Specification and Checking of Software Contracts for Conditional Information Flow*

Torben Amtoft[1], John Hatcliff[1], Edwin Rodríguez[1], Robby[1], Jonathan Hoag[1], David Greve[2]

[1] Kansas State University
Manhattan, KS 66506, USA
{tamtoft,hatcliff,edwin, robby,jch5588}@cis.ksu.edu

[2] Rockwell Collins
Cedar Rapids, IA, USA
dagreve@rockwellcollins.com

Abstract. Information assurance applications built according to the MILS (Multiple Independent Levels of Security) architecture often contain information flow policies that are *conditional* in the sense that data is allowed to flow between system components only when the system satisfies certain state predicates. However, existing specification and verification environments, such as SPARK Ada, used to develop MILS applications can only capture unconditional information flows. Motivated by the need to better formally specify and certify MILS applications in industrial contexts, we present an enhancement of the SPARK information flow annotation language that enables specification, inferring, and compositional checking of conditional information flow contracts. We report on the use of this framework for a collection of SPARK examples.

1 Introduction

National and international infrastructures as well as commercial services are increasingly relying on complex distributed systems that share information with *Multiple Levels of Security* (MLS). These systems often seek to coalesce information with mixed security levels into information streams that are targeted to particular clients. For example, in a national emergency response system, some data will be privileged (*e.g.*, information regarding availability of military assets, and deployment orders for those assets) and some data will be public (*e.g.*, weather and mapping information). In such systems there is a huge tension between providing aggressive information flow in order to gain operational advantage while preventing the flow of information to unauthorized parties. Specification and verification of security policies and providing end-to-end guarantees in this context is exceedingly difficult.

The *Multiple Independent Levels of Security* (MILS) architecture [29] proposes to make development, accreditation, and deployment of MLS-capable systems more practical, achievable, and affordable by providing a certified infrastructure *foundation for systems that require assured information sharing*. In the MILS architecture, systems are developed on top of: (a) a "separation kernel", a concept due to Rushby [25] which guarantees isolation and controlled communication between application components deployed in different virtual "partitions" supported by the kernel, and (b) MILS middleware services such as "high assurance guards" that allow information to flow between various partitions and between trusted and untrusted segments of a network only when certain *conditions* are satisfied.

* This work was supported in part by the US National Science Foundation (NSF) awards 0454348, 0429141, and CAREER award 0644288, the US Air Force Office of Scientific Research (AFOSR), and Rockwell Collins.

Researchers at Rockwell Collins Advanced Technology Center (RC ATC) are industry leaders in certifying MILS components according to standards such as the Common Criteria (EAL 6/7) that mandate the use of formal methods. For example, RC ATC engineers carried out the certification of the hardware-based separation kernel in RC's AAMP7 processor (this was the first such certification of a MILS separation kernel and it formed the initial draft of the Common Criteria Protection Profile for Separation Kernels) as well as the software-based kernel in the Green Hills Integrity 178B RTOS.

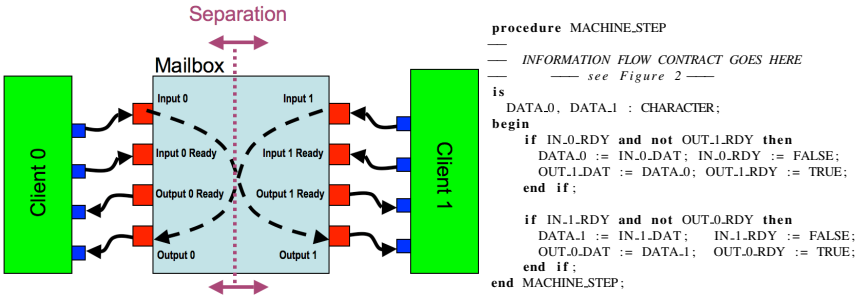
Seeking to leverage the groundbreaking work on the AAMP7 separation kernel, Rockwell Collins product groups that include 200+ developers are building several different information assurance products on top of the AAMP7 following the MILS architecture. These products are programmed using the SPARK subset of Ada [8]. One of the primary motivating factors for the use of SPARK is that it includes annotations (formal contracts for procedure interfaces) for specifying and checking information flow [11]. The use of these annotations plays a key role in the certification cases for the products. The SPARK language and associated tool-set is the only commercial product that we know of which can support checking of code-level information flow contracts, and SPARK provides a number of well-designed and effective capabilities for specifying and verifying properties of implementations of safety-critical systems.

However, Rockwell Collins developers are struggling to provide precise arguments for correctness in information assurance certification due to several limitations of the SPARK information flow framework. A key limitation is that SPARK information flow annotations are unconditional (e.g., they capture such statements as "executing procedure P may cause information to flow from input variable X to output variable Y"), but MILS security policies are often conditional (e.g., data from partition A is only allowed to flow to partition B when state variables G_1 and G_2 satisfy certain conditions). Thus, SPARK cannot capture nor support verification of critical aspects of MILS policies (treating such conditional flows as unconditional flows in SPARK is an over-approximation that leads to many false alarms).

In previous work, Banerjee and the first author have developed Hoare logics that enable compositional reasoning about information flow [3, 2]. Inspired by challenge problems from Rockwell Collins, this logic was extended to support conditional information flow [5]. While the logic as presented in [5] exposed some foundational issues, it only supported intraprocedural analysis, it required developers to specify information flow loop invariants, the verification algorithm was not yet fully implemented (and thus no experience was reported), and the core logic was not mapped to a practical method contract language capable of supporting compositional reasoning in industrial settings.

In this paper, we address these limitations by describing how the logic can provide a foundation for a practical information flow contract language capable of supporting compositional reasoning about conditional information flows. The specific contributions of our work are as follows:

- we propose an extension to SPARK's information flow contract language that supports conditional information flow, and we describe how the logic of [5] can be used to provide a semantics for the resulting framework,
- we extend the verification generation rules of [5] to support procedure calls and compositional checking,
- we present a strategy for automatically inferring conditional information flow invariants for while loops, thus significantly reducing developers' annotation burden,




```
procedure MACHINE_STEP
--
--   INFORMATION FLOW CONTRACT GOES HERE
--          -- see Figure 2 --
is
   DATA_0, DATA_1 : CHARACTER;
begin
      if IN_0_RDY and not OUT_1_RDY then
         DATA_0 := IN_0_DAT; IN_0_RDY := FALSE;
         OUT_1_DAT := DATA_0; OUT_1_RDY := TRUE;
      end if;

      if IN_1_RDY and not OUT_0_RDY then
         DATA_1 := IN_1_DAT;    IN_1_RDY := FALSE;
         OUT_0_DAT := DATA_1;   OUT_0_RDY := TRUE;
      end if;
end MACHINE_STEP;
```

Fig. 1. Simple MILS Guard - mailbox mediates communication between partitions.

