. The algorithm *does not* always compute the weakest precondition; a small degree of imprecision may be introduced when treating loops and procedure calls. The former is due to the need to ensure termination of the analysis, the latter is due to the need for modularity so that the analysis uses a procedure's specification rather than its actual code. As a result, antecedents may be too weak, yielding too strong 2-assertions.

This algorithm extends our earlier work [5] by adding fine grained treatment of array updates/accesses, the notion of universal quantification for reasoning about arrays, and a method for inferring universally quantified preconditions for certain **for**-loop structures. The following theorem summarizes the correctness of the algorithm (for a sketch of the proof for this theorem, we refer the reader to [4]).

Theorem 1. *For all S, Θ, Θ', if $\{\Theta\} \Longleftarrow S\ \{\Theta'\}$ holds, then $\{\Theta\}S\{\Theta'\}$ holds.*

The algorithm can be applied to automatically check or infer information flow contracts. For implementing checking, the algorithm would be used to compute a candidate precondition from the stated postcondition, and then a supplementary algorithm would check that the stated precondition entails the computed precondition (this functionality is present in our implementation using theorem-prover technology). We focus on contract inference in the remainder of our discussion.

As with conventional forms of compositional contract-based reasoning, when processing the body of some procedure p_1, our algorithm assumes that any procedure called by p_1 already has an associated contract (since SPARK does not include recursion, contract inference for all procedures in the program can be carried out via a bottom up traversal of the call graph).

The Purify **Function:** As noted earlier, our algorithm generates impure agreement assertions Θ' that may include expressions of the form $H_0\{A_0 : A_1\}$. Since this expression form is not found in the programming language, to phrase a contract in terms of the programming language's expressions, we need to strengthen Θ' to obtain a pure assertion Θ to be used in the contract such that $\Theta \rhd_2 \Theta'$. The following lemma establishes the existence of such a Θ. The intuition is that the implicit "case analysis" in $H_0\{A_0 : A_1\}$ (*i.e.*., yield the value of A_1 when the index has value A_0 and consult H_0 otherwise) can be represented using logical implications that "trigger" based on different cases of the index value.

$\{\Theta\} \Longleftarrow \mathbf{skip}\ \{\Theta'\}$ iff $\Theta = \Theta'$

$\{\Theta\} \Longleftarrow \mathbf{assert}(\phi_0)\ \{\Theta'\}$
iff $\Theta = \{(\phi \wedge \phi_0) \Rightarrow E\ltimes \mid \phi \Rightarrow E\ltimes \in \Theta'\}$

$\{\Theta\} \Longleftarrow x := A\ \{\Theta'\}$ iff $\Theta = \Theta'[A/x]$

$\{\Theta\} \Longleftarrow h[A_0] := A\ \{\Theta'\}$ iff $\Theta = \Theta'[h\{A_0 \ : \ A\}/h]$

$\{\Theta\} \Longleftarrow h := \mathbf{new}\ \{\Theta'\}$ iff $\Theta = \Theta'[Z/h]$

$\{\Theta\} \Longleftarrow S_1\ ;\ S_2\ \{\Theta'\}$
iff $\{\Theta''\} \Longleftarrow S_2\ \{\Theta'\}$ and $\{\Theta\} \Longleftarrow S_1\ \{\Theta''\}$

$\{\Theta\} \Longleftarrow \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \{\Theta'\}$
iff $\Theta = \bigcup_{\theta \in \Theta'} \mathsf{Pre}_{\mathrm{if}}(\theta)$ where
$\mathsf{Pre}_{\mathrm{if}}(\phi \Rightarrow E\ltimes) =$
let $\{\Theta_1\} \Longleftarrow S_1\ \{\phi \Rightarrow E\ltimes\}$
let $\{\Theta_2\} \Longleftarrow S_2\ \{\phi \Rightarrow E\ltimes\}$
in if $\mathsf{Preserves}(S_1, E)$ and $\mathsf{Preserves}(S_2, E)$
then $\{\phi_1 \wedge B \vee \phi_2 \wedge \neg B \Rightarrow E\ltimes \mid \phi_i \Rightarrow \text{-}\ltimes \in \Theta_i (i = 1, 2)\}$
else $\{\phi_1 \wedge B \Rightarrow E_1\ltimes \mid \phi_1 \Rightarrow E_1\ltimes \in \Theta_1\} \cup \{\phi_2 \wedge \neg B \Rightarrow E_2\ltimes \mid \phi_2 \Rightarrow E_2\ltimes \in \Theta_2\} \cup \{\phi_1 \wedge B \vee \phi_2 \wedge \neg B \Rightarrow B\ltimes \mid \phi_i \Rightarrow \text{-}\ltimes \in \Theta_i (i = 1, 2)\}$

$\{\Theta\} \Longleftarrow \mathbf{call}\ p\ \{\Theta'\}$ iff $\Theta = \bigcup_{\theta \in T} \mathsf{Pre}_{\mathrm{call}}(\theta) \cup R$
where $(R, T) = \mathsf{PreProc}(\Theta')$ and
$\mathsf{Pre}_{\mathrm{call}}(\phi' \Rightarrow E\ltimes) =$
let $\phi = NPC(p, \phi')$
in case E of
$w \Rightarrow \{\phi \wedge \phi_w \Rightarrow E_w\ltimes \mid \phi_w \Rightarrow E_w\ltimes \in 2PC^p_w\}$
$h[A] \Rightarrow \{\phi \wedge \phi_h[A/u] \Rightarrow E_h[A/u]\ltimes \mid \phi_h \Rightarrow E_h\ltimes \in \Theta_h\}$
where $2PC^p_{h[\_]} = \forall u.\Theta_h$

$\{\Theta\} \Longleftarrow \mathbf{for}\ q \leftarrow 1\ \mathbf{to}\ y\ \mathbf{do}\ S_0 (= S)\ \{\Theta'\}$
iff $\Theta = \Theta_f \cup \Theta_a \cup \Theta_w[1/q] \cup R$
and $(R, T) = \mathsf{PreProc}(\Theta')$ where
$\Theta'_a = \{\theta \in T \mid \exists h, A : con(\theta) = h[A]\}$
$\Theta'_f = \{\theta \in \Theta'_a \mid \mathsf{Pre}_{\mathrm{for}}(S_0, q, y, \theta)\ \text{succeeds}\}$
$\Theta'_w = \Theta'_a \setminus \Theta'_f$
$\Theta_f = \bigcup_{\theta \in \Theta'_f} \mathsf{Pre}_{\mathrm{for}}(S_0, q, y, \theta)$
$\Theta_a = \{NPC(S, \phi) \Rightarrow A\ltimes \mid \phi \Rightarrow h[A]\ltimes \in \Theta'_w\}$
$\Theta'_h = \{\phi \Rightarrow h\ltimes \mid \phi \Rightarrow h[A]\ltimes \in \Theta'_w\}$
$\Theta_w = \mathsf{Pre}_{\mathrm{while}}((S_0\ ;\ q := q + 1), q \leq y, \Theta'_h \cup (T \setminus \Theta'_a))$

$\{\Theta\} \Longleftarrow \mathbf{while}\ B\ \mathbf{do}\ S_0\ \mathbf{od} (= S)\ \{\Theta'\}$
iff $\Theta = \Theta_w \cup \Theta_a \cup R$ and $(R, T) = \mathsf{PreProc}(\Theta')$ where
$\Theta_a = \{NPC(S, \phi) \Rightarrow A\ltimes \mid \phi \Rightarrow h[A]\ltimes \in T\}$
$\Theta'_h = \{\phi \Rightarrow h\ltimes \mid \phi \Rightarrow h[A]\ltimes \in T\}$
$\Theta'_w = \{\theta \in T \mid con(\theta)\ \text{is a variable}\}$
$\Theta_w = \mathsf{Pre}_{\mathrm{while}}(S_0, B, \Theta'_h \cup \Theta'_w)$

Helper functions:

$\mathsf{Pre}_{\mathrm{for}}(S, q, y, \phi \Rightarrow h[u]\ltimes) =$
let A_j be all expressions such that S_j is a subcommand of S
and S_j is of the form $h[A_j] := \_$ with $j \in J$
and $J = \{1, 2, \ldots \#\mathrm{ArrUpd}(S)\}$
in
let $\forall j \in J\ .\ \{\Theta_j\} \Longleftarrow S\ \{h[A_j]\}$
in
if $(\forall S_i\ .\ S_i$ occurs in $S \wedge (S_i = \mathbf{call}\ p) \Rightarrow \mathsf{Preserves}(S_i, h))$
$\wedge\ (\forall j \in J\ .\ \mathsf{Preserves}(S, A_j))$
$\wedge\ (\forall j \in J\ .\ \exists A'_j\ .\ (\mathrm{fv}(A'_j) \subseteq \{u\} \cup \mathrm{fv}(A_j) \setminus \{q\} \wedge \forall s\ .\ [\![u = A_j]\!]s = [\![q = A'_j]\!]s$
$\wedge\ (\forall j \in J\ .\ \exists \phi_j\ .\ (\mathrm{fv}(\phi_j) \subseteq \{u\} \cup \mathrm{fv}(A_j) \setminus \{q\} \wedge (\forall s, n\ .\ [s \mid u \mapsto n] \models \phi_j \Leftrightarrow n \in \{[\![A_j]\!]_{[s|q \mapsto i]} \mid 1 \leq i \leq s(y)\}))))$
$\wedge\ w \in fv(\Theta_j) \wedge \neg\mathsf{Preserves}(S, w) \Rightarrow w = h$
$\wedge\ \forall j \in J, h[A]\ .\ h[A]$ is a subexpression of $\Theta_j \Rightarrow$
$(\forall s \in Store, i, i' \in \{1 \ldots s(y)\}\ .\ [\![A]\!]_{[s|q \mapsto i']} = [\![A_j]\!]_{[s|q \mapsto i]} \Rightarrow i' \leq i)$
then $\{\phi_j \Rightarrow \Theta_j[A'_j/q]\ltimes \mid j \in J\}$
$\cup \{\wedge_{j \in J} \neg\phi_j \Rightarrow h[u]\ltimes\}$
$\cup \{x\ltimes \mid \exists j \in J : x \in \mathrm{fv}(A_j) \setminus \{q\}\}$
$\cup \{y\ltimes\}$
(succeeds)
else
$\emptyset$ (fails)

