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

Much effort has been spent on developing techniques to analyze information flow in computer programs [27] – leading to several languages such as Myers' JFlow [21], and FlowCaml [28], 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 [23] and the US National Security Agency (NSA) [7], 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 architectures such as MILS (Multiple Independent

A.D. Gordon (Ed.): ESOP 2010, LNCS 6012, pp. 43–63, 2010.

Levels of Security) [10]. 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. 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.

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,
- 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 [25], 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. 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 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 including the UK's iFACTS next generation air traffic control system.

Figure 1 (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`. SPARK also provides other annotation

```
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. 1. (a) Limitations of SPARK annotations and (b) proposed enhancements

mechanisms to specify pre- and postconditions, but for this discussion we will focus on those directly related to information flow analysis.

While the semantics of existing SPARK contracts, as presented in Figure 1 (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 1 (a) presents a series of micro-examples that illustrate the deficiencies of current SPARK annotations for arrays, and Fig. 1 (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 1 (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. 1 (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 1 (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 flows 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 an 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. 1 (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. 1 (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 this enables us to 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.

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:

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

Fig. 2. Syntax of a simple imperative language

In Fig. 2, we present the syntax of the simple imperative language. For commands, procedures are parameterless; this simplifies our exposition but 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 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.

We use E to range over expressions which include arithmetic, boolean, and array expressions. Boolean expressions are also used as assertions. We use x to range over integer (scalar) variables (but q to range over such when used as counters in **for** loops), h to range over array variables, u to range over universally quantified variables; we shall use w, z to range over all kind of variables. We use 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 $H\{A_0 : A_1\}$, which represents the value of array H except that index A_0 now has value A_1. We also use Z to denote an initial array as created by the command $h := \mathbf{new}$. We require a program (command) submitted for verification to be *pure* in the sense that it does not contain these additional array constructs. Thus, in a pure entity, all array accesses are of the form $h[A]$ with h a variable. Similarly, universal variables u are used only in specifications; *programs* submitted for verification cannot 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. 2 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 in E and write $E[A/x]$ for the result of substituting in E all occurrences of x by A (similarly for $E[H/h]$).

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:

$$s\ [\![x := A]\!]\ s' \text{ iff } \exists v : v = [\![A]\!]_s \text{ and } s' = [s \mid x \mapsto v]$$
$$s\ [\![\mathbf{assert}(\phi)]\!]\ s' \text{ iff } s \models \phi \text{ and } s' = s$$
$$s\ [\![\mathbf{call}\ p]\!]\ s' \text{ iff } s\ \mathcal{P}(p)\ s'$$
$$s\ [\![\mathbf{for}\ q \leftarrow 1\ \mathbf{to}\ y\ \mathbf{do}\ S]\!]\ s' \text{ iff } \exists n \geq 1 : n = s(y) \text{ and } \forall i \in \{0 \ldots n\}\ \exists s_i : s_0 = s \text{ and } s' = [s_n \mid q \mapsto n+1] \text{ and } \forall j \in \{1 \ldots n\} : [s_{j-1} \mid q \mapsto j]\ [\![S]\!]\ s_j$$
$$s\ [\![h[A_0] := A]\!]\ s' \text{ iff } \exists n, v : n = [\![A_0]\!]_s,\ v = [\![A]\!]_s \text{ and } s' = [s \mid h(n) \mapsto v]$$
$$s\ [\![h := \mathbf{new}]\!]\ s' \text{ iff } s' = [s \mid h \mapsto \lambda n.0]$$

Fig. 3. Semantics of the Simple Programming Language (excerpts)

Fig. 3 gives excerpts of the language semantics definition (the definitions for conditionals and while loops are standard and omitted). 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; we write $[a \mid n \mapsto v]$ for the array that is like a except that it maps n into v. We shall ignore bounds and range checks (unlike [15] where array length may be revealed separately from array content) and assume that an array reference $a(n)$ is always well-defined (the typical SPARK development process will prove statically that array-out-of-bounds exceptions cannot occur).