- we provide an implementation that can both: (a) check method implementations against information flow contracts, and (b) automatically mine conditional information flow contracts from implementations, and
- we report on experiments applying the implementation to a collection of examples.

Recent efforts for certifying MILS separation kernels [16, 17] used formal models in ACL2 [20] or PVS [23] theorem provers that were developed by hand from source code, and extensive inspections were required by certification authorities to establish the validity of these manual steps. Because our approach is directly integrated with code, it complements these earlier efforts by: (a) removing the "trust gaps" associated with manually building and inspecting behavioral models, and (b) allowing many verification obligations to be discharged earlier in the life cycle by developers while leaving only the most complicated obligations to certification teams. In addition, the logic-based approach presented in this paper provides a foundation for producing independently auditable and machine-checkable *evidence* (in the form of proofs within the logic) of correctness and MILS policy satisfaction as recommended by the National Research Council's Committee on Certifiably Dependable Software Systems [19].

2 Example

Figure 1 illustrates the conceptual information flows in a fragment of an idealized MILS infrastructure component used by Rockwell Collins engineers to demonstrate specification and verification of information flow issues in MILS components running on top of the AAMP7 separation kernel for NSA and industry representatives. This demonstration was the first iteration of what is now a much more sophisticated high assurance network guard product line at Rockwell Collins. The "Mailbox" component in the center of the diagram mediates communication between two client processes – each running on its own partition in the separation kernel. *Client 0* writes data to communicate in the memory segment *Input 0* that is shared between *Client 0* and the mailbox, then it sets the *Input 0 Ready* flag. The mailbox process polls its ready flags; when it finds that, *e.g.*, *Input 0 Ready* is set and *Output 1 Ready* is cleared (indicating that *Client 1* has already consumed data deposited in the *Output 1* slot in a previous communication), then it copies the data from *Input 0* to *Output 1* and clears *Input 0 Ready* and sets *Output 1 Ready*. The communication from *Client 1* to *Client 0* follows a symmetric set of steps.

Figure 2(a) shows the SPARK Ada annotations for the `MACHINE_STEP` procedure (shown in Fig. 1, next to the diagram) of the Mailbox example that implements the actions to be taken in each execution frame. SPARK `derives` annotations are used to capture the information flow properties of the example. It requires that each parameter and each global variable referenced by the procedure be classified as **in** (read only),

```
--# global in out IN_0_RDY, IN_1_RDY, OUT_0_RDY, OUT_1_RDY,
--#                OUT_0_DAT, OUT_1_DAT;
--#         in     IN_0_DAT, IN_1_DAT;
--# derives
--#   OUT_0_DAT from IN_1_DAT, OUT_0_DAT, OUT_0_RDY, IN_1_RDY &
--#   OUT_1_DAT from IN_0_DAT, OUT_1_DAT, IN_0_RDY, OUT_1_RDY &
--#   IN_0_RDY  from IN_0_RDY, OUT_1_RDY &
--#   IN_1_RDY  from INP_1_RDY, OUT_0_RDY &
--#   OUT_0_RDY from OUT_0_RDY, IN_1_RDY &
--#   OUT_1_RDY from OUT_1_RDY, IN_0_RDY;
```

(a)

```
--# ...
--# derives
--#   OUT_0_DAT from
--#     IN_1_DAT when (IN_1_RDY and not OUT_0_RDY),
--#     OUT_0_DAT when (not IN_1_RDY or OUT_0_RDY),
--#     OUT_0_RDY, IN_1_RDY &
--#   OUT_1_DAT from
--#     IN_0_DAT when (IN_0_RDY and not OUT_1_RDY),
--#     OUT_1_DAT when (not IN_0_RDY or OUT_1_RDY),
--#     OUT_1_RDY, IN_0_RDY
--# ...
```

(b)

Fig. 2. (a) SPARK information flow contract for Mailbox example. (b) Fragment of same example with proposed conditional information flow extensions.

out (written, and initial values [values at the point of procedure call] are unread), or **in out** (written, and initial values read). For a procedure P, variables annotated as `in` or `in out` are called *input variables* and denoted IN_P; while variables annotated as `out` or `in out` are *output variables* and denoted as OUT_P. Each output variable x_o must have a `derives` annotation indicating the input variables whose initial values are used to directly or indirectly calculate the final value of x_o. One can also think of each derives clause as expressing a dependence relation or program slice between an output variable and the input variables that it transitively depends on (via both data and control dependence). For example, the second `derives` clause specifies that on each `MACHINE_STEP` execution the output value of `OUT_1_DAT` is possibly determined by the input values of several variables: from `IN_0_DAT` when the Mailbox forwards data supplied by *Client 0*, from `OUT_1_DAT` when the conditions on the ready flags are not satisfied (`OUT_1_DAT`'s output value then is its input value), and from `OUT_1_RDY` and `IN_0_RDY` because these variables *control* whether or not data flows from *Client 0* on a particular machine step (*i.e.*, they *guard* the flow).

While upper levels of the MILS architecture require reasoning about lattices of security levels (*e.g.*, *unclassified*, *secret*, *top secret*), the policies of infrastructure components such as separation kernels and guard applications usually focus on data separation policies (reasoning about flows between components of program state), and we restrict ourselves to such reasoning in this paper.