$\mathsf{PreProc}(\Theta') =$
$P := Purify(\Theta')$
$R := \emptyset$
$T := \emptyset$
while $P \neq \emptyset$
remove $(\phi \Rightarrow E\ltimes)$ from P and do
if $\mathsf{Preserves}(p, E)$
then $R := R \cup \{NPC(S, \phi) \Rightarrow E\ltimes\}$
else case E of
E_1 op E_2 or E_1 bop E_2
$\Rightarrow P := P \cup \{\phi \Rightarrow E_1\ltimes, \phi \Rightarrow E_2\ltimes\}$
$w \Rightarrow T := T \cup \{\phi \Rightarrow w\ltimes\}$
$h[A] \Rightarrow$
if $\mathsf{Preserves}(p, h) \wedge \neg\mathsf{Preserves}(p, A)$
then $R := R \cup \{NPC(S, \phi) \Rightarrow h\ltimes\}$
else if $\neg\mathsf{Preserves}(p, h) \wedge \mathsf{Preserves}(p, A)$
then $T := T \cup \{\phi \Rightarrow h[A]\ltimes\}$
else if $\neg\mathsf{Preserves}(p, h) \wedge \neg\mathsf{Preserves}(p, A)$
then $T := T \cup \{\phi \Rightarrow h\ltimes\}$
return (R, T)

$\mathsf{Pre}_{\mathrm{while}}(S, B, \Theta') = \Theta^j$ such that
for each $w \in X$ (with $X = fv(S) \cup fv(B) \cup U$), where U is the set of all variables involved in procedure calls occurring in S, we inductively in i define
$\phi^i_w, \Theta^i, \Psi^i, \psi^i_w$ by
$\phi^0_w = \bigvee\{\phi \mid \exists E : (\phi \Rightarrow E\ltimes) \in \Theta' \wedge w \in \mathrm{fv}(E)\}$,
$\Theta^i = \{\phi^i_w \Rightarrow w\ltimes \mid w \in X\},\ \{\Psi^i_w\} \Longleftarrow S\ \{\phi^i_w \Rightarrow w\ltimes\}$
$\Psi^i = \bigcup_{w \in X} \Psi^i_w$
$\psi^i_w = \bigvee\{\phi \mid \exists (\phi \Rightarrow E\ltimes) \in \Psi^i$ with $w \in \mathrm{fv}(E)$ or $w \in \mathrm{fv}(B)$ and $\exists z \in X\ .\ \neg\mathsf{Preserves}(S, z) \wedge (\phi = \phi^i_z \vee \phi \Rightarrow \text{-}\ltimes \in \Psi^i_z)\}$
$\phi^{i+1}_w = \text{if } \psi^i_w \rhd_1 \phi^i_w \text{ then } \phi^i_w \text{ else } \phi^i_w \bigtriangledown \psi^i_w$
and j is the least i such that $\Theta^i = \Theta^{i+1}$

Fig. 6. The Precondition Generator

Lemma 1 (refine impure to pure). *There exists a function $Purify$ such that*
- *for all 1-assertion ϕ', $Purify(\phi') = \phi$, where ϕ is pure and $\phi \equiv_1 \phi'$,*
- *for all 2-assertions Θ', $Purify(\Theta') = \Theta$ where Θ is pure and $\Theta \rhd_2 \Theta'$.*

Now, we can go ahead and define a function Purify that refines 2-assertions while removing non-pure constructs from their consequents. To be precise, with $\Theta_0 = \mathsf{Purify}(\Theta)$ it holds that $\Theta_0 \rhd_2 \Theta$ and that $con(\theta)$ is pure for all $\theta \in \Theta_0$; the former requirement ensures that if a command establishes Θ_0 then it also establishes Θ. We define

$$\mathsf{Purify}(\Theta) = \bigcup_{\theta \in \Theta} \mathsf{Purify}(\theta)$$

where $\mathsf{Purify}(\theta)$ with $\theta = \phi \Rightarrow E\ltimes$ is defined by case analysis on E:

– If E is an arithmetic expression A, apply Lemma 1 to find pure $A_1 \dots A_k$ and pure $\phi_1 \dots \phi_k$ such that $[\![A]\!]_s = v$ iff there exists $i \in \{1 \dots k\}$ such that $s \models \phi_i$ and $[\![A_i]\!]_s = v$. Then

$$\begin{aligned}\mathsf{Purify}(\theta) = \{&\phi \wedge \phi_i \Rightarrow A_i{\ltimes} \mid i \in \{1 \dots k\}\} \\ \cup\, \{&\phi \Rightarrow \phi_i{\ltimes} \mid i \in \{1 \dots k\}\}\end{aligned}$$

– If E is an assertion ϕ_1, apply Lemma 1 to find pure ϕ_0 such that $\phi_1 \equiv_1 \phi_0$. Then $\mathsf{Purify}(\theta) = \{\phi \Rightarrow \phi_0{\ltimes}\}$.
– Otherwise, E is an array expression H. If H is an array variable h we have $\mathsf{Purify}(\theta) = \{\phi \Rightarrow h{\ltimes}\}$; if H is Z then $\mathsf{Purify}(\theta) = \emptyset$. Otherwise, H is of the form $H = H_0\{A_0 : A_1\}$ and we call Purify recursively:

$$\begin{aligned}\mathsf{Purify}(\theta) = &\ \mathsf{Purify}(\phi \Rightarrow H_0{\ltimes}) \\ \cup\, &\ \mathsf{Purify}(\phi \Rightarrow A_0{\ltimes}) \\ \cup\, &\ \mathsf{Purify}(\phi \Rightarrow A_1{\ltimes}).\end{aligned}$$

For example, the 2-assertion $\phi \Rightarrow h\{A_0 : A'\}[A]{\ltimes}$ is refined by $\{\theta_1, \theta_2, \theta_3\}$ where

$$\begin{aligned}\theta_1 &= (\phi \wedge A = A_0) \Rightarrow A'{\ltimes} \\ \theta_2 &= (\phi \wedge A \neq A_0) \Rightarrow h[A]{\ltimes} \\ \theta_3 &= \phi \Rightarrow (A = A_0){\ltimes}\end{aligned}$$

θ_1 and θ_2 represent a case analysis on the index expression, and θ_3 imposes a "control" agreement stating that for any two states, the same case must be taken whenever ϕ holds. Looking back now at Fig. 5, we can see how assertion (2) is turned into assertion (1) using that Purify function.

The Preserves **Function:** There are several instances in Figure 6 where the generation of the precondition for S from its post-condition $\phi \Rightarrow E{\ltimes}$ can be simplified if S preserves the semantics of E. Accordingly, we utilize a predicate $Preserves(S, E)$ such that if $Preserves(S, E)$ holds then whenever $s\ [\![S]\!]\ s'$ we have $[\![E]\!]_s = [\![E]\!]_{s'}$. For decidability reasons we cannot hope for the opposite direction always to hold, but it is we shall aim at a good approximation, and define the predicate $Preserves(S, E)$ recursively, by case analysis on S:

– for an assignment $x := A$ we demand $x \notin \mathrm{fv}(E)$;
– for a conditional $\mathbf{if}\ \phi\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2$, or a sequential composition $S_1\ ; S_2$, we demand $Preserves(S_1, E)$ and $Preserves(S_2, E)$;
– for iterations $\mathbf{while}\ \phi\ \mathbf{do}\ S_0\ \mathbf{od}$, or $\mathbf{for}\ q \leftarrow 1\ \mathbf{to}\ y\ \mathbf{do}\ S_0$, we demand $Preserves(S_0, E)$.
– for an array update $h[A_0] := A_1$, we demand either that $h \notin \mathrm{fv}(E)$, or that E is a pure arithmetic or boolean expression where for all syntactic subparts of the form $h[A]$, it holds for all s that $[\![A_0]\!]_s \neq [\![A]\!]_s$.
– for a procedure call $\mathbf{call}\ p$ we demand:
  - for all $x \in \mathrm{fv}(E)$, mod_x^p is equivalent to *false*.
  - for all $h \in \mathrm{fv}(E)$, with mod_h^p of the form $\forall u.\phi_h$, either ϕ_h is equivalent to *false*, or E is a pure arithmetic or boolean expression where for all A such that $h[A]$ is a syntactic subpart of E, it holds that there is no s with $s \models \phi_h[A/u]$.

We shall often write $Preserves(p, E)$ for $Preserves(\mathbf{call}\ p, E)$.

The NPC **Function:** When generating a precondition for S for post-condition $\phi \Rightarrow E\ltimes$ where $Preserves(S, E)$ holds, whereas S may affect the antecedent ϕ, we must compute a new antecedent ϕ' so that $\{\phi' \Rightarrow E\ltimes\}S\{\phi \Rightarrow E\ltimes\}$; For this to be the case, we must ensure that if two post-states satisfy ϕ then the pre-states satisfy ϕ' and hence $E\ltimes$. Accordingly, we utilize a function NPC computing a "necessary precondition" for ϕ to hold after S. That is, with $\phi_0 = NPC(S, \phi)$ it holds that if $s\ [\![S]\!]\ s'$ and $s' \models \phi$ then $s \models \phi_0$. Note that we can always pick $\phi_0 = true$, and that if S *preserves* ϕ then we can pick $\phi_0 = \phi$. We assume that each procedure p is equipped with such a function, such that if $s\ \mathcal{P}(p)\ s'$ and $s' \models \phi$ then with $\phi_0 = NPC(p, \phi)$ we have $s \models \phi_0$. In our implementation, NPC is computed by performing a gradual weakening of the antecedents, in the worst case returning *true*. It may seem counterintuitive that we are talking about *necessary* precondition instead of *weakest* precondition, but this stems from the contravariant nature of the antecedent component of 2-assertions.

The PreProc **Function:** The computation of preconditions for procedure calls and loops shares certain steps that can be broken out into a preprocessing phase realized by a common function, called PreProc and listed in Fig. 6. Preprocessing includes two main ideas: (1) strengthening 2-assertions to a canonical form $\phi \Rightarrow E_{con}\ltimes$ where E_{con} must be a variable name or array access expression (but not an operation), and (2) the immediate construction of preconditions (the R component) which is possible for 2-assertions whose consequents are not modified by the command under consideration. Point (1) is required for, e.g., the identification of dependence connections between a calling context and the contract of the called procedure.