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 $dom(s)$ for the domain of s and write $[s \mid x \mapsto v]$ ($[s \mid h \mapsto a]$) for the store that is like s except that it maps x into v (maps h into a), and write $[s \mid h(n) \mapsto v]$ for $[s \mid h \mapsto [s(h) \mid n \mapsto v]]$. We write $s \models \phi$ for $[\![\phi]\!]_s = \mathsf{True}$. We define ϕ and ϕ' to be 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_p the body of p, we have $s\ [\![S_p]\!]\ s'$. 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

To motivate our treatment of information flow, consider the code

```
procedure p begin x := a+1; y := b * 2; end p;
```

where there are two "channels" of information flow associated with x and y: (1) from a to x, and (2) from b to y Using SPARK to specify these flows, we would write: `derives x from a & y from b;`

We may express the "non-interference" [16] 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. Thus x depends on a but on no other variables, cf. Cohen[12]. We desire to state such properties (which would provide a semantic foundation for `derives` contracts), using program level assertions. However, the 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 $\{a{\ltimes}\}\ S\ \{x{\ltimes}\}$. 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 deal with one channel at a time.

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 [3]. 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: $s\ \&\ \sigma \models \phi \Rightarrow E{\ltimes}$ iff whenever $s \models \phi$ and $\sigma \models \phi$ then $[\![E]\!]_s = [\![E]\!]_\sigma$.

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, $s\&\sigma \models \Theta$ iff $\forall \theta \in \Theta : s\&\sigma \models \theta$.

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 $s'\&\sigma' \models \Theta'$.

We define $\Theta \rhd_2 \Theta'$, pronounced "Θ 2-implies Θ'", to hold iff for all s, σ: whenever $s\&\sigma \models \Theta$ then also $s\&\sigma \models \Theta'$. 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.

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

$\{\Theta\} \Longleftarrow h := \mathbf{new}\ \{\Theta'\}$ iff $\Theta = \Theta'[Z/h]$ $\quad$ $\{\Theta\} \Longleftarrow h[A_0] := A_1\ \{\Theta'\}$ iff $\Theta = \Theta'[h\{A_0 : A_1\}/h]$

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

$\{\Theta\} \Longleftarrow S_1\ ;\ S_2\ \{\Theta'\}$ 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 $\{\Theta_i\} \Longleftarrow S_i\ \{\phi' \Rightarrow E\ltimes\}$ for $i = 1, 2$
in if S_1 *preserves* E and S_2 *preserves* E
then $\{(\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B) \Rightarrow E\ltimes \mid \phi_i \Rightarrow \_\ltimes \in \Theta_i\ (i = 1, 2)\}$
else $\{\phi_1 \wedge B \Rightarrow E_1\ltimes \mid \phi_1 \Rightarrow E_1\ltimes \in \Theta_1\} \cup \{\phi_2 \wedge \neg B \Rightarrow E_2\ltimes \mid \phi_2 \Rightarrow E_2\ltimes \in \Theta_2\} \cup$
$\{(\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B) \Rightarrow B\ltimes \mid \phi_i \Rightarrow \_\ltimes \in \Theta_i\ (i = 1, 2)\}$

$\{\Theta\} \Longleftarrow \mathbf{call}\ p\ \ (= S)\ \{\Theta'\}$ iff $\Theta = R \cup \bigcup_{\theta \in T} \mathsf{Pre}_{\mathrm{call}}(\theta)$ where
$(R, T) = \mathsf{PreProc}(S, \Theta')$ and
$\mathsf{Pre}_{\mathrm{call}}(\phi' \Rightarrow E\ltimes) =$ let $\phi_0 = NPC(S, \phi')$ in case E of
$w : \{\phi_0 \wedge \phi_w \Rightarrow E_w\ltimes \mid \phi_w \Rightarrow E_w\ltimes \in 2PC^p_w\}$