No other commercial language framework provides automatically checkable information flow specifications, so the use of the information flow checking framework in SPARK is a significant step forward. As illustrated above, SPARK `derives` clauses can be used to specify flows of information from input variables to output variables, but they do not have enough expressive power to state that information only flows under specific conditions. For example, in the Mailbox code, information from `IN_0_DAT` only flows to `OUT_1_DAT` when the flag `IN_0_RDY` is set and `OUT_1_READY` is cleared, otherwise `OUT_1_DAT` remains unchanged. In other words, the flags `IN_0_RDY` and `OUT_1_RDY` *guard* the flow of information through the mailbox. Unfortunately, the SPARK `derives` cannot distinguish the flag variables as guards nor phrase the conditions under which the guards allow information to pass or be blocked. This means that guarding logic, which is central to many MLS applications including those developed at Rockwell Collins, *is completely absent from the checkable specifications* in SPARK.

In general, the lack of ability to express *conditional* information flow not only inhibits automatic verification of guarding logic specifications, but also results in imprecision which cascades and builds throughout the specifications in the application.

3 Foundations of SPARK Conditional Information Flow

The SPARK subset of Ada is designed for programming and verifying safety critical applications such as avionics applications certified to DO-178B Level A. It deliberately

omits constructs that are difficult to reason about such as dynamically created data, pointers, and exceptions. Below, we present the syntax of a simple imperative language with assertions that one can consider to be an idealized version of SPARK.

Assertions	*Expressions*	*Commands*
$\phi ::= B \mid \phi \wedge \phi$	$A ::= x \mid c \mid A \;\mathsf{op}\; A$	$S ::= \mathbf{skip} \mid x := A \mid S\,;S \mid \mathbf{assert}(\phi)$
$\mid \phi \vee \phi \mid \neg\phi$	$B ::= A \;\mathsf{bop}\; A$	$\mid \mathbf{if}\ B\ \mathbf{then}\ S\ \mathbf{else}\ S \mid \mathbf{while}\ B\ \mathbf{do}\ S \mid \mathbf{call}\ p$

Features of SPARK that we do not consider here include the package and inheritance structure, records, and arrays. From these, only arrays present conceptual challenges. Our current implementation treat arrays as atomic entities just as SPARK does. The extended version of this paper [6] describes how our logical approach can reason about individual elements of arrays (giving more precision than SPARK), but that extension is not yet included in our implementation. We consider both arithmetic (A) and boolean (B) expressions where we use $x, y, \ldots$ to range over variables, c to range over integer constants, p to range over named (parameterless) procedures, op to range over arithmetic operators in $\{+, \times, \mathsf{mod}, \ldots\}$, and bop to range over comparison operators in $\{=, <, \ldots\}$. Using parameterless procedures simplifies our exposition; our implementation supports procedures with parameters (there are no conceptual challenges in this extended functionality). For an expression E (arithmetic or boolean), we write $\mathrm{fv}(E)$ for the variables occurring free in E, and $E[A/x]$ for the result of substituting in E all occurrences of x by A. For assertions ϕ, where as usual we define $\phi_1 \rightarrow \phi_2$ as $\neg\phi_1 \vee \phi_2$, we also define *true* as $0 = 0$, and *false* as $0 = 1$.

The semantics of an arithmetic expression $[\![A]\!]$ is a function from stores into values, where a value ($v \in \mathit{Val}$) is an integer n and where a *store* $s \in \mathit{Store}$ maps variables to values; we write $\mathit{dom}(s)$ for the domain of s and write $[s|x \mapsto v]$ for the store that is like s except that it maps x into v. Similarly, $[\![B]\!]_s$ denotes a boolean. A command transforms the store into another store; hence its semantics is given in relational style, in the form $s\,[\![S]\!]\,s'$. For some S and s, there may not exist any s' such that $s\,[\![S]\!]\,s'$; this can happen if a $\mathbf{while}$ loop does not terminate, or an $\mathbf{assert}$ fails. The details of the semantics are standard and thus omitted; implicitly we assume a global procedure environment P that for each p returns a relation between input and output stores.

The satisfaction relation for assertions reads $s \models \phi$ and denotes that ϕ holds in s following the standard semantics. 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, "ϕ 1-implies ϕ'", written $\phi \rhd_1 \phi'$, when ϕ logically implies ϕ'.

Reasoning about information flow in terms of non-interference: MILS seeks to prevent security breaches that can occur via unauthorized/unintended information flow from one partition to another; thus previous certification efforts for MILS components have among the core requirements included the classical property of *non-interference* [15] which (in this setting) states: for every pair of runs of a program, if the runs agree on the initial values of one partition's data (but may disagree on the data of other partitions) then the runs also agree on the final values of that partition's data.

Capturing non-interference and secure information flow in a compositional logic: The logic developed in [3] 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" 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

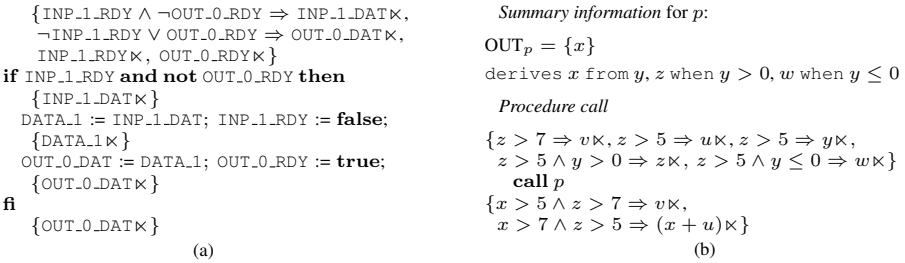


Fig. 3. (a) A derivation for the mailbox example, illustrating the handling of conditionals. (b) An example illustrating the handling of procedure calls.

the *agreement assertion* $x\ltimes$ is satisfied by a *pair* of states, s_1 and s_2, if $s_1(x) = s_2(x)$. With P the example program from Sect. 2, we would have, e.g.,

$$\{\texttt{INP\_1\_DAT}\ltimes, \texttt{OUT\_0\_DAT}\ltimes, \texttt{INP\_1\_RDY}\ltimes, \texttt{OUT\_0\_RDY}\ltimes\}\ P\ \{\texttt{OUT\_0\_DAT}\}.$$

To capture conditional information flow, recent work [5] by Banerjee and the first author introduced *conditional* agreement assertions, also called *2-assertions*. They are of the form $\phi \Rightarrow E\ltimes$ which is satisfied, intuitively, by a pair of stores if either at least one of them does not satisfy ϕ, or they agree on the value of E. We use $\theta \in \mathbf{2Assert}$ to range over 2-assertions, and define $s\ \&\ s_1 \models \theta$ by:

$$s\ \&\ s_1 \models \phi \Rightarrow E\ltimes \text{ iff whenever } s \models \phi \text{ and } s_1 \models \phi \text{ then } [\![E]\!]_s = [\![E]\!]_{s_1}.$$

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 $\forall \theta \in \Theta \bullet s\&s_1 \models \theta$.