As a first step, $Purify$ is applied to Θ'. The preprocessing algorithm maintains three sets of 2-assertions: P – a set of pending assertions, still to be preprocessed; R – is a set of assertions that can immediately be put in precondition Θ without further processing; T – a set of assertions already preprocessed, but still need to go through the precondition generator. The following invariant holds for these three sets: if $\{\Theta\}S\{T \cup P\}$ then $\{\Theta \cup R\}S\{\Theta'\}$. Initially P is set to $Purify(\Theta')$, whereas both T and R are set to $\emptyset$. Each iteration of the preprocessing algorithm removes an element $\phi \Rightarrow E\ltimes$ from P, and performs the following steps:

1. If S *preserves* E then the 2-assertion $NPC(S, \phi) \Rightarrow E\ltimes$ is inserted into R – bypassing further computation as discussed in the definitions of *preserves* and NPC.
2. Otherwise, if E is of the form E_1 op E_2 or E_1 bop E_2, the 2-assertions $\phi \Rightarrow E_1\ltimes$ and $\phi \Rightarrow E_2\ltimes$ are inserted into P for further preprocessing – leading to notion of canonical form discussed above.
3. Otherwise, if E is a variable w (scalar or array), the 2-assertion $\phi \Rightarrow w\ltimes$ is inserted into T.
4. Otherwise, E must be of the form $h[A]$. There are three sub-cases:
   - If S *preserves* h and not S *preserves* A, the assertion $NPC(S, \phi) \Rightarrow h\ltimes$ is inserted into R. Since S does not modify h we can add the assertion directly to the preconditions. However, we can not use A to access the particular location because A is modified by S, so we strengthen the assertion to require that *all* of the array has the same value on the prestate for any 2 runs[2].

[2] Note that since this language has no pointers (just as SPARK), then the assertion $h\ltimes$ applies on the array itself, not on its *l-value* as some might expect.

```
procedure flipHalves
--# global in out h;
--# derives h from *;
is
 t : h.content;
 m : h.Range_type;
 begin
    m := h.Range'Last/2;
    for q in range
     h.Range'First .. m loop
       t := h(q);
       h(q) := h(q+m);
       h(q+m) := t;
    end loop;
end flipHalves;
```

(a)

```
procedure flipHalves_Refactored
 (q : h.Range'First .. h.Range'Last/2)
--# global in out h(*);
--# derives for all I in h.Range =>
--#     (h(I) from h(q+m) when I = q,
--#                 h(q) when I = q+m,);
is
 t : h.content;
 m : h.Range_type;
 begin
    m := h.Range'Last/2;

    t := h(q);
    h(q) := h(q+m);
    h(q+m) := t;

end flipHalves_Refactored;
```

(b)

```
procedure flipHalves
--# global in out h;
--# derives for all I in h.Range =>
--#     (h(I) from h(I+h.Range'Last/2)
--#         when I >= 1 && I <= h.Range'Last/2,
--#        h(I-h.Range'Last/2)
--#         when I > h.Range'Last/2 && I <= h.Range,
--#        * when I < 1 || I > h.Range);
is
 t : h.content;
 m : h.Range;
 begin
    m := h.Range'Last/2;
    for q in range h.Range'First .. m loop
       t := h(q);
       h(q) := h(q+m);
       h(q+m) := t;
    end loop;
end flipHalves;
```

(c)

Fig. 7. `for` loop example

- If S *preserves* h does not hold, but S *preserves* A holds, then $\phi \Rightarrow h[A]\ltimes$ is inserted into T.
- If neither S *preserves* h nor S *preserves* A hold, then $\phi \Rightarrow h\ltimes$ is inserted into T – using a strengthening similar to the first sub-case. However, since S modifies h in this case, we require the resulting assertion to go through full precondition generation (via T) instead of moving immediately to the precondition (via R).

The algorithm terminates when P is $\emptyset$. Upon termination, T is the set of assertions that will be passed on to the particular precondition generator (procedure calls or loops), the output of which will be joined with R to form the desired precondition for S.

The $\mathsf{Pre}_{\text{for}}$ **Function:** The rule (Fig. 6) for **for**-loops, with associated helper function $\mathsf{Pre}_{\text{for}}$, lies at the core of the generation of universally quantified information flow assertions for arrays, and is one of the main innovations of this paper. The idea behind this function is to identify and exploit a common pattern: **for**-loops are often used to traverse arrays to perform updates or other processing on a per-location basis *and* the processing is often done in a manner in which the current iteration does not depend on previous iterations, *i.e.*, there are no *loop-carried-dependencies* [21].

Consider the procedure `flipHalves` in Fig. 7 (a) that flips the values between the upper and lower halves of an array – resulting in information flow between the two halves of the array. However, if we apply the approach to loop processing from our previous work [5], we obtain a contract that merely says that the final value of the array is derived from its original value (`h from *`), but nothing more precise.

Still, this procedure possesses no loop-carried-dependencies: changes made in the current iteration do not depend on previous ones. So, we should be able to reason about the flows in all iterations of this loop (and analogously, flows related to all index positions of array h) using a single "schematic" iteration (and analogously, a single "schematic" index position h). We try this by refactoring the body of the loop into a separate procedure `flipHalves_Refactored` (Fig. 7 (b)) that performs the flip on a single pair of index positions, *i.e.*, just one iteration of the loop. When we run our algorithm now, we obtain a contract showing the flow between the two locations on the separate halves of the array. It's not hard to see from this point that what we want is a quantified version of the specification obtained on the refactored procedure.

To ensure discovery of appropriate quantifications when multiple array updates occur in a loop body S, the definition $\mathsf{Pre}_{\text{for}}$in Fig. 6 has some additional concepts. `#ArrUpd(S)` returns the number of subcommands in S that are of the form $h[A] := \_$,

i.e., the number of array updates. With J is a set of indices ranging over those subcommands, for each $j \in J$ we let A_j be the corresponding array indexing expression. With these definitions at hand, $\mathsf{Pre}_{\text{for}}$ checks for the following conditions to hold on the body of the loop:

1. For all $\mathbf{call}\ p$ syntactically occurring in S, we have p *preserves* h.
2. For all $j \in J$ we have S *preserves* A_j.
3. Every A_j has an "inverse" A'_j, satisfying that $A_j = u$ iff $A'_j = q$. For example, if $A_j = q + 1$, then $A'_j = u - 1$.
4. The range of each A_j can be expressed. For example, if q ranges from 1 to 10, and $A_j = q + 1$, then the range of A_j is $1 + 1 \leq u \leq 10 + 1$.
5. Nothing in the precondition is modified except possibly h; that is, there are no loop carried dependencies between scalar variables.
6. There are no loop-carried dependencies between array locations. That is, an array location is not read *after* it has been updated.

In summary, $\mathsf{Pre}_{\text{for}}$ leverages the intuition obtained above, and upon discharging several sanity conditions, among those the absence of loop-carried-dependencies (condition 6), analyzes a single iteration of the loop and derives a quantified information flow specification from it. $\mathsf{Pre}_{\text{for}}$ takes as input, together with a command $S = \mathbf{for}\ q \leftarrow 1\ \mathbf{to}\ y\ \mathbf{do}\ S_0$, a postcondition $\theta' = \phi \Rightarrow h[A']\ltimes$ where S_0 *preserves* A' holds but S_0 *preserves* h does not hold.

Conditions 3 and 4 ensure that contracts can be expressed, whereas the absence of loop-carried dependencies, as formalized in conditions 5 and 6, ensures the soundness of quantification: we can reason about a single run of the loop and generalize the result, because there is no interdependence among the different iterations. Now, with $\mathsf{Pre}_{\text{for}}$ at hand, we try again with `flipHalves` and obtain the desired specification shown in Fig. 7 (c). If any of the conditions above is not satisfied, then the loop is treated as a `while` loop, and the corresponding rule is applied (which prevents obtaining a quantified information flow precondition).

To better illustrate the algorithm, we have casted the SPARK example shown in Fig. 7(a) into our simple imperative language, to show, step-by-step, how the contract shown in Fig. 7(c) is computed:

Example 2. Consider the following program (shown as SPARK in Fig. 7(a)):

$$\mathbf{for}\ q \leftarrow 1\ \mathbf{to}\ m\ \mathbf{do}\ (t := h[q]\ ; h[q] := h[q+m]\ ; h[q+m] := t)$$

With $J = \{1, 2\}$ we have

$$A_1 = q\ ,\ A_2 = q + m$$

Our algorithm then computes:

$$A'_1 = u\ ,\ A'_2 = u - m$$

which satisfies Condition 3 since $(u = q) \equiv_1 (q = u)$ and $(u = q + m) \equiv_1 (q = u - m)$.

Next, we compute the ranges for expressions:

$$\phi_1 = 1 \leq u \leq m\ ,\ \phi_2 = m + 1 \leq u \leq m + m$$

This satisfies Condition 4 since for all s and for all n,

$$[s \mid u \mapsto n] \models 1 \leq u \leq m\ ,\ \text{iff}\ n \in \{[\![q]\!]_{[s|q \mapsto i]} \mid 1 \leq i \leq s(m)\}$$

```
while i < 7 do
   if odd(i)
    then r := r + v; v := v + h
    else v := x;
   i := i + 1
{r⋉}
```

Iteration	0	1	2	3	
	false	*false*	*false*	*false*	$\Rightarrow$ h⋉
	false	*true*	*true*	*true*	$\Rightarrow$ i⋉
	true	*true*	*true*	*true*	$\Rightarrow$ r⋉
	false	odd(i)	odd(i)	odd(i)	$\Rightarrow$ v⋉
	false	*false*	¬odd(i)	*true*	$\Rightarrow$ x⋉

Fig. 8. Iterative analysis of while loop. (We use $\mathsf{odd}(i)$ as a shorthand for $i \bmod 2 = 1$.)

as both sides reduce to $1 \leq n \leq s(m)$, and also

$$[s \mid u \mapsto n] \models m+1 \leq u \leq m+m \text{ , iff } n \in \{[\![q+m]\!]_{[s|q \rightarrow i]} \mid 1 \leq i \leq s(m)\}$$

as both sides reduce to $s(m)+1 \leq n \leq s(m)+s(m)$.