$h[A]$: let $2PC^p_{h[\_]} = \forall u.\Theta_h$ // *S preserves A*
in $\{\phi_0 \Rightarrow A\ltimes\} \cup \{\phi_0 \wedge \phi_h[A/u] \Rightarrow E_h[A/u]\ltimes \mid \phi_h \Rightarrow E_h\ltimes \in \Theta_h\}$

$\{\Theta\} \Longleftarrow \mathbf{while}\ B\ \mathbf{do}\ S_0\ \mathbf{od}\ \ (= S)\ \{\Theta'\}$ iff $\Theta = R \cup \Theta_A \cup \Theta_W$ where
$(R, T) = \mathsf{PreProc}(S, \Theta')$ $\quad$ $\Theta_A = \{NPC(S, \phi) \Rightarrow A\ltimes \mid \phi \Rightarrow h[A]\ltimes \in T\}$
$\Theta_W = \mathsf{Pre}_{\mathrm{while}}(S_0, B, T_W)$ $\quad$ $T_w = \{\phi \Rightarrow w\ltimes \in T\} \cup \{\phi \Rightarrow h\ltimes \mid \phi \Rightarrow h[A]\ltimes \in T\}$

$\{\Theta\} \Longleftarrow \mathbf{for}\ q \leftarrow 1\ \mathbf{to}\ m\ \mathbf{do}\ S_0\ \ (= S)\ \{\Theta'\}$ iff $\Theta = R \cup \Theta_A \cup \Theta_W \cup \Theta_F$ where
$(R, T) = \mathsf{PreProc}(S, \Theta')$ $\ u$ is fresh $\quad$ $\Theta_A = \{NPC(S, \phi) \Rightarrow A\ltimes \mid \phi \Rightarrow h[A]\ltimes \in T\}$
$\Theta_W = \mathsf{Pre}_{\mathrm{while}}((S_0\ ;\ q := q + 1), q \leq m, T_W)[1/q]$
$\Theta_F = \{NPC(S, \phi) \wedge \phi_1[A/u] \Rightarrow E_1[A/u]\ltimes \mid \phi_1 \Rightarrow E_1\ltimes \in \Theta_h,\ \phi \Rightarrow h[A]\ltimes \in T,\ \Theta_h \neq \mathbf{fail}\}$
$T_w = \{\phi \Rightarrow w\ltimes \in T\} \cup \{\phi \Rightarrow h\ltimes \mid \phi \Rightarrow h[A]\ltimes \in T,\ \Theta_h = \mathbf{fail}\}$
$\Theta_h = \mathsf{Pre}_{\mathrm{for}}(S_0, q, m, h[u]\ltimes)$ (for all h)

Fig. 4. The Precondition Generator

5 Computing Preconditions

Figure 4, 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 uses some auxiliary functions, defined in Fig. 5, as well as some other functions that will be sketched below but for whose complete definitions we refer to [4].

The algorithm *does not* always compute the weakest precondition; main sources of imprecision are: on loops, approximations have to be made to ensure termination of the analysis; on procedure calls, the analysis (for the sake of modularity) uses the 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 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:

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

For a detailed proof of this theorem, we refer the reader to [4]. The main structure of the proof is quite similar to our earlier work [3,5] though a main difference is that we

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

$\mathsf{Pre}_{\mathrm{for}}(S_0, q, m, h[u]\ltimes) =$
  let $\{A_j \mid j \in J\}$ be all occurrences such that $h[A_j] :=$ _ is a subcommand of S_0
  let $\{\Theta_j\} \Longleftarrow S_0\ \{h[A_j]\ltimes\}$ (for all $j \in J$)
  in if 1. **call** p *preserves* h for all **call** p occurring in S_0, and for all $j \in J$ it holds that
      2. S_0 *preserves* A_j
      3. there exists A'_j with $\mathrm{fv}(A'_j) \subseteq \{u\} \cup \mathrm{fv}(A_j) \setminus \{q\}$ where for all s,n with $dom(s) \subseteq \mathrm{fv}(A_j)$,
          $[\![n = A_j]\!]_s = [\![q = A'_j[n/u]]\!]_s$
      4. there exists ϕ_j with $\mathrm{fv}(\phi_j) \subseteq \{u\} \cup \mathrm{fv}(A_j) \setminus \{q\}$ where for all s,n with $dom(s) \subseteq \mathrm{fv}(A_j)$,
          $n \in \{[\![A_j]\!]_{[s|q\mapsto i]} \mid 1 \leq i \leq s(m)\}$ iff $s \models \phi_j[n/u]$
      5. if $w \in \mathrm{fv}(\Theta_j)$ with $w \neq h$ then S_0 *preserves* w
      6. if h occurs in Θ_j it is in the context $h[A]$ where for all $j_1 \in J$, all s, all $i, i' \in \{1 \ldots s(m)\}$:
          if $[\![A]\!]_{[s|q\mapsto i']} = [\![A_{j_1}]\!]_{[s|q\mapsto i]}$ then $i' \leq i$
    then **succeed** and return $\{\phi_j \Rightarrow \Theta_j[A'_j/q]\ltimes \mid j \in J\} \cup$
      $\{\wedge_{j\in J}\neg\phi_j \Rightarrow h[u]\ltimes\}\ \cup\ \{x\ltimes \mid \exists j \in J : x \in \mathrm{fv}(A_j) \setminus \{q\}\}\ \cup\ \{m\ltimes\}$
    else **fail**

$\mathsf{Pre}_{\mathrm{while}}(S_0, B, \Theta') =$
  $\psi_w \leftarrow \emptyset$ for all variables w (including a dummy variable d)
  for $\phi \Rightarrow w\ltimes \in \Theta'$ do
    $\psi_w \leftarrow \psi_w \vee (\phi \wedge \neg B)$; if $w \notin \mathrm{fv}(B)$ and not (S_0 *preserves* w) then $\psi_d \leftarrow \psi_d \vee (\phi \wedge \neg B)$
  repeat
    for each variable w do $\{\Theta_w\} \Longleftarrow S_0\ \{\psi_w \Rightarrow w\ltimes\}$;
      for each $\phi \Rightarrow E\ltimes \in \Theta_w$ do
        for each $z \in \mathrm{fv}(E)$ do $\psi_z \leftarrow \psi_z \vee (\phi \wedge B)$;
        if $w \in \mathrm{fv}(B)$ or S_0 *preserves* w then $\psi_w \leftarrow \psi_w \vee (\phi \wedge B)$
    for all $w \in \mathrm{fv}(B)$, for all z with not (S_0 *preserves* z) do $\psi_w \leftarrow \psi_w \vee \psi_z$
  until each ψ_w stabilizes (through *widening*) into Ψ_w
  return $\Theta = \{\Psi_w \Rightarrow w\ltimes \mid w \text{ is variable}\}$

Fig. 5. The Precondition Generator, Helper functions

have disposed with the "R-component"; this allows for a more streamlined presentation. Quite similar to those earlier works, we need the following lemma:

Lemma 1. *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. 4 so that Lemma 1 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.

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, our algorithm assumes that any procedure called by p already has an associated contract: for each w that may be modified by p, the contract contains a precondition $2PC^p_w$ (at least one assertion in which must be unconditional) such that $\{2PC^p_w\}p\{w{\ltimes}\}$; for each h that may be modified by p, the contract contains a precondition $2PC^p_{h[\_]}$ which is a *quantified set of 2-assertions* of the form $\forall u.\Theta$ where we demand that $\{\Theta\}p\{h[u]{\ltimes}\}$. 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.

Concerning the roles of universal variables, they are introduced in two situations: when analyzing a **for** loop (the output of $\mathsf{Pre}_{\text{for}}$), and when looking up $2PC^p_{h[\_]}$ for procedure calls. In both cases, they are instantiated immediately afterwards. When we *compute* summaries, however, universal variables are present throughout the derivation.

For most language constructs, the corresponding rule in Fig. 4 is straightforward. Assignments, to variables as well as array elements, are handled by syntactic replacement, as in classical Hoare logic.

For a conditional $\mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2$, if E is such that neither S_1 nor S_2 modifies E, the the precondition for $\phi \Rightarrow E{\ltimes}$ does not need to involve $B{\ltimes}$. There are several other instances 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 *S preserves E* such that if *S preserves E* holds then whenever $s\ [\![S]\!]\ s'$ we have $[\![E]\!]_s = [\![E]\!]_{s'}$. *$S$ preserves E* can be computed in a straightforward manner by detecting if S modifies variables occuring in E either directly via an assignment or indirectly via updates in a procedure call (in which case, the procedure's contract is consulted).