Fig. 3(a) illustrates a simple derivation using conditional information flow assertions that answers the question: what is the source of information flowing into variable OUT_0_DAT? The natural way to read the derivation is from the bottom up (this is the way our algorithm actually generates the derivation). Thus, for $\texttt{OUT\_0\_DAT}\ltimes$ to hold after execution of P, we must have $\texttt{DATA\_1}\ltimes$ before line 3 (since data flows from DATA_1 to OUT_0_DAT), $\texttt{INP\_1\_DAT}\ltimes$ before line 2 (since data flows from INP_1_DAT to DATA_1), and finally $\texttt{INP\_1\_RDY}\ltimes$ and $\texttt{OUT\_0\_RDY}\ltimes$ (since they *control* which branch of the condition is taken), along with conditional assertions. The pre-condition shows that the value of OUT_0_DAT depends *unconditionally* on INP_1_RDY and OUT_0_RDY, and *conditionally* on INP_1_DAT and OUT_0_DAT, just as we would expect.

Relations between agreement assertions: 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 $s\&s_1 \models \Theta'$. Θ and Θ' are *2-equivalent*, written $\Theta \equiv_2 \Theta'$, iff $\Theta \rhd_2 \Theta'$ and $\Theta' \rhd_2 \Theta$. 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. For example, $\{x\ltimes, y\ltimes\}$ refines $x\ltimes$ by adding an (unconditional) agreement assertion, and $z < 10 \Rightarrow x\ltimes$ refines $z < 7 \Rightarrow x\ltimes$ by weakening the antecedent of a 2-assertion. We define a function *decomp* that converts arbitrary 2-assertions into assertions with only variables as consequents: $decomp(\Theta) = \{\phi \Rightarrow x\ltimes \mid \phi \Rightarrow E\ltimes \in \Theta, x \in \mathrm{fv}(E)\}$.

Fact 1 *For all Θ, $decomp(\Theta)$ is a refinement of Θ.*

The converse does not hold; for example we do not have $\{(x+y)\ltimes\} \rhd_2 \{x\ltimes, y\ltimes\}$ since we might have $s\&s_1 \models (x+y)$ but not $s\&s_1 \models x$ or $s\&s_1 \models y$, as when $s(x) = s_1(y) = 3$, $s(y) = s_1(x) = 7$.

4 Conditional Information Flow Contracts

4.1 Foundations of flow contracts

The syntax of a SPARK `derives` annotation for a procedure P (as illustrated in Figure 2(a)) can be represented formally as a relation $\mathcal{D}_P$ between OUT_P and $\mathcal{P}(\text{IN}_P)$. A particular clause $\mathsf{derives}(x, \bar{y}) \in \mathcal{D}_P$ declares that the final value of output variable x depends on the input values of variables $\bar{y} = y_1, \ldots, y_k$. The correctness of such a clause as a contract for P can be expressed in terms of the logic of the preceding section, as requiring the triple $\{\bar{y}\ltimes\}\ S\ \{x\ltimes\}$ where S is the body of procedure P and where $\bar{y}\ltimes$ is a shorthand for $\{y_1\ltimes, \ldots, y_k\ltimes\}$.

Because $\mathcal{D}_P$ contains multiple clauses (one for each output variable of P), it captures multiple "channels" of information flow through P. Therefore, we cannot simply describe the semantics of a multi-clause derives contract $\{\mathsf{derives}(x, \bar{y}), \mathsf{derives}(z, \bar{w})\}$ as $\{(\bar{y}\bar{w})\ltimes\}\ S\ \{x\ltimes, z\ltimes\}$ because this would confuse the dependencies associated with x and z, *i.e.*, it would allow z to depend on $\bar{y}$. Accordingly, the full semantics of SPARK derives contracts is supported by what we term a *multi-channel version* of the logic which is extended to include *indexed agreement assertions* $x\ltimes_c$ indexed by a channel identifier c – which one can usually associate with a particular output variable to specify the flows/channel associated with a specific derives clause. In the multi-channel logic, the confused triple above can now be correctly stated as $\{\bar{y}\ltimes_x, \bar{w}\ltimes_z\}\ S\ \{x\ltimes_x, z\ltimes_z\}$. The algorithm to be given in Sect. 5 extends to the multi-channel version of the logic in a straightforward manner; hence the implementation described subsequently supports the multi-channel version of the logic. For notational simplicity, we continue the discussion of the semantics of contracts using the single-channel version of the logic.

We now give a more convenient notation for triples of the form $\{\Theta\}P\{\Theta'\}$. This will provide a formal interpretation for method contracts that capture conditions of flows from beginning to end of a method P. A flow judgement κ is of the form $\Theta \rightsquigarrow \Theta'$, with Θ the precondition and with Θ' the postcondition. We say that $\Theta \rightsquigarrow \Theta'$ is valid for command S, written $S \models \Theta \rightsquigarrow \Theta'$, if whenever $s_1 \& s_2 \models \Theta$ and $s_1 [\![S]\!] s_1'$ and $s_2\ [\![S]\!]\ s_2'$ then also $s_1' \& s_2' \models \Theta'$ (if the 2-assertions in the precondition hold for input states s_1 and s_2, the postcondition must also hold for associated output states s_1' and s_2').

4.2 Language design for conditional SPARK contracts

The logic of the preceding section is potentially much more powerful than what we actually want to expose to developers – instead, we view it as a "core calculus" in which information flow reasoning is expressed. Our design goals that determine how much of the power of the logic we wish to expose to developers in enhanced SPARK conditional information flow contracts are: (1) the effort required to write the contracts should be as simple as possible, (2) the contracts should be able to capture common idioms of MILS information guarding, (3) the contract checking framework should be compositional to support MILS goals, and (4) there should be a natural progression (e.g., via formal refinements) from unconditional `derives` statements to conditional statements.