With S_0 the body of the **for** loop we now compute

$$\{\Theta_1\} \Longleftarrow S_0\ \{h[q]\ltimes\}\ ,\ \{\Theta_2\} \Longleftarrow S_0\ \{h[q+m]\ltimes\}$$

where it is easy to see, computing the preconditions with the logic, that Θ_2 simplifies to $h[q]\ltimes$, and that Θ_1 simplifies – assuming we know that $m \geq 1$ – to $h[q+m]\ltimes$.

The only non-trivial requirement which is left to check is condition 6 which splits into 4 proof obligations, all given s and i, i' with $i, i' \in \{1 \ldots s(m)\}$:

$$\begin{aligned}
[\![q+m]\!]_{[s|q \rightarrow i']} &= [\![q]\!]_{[s|q \rightarrow i]} \text{ implies } i' \leq i \\
[\![q+m]\!]_{[s|q \rightarrow i']} &= [\![q+m]\!]_{[s|q \rightarrow i]} \text{ implies } i' \leq i \\
[\![q]\!]_{[s|q \rightarrow i']} &= [\![q]\!]_{[s|q \rightarrow i]} \text{ implies } i' \leq i \\
[\![q]\!]_{[s|q \rightarrow i']} &= [\![q+m]\!]_{[s|q \rightarrow i]} \text{ implies } i' \leq i
\end{aligned}$$

The second and third obligations are trivially true since the antecedents reduce to $i' + s(m) = i + s(m)$ and to $i' = i$; the first and fourth obligations are vacuously true, given the bounds on i, i', since the antecedents reduce to $i' + s(m) = i$ and to $i' = i + s(m)$. As all requirements are fulfilled, we see that $\mathsf{Pre}_{\mathrm{for}}$ succeeds for the given program. After some simplifications, we end up with the preconditions

$$\begin{aligned}
&1 \leq u \leq m \Rightarrow h[u+m]\ltimes \\
&m \leq u \leq (m+m) \Rightarrow h[u-m]\ltimes \\
&(1 > u) \vee (u > m+m) \Rightarrow h[u]\ltimes \\
&m\ltimes
\end{aligned}$$

which is exactly what we would expect, and which is listed in Fig. 7(c).

The $\mathsf{Pre}_{\mathrm{while}}$ Function

We shall now present the function $\mathsf{Pre}_{\mathrm{while}}$, which is much as in [5]. As seen in Fig. 6, it takes as input, together with a command $S = \mathbf{while}\ B\ \mathbf{do}\ S_0\ \mathbf{od}$, a postcondition Θ' where each $\theta' \in \Theta'$ is of the form $\phi \Rightarrow w\ltimes$ where w is a scalar variable or an array variable.

The idea is to consider assertions of the form $\phi_w \Rightarrow w\ltimes$ and then repeatedly analyze the loop body so as to iteratively weaken the antecedents until a fixed point is reached. To illustrate the overall behavior, consider the example in Fig. 8 where we are given r⋉ as postcondition; hence the initial value of r's antecedent is *true* whereas all other

antecedents are initialized to *false*. The first iteration updates `v`'s antecedent to $\texttt{odd}(\texttt{i})$, since `v` is used to compute `r` when `i` is odd, and also updates `i`'s antecedent to *true*, since (the parity of) `i` is used to decide whether `r` is updated or not. The second iteration updates `x`'s antecedent to $\neg\texttt{odd}(\texttt{i})$, since in order for two runs to agree on `v` when `i` is odd, they must have agreed on `x` in the previous iteration when `i` was even. The third iteration updates `x`'s antecedent to *true*, since in order for two runs to agree on `x` when `i` is even, then must agree on `x` always (as `x` doesn't change). We have now reached a fixed point.

It is noteworthy that even though the postcondition mentions $\texttt{r}\ltimes$, and `r` is updated using `v` which in turn is updated using `h`, the generated precondition does not mention `h`, since the parity of `i` was exploited. This shows [3] that even if we should only aim at producing contracts where all assertions are unconditional, precision may still be improved if the analysis engine makes internal use of *conditional* assertions.

To ensure termination in the general case, we need a "widening operator" $\bigtriangledown$ on 1-assertions, with the following properties:

(a) for all ϕ and ψ, ψ logically implies $\psi \bigtriangledown \phi$, and also ϕ logically implies $\psi \bigtriangledown \phi$;
(b) if for all i we have that ϕ^{i+1} is of the form $\psi \bigtriangledown \phi^i$, then the chain $\{\phi^i \mid i \geq 0\}$ eventually stabilizes.

A trivial widening operator is the one that always returns *true*, in effect converting conditional agreement assertions into unconditional. A less trivial option will utilize a number of assertions, say $\psi_1 \ldots \psi_k$, and allow $\psi \bigtriangledown \phi = \psi_j$ if ψ_j is logically implied by ψ as well as by ϕ; such assertions may be given by the user if he has a hint that a suitable invariant may have one of $\psi_1 \ldots \psi_k$ as antecedent. Our implementation attempts to infer these invariants by using disjunction as a widening operator which returns *true* if convergence is not achieved after a certain number of iterations. For more formal details about this algorithm, we refer the reader to [5].

5 Experimental Assessment

To assess the ideas presented in this paper, we have developed an implementation that checks and infers information flow contracts for SPARK using our more precise enhanced contract language. The algorithm extends our implementation for conditional contracts described in [5] to support arrays, universally quantified flow contracts, and precise processing of `for` loops as detailed in previous sections.

We tested this implementation on an information assurance application (a MILS Message Router) that presents a number of challenges due to its extensive use of arrays, a collection of embedded applications (an Autopilot, a Minepump, a Water Boiler monitor, and a Missile Guidance system – all developed outside of our research group), and a collection of small programs that we developed ourselves to highlight common array idioms that we discovered in information assurance applications. We provide a more detailed assessment of the MMR example after summarizing the results of the experiments and illustrating the following array idiom examples (see Fig. 9).

- **ArrayInit:** A procedure that initializes all elements of an array to a particular value.
- **ArrayScrub:** A procedure that re-writes the replaces the elements of an array that satisfy a predetermined condition, with a particular value.
- **ArrayTransfer:** A procedure that transfer the elements from one array to another.
- **ArrayPartitionedTransfer:** Similar to the previous one except that the transfer from one array to the other is done only within certain partitions (ranges) defined in each array.

```
procedure ArrayInit
--# global out A(*);
--# derives for all I in A.Range => (A(I) from {});
is
begin
  for I in A.Range loop
    A(I) := 0;
  end loop;
end ArrayInit;
```

```
procedure ArrayScrub
--# global in Scrub_Constant,
--#         out A(*);
--# derives for all I in A.Range =>
--#                (A(I) from Scrub_Constant
--#                 when should_scrub(A(I)));
is
 begin
    for I in A.Range loop
      if should_scrub(A(I)) then
        A(I) := Scrub_Constant;
      end if;
    end loop;
end ArrayScrub;
```

```
procedure ArrayTransfer
--# global in B(*),
--#         out A(*);
--# derives for all I in A.Range => (A(I) from B(I));
is
begin
  for I in A.Range loop
    A(I) := B(I);
  end loop;
end ArrayTransfer;
```

```
procedure ArrayPartitionedTransfer
--# global in B(*), C(*), K,
--#         out A(*);
--# derives for all I in range
--#                A'First .. K => (A(I) from B(I)) &
--#          for all I in range
--#                K+1 .. A'Last => (A(I) from C(I-K));
is
 begin
    for I in range A'First .. K loop
      A(I) := B(I);
    end loop;

    for I in range k+1 .. A'Last loop
      A(I) := C(I-K);
    end loop;
end ArrayPartitionedTransfer;
```

Fig. 9. Information flow contracts inferred by our implementation for a selection of examples.

In each of these examples, using original SPARK contracts/analysis would have allowed us to specify only that information is flowing from one entire array to another. Fig. 9 illustrates how our conditional and quantified contracts allow a much more precise verified specification of the flows.

A total of **66 procedures** were analyzed, and information flow contracts were inferred for all of them, each taking **less than two seconds for each** to run on a Core 2 Duo 2.2GHz processor and 3 GB of RAM. Of these procedures, ten included array manipulations that tested our new extensions to the logic. In all of these cases, our implementation generates a quantified information flow specification showing the dependence dynamics in the arrays.

The MMR Example: The MMR (MILS Message Router) is an idealized version of a MILS infrastructure component (first proposed by researchers at the University of Idaho [26]) designed to mediate communication between partitions in a *separation kernel* [27] – the foundation of specialized real-time platforms used in security contexts to provide strong data and temporal separation.

Fig. 10 illustrates a set of partition processes that execute in a static round-robin schedule. During each schedule cycle, each process is allowed to post up to one bounded-size message to each of the other partitions and receive messages from partitions sent during the previous cycle. Different partitions do not communicate directly. Instead, they post messages to the MMR, which only propagates a message if it conforms to a static security policy represented by a two dimensional boolean array *Policy* indexed by process IDs. In Fig. 10, a shaded square (representing the value *True*) in the *Policy* array indicates that the row process (*e.g.*, B) is allowed to send messages to the column process (*e.g.*, D). The figure illustrates that unidirectional communication can be enforced (*e.g.*, D is not allowed to send messages to B).

During the posting, the infrastructure attaches an unspoofable header to the message indicating the ID of sender process and the ID of the destination process. The MMR places each posted message in a pool of shared *Memory* slots (represented as an array of messages), and updates *Pointers* (represented as a two-dimensional array of indices into *Memory*) that organizes incoming/outgoing messages. During posting, a *Memory* cell indexed by row *A*, column *B* holding pointer *X* indicates that the memory location pointed to by *X* is "owned" by process *A* and holds a message from process *A* destined

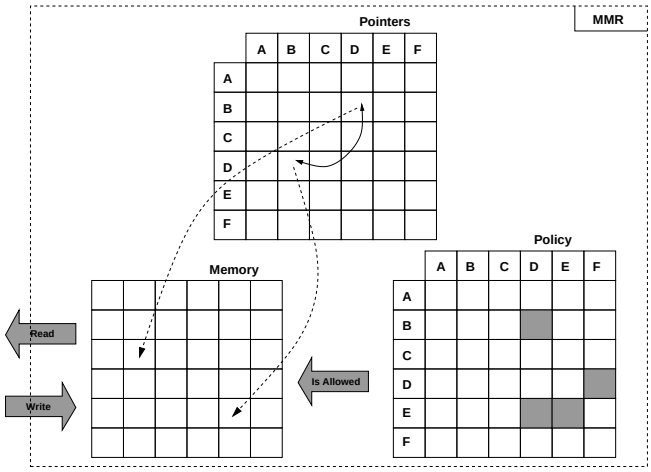

Fig. 10. Diagram of the MILS Message Router.

for process *B*. Entries in Flags (an array of boolean values with the same dimensions as *Pointers*) indicate if the corresponding entry in *Pointers* represents a valid message or a "place holder" default message that will not be propagated by the MMR.