The NPC Function: When generating a precondition for S for post-condition $\phi' \Rightarrow E{\ltimes}$ where *S preserves E* holds, but 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 $\phi = NPC(S, \phi')$ (we can assume ϕ' to be pure) it holds that if $s\ [\![S]\!]\ s'$ and $s' \models \phi'$ then $s \models \phi$. 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.

Note that if *S preserves ϕ* then we can pick $\phi_0 = \phi$, and that we can always pick $\phi_0 = true$, but often we can compute something stronger. Our implementation, which assumes that each procedure p is equipped with a function that computes $NPC(p, \_)$, contains rules such as $NPC(x := A, \phi) = \phi[A/x]$ and
$NPC(\mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2, \phi) = (NPC(S_1, \phi) \wedge B) \vee (NPC(S_2, \phi) \wedge \neg B)$.

The *Purify* Function: As noted earlier, the rules for array update (creation) may generate a precondition that include impure expressions of the form $H_0\{A_0 : A_1\}$ (or Z) that we would not like to see in contracts. We therefore employ a function *Purify* with the following properties:

1. given ϕ, with $\phi_0 = \mathit{Purify}(\phi)$ we have $\phi \equiv_1 \phi_0$ with ϕ_0 pure.
2. given A, Purify returns pure $\phi_1 \ldots \phi_k$, and pure $A_1 \ldots A_k$, such that for all $i \in 1..k$: if $s \models \phi_i$ then $[\![A]\!]_s = [\![A_i]\!]_s$.
3. given Θ, with $\Theta_0 = \mathit{Purify}(\Theta)$ we have $\Theta_0 \rhd_2 \Theta$ with Θ_0 pure, and for all $\phi \Rightarrow \_\ltimes \in \Theta$ there exists $\phi_0 \Rightarrow \_\ltimes \in \Theta_0$ with $\phi \rhd_1 \phi_0$.

As an example of case 2, if A is given by $h\{x : y\}[z]$ then Purify returns ϕ_1, ϕ_2 given by $z = x$ and $z \neq x$, and A_1, A_2 given by y and $h[z]$. As an example of case 3, with A as above then $\mathit{Purify}(y > 0 \Rightarrow A\ltimes)$ is given by

$$\{y > 0 \wedge z = x \Rightarrow y\ltimes,\ y > 0 \wedge z \neq x \Rightarrow h[z]\ltimes,\ y > 0 \Rightarrow (z = x)\ltimes\}.$$

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. 5. 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, 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. Formally, we have: $\mathsf{PreProc}(S, \Theta')$ always terminates and returns R,T such that

1. for all Θ, if $\{\Theta\}S\{T\}$ then $\{\Theta \cup R\}S\{\Theta'\}$.
2. T is pure, and if $\phi \Rightarrow E\ltimes \in T$ then either $E = w$ where S *preserves* w does not hold, or $E = h[A]$ where S *preserves* A holds but S *preserves* h does not hold.

To prove this result, we observe that an invariant for the loop inside PreProc is: for all Θ, if $\{\Theta\}S\{T \cup P\}$ then $\{\Theta \cup R\}S\{\Theta'\}$.

The $\mathsf{Pre}_{\mathrm{for}}$ **Function:** The rule (Fig. 4) for **for**-loops, with associated helper function $\mathsf{Pre}_{\mathrm{for}}$ (Fig. 5), generates 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* [20]. Consider the following procedure body

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

that flips the values between the upper and lower halves of an array, resulting in information flow between the two halves. 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 $h[u]$). And indeed, replacing the **for** loop by its body (thus being iterated once only) will result in a contract showing the flow between the two locations on the separate halves of the array. What we want is a quantified version of that specification.