Simplifying assertions: The agreement assertions from the logic of Sect. 3 have the form $\phi \Rightarrow E\ltimes$. Here E is an arbitrary expression (not necessarily a variable), whereas SPARK `derives` statements are phrased in terms of IN/OUTvariables only. We believe that including arbitrary expressions in SPARK conditional `derives` statements would add significant complexity for developers, and our experimental studies have shown that little increase in precision would be gained by such an approach. Instead, we retain

the use of expression-based assertions $\phi \Rightarrow E\ltimes$ only during intermediate (automated) steps of the analysis, and appealing to Fact 1, we have a canonical way of strengthening $\phi \Rightarrow E\ltimes$ to $\phi \Rightarrow x_1\ltimes, \ldots, \phi \Rightarrow x_k\ltimes$ where $x_1, \ldots, x_k \in \mathrm{fv}(E)$ for contracts at procedure boundaries. A second simplification relates to the fact that the core logic allows both pre- and post-conditions to be conditional (e.g., $\{\phi_1 \Rightarrow E_1\ltimes\}\ P\ \{\phi_2 \Rightarrow E_2\ltimes\}$ where ϕ_1 and ϕ_2 represent different conditions). Based on discussions with developers at Rockwell Collins and initial experiments, we believe that this would expose too much power/complexity to developers leading to unwieldy contracts and confusion about the underlying semantics. Accordingly, we are currently pursuing an approach in which only preconditions can be conditional. Combining these two simplifications, SPARK `derives` clauses are extended to allow conditions on input variables as follows:

$$\texttt{derives}\ x\ \texttt{from}\ y_1\ \texttt{when}\ \phi_1, \ldots, y_k\ \texttt{when}\ \phi_k$$

Here $\phi_1 \ldots \phi_k$ are boolean expressions on the pre-state of the associated procedure P. Thus, the above specification can be read as "The value of variable x at the conclusion of executing P (for *any* final state s') is derived from y_j when ϕ_j holds in the pre-state s from which s' is computed (if also ϕ_i holds in s, then x depends also on y_i)." Additional syntactic sugar can be introduced to simplify the contract notation, *e.g.*, when input variables are conditioned (guarded) using the same expression. Figure 2(b) shows how this can be used to specify conditional flows for procedure `MACHINE_STEP` in Fig. 1.

Design methodology separating guard logic from flow logic: The lack of conditional assertions in post-conditions has the potential to introduce imprecision. Yet, we believe the above approach to conditional expressions can be effective for the following reason: we have observed that information assurance application design tends to factor out the *guarding logic* (i.e., the pieces of state and associated state changes that determine *when* information can flow) from the code which propagates information. This follows a common pattern in embedded systems in which the control logic is often factored out from data computation logic.

This informal design strategy can be firmed up and presented as an effective design methodology: some procedures act to modify the conditions under which information flows (the guard logic) while other procedures actually realize the flows for a particular value of the guard state. This could be enhanced by an explicit declaration of the *guard state*, i.e., declaration of the program variables that can be observed by guards. Guard state variables would be modified in guard logic procedures, but not be modified in any procedure that declares conditions based on those guards. SPARK's existing IN/OUTvariable annotations can capture this requirement (no variable appearing in a condition can be in OUT).

Contract abstraction and refinement: For a practical design and development methodology, it is important to consider notions of contract abstraction (generalization) and refinement – ideally, conditional contracts should be a refinement of unconditional contracts. For example, we believe it will be easier to introduce conditional contracts into workflows if developers can: (1) make a rough cut at specifying information flows without conditions, and (2) systematically refine to produce conditional contracts. In addition, in situations where developers have trouble capturing flow policies, they can state flows without conditions and expert verification engineers can later refine those into conditional contracts. Conversely, it is important for managers to understand that they are not locked into our emerging technology; if they decide not to pursue a verification approach based on conditional SPARK contracts, they can safely abstract all conditional contracts back to unconditional contracts.

$\{\Theta\}\ (R) \Longleftarrow \mathbf{skip}\ \{\Theta'\}$ iff $R = \{(\theta, u, \theta) \mid \theta \in \Theta'\}$ and $\Theta = \Theta'$

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

$\{\Theta\}\ (R) \Longleftarrow x := A\ \{\Theta'\}$ iff $R = \{(\phi[A/x] \Rightarrow E[A/x]\ltimes, \gamma, \phi \Rightarrow E\ltimes) \mid \phi \Rightarrow E\ltimes \in \Theta'\}$,
where $\gamma = m$ iff $x \in \mathrm{fv}(E)$ and $\Theta = dom(R)$

$\{\Theta\}\ (R) \Longleftarrow S_1\ ;\ S_2\ \{\Theta'\}$ iff $\{\Theta''\}\ (R_2) \Longleftarrow S_2\ \{\Theta'\}$ and $\{\Theta\}\ (R_1) \Longleftarrow S_1\ \{\Theta''\}$
and $R = \{(\theta, \gamma, \theta') \mid \exists \theta'', \gamma_1, \gamma_2 \bullet (\theta, \gamma_1, \theta'') \in R_1,\ (\theta'', \gamma_2, \theta') \in R_2\}$, where $\gamma = m$ iff $\gamma_1 = m$ or $\gamma_2 = m$

$\{\Theta\}\ (R) \Longleftarrow \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \{\Theta'\}$
iff $\{\Theta_1\}\ (R_1) \Longleftarrow S_1\ \{\Theta'\}$, $\{\Theta_2\}\ (R_2) \Longleftarrow S_2\ \{\Theta'\}$, $R = R'_1 \cup R'_2 \cup R'_0 \cup R_0$, and $\Theta = dom(R)$,
where $R'_1 = \{((\phi_1 \wedge B) \Rightarrow E_1\ltimes, m, \theta') \mid \theta' \in \Theta'_m,\ (\phi_1 \Rightarrow E_1\ltimes, \_, \theta') \in R_1\}$
and $R'_2 = \{((\phi_2 \wedge \neg B) \Rightarrow E_2\ltimes, m, \theta') \mid \theta' \in \Theta'_m,\ (\phi_2 \Rightarrow E_2\ltimes, \_, \theta') \in R_2\}$
and $R'_0 = \{(((\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)) \Rightarrow B\ltimes, m, \theta')$
$\mid \theta' \in \Theta'_m,\ (\phi_1 \Rightarrow E_1\ltimes, \_, \theta') \in R_1, (\phi_2 \Rightarrow E_2\ltimes, \_, \theta') \in R_2\}$
and $R_0 = \{(((\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)) \Rightarrow E\ltimes, u, \theta')$
$\mid \theta' \in \Theta'_u,\ (\phi_1 \Rightarrow E\ltimes, u, \theta') \in R_1, (\phi_2 \Rightarrow E\ltimes, u, \theta') \in R_2\}$
and $\Theta'_m = \{\theta' \in \Theta' \mid \exists(\_, m, \theta') \in R_1 \cup R_2\}$ and $\Theta'_u = \Theta' \setminus \Theta'_m$