Fig. 11 (a) displays the SPARK code for procedure `Route` that implements part of the MMR routing phase. Conceptually, messages are routed by swapping `Pointers` entries. Before `Route` is executed, the array of pointers points to *outgoing* messages, whereas after routing it points to *incoming* messages. Specifically, after routing, a `Memory` cell indexed by `Pointers` row *A*, column *B* holding pointer *X* indicates that the memory location pointed to by *X* is "owned" by process *A* and holds a message from process *B* sent to process *A*. For any two processes *A* and *B*, the first two conditional blocks in `Route` determine if messages from *A* and *B* (and vice versa) are allowed by the security policy. If a message is not allowed, then the memory location holding it is cleared with a default message and the corresponding `Flags` entry is set to *false*. Then, if there remains a valid message flowing in either direction, `Route` swaps the `Memory` cell indices in `Pointers` so that the ownership between the memory locations is exchanged among the processes (note that if a message is allowed in one direction but not the other, the swap will propagate a default message in the "disallowed" direction).

As stated before, initially the array of pointers indexes the regions of memory holding outgoing messages. The way that the procedure `Route` forwards the messages is by exchanging ownership of those memory locations between processes. For any two procesees, say *A* and *B*, `Route` checkes whether there is a message outgoing between *A* and *B*, in any direction, and whether it doesn't violate the security policy. If there is a message that is not allowed by the security policy, then the memory location holding it is cleared with a default message. If after the pre-check there's still a message to be forwarded, `Route` swaps the pointers between in the array of pointers, so that the ownership between the memory locations is exhanged among the processes. In effect, the array cell indexed by column*A* and row *B* now holds a pointer to a memory location that contains a message from *B* to *A*, and is owned by process *A*. The process is such that before `Route` is executed, the array of pointers points to *outgoing* messages, and after the routing is done it points to *incoming* messages.

The idea behind the MMR is that of a system that functions as a mediator in the communication between processes in an operating system environment. In this context, processes communicate sending/receiving messages to/from each other, and all messages have to go through the MMR. At the same time, the message routing is performed based on a security policy.

The architecture we chose for our implementation is a slight modification from the one proposed in [26], and as mentioned before, it is shown in Fig. 10. The communication policy is very simple and simply states whether a process is allowed to send messages to another particular process (or whether the latter process is allowed to receive messages from the former). The policy is sensitive to the direction of communication, that is, if process A is not allowed to send messages to process B, it doesn't mean that process B is not allowed to send messages to process A. As shown in Fig. 10, this policy is implemented as a 2-dimensional array. In the figure, a shaded square indicates that the row process is allowed to send messages to the column process. Similarly, in our implementation, we simulate memory as array. Messages between processes in the system are stored in memory. When a process requests the MMR to send a message, the message is put in an available memory space, and the process is assigned a pointer to that memory location, which is stored in another array, labeled *Pointers* in Fig. 10.

The array of pointers is another 2-dimensional array, similar to the *Policy* array. However, instead of denoting whether communication between particular processes is allowed, this array holds the pointers that denote the memory locations containing the messages. This array basically summarizes the memory ownership by the processes of the portion of memory that holds the messages. So, if the cell indexed by column *A*, row *B*, holds pointer *X*, then it means that the memory location pointed to by *X*, is owned by process *A*, and holds (at the beginning of the routing cycle) a message from process *A* to process *B*.

The way the routing of messages works is through a process of pointers-swapping on the array of pointers. The code for the routing procedure is presented in Fig. 11. As stated before, initially the array of pointers indexes the regions of memory holding outgoing messages. The way that the procedure `Route` forwards the messages is by exchanging ownership of those memory locations between processes. For any two procesees, say *A* and *B*, `Route` checkes whether there is a message outgoing between *A* and *B*, in any direction, and whether it doesn't violate the security policy. If there is a message that is not allowed by the security policy, then the memory location holding it is cleared with a default message. If after the pre-check there's still a message to be forwarded, `Route` swaps the pointers between in the array of pointers, so that the ownership between the memory locations is exhanged among the processes. In effect, the array cell indexed by column*A* and row *B* now holds a pointer to a memory location that contains a message from *B* to *A*, and is owned by process *A*. The process is such that before `Route` is executed, the array of pointers points to *outgoing* messages, and after the routing is done it points to *incoming* messages.

There are multiple reasons why it is very difficult to verify statically that the MMR conform to the end-to-end information flow policy as captured by the *Policy* matrix. First, the examples of Section 2 illustrated the difficulties of statically reasoning about individual cells of an array, and, in the MMR, invalid message channels are "squelched" by clearing out (with a default message) individual cells within a large array. Second, the information flow path between two partitions is not implemented via direct reference to source and destination memory cells, but instead involves a level of indirection via the *Pointers* array. Third, the information flow path through the MMR between two

```
procedure Route
--# global in Policy.Comm_Policy;
--#        in out Flags, Pointers, Memory.Mem_Space;
--# derives Pointers from *, Policy.Comm_Policy, Flags &
--#          Memory.Mem_Space from *, Policy.Comm_Policy,
--#              Pointers, Flags &
--#          Flags from *, Policy.Comm_Policy;
is
  T : Lbl_t.Pointer;
begin
 for I in Lbl_t.Proc_ID loop
  for J in Lbl_t.Proc_ID range
   I .. Lbl_t.Proc_ID'Last loop
   if not Policy.Is_Allowed(I,J) then
    Memory.Write(Msg_t.Def_Msg, Pointers(I,J));
    Flags(I,J) := FALSE;
   end if;
   if not Policy.Is_Allowed(J,I) then
    Memory.Write(Msg_t.Def_Msg, Pointers(J,I));
    Flags(J,I) := FALSE;
   end if;
   if Flags(I,J) or Flags(J,I) then
    T := Pointers(I,J);
    Pointers(I,J) := Pointers(J,I);
    Pointers(J,I) := T;
   end if;
  end loop;
 end loop;
end Route;
```

(a)

```
procedure Route
--# global in Policy.Comm_Policy;
--#        in out Flags, Pointers, Memory.Mem_Space;
--# derives for all I in Lbl_t.Proc_ID => (
--#           for all J in Lbl_t.Proc_ID => (
--#             Pointers(I,J) from
--#               Pointers(J,I) when
--#                 (Policy.Is_Allowed(I,J
--#                       and (Flags(I,J))
--#                    or (Policy.Is_Allowed(J,I)
--#                       and Flags(J,I)),
--#               * when
--#                 (not (Policy.Is_Allowed(I,J)
--#                       and Flags(I,J))) and
--#                    (not (Policy.Is_Allowed(J,I)
--#                       and Flags(J,I))) &
--#         for all I in Lbl_t.Proc_ID => (
--#           for all J in Lbl_t.Proc_ID => (
--#             Memory.Mem_Space(Pointers(I,J)) from
--#               {Msg_t.Def_Msg} when
--#                 not Policy.Is_Allowed(I,J),
--#               * when
--#                 Policy.Is_Allowed(I,J))) &
--#         for all I in Lbl_t.Proc_ID => (
--#           for all J in Lbl_t.Proc_ID => (
--#             Flags(I,J) from
--#               {FALSE} when
--#                 not Policy.Is_Allowed(I,J),
--#               * when
--#                 Policy.Is_Allowed(I,J)));
```

(b)

Fig. 11. Source code and initial specification for procedure `Routing` of the MILS Message Router (a), and information flow specification for the same procedure using extended specification and analysis techniques for arrays (b).

partitions is not static (*e.g.*, as is the case for information flow between two variables of scalar type), but it is changing – information for the same conceptual path flows through different *Memory* cells whose "ownership" changes on different iterations.

As anticipated, Figure 11 (a) illustrates that the original SPARK annotations for `Route` are far too imprecise to support verification of the desired end-to-end policy. For example, the `derives` clause for `Pointers` states that the final value of the array is derived from its initial value (`*`), from the communication policy (`Policy.Comm_Policy`), and from the array of flags (`Flags`). The problem here is that the forced abstraction of `Pointers` array cells into a single atomic entity collapses the individual *allowed* inter-partition information flow channels (where we needed to verify *separation of channels*) and does not capture the fact that some inter-partition flows are *disallowed*. Furthermore, we have lost information about the specific conditions of the `Policy` that enable or disable corresponding flows in `Pointers`. Finally, without precise accounting of flows for `Pointers`, it is impossible to get a handle on what we are most interested in: flows of the actual messages through `Memory`.

We focus our attention on the annotations and identify the problems we are addressing in this work. From the information flow perspective we are mainly concerned with the `derives` annotations, and sice we have already discussed that the routing of messages is done by swapping pointers, we will focus our attention on the information flow specification for the array `Pointers`. The `derives` annotation presented in Fig. 11 corresponds to the information flow specification inferred by our previous algorithm, as presented in [5]. The `derives` clause for `Pointers` states that the procedure modifies the array such that its value is derived from itself (`*`), from the communication policy (`Policy.Comm_Policy`), and from the array of flags (`Flags`). The problem here is that the information flow channels are all collapsed. We have lost information about which variables directly affect the final value of `Pointers` and under what conditions that happens. Furthermore, although we know that the array `Pointers` derives its final value from itself (as there is inter-flow between the array's internal locations), we have

lost information about the different information flow channels within the array: given a particular array location, from what other location is it getting its final value from?

The limitations with the information flow specification inference described above are caused by two separate issues. First, there is currently no way in SPARK to specify information flows for a particular array location. Instead, arrays are treated like a single variable, and all flows into the array are collapsed into a single one. This approach is too coarse, in particular for our example where we would like to understand the interflows between array locations. The other problem is the loops: even if we were able to specify flows at the level of particular array locations, the conditional flows generated would most likely be lost once a suitable invariant is seached for the outer loops, and a widening process is applied.