The definition of $\mathsf{Pre}_{\mathrm{for}}$ given in Fig. 5 implements the above intuition, for a given array h. (If multiple arrays are updated in the same loop, $\mathsf{Pre}_{\mathrm{for}}$ must be called separately on each array.) To handle also multiple updates, none of which can happen indirectly through procedure calls (condition 1), we let J range over all occurrences of such updates. Thus each array update is of the form $h[A_j] := \_$ where (condition 2) we can not allow A_j to be modified by the loop body (but we certainly expect A_j to contain the loop counter). Condition 3 states that each A_j must have an "inverse" A'_j. For example, if $A_j = q + 1$, then $A'_j = u - 1$. Condition 4 states that 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 determined by the predicate ϕ_j given by $1 + 1 \leq u \leq 10 + 1$. Condition 5 states that nothing in the precondition is modified except possibly h; that is, there are no loop carried dependencies between scalar variables. Condition 6 states that there are no loop-carried dependencies between array locations. That is, an array location is not read *after* it has been updated.

Thus 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. If any of the conditions is not satisfied, then the loop is treated as a `while` loop, in effect smashing together all array entries without obtaining a quantified information flow precondition.

The following lemma is a key step in the proof of Theorem 1.

Lemma 2. *Let S be* **for** $q \leftarrow 1$ **to** m **do** S_0. *Assume* $\mathsf{Pre}_{\mathrm{for}}(S_0, q, m, h[u]\ltimes)$ *succeeds, with result Θ. Then for all integer constants c we have* $\{\Theta[c/u]\}S\{h[c]\ltimes\}$.

Example 1. Consider the **for**-loop from (1). 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 $(n = q) \equiv_1 (q = n)$ and $(n = q + m) \equiv_1 (q = n - m)$.

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,

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

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 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 equations that *each* should imply $i' \leq i$ (given s and i, i' with $i, i' \in \{1 \ldots s(m)\}$):

(1) $[\![q + m]\!]_{[s|q \rightarrowtail i']} = [\![q]\!]_{[s|q \rightarrowtail i]}$ (3) $[\![q]\!]_{[s|q \rightarrowtail i']} = [\![q]\!]_{[s|q \rightarrowtail i]}$

(2) $[\![q + m]\!]_{[s|q \rightarrowtail i']} = [\![q + m]\!]_{[s|q \rightarrowtail i]}$ (4) $[\![q]\!]_{[s|q \rightarrowtail i']} = [\![q + m]\!]_{[s|q \rightarrowtail i]}$

Here (2) and (3) trivially imply $i' \leq i$ since they reduce to $i' + s(m) = i + s(m)$ and to $i' = i$; (1) and (4) vacuously imply $i' \leq i$ since they reduce to $i' + s(m) = i$ and to $i' = i + s(m)$ which both are impossible given $1 \leq i, i' \leq s(m)$. 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 expected preconditions

$$1 \leq u \leq m \Rightarrow h[u+m]{\ltimes},\ m+1 \leq u \leq (m+m) \Rightarrow h[u-m]{\ltimes},$$
$$(1 > u) \vee (u > m+m) \Rightarrow h[u]{\ltimes},\ m{\ltimes}.$$

The $\mathsf{Pre}_{\mathrm{while}}$ Function For the analysis of **while** loops (or **for** loops with loop-carried dependencies), we employ the function $\mathsf{Pre}_{\mathrm{while}}$ (Fig. 5) which expects a postcondition Θ' where each $\theta' \in \Theta'$ is of the form $\phi \Rightarrow w{\ltimes}$ (w a scalar or 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 ensure termination, we need a "widening operator" [13] on 1-assertions. A trivial widening operator is the one that always returns *true*, in effect converting conditional agreement assertions into unconditional. Our implementation uses disjunction as a widening operator but returns *true* if convergence is not achieved after a certain number of iterations. Space constraints prevent us from further explaining the algorithm (a variant of which was presented in [5]); we refer the reader to [4].

6 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. 6).

- **ArrayInit:** A procedure that initializes all elements of an array to a particular value.
- **ArrayScrub:** A procedure that 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.

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. 6

```
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. 6. Information flow contracts inferred by our implementation for a selection of examples

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, 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 [25]) designed to mediate communication between partitions in a *separation kernel* [26] – the foundation of specialized real-time platforms used in security contexts to provide strong data and temporal separation.