$\{\Theta\}\ (R) \Longleftarrow \mathbf{call}\ p\ \{\Theta'\}$
iff $R = R_u \cup R_0 \cup R_m$ and $\Theta = dom(R)$,
where $R_u = \{(rm^{+}_{\mathrm{OUT}_P}(\phi) \Rightarrow E\ltimes, u, \phi \Rightarrow E\ltimes) \mid (\phi \Rightarrow E\ltimes) \in \Theta' \wedge \mathrm{fv}(E) \cap \mathrm{OUT}_P = \emptyset\}$
and $R_0 = \{(rm^{+}_{\mathrm{OUT}_P}(\phi) \Rightarrow x\ltimes, m, \phi \Rightarrow E\ltimes)$
$\mid (\phi \Rightarrow E\ltimes) \in \Theta' \wedge \mathrm{fv}(E) \cap \mathrm{OUT}_P \neq \emptyset \wedge x \in \mathrm{fv}(E) \wedge x \notin \mathrm{OUT}_P\}$
and $R_m = \{(rm^{+}_{\mathrm{OUT}_P}(\phi) \wedge \phi^{y}_{x} \Rightarrow y\ltimes, m, \phi \Rightarrow E\ltimes)$
$\mid (\phi \Rightarrow E\ltimes) \in \Theta' \wedge x \in \mathrm{fv}(E) \cap \mathrm{OUT}_P \wedge \phi^{y}_{x} \Rightarrow y\ltimes$ among preconditions for $x\ltimes$ in p's summary $\}$

$\{\Theta\}\ (R) \Longleftarrow \mathbf{while}\ B\ \mathbf{do}\ S_0\ \{\Theta'\}$
iff $R = R_u \cup R_m$ and $\Theta = dom(R)$,
where for each x (in X) we inductively in i define ϕ^{i}_{x}, Θ^{i}, R^{i}, ψ^{i}_{x} by
$\phi^{0}_{x} = \bigvee\{\phi \mid \exists E : (\phi \Rightarrow E\ltimes) \in \Theta' \wedge x \in \mathrm{fv}(E)\}$, $\Theta^{i} = \{\phi^{i}_{x} \Rightarrow x\ltimes \mid x \in X\}$, $\{\_\}\ (R^{i}) \Longleftarrow S_0\ \{\Theta^{i}\}$
$\psi^{i}_{x} = \bigvee\{\phi \mid \exists(\phi \Rightarrow E\ltimes, \_, \_) \in R^{i}$ with $x \in \mathrm{fv}(E)$
or $x \in \mathrm{fv}(B)$ and $\exists(\theta, m, \theta') \in R^{i}$ with $\phi = ant(\theta)$ or $\phi = ant(\theta')\}$
$\phi^{i+1}_{x} =$ if $\psi^{i}_{x} \rhd_1 \phi^{i}_{x}$ then ϕ^{i}_{x} else $\phi^{i}_{x}\ \bigtriangledown\ \psi^{i}_{x}$
and j is the least i such that $\Theta^{i} = \Theta^{i+1}$
and $R_u = \{(\phi \Rightarrow E\ltimes, u, \theta') \mid \theta' \in \Theta'_u, E = con(\theta'), (\mathrm{fv}(E) = \emptyset, \phi = \mathit{true}) \vee (\mathrm{fv}(E) \neq \emptyset, \phi = \bigvee_{x \in \mathrm{fv}(E)}(\phi^{j}_{x}))\}$
and $R_m = \{(\theta, m, \theta') \mid \theta' \in \Theta'_m \wedge \theta \in \Theta^{j} \cup \{\mathit{true} \Rightarrow 0\ltimes\}\}$
and $\Theta_m = \{\theta' \in \Theta' \mid \exists x \in \mathrm{fv}(con(\theta')) : \exists(\_, m, \_ \Rightarrow x\ltimes) \in R^{j}\}$ and $\Theta_u = \Theta' \setminus \Theta'_m$

Fig. 4. The Precondition Generator

We now establish the desired notion of contract refinement (in terms of the general underlying calculus instead of its limited exposure in SPARK), by defining a relation between flow judgements: $\kappa_1 \rhd_\kappa \kappa_2$, pronounced "$\kappa_1$ refines κ_2", to hold iff for all commands S, whenever $S \models \kappa_1$ then also $S \models \kappa_2$.

To gain the proper intuition about contract refinement, it is important to note that the refinement relation is contra-variant in the pre-condition and co-variant in the post-condition: given $\kappa_1 \equiv \Theta_1 \rightsquigarrow \Theta'_1$ and $\kappa_2 \equiv \Theta_2 \rightsquigarrow \Theta'_2$, if $\Theta_2 \rhd_2 \Theta_1$ and $\Theta'_1 \rhd_2 \Theta'_2$ then $\kappa_1 \rhd_\kappa \kappa_2$. For example, $x\ltimes \rightsquigarrow y\ltimes \rhd_\kappa x\ltimes, y\ltimes \rightsquigarrow y\ltimes$ holds because $x\ltimes, y\ltimes \rhd_2 x\ltimes$ (Section 3). Intuitively, this captures the fact that a contract can always be *abstracted* to a weaker one by stating that the output variables may depend on additional input variables. This illustrates that our contracts capture "may" dependence modalities: output y *may* depend on both inputs x and y, but a refinement $x\ltimes \rightsquigarrow y\ltimes$ shows that output y need not depend on input y (the contract before refinement is an *over-approximation* of dependence information. Also, we have $(z < 7 \Rightarrow x\ltimes \rightsquigarrow y\ltimes) \rhd_\kappa (x\ltimes \rightsquigarrow y\ltimes)$ which realizes our design goals of achieving: (a) a formal refinement by adding conditions to a contract, and (b) a formal (safe) abstraction by removing conditions.