Fortunately applying the theory and techniques discussed so far in this paper, we are able to perform a much more precise and detailed analysis and specification of this procedure. Figure 11(b) shows the result of applying the techniques developed in this work to the procedure `Route`. The new specification uses the extended syntax with quantification operators, and completely grasps the underlying conditional information flow policy for this procedure and the MMR system. Keep in mind that we are able to obtain such a detailed information flow specification, describing the information dynamics of the procedure's loops, because both loops are *loop-carried-dependence-free*, and so we are able to reason about each iteration on its own, which in turn enables the seamless introduction of quantifications.

Figure 11 (b) displays a contract in our extended contract language that is automatically inferred using the precondition generation algorithm of the preceding section. The `derives` clause for `Pointers` uses nested quantification (derived from the nested loop structure) to capture the "swapping" action of `Route`. Moreover, it includes the conditions under which the swapping occurs or under which `Pointers(I,J)` retains its value. The `Memory derives` clause correctly captures the fact that the cell holding an outgoing message is "cleared" with the default message when the policy disallows communication between the sender and destination (the `derives` clause for `Flags` has a similar structure).

6 Related Work

The first theoretical framework for SPARK information flow is provided by Bergeretti and Carré [10] who present a compositional method for inferring and checking dependencies among variables. That approach is flow-sensitive, unlike most security type systems [32, 6] that rely on assigning a security level ("high" or "low") to each variable. Chapman and Hilton [12] describe how SPARK information flow contracts could be extended with lattices of security levels and how the SPARK Examiner could be enhanced correspondingly. Rossebo *et al.*[26] show how the existing SPARK framework can be applied to verify various *unconditional* properties of a MILS Message Router. Apart from Spark Ada, there exists several tools for analyzing information flow, notably Jif (Java + information flow) which is based on [22]), and FlowCaml [29].

The seminal work on (inherently flow-sensitive) agreement assertions is [2] which comes with an algorithm for computing (weakest) preconditions, but the approach does not integrate with programmer assertions. To address that, and to analyze heap-manipulating languages, the logic of [1] employs *three* kinds of primitive assertions: agreement, programmer, and region (for a simple alias analysis). But, since those can be combined only through conjunction, programmer assertions are not smoothly integrated, and one cannot capture *conditional* information flows. This motivated Amtoft

& Banerjee [3] to introduce conditional agreement assertions (for a heap-manipulating language); in [5] that approach was applied to the (heap-free) SPARK setting and worked out extensively, with an algorithm for computing loop invariants and with reports from an implementation. All of these works treat arrays as indivisible entities.

Reasoning about individual array elements is desirable for the precise analysis of a loop that traverses an array. We have established syntactic and semantic conditions for when we can allow such fine-grained analysis; these conditions include what is essentially the absence of loop-carried dependencies. This suggests a relationship to the body of work, with [25] as a seminal paper, addressing when loops can be parallellized. Our conditions are more permissive though since they allow a location to be read *before* it is written, as for the loop body $h[q] := h[q+1]$ (whereas we do not allow $h[q+1] := h[q]$).

Rather than designing a specific logic for information flow, one can employ general logic as does the recently popular *self-composition* technique, first proposed by Barthe et al. [9] and later extended by, e.g., Terauchi and Aiken [31] and (for heap-manipulating programs) Naumann [23]. The information flow property which we encode as $\{x\ltimes\}\ S\ \{y\ltimes\}$ would in those papers be encoded as $\{x = x'\}\ S; S'\ \{y = y'\}$ where S' is a copy of S with all variables renamed (primed); such a property can be checked using existing static verifiers. Alternatively, one could use dynamic logic [13].

When it comes to *conditional* information flow, the most noteworthy existing tool is the slicer by Snelting et al [30] which generates *path conditions* in program dependence graphs for reasoning about end-to-end flows between specified program points/variables. In contrast, we provide a contract-based approach for *compositional* reasoning about conditions on flows with an underlying logic representation that can provide external evidence for conformance to conditional flow properties. We plan to investigate the deeper technical connections between the two approaches.

Ground-breaking efforts in certification of MILS infrastructure [17, 19] have used approaches in which source code has been hand-translated into executable models in theorem provers such as ACL2 and PVS. While the direct theorem-proving approach followed in these efforts enables proofs of very strong properties beyond what our framework can currently handle, our aim is to dramatically reduce the labor required, and the potential for error, by integrating automated techniques directly on code, models, and developer workflows to allow many information flow verification obligations to be discharged earlier in the life cycle.

7 Correctness Proof

In this section we shall present some key steps in a proof that (a slightly modified[3] version of) the Algorithm in Fig. 6 is correct. We plan to fill in all details soon.

As in our previous works, we need the following

Lemma 2. *: Assume that* $\{\Theta\} \Longleftarrow S\ \{\Theta'\}$ *. For all* $\phi' \Rightarrow \_\ltimes \in \Theta'$*, there exists* $\phi \Rightarrow \_\ltimes \in \Theta$ *such that whenever* $s\ [\![S]\!]\ s'$ *and* $s' \models \phi'$ *then* $s \models \phi$*.*

Observe that it is easy to modify Fig. 6 so that Lemma 2 trivially holds, for example by adding *true* $\Rightarrow 0\ltimes$ to all preconditions, but the analysis of a command may become less precise if the analysis of a subcommand is augmented in that way.

[3] Working out the proof in detail revealed some inaccuracies in Fig. 6; these will be highlighted in footnotes along the way.

Proof. Induction in S, where we do a case analysis on the form of S. So far, we consider only the case where S is a conditional $\mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2$. Given $\phi' \Rightarrow \_\ltimes \in \Theta'$, we can inductively assume that for $i = 1, 2$ there exists $\phi_i \Rightarrow \_\ltimes \in \Theta_i$ such that if $s_i\ [\![S_i]\!]\ s_i'$ and $s_i' \models \phi'$ then $s_i \models \phi_i$. Now assume that $s\ [\![S]\!]\ s'$ and $s' \models \phi'$. If $s \models B$ then $s\ [\![S_1]\!]\ s'$ and thus $s \models \phi_1$; if $s \models \neg B$ then $s\ [\![S_2]\!]\ s'$ and thus $s \models \phi_2$. We infer that in all cases, we have $s \models (\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)$. This yields the claim, since Θ will always contain an assertion with antecedent $(\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)$, no matter whether S_1 *preserves* E and S_2 *preserves* E holds or not.

We are now ready for the main theorem.

Theorem 2 (Correctness of Pre**).** *For all S, Θ, Θ', if $\{\Theta\} \Longleftarrow S\ \{\Theta'\}$ holds, then $\{\Theta\}S\{\Theta'\}$ holds.*

That is, for all s, s', σ, σ', if $s\ [\![S]\!]\ s'$ and $\sigma\ [\![S]\!]\ \sigma'$, and also $s\&\sigma \models \Theta$, then $s'\&\sigma' \models \Theta'$.

The proof is by induction in S, where we do a case analysis on the form of S; in some cases, we need auxiliary results to be proved along the way. So far, we consider only the case where S is a conditional, and also the case where S is a `for`-loop where Θ' is of the form $\{\phi \Rightarrow h[A]\ltimes\}$ and $\mathsf{Pre}_{\mathrm{for}}$ succeeds on $h[u]\ltimes$. In general, we shall assume the terminology of Fig. 6.

Conditionals. First assume that S is a conditional $\mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2$. Let $\phi \Rightarrow E\ltimes \in \Theta'$, let $s\ [\![S]\!]\ s'$ and $\sigma\ [\![S]\!]\ \sigma'$, let $s\&\sigma \models \Theta$, and assume that $s' \models \phi$ and $\sigma' \models \phi$ so as to prove $[\![E]\!]_{s'} = [\![E]\!]_{\sigma'}$. By Lemma 2, for $i = 1, 2$ there exists $\phi_i \Rightarrow E_i\ltimes \in \Theta_i$ such that if $s_i\ [\![S_i]\!]\ s_i'$ and $s_i' \models \phi$ then $s_i \models \phi_i$. Let $\phi_0 = (\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)$. From $s\ [\![S]\!]\ s'$ we see that either $s \models B$ and $s\ [\![S_1]\!]\ s'$ so that $s \models \phi_1$, or $s \models \neg B$ and $s\ [\![S_2]\!]\ s'$ so that $s \models \phi_2$; in either case, $s \models \phi_0$. Similarly, $\sigma \models \phi_0$. There are now two cases.

First assume that S_1 *preserves* E and S_2 *preserves* E both hold; thus S *preserves* E also holds. Then $\phi_0 \Rightarrow E\ltimes \in \Theta$, so from $s\&\sigma \models \Theta$ and $s \models \phi_0$ and $\sigma \models \phi_0$, we get $[\![E]\!]_s = [\![E]\!]_\sigma$. But since $[\![E]\!]_s = [\![E]\!]_{s'}$ and $[\![E]\!]_\sigma = [\![E]\!]_{\sigma'}$, we infer the desired $[\![E]\!]_{s'} = [\![E]\!]_{\sigma'}$.

Now assume that S_1 *preserves* E and S_2 *preserves* E do not both hold. Then $\phi_0 \Rightarrow B\ltimes \in \Theta$, so from $s\&\sigma \models \Theta$ and $s \models \phi_0$ and $\sigma \models \phi_0$, we get $[\![B]\!]_s = [\![B]\!]_\sigma$. Assume, wlog, that $[\![B]\!]_s = [\![B]\!]_\sigma = \mathsf{False}$. Thus $s\ [\![S_2]\!]\ s'$ and $\sigma\ [\![S_2]\!]\ \sigma'$, and also $s \models \phi_2$ and $\sigma \models \phi_2$.

It is sufficient to show $s\&\sigma \models \Theta_2$ since then our induction hypothesis, applied to S_2, gives the desired $s'\&\sigma' \models \phi \Rightarrow E\ltimes$. Therefore consider $\phi_2 \Rightarrow E_2\ltimes \in \Theta_2$, so as to prove $[\![E_2]\!]_s = [\![E_2]\!]_\sigma$. But since $\phi_2 \wedge \neg B \Rightarrow E_2\ltimes \in \Theta$, this follows from $s\&\sigma \models \Theta$ and $s \models \phi_2 \wedge \neg B$ and $\sigma \models \phi_2 \wedge \neg B$.