Fig. 7 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. 7, 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 the 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

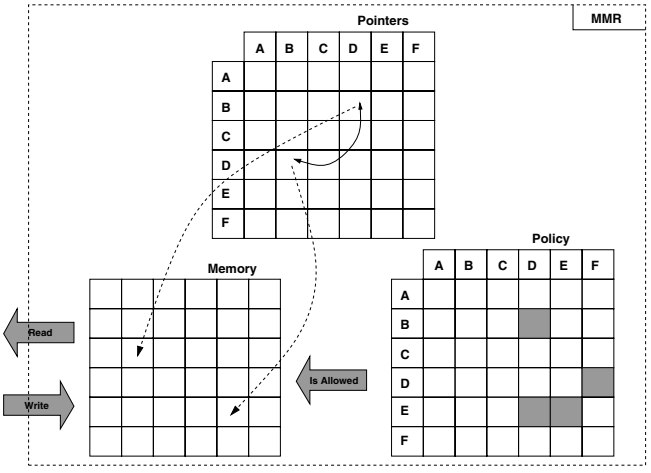

Fig. 7. Diagram of the MILS Message Router

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 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. 8 (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. 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).

There are multiple reasons why it is very difficult to verify statically that the MMR conforms 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

```
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. 8. 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)

the *Pointers* array. Third, the information flow path through the MMR between two 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 8 (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`.

Figure 8 (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).

7 Related Work

The first theoretical framework for SPARK information flow is provided by Bergeretti and Carré [9] who present a compositional method for inferring and checking dependencies among variables. That approach is flow-sensitive, unlike most security type systems [31,6] that rely on assigning a security level ("high" or "low") to each variable. Chapman and Hilton [11] 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.*[25] 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 [21]), and FlowCaml [28].

Agreement assertions (inherently flow-sensitive) were introduced in [2] together 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 [24] 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 $h[q+1] := h[q]$). Even though our focus is on the flow between between array elements, not their actual content, we might look into the body of work on static analysis of array content to see if some techniques may improve the precision of our analysis.

Rather than designing a specific logic for information flow, one can employ general logic as does the recently popular self-composition technique. Here the information flow property which we encode as $\{x\ltimes\}\ S\ \{y\ltimes\}$ is 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. This is the approach by Barthe et al. [8] that was extended by, e.g., Terauchi and Aiken [30] and Naumann [22]. The effect of self-composition can also be obtained through dynamic logic, as done by Darvas et al [14].

When it comes to *conditional* information flow, the most noteworthy existing tool is the slicer by Snelting et al [29] 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.

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., *object invariants* required to handle the heap[3]) 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.

Acknowledgements. This work is funded in part by the Air Force Office of Scientific Research (FA9550-09-1- 0138) and Rockwell Collins. We would like to thank Andrew Cousino for reading a draft of this paper, catching a subtle error, and assisting with some of the correctness proofs. We would also like to thank the anonymous referees for constructive comments and suggestions.