5 A Precondition Generation Algorithm

We define in Fig. 4 an algorithm Pre for inferring preconditions from postconditions. We write $\{\Theta\}\ (R) \Longleftarrow S\ \{\Theta'\}$ when, given command S and postcondition Θ', Pre

returns a precondition Θ for S that is designed so as to be sufficient to establish Θ' and a relation R that associates each 2-assertion $\theta \in \Theta'$ with the 2-assertions in Θ needed to establish θ. R captures dependences between variables before and after the execution of S, and it also supports reasoning about multiple channels of information flow as discussed in Sect. 4.1, *e.g.*, if $\{y_1 y_2 \ltimes_x, y_1 y_3 \ltimes_z\}\ S\ \{x \ltimes_x, z \ltimes_z\}$ then R will relate y_1 to x and to z, y_2 to x, and y_3 to z. More precisely, we have $R \subseteq \Theta \times \{m, u\} \times \Theta'$ where tags m,u are mnemonics for "modified" and "unmodified"; if $(\theta, u, \theta') \in R$ then additionally it holds that S modifies no "relevant" variable, where a "relevant" variable is one occurring in the consequent of θ'. We use γ to range over $\{m, u\}$, and write $dom(R) = \{\theta \mid \exists(\theta, \_, \_) \in R\}$ and $ran(R) = \{\theta' \mid \exists(\_, \_, \theta') \in R\}$.

Correctness results: If $\{\Theta\}\ (\_) \Longleftarrow S\ \{\Theta'\}$ then Θ is indeed a precondition (but not necessarily the *weakest* such) that is strong enough to establish Θ', as stated by:

Theorem 2 (Correctness). *Assume* $\{\Theta\}\ (\_) \Longleftarrow S\ \{\Theta'\}$ *. Then* $S \models \Theta \rightsquigarrow \Theta'$*. That is, if* $s \& s_1 \models \Theta$*, and* s', s'_1 *are such that* $s\ [\![S]\!]\ s'$ *and* $s_1\ [\![S]\!]\ s'_1$*, then* $s' \& s'_1 \models \Theta'$*.*

Note that Theorem 2 is termination-*in*sensitive; this is not surprising given our choice of a relational semantics (but see [4] for a logic-based approach that is termination-sensitive). Also note that correctness is phrased directly wrt. the underlying semantics, unlike [3, 2] which first establish the semantic soundness of a logic and next provide a sound implementation of that logic. Theorem 2 is proved [6] much as the corresponding result [5] (that handled a language with heap manipulation but without procedure calls and without automatic computation of loop invariants), by establishing some auxiliary properties (e.g., the R component) that have largely determined the design of Pre.

Intraprocedural analysis: We now explain the various clauses of Pre in Fig. 4, where the clause for **skip** is trivial. For an assignment $x := A$, each 2-assertion $\phi \Rightarrow E\ltimes$ in Θ' produces exactly one 2-assertion in Θ, given by substituting A for x (as in standard Hoare logic) in ϕ as well as in E; the connection is tagged m when x occurs in E. For example, if S is $x := w$ then R might contain the triplets $(q > 4 \Rightarrow w\ltimes, m, q > 4 \Rightarrow x\ltimes)$ and $(w > 3 \Rightarrow z\ltimes, u, x > 3 \Rightarrow z\ltimes)$. The rule for $S_1\,; S_2$ works backwards, first computing S_2's precondition which is then used to compute S_1's; the tags express that a consequent is modified iff it has been modified in either S_1 or S_2. The rule for **assert** allows us to weaken 2-assertions, by strengthening their antecedents; this is sound since execution will abort from stores not satisfying the new antecedents.

To illustrate and motivate the rule for conditionals, we shall use Fig. 3(a) where, given postcondition `OUT_0_DAT`$\ltimes$, the **then** branch generates (as the domain of R_1) precondition `INP_1_DAT`$\ltimes$ which by R'_1 contributes the first conditional assertion of the overall precondition. The **skip** command in the implicit **else** branch generates (as the domain of R_2) precondition `OUT_0_DAT`$\ltimes$ which by R'_2 contributes the second conditional assertion of the overall precondition. We must also capture that two runs, in order to agree on `OUT_0_DAT` after the conditional, must agree on the value of the test B; this is done by R'_0 which generates the precondition $(\mathit{true} \wedge B) \vee (\mathit{true} \wedge \neg B) \Rightarrow B\ltimes$; optimizations (not shown) in our algorithm simplify this to $B\ltimes$ and then use Fact. 1 to split out the variables in the conjuncts of B into the two unconditional assertions of the overall precondition. Finally, assume the postcondition contained an assertion $\phi \Rightarrow E\ltimes$ where E is not modified by either branch: if also ϕ is not modified then $\phi \Rightarrow E\ltimes$ belongs to both R_1 and R_2, and hence by R_0 also to the overall precondition; if ϕ is modified by one or both branches, R_0 generates a more complex antecedent for $E\ltimes$.

Interprocedural analysis: Recall from Sect. 4.2 that for a procedure summary, we allow only variables as consequents, and allow conditional assertions only in the preconditions. At a call site **call** p, antecedents in the call's postcondition will carry over to the precondition, *provided* that they do not involve variables in OUT_P. Otherwise, since our summaries express variable dependencies but not functional relationships, we cannot state an exact formula for modifying antecedents (unlike what is the case for assignments). Instead, we must conservatively strengthen the preconditions, by weakening their antecedents; this is done by an operator rm^+ such that if $\phi' = rm^+_X(\phi)$ (where $X = \text{OUT}_p$) then ϕ logically implies ϕ' where ϕ' does not contain any variables from X. A trivial definition of rm^+ is to let it always return *true* (which drops all conditions associated with X), but we can often get something more precise; for instance, we can choose $rm^+_{\{x\}}(x > 7 \wedge z > 5) = (z > 5)$.