For Loops We first need to establish a lemma about $\mathsf{Pre}_{\mathrm{for}}$:

Lemma 3. *Let S be* $\mathbf{for}\ q \leftarrow 1\ \mathbf{to}\ m\ \mathbf{do}\ S_0$. *Assume that calling* $\mathsf{Pre}_{\mathrm{for}}$ *on S with postcondition*[4] *$h[u]\ltimes$ succeeds, with result Θ. Then for all integer constants c we have $\{\Theta[c/u]\}S\{h[c]\ltimes\}$.*

[4] Our original figure inexplicably provided $h[u]\ltimes$ with an antecedent ϕ.

Proof. First recall that the following (sets of) assertions are in Θ:

$$\begin{aligned}
\textsf{UPDATED}: &\ \{\phi_j \Rightarrow \Theta_j[A'_j/q]\ltimes \mid j \in J\}\\
\textsf{OUTSIDE}: &\ (\wedge_{j\in J}\neg\phi_j) \Rightarrow h[u]\ltimes\\
\textsf{INDEX}: &\ \{w\ltimes \mid \exists j \in J : w \in \mathrm{fv}(A_j) \setminus \{q\}\}\\
\textsf{BOUND}: &\ \{m\ltimes\}
\end{aligned}$$

where we have the following entities and conditions:

1. J ranges over the set of assignments to h in S_0, with each such assignment being of the form $h[A_j] := E$ for some $j \in J$
2. for each procedure p called within S_0 we have p *preserves* h.
3. for each $j \in J$, S_0 *preserves* A_j.
4. for each $j \in J$, A'_j is such that $\mathrm{fv}(A'_j) \subseteq \mathrm{fv}(A_j) \cup \{u\} \setminus \{q\}$ and [5] such that for all s with $dom(s) \subseteq \mathrm{fv}(A_j)$, and for all natural numbers n:
$$[\![n = A_j]\!]_s = [\![q = A'_j[n/u]]\!]_s.$$
5. for each $j \in J$, ϕ_j is such that $\mathrm{fv}(\phi_j) \subseteq \mathrm{fv}(A_j) \cup \{u\} \setminus \{q\}$ and such that for all s with $dom(s) \subseteq \mathrm{fv}(A_j)$, and for all natural numbers n:
$$n \in \{[\![A_j]\!]_{[s|q\mapsto i]} \mid i \in 1..s(m)\} \text{ iff } s \models \phi_j[n/u].$$
6. for each $j \in J$, Θ_j is such that
$$\{\Theta_j\} \Longleftarrow S_0 \ \{h[A_j]\ltimes\}\ .$$
7. for each $j \in J$, if $w \in \mathrm{fv}(\Theta_j)$ with $w \neq h$ then S_0 *preserves* w.
8. for each $j \in J$, if h occurs in Θ_j then it is in the context[6] $h[A]$, where for all[7] $j_1 \in J$, all s, all $i, i' \in 1..s(y)$:
   if $[\![A]\!]_{[s|q\mapsto i']} = [\![A_{j_1}]\!]_{[s|q\mapsto i]}$ then $i' \leq i$.

So consider s, σ such that $s\&\sigma \models \Theta[c/u]$, or equivalently $[s \mid u\mapsto c]\&[\sigma \mid u\mapsto c] \models \Theta$, and such that $s\ [\![S]\!]\ s'$ and $\sigma\ [\![S]\!]\ \sigma'$, so as to prove $s'\&\sigma' \models h[c]\ltimes$. Since by BOUND we have $m\ltimes \in \Theta$, there exists M such that $s(m) = \sigma(m) = M$.

From the semantics of the `for` loop, as presented in Fig. 4, we see that for $i \in 0..M$ there exists s_i with the following properties:

- $s_0 = s$, and
- for $i \in 1..M$, s_i is such that $[s_{i-1} \mid q\mapsto i]\ [\![S_0]\!]\ s_i$.

Also, we have $s' = [s_M \mid q\mapsto M+1]$. Similarly, for $i \in 0..M$ there exists σ_i such that $\sigma_0 = \sigma$, and for $i \in 1..M$, σ_i is such that $[\sigma_{i-1} \mid q\mapsto i]\ [\![S_0]\!]\ \sigma_i$.

Observe that by INDEX we have $[\![A_j]\!]_{[s|q\mapsto i]} = [\![A_j]\!]_{[\sigma|q\mapsto i]}$ for all $j \in J$ and all $i \in 1..M$, and by item 3 we have $[\![A_j]\!]_{[s|q\mapsto i]} = [\![A_j]\!]_{[s_{i'}|q\mapsto i]}$ for all $i' \in 0..i-1$. With a similar result for σ, for all $i', i'' \in 0..i-1$ we thus have

$$[\![A_j]\!]_{[s_{i'}|q\mapsto i]} = [\![A_j]\!]_{[\sigma_{i''}|q\mapsto i]}.$$

[5] We added the condition on $dom(s)$, for A'_j and for ϕ_j.
[6] We added the condition that h must occur in context $h[A]$.
[7] Our original figure inexplicably assumed that $j_1 = j$.

We can therefore split into two cases, depending on whether the given c can occur as an update index or not.

The first case is where for all $i \in 1..M$, and all $j \in J$,

$$[\![A_j]\!]_{[s_{i-1}|q\mapsto i]} = [\![A_j]\!]_{[\sigma_{i-1}|q\mapsto i]} \neq c.$$

In that case, by item 2, we can infer

$$[\![h[c]]\!]_s = [\![h[c]]\!]_{s'} \text{ and } [\![h[c]]\!]_\sigma = [\![h[c]]\!]_{\sigma'}. \tag{1}$$

Also, since $c \notin \{[\![A_j]\!]_{[s|q\mapsto i]} \mid i \in 1..M\}$, we by item 5 infer that for no $j \in J$ do we have $s \models \phi_j[c/u]$. Equivalently, $[s \mid u \mapsto c] \models \phi_j$ holds for no $j \in J$. Thus we have

$$[s \mid u \mapsto c] \models \wedge_{j\in J} \neg\phi_j$$

and similarly $[\sigma \mid u \mapsto c] \models \wedge_{j\in J}\neg\phi_j$. Since $[s \mid u \mapsto c]\&[\sigma \mid u \mapsto c] \models \Theta$, we from OUTSIDE infer $[s \mid u \mapsto c]\&[\sigma \mid u \mapsto c] \models h[u]\ltimes$ and thus $s\&\sigma \models h[c]\ltimes$. By (1), this implies the desired $s'\&\sigma' \models h[c]\ltimes$.

The second case is where there exists $i \in 1..M$ and $j \in J$ such that

$$[\![A_j]\!]_{[s_{i-1}|q\mapsto i]} = [\![A_j]\!]_{[\sigma_{i-1}|q\mapsto i]} = c.$$

(Notice that this does not necessarily imply that $h[c]$ was written in the i'th iteration, as the assignment $h[A_j] := \_$ could have been inside a non-executed branch of a conditional.) Let I be the *largest* such i, and let j be such that $[\![A_j]\!]_{[s_{I-1}|q\mapsto I]} = [\![A_j]\!]_{[\sigma_{I-1}|q\mapsto I]} = c$. Since $[\![h[c]]\!]_{s_I} = [\![h[c]]\!]_{s'}$ and $[\![h[c]]\!]_{\sigma_I} = [\![h[c]]\!]_{\sigma'}$, it suffices to prove $s_I\&\sigma_I \models h[c]\ltimes$ which by our case assumption, and item 3, amounts to $s_I\&\sigma_I \models h[A_j]\ltimes$. But since $[s_{I-1} \mid q \mapsto I]\ [\![S_0]\!]\ s_I$ and $[\sigma_{I-1} \mid q \mapsto I]\ [\![S_0]\!]\ \sigma_I$, this will follow from the overall induction hypothesis applied to item 6, assuming that we can show

$$[s_{I-1} \mid q \mapsto I]\&[\sigma_I \mid q \mapsto I] \models \Theta_j. \tag{2}$$

We now embark on showing (2). Since $c \in \{[\![A_j]\!]_{[s|q\mapsto I]} \mid i \in 1..M\}$, we by item 5 infer $s \models \phi_j[c/u]$ which amounts to $[s \mid u \mapsto c] \models \phi_j$. Similarly, $[\sigma \mid u \mapsto c] \models \phi_j$. Since $[s \mid u \mapsto c]\&[\sigma \mid u \mapsto c] \models \Theta$, from UPDATED we infer $[s \mid u \mapsto c]\&[\sigma \mid u \mapsto c] \models \Theta_j[A'_j/q]$ which (as $u \notin \mathrm{fv}(\Theta_j)$) amounts to

$$[s \mid q \mapsto [\![A'_j]\!]_{[s|u\mapsto c]}]\&[\sigma \mid q \mapsto [\![A'_j]\!]_{[\sigma|u\mapsto c]}] \models \Theta_j. \tag{3}$$

Since $c = [\![A_j]\!]_{[s|q\mapsto I]}$ we have $[\![c = A_j]\!]_{[s|q\mapsto I]} = \mathsf{True}$, and by item 4 thus also $[\![q = A'_j[c/u]]\!]_{[s|q\mapsto I]} = \mathsf{True}$. Hence $I = [\![A'_j[c/u]]\!]_{[s|q\mapsto I]}$, which amounts to $I = [\![A'_j]\!]_{[s|u\mapsto c]}$. Using a similar result for σ, we can thus simplify (3) to

$$[s \mid q \mapsto I]\&[\sigma \mid q \mapsto I] \models \Theta_j.$$

This enables us to show (2) provided we can prove that Θ_j is not modified by iteration $1..I-1$. To be more precise, we shall prove by structural induction that if E is a subpart of Θ_j and $i \in 1..I-1$ then

$$[\![E]\!]_{[s_{i-1}|q\mapsto I]} = [\![E]\!]_{[s_i|q\mapsto I]}.$$

By item 8, it is sufficient to prove the claim for $E \neq h$.

Most cases are straightforward applications of the induction hypothesis. For a variable w, we must have $w \neq h$ and then the claim follows from our item 7. The interesting case is when we are dealing with an array access $h[A]$. Inductively, we can assume that there is n such that $[\![A]\!]_{[s_{i-1}|q\rightarrow I]} = [\![A]\!]_{[s_i|q\rightarrow I]} = n$. Now assume, to get a contradiction, that $h[n]$ is modified in the i'th iteration. That is, there exists $j' \in J$ and $i \in 1..I-1$ such that $[\![A_{j'}]\!]_{[s_{i-1}|q\rightarrow i]} = n$. By item 8 we now infer $I \leq i$, but since we assumed $i \in 1..I-1$, this yields the desired contradiction.