References

1. Amtoft, T., Bandhakavi, S., Banerjee, A.: 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). ACM Press, New York (2006)
2. Amtoft, T., Banerjee, A.: Information flow analysis in logical form. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 100–115. Springer, Heidelberg (2004)
3. Amtoft, T., Banerjee, A.: Verification condition generation for conditional information flow. In: 5th ACM Workshop on Formal Methods in Security Engineering (FMSE 2007), George Mason University, pp. 2–11. ACM, New York (2007); The full paper appears as Technical report 2007-2, Department of Computing and Information Sciences, Kansas State University (August 2007)
4. Amtoft, T., Hatcliff, J., Rodríguez, E.: 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), `http://www.cis.ksu.edu/~edwin/papers/TR-esop10.pdf`
5. Amtoft, T., Hatcliff, J., Rodríguez, E., Robby, Hoag, J., Greve, D.: Specification and checking of software contracts for conditional information flow. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 229–245. Springer, Heidelberg (2008)
6. Banerjee, A., Naumann, D.A.: History-based access control and secure information flow. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 27–48. Springer, Heidelberg (2005)
7. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer enclave protection software. In: Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE 2006). IEEE Press, Los Alamitos (2006)
8. Barthe, G., D'Argenio, P., Rezk, T.: Secure information flow by self-composition. In: Proceedings of the 17th IEEE Computer Security Foundations Workshop, Pacific Grove, California, USA, June 28 - 30, pp. 100–114. IEEE Computer Society Press, Los Alamitos (2004)
9. Bergeretti, J.-F., Carré, B.A.: Information-flow and data-flow analysis of while-programs. ACM Transactions on Programming Languages and Systems 7(1), 37–61 (1985)
10. Boettcher, C., Delong, R., Rushby, J., Sifre, W.: The MILS component integration approach to secure information sharing. In: 27th IEEE/AIAA Digital Avionics Systems Conference (DASC 2008). IEEE, Los Alamitos (2008)
11. Chapman, R., Hilton, A.: Enforcing security and safety models with an information flow analysis tool. ACM SIGAda Ada Letters XXIV(4), 39–46 (2004)
12. Cohen, E.S.: Information transmission in sequential programs. In: DeMillo, R.A., Dobkin, D.P., Jones, A.K., Lipton, R.J. (eds.) Foundations of Secure Computation, pp. 297–335. Academic Press, London (1978)
13. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages, pp. 238–252 (1977)
14. Darvas, A., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer, Heidelberg (2005)
15. Deng, Z., Smith, G.: Lenient array operations for practical secure information flow. In: 17th IEEE Computer Security Foundations Workshop, Pacific Grove, California, June 2004, pp. 115–124 (2004)
16. Goguen, J., Meseguer, J.: Security policies and security models. In: Proceedings of the 1982 Symposium on Security and Privacy, pp. 11–20. IEEE, Los Alamitos (1982)

17. Greve, D., Wilding, M., Vanfleet, W.M.: A separation kernel formal security policy. In: 4th International Workshop on the ACL2 Theorem Prover and its Applications (2003)
18. Gries, D.: The Science of programming. Springer, New York (1981)
19. Heitmeyer, C.L., Archer, M., Leonard, E.I., McLean, J.: 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), pp. 346–355. ACM, New York (2006)
20. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco (1997)
21. Myers, A.C.: JFlow: practical mostly-static information flow control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 1999), pp. 228–241. ACM Press, New York (1999)
22. Naumann, D.A.: From coupling relations to mated invariants for checking information flow. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 279–296. Springer, Heidelberg (2006)
23. Praxis High Integrity Systems. Rockwell Collins selects SPARK Ada for high-grade programmable cryptographic engine. Press Release (2006), `http://www.praxis-his.com/sparkada/pdfs/praxis_rockwell_final_pr.pdf`
24. Pugh, W.: A practical algorithm for exact array dependence analysis. Commun. ACM 35(8), 102–114 (1992)
25. Rossebo, B., Oman, P., Alves-Foss, J., Blue, R., Jaszkowiak, P.: Using Spark-Ada to model and verify a MILS message router. In: Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering, pp. 7–14. IEEE, Los Alamitos (2006)
26. Rushby, J.: The design and verification of secure systems. In: Proceedings of the 8th ACM Symposium on Operating System Principles (SOSP 1981), Asilomar, CA, pp. 12–21. ACM Press, New York (1981); ACM Operating Systems Review 15(5)
27. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal On Selected Areas in Communications 21(1), 5–19 (2003)
28. Simonet, V., Rocquencourt, I.: Flow Caml in a nutshell. In: Proceedings of the first APPSEM-II workshop, pp. 152–165 (2003)
29. Snelting, G., Robschink, T., Krinke, J.: Efficient path conditions in dependence graphs for software safety analysis. ACM Transactions on Software Engineering and Methodology 15(4), 410–457 (2006)
30. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005)
31. Volpano, D.M., Smith, G.: A type-based approach to program security. In: Bidoit, M., Dauchet, M. (eds.) CAAP 1997, FASE 1997, and TAPSOFT 1997. LNCS, vol. 1214, pp. 607–621. Springer, Heidelberg (1997)