This concludes the proof of Lemma 3.

We shall now consider the clause[8] for `for`-loops, but will for now only consider the special (but very typical) situation where with $S = \mathbf{for}\ q \leftarrow 1\ \mathbf{to}\ m\ \mathbf{do}\ S_0$ we have $\{\Theta\} \Longleftarrow S\,\{\Theta'\}$ where $\Theta' = \{\phi \Rightarrow h[A]\ltimes\}$ and where S *preserves* A, and also $\mathsf{Pre}_{\mathrm{for}}$ succeeds when called on S and $h[u]\ltimes$. Recall that we assume that $s\ [\![S]\!]\ s'$, $\sigma\ [\![S]\!]\ \sigma'$, and $s\&\sigma \models \Theta$; we then further assume $s' \models \phi$ and $\sigma' \models \phi$ so as to show $[\![h[A]]\!]_{s'} = [\![h[A]]\!]_{\sigma'}$. With $\phi_0 = NPC(S,\phi)$ we have, by the requirements to NPC, $s \models \phi_0$ and $\sigma \models \phi_0$.

With terminology as in Fig. 6, there are two cases. First assume that S *preserves* $h[A]$. Then $R = \{\phi_0 \Rightarrow h[A]\ltimes\}$ and $T = \emptyset$, implying that $\Theta = \{\phi_0 \Rightarrow h[A]\ltimes\}$. From $s \models \phi_0$, $\sigma \models \phi_0$, and $s\&\sigma \models \Theta$, we now infer that $s\&\sigma \models h[A]\ltimes$. Since S *preserves* $h[A]$, this yields the desired $s'\&\sigma' \models h[A]\ltimes$.

The second case is when S *preserves* h does not hold, while S *preserves* A does hold. Then $R = \{\phi_0 \Rightarrow A\ltimes\}$ and $T = \{\phi \Rightarrow h[A]\ltimes\}$, so in this case we run $\mathsf{Pre}_{\mathrm{for}}$ on S and $h[u]\ltimes$. Our assumption is that this call succeeds; with Θ_0 its result, we return

$$\Theta = \{(\phi_0 \wedge \phi_1[A/u] \Rightarrow E[A/u]\ltimes) \mid (\phi_1 \Rightarrow E\ltimes) \in \Theta_0\} \cup \{\phi_0 \Rightarrow A\ltimes\}.$$

From $s \models \phi_0$, $\sigma \models \phi_0$, and $s\&\sigma \models \Theta$, we infer $[\![A]\!]_s = [\![A]\!]_\sigma$. Since S *preserves* A there exists c such that $c = [\![A]\!]_s = [\![A]\!]_\sigma = [\![A]\!]_{s'} = [\![A]\!]_{\sigma'}$.

We shall now prove that $s\&\sigma \models \Theta_0[c/u]$. For let $\phi_1 \Rightarrow E\ltimes\ \in \Theta_0$ be given, and assume $s \models \phi_1[c/u]$ and $\sigma \models \phi_1[c/u]$, so as to prove $[\![E[c/u]]\!]_s = [\![E[c/u]]\!]_\sigma$. But as our assumption amounts to $s \models \phi_1[A/u]$ and $\sigma \models \phi_1[A/u]$, and we have also $s \models \phi_0$ and $\sigma \models \phi_0$, we from $s\&\sigma \models \Theta$ infer $s\&\sigma \models E[A/u]\ltimes$ which amounts to the desired $[\![E[c/u]]\!]_s = [\![E[c/u]]\!]_\sigma$.

We have proved $s\&\sigma \models \Theta_0[c/u]$, and since Lemma 3 gives us $\{\Theta_0[c/u]\}S\{h[c]\ltimes\}$, we conclude that $s'\&\sigma' \models h[c]\ltimes$. That is, $[\![h[c]]\!]_{s'} = [\![h[c]]\!]_{\sigma'}$, which amounts to the desired $[\![h[A]]\!]_{s'} = [\![h[A]]\!]_{\sigma'}$.

This concludes the current proof, covering selected but typical and non-trivial cases, of the Correctness Theorem.

[8] Concerning the procedure PreProc in Fig. 6: It needs to have the command S as a parameter, and ask "*S preserves* " rather than "*p preserves* ". When S *preserves* A does not hold, we need to insert $\phi \Rightarrow A\ltimes$ into T. When S *preserves* A does hold, we need to insert $NPC(S,\phi) \Rightarrow A\ltimes$ into R.

The clause for `for` needs to be modified as follows: when encountering $\phi \Rightarrow h[A]\ltimes\ \in T$ (in which case we have S *preserves* A), call $\mathsf{Pre}_{\mathrm{for}}$ on $h[u]$. If it succeeds with result Θ, return

$$\{(NPC(S,\phi) \wedge \phi_1[A/u] \Rightarrow E_1[A/u]\ltimes) \mid (\phi_1 \Rightarrow E_1\ltimes) \in \Theta\}.$$

If it fails, call $\mathsf{Pre}_{\mathrm{while}}$ as is currently done.

8 Conclusions and Future Work

We believe that the results of this paper provide another demonstration that information flow logic as introduced in [2] provides a powerful and flexible framework for precise compositional reasoning about information flow. The logic seems particularly well-suited for SPARK because (a) it naturally provides a semantics for SPARK's original flow contracts, and (b) SPARK's simplicity means that extensive use of the more complicated aspects of the logic (e.g., *region* annotations required to handle the heap[1]) can be avoided while still yielding significant increases in precision compared to the original SPARK contract language.

Several challenges remain as we transition this work into an environment that will be used by industrial engineers. First, the contracts that we infer can be so precise that they become large and unwieldy. The complexity of the contracts in these cases often results when the contract makes distinctions between different conditional flows that are unnecessary for establishing the desired end-to-end flow policy of a system or subsystem. We are developing tool-supported methodologies that guide programmers in writing more abstract specifications that capture distinctions required for end-to-end policies. Second, although our treatment of arrays using quantification works well for buffer manipulations often seen in information assurance applications, it works less well when trying to describe flows between elements of data structures such as trees implemented using arrays. We are investigating how separation logic might be able to provide a solution for this.

References

1. T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2006)*. Springer-Verlag, 2006.
2. T. Amtoft and A. Banerjee. Information flow analysis in logical form. In R. Giacobazzi, editor, *SAS 2004 (11th Static Analysis Symposium), Verona, Italy, August 2004*, volume 3148 of *LNCS*, pages 100–115. Springer-Verlag, 2004.
3. T. Amtoft and A. Banerjee. Verification condition generation for conditional information flow. In *5th ACM Workshop on Formal Methods in Security Engineering (FMSE'07), George Mason University*, pages 2–11. ACM, Nov. 2007. The full paper appears as Technical report 2007-2, Department of Computing and Information Sciences, Kansas State University, August 2007.
4. T. Amtoft, J. Hatcliff, and E. Rodríguez. Precise and automated contract-based reasoning for verification and certification of information flow properties of programs with arrays. Technical report, Kansas State University, October 2009. Available from http://www.cis.ksu.edu/ edwin/papers/TR-esop10.pdf.
5. T. Amtoft, J. Hatcliff, E. Rodríguez, Robby, J. Hoag, and D. Greve. Specification and checking of software contracts for conditional information flow. In *15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *LNCS*, pages 229–245. Springer-Verlag, 2008.
6. A. Banerjee and D. A. Naumann. History-based access control and secure information flow. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *LNCS*, pages 27–48. Springer-Verlag, 2005.
7. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
8. J. Barnes, R. Chapman, R. Johnson, J. Widmaier, D. Cooper, and B. Everett. Engineering the tokeneer enclave protection software. In *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE 2006)*. IEEE Press, 2006.

9. B. Barthes, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, June 28 - 30, 2004, Pacific Grove, California, USA, pages 100–114. IEEE Computer Society Press, 2004.
10. J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, 1985.
11. C. Boettcher, R. Delong, J. Rushby, and W. Sifre. The MILS component integration approach to secure information sharing. In *27th IEEE/AIAA Digital Avionics Systems Conference (DASC 2008)*. IEEE, 2008.
12. R. Chapman and A. Hilton. Enforcing security and safety models with an information flow analysis tool. *ACM SIGAda Ada Letters*, XXIV(4):39–46, 2004.
13. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Proceedings of the 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193 – 209. Springer-Verlag, 2005.
14. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.
15. N. Gehani. *Ada: an advanced introduction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1983.
16. J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.
17. D. Greve, M. Wilding, and W. M. Vanfleet. A separation kernel formal security policy. In *4th International Workshop on the ACL2 Theorem Prover and its Applications*, 2003.
18. D. Gries. *The Science of programming*. Springer-Verlag, New York, 1981.
19. C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*, pages 346–355, New York, NY, USA, 2006. ACM.
20. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.
21. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
22. A. C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 1999*, pages 228–241, New York, NY, USA, 1999. ACM Press.
23. D. A. Naumann. From coupling relations to mated invariants for checking information flow. In *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS 2006)*, volume 4189 of *LNCS*, pages 279–296. Springer, 2006.
24. Praxis High Integrity Systems. Rockwell Collins selects SPARK Ada for high-grade programmable cryptographic engine. Press Release. Available from: http://www.praxis-his.com/sparkada/pdfs/praxis_rockwell_final_pr.pdf, 2006.
25. W. Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, Aug. 1992.
26. B. Rossebo, P. Oman, J. Alves-Foss, R. Blue, and P. Jaszkowiak. Using Spark-Ada to model and verify a MILS message router. In *Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*, pages 7–14. IEEE, 2006.
27. J. Rushby. The design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating System Principles (SOSP 1981)*, pages 12–21, Asilomar, CA, December 1981. ACM. (ACM *Operating Systems Review*, Vol. 15, No. 5).
28. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal On Selected Areas in Communications*, 21(1):5–19, January 2003.
29. V. Simonet and I. Rocquencourt. Flow Caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*, pages 152–165, 2003.

30. G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology*, 15(4):410–457, 2006.
31. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Proceedings of the 12th International Static Analysis Symposium (SAS 2005)*, volume 3672 of *LNCS*, pages 352–367, September 2005.
32. D. M. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 607–621. Springer-Verlag, 1997.