Equipped with rm^+, we can now define the analysis of procedure call, as done in Fig. 4 and illustrated in Fig. 3(b). In Fig. 4, R_u deals with assertions (such as $x > 5 \wedge z > 7 \Rightarrow v\ltimes$ in the example) whose consequent has not been modified by the procedure call (its "frame conditions" determined by the OUT declaration). For an assertion whose consequent E has been modified (such as $x > 7 \wedge z > 5 \Rightarrow (x + u)\ltimes$), we must ensure that the variables of E agree after the procedure call (when the antecedent holds). For those not in OUT_p (such as u), this is done by R_0 (which expresses some "semi frame conditions"); for those in OUT_p (such as x), this is done by R_m which utilizes the procedure summary (contract) of the called procedure.

Analyzing iteration: For while loops, the idea is to consider assertions of the form $\phi_x \Rightarrow x\ltimes$ and then repeatedly analyze the loop body so as to iteratively weaken the antecedents until a fixed point is reached. The details are given in Fig. 4 but space does not permit us to explain each line; instead we illustrate the overall behavior below.

```
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*	$\Rightarrow h\ltimes$
	false	*true*	*true*	*true*	$\Rightarrow i\ltimes$
	true	*true*	*true*	*true*	$\Rightarrow r\ltimes$
	false	$\mathsf{odd}(i)$	$\mathsf{odd}(i)$	$\mathsf{odd}(i)$	$\Rightarrow v\ltimes$
	false	*false*	$\neg\mathsf{odd}(i)$	*true*	$\Rightarrow x\ltimes$

Here we are given $r\ltimes$ 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 $\mathsf{odd}(i)$ (we use $\mathsf{odd}(i)$ as a shorthand for $i \bmod 2 = 1$), 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 $\neg\mathsf{odd}(i)$, 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 $r\ltimes$, 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 [5] 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.

In the general case, however, fixed point iteration may not terminate. To ensure termination, 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*,

Package.Procedure Name	LoC	C	L	P	O	SF	Flows		Cond. Flows		Gens.		Time (seconds)	
							1	2	1	2	1	2	1	2
Autopilot.AP.Altitude.Pitch.Rate.History_Average	10	0	1	0	1	2	5	3	0	0	0	0	0.047	0.063
Autopilot.AP.Altitude.Pitch.Rate.History_Update	8	1	1	0	1	2	3	3	0	0	0	0	0.000	0.157
Autopilot.AP.Altitude.Pitch.Rate.Calc_Pitchrate	13	2	0	2	2	7	17	8	0	0	15	15	0.000	0.015
Autopilot.AP.Altitude.Pitch.Target_ROC	9	2	0	0	1	2	7	3	6	2	0	0	0.000	0.000
Autopilot.AP.Altitude.Pitch.Target_Rate	17	4	0	1	1	3	53	4	42	0	142	46	0.015	0.015
Autopilot.AP.Altitude.Pitch.Calc_Elevator_Move	7	0	0	1	1	3	4	4	0	0	0	0	0.000	0.000
Autopilot.AP.Altitude.Pitch.Pitch_AP	7	0	0	4	2	11	54	11	42	0	0	0	0.015	0.000
Autopilot.AP.Altitude.Maintain	9	2	0	1	4	19	36	23	23	19	0	0	0.000	0.015
Autopilot.AP.Heading.Roll.Target_ROR	15	3	0	1	1	2	4	3	0	0	26	26	0.000	0.000
Autopilot.AP.Heading.Roll.Target_Rate	11	2	0	1	1	3	9	4	0	0	14	14	0.000	0.000
Autopilot.AP.Heading.Roll.Calc_Aileron_Move	7	0	0	1	1	3	4	4	0	0	0	0	0.000	0.000
Autopilot.AP.Heading.Roll.Roll_AP	7	0	0	4	2	7	9	7	0	0	0	0	0.000	0.000
Autopilot.AP.Control	19	1	0	13	8	46	58	54	0	0	63	51	0.016	0.032
Autopilot.AP.Heading.Yaw.Calc_Rudder_Move	7	0	0	1	1	2	4	3	0	0	0	0	0.000	0.000
Autopilot.AP.Heading.Yaw.Yaw_AP	5	0	0	3	2	5	5	5	0	0	0	0	0.000	0.000
Autopilot.Scale.Inverse	4	0	0	0	1	1	1	1	0	0	0	0	0.000	0.016
Autopilot.Scale.Scale_Movement	22	4	0	2	1	4	47	10	46	9	0	0	0.016	0.000
Autopilot.Scale.Heading_Offset	7	1	0	0	1	1	3	1	2	0	0	0	0.000	0.000
Autopilot.Heading.Maintain	6	1	0	2	4	15	28	20	16	16	0	0	0.000	0.000
Autopilot.Main	5	0	1	1	8	47	176	54	0	0	0	0	0.031	0.031
Minepump.Logbuffer.ProtectedWrite	8	1	0	0	5	9	9	9	4	4	0	0	0.031	0.047
Minepump.Logbuffer.ProtectedRead	6	0	0	0	5	6	7	7	0	0	0	0	0.000	0.000
Minepump.Logbuffer.Write	2	0	0	1	5	9	11	9	3	1	0	0	0.000	0.000
Mailbox.MACHINE_STEP	17	2	0	0	6	16	18	18	12	12	0	0	0.047	0.062
Mailbox.Main	6	0	1	1	6	16	54	22	0	0	2	2	0.031	0.016
BoilerWater-Monitor.FaultIntegrator.Test	11	3	0	0	4	11	46	22	42	18	0	0	0.047	0.047
BoilerWater-Monitor.FaultIntegrator.ControlHigh	8	1	0	2	2	4	6	5	0	0	2	2	0.000	0.000
BoilerWater-Monitor.FaultIntegrator.ControlLow	8	1	0	2	2	4	6	5	0	0	2	2	0.000	0.000
BoilerWater-Monitor.FaultIntegrator.Main	11	0	1	6	2	2	14	4	0	0	0	0	0.016	0.016
Lift-Controller.Next_Floor	9	2	0	0	1	2	7	4	6	3	0	0	0.047	0.047
Lift-Controller.Poll	22	2	1	3	2	9	77	12	43	0	0	0	0.031	0.031
Lift-Controller.Traverse	18	0	1	11	3	10	210	13	66	0	0	0	0.281	0.063
Missile_Guidance.Clock_Read	12	2	0	0	3	5	13	11	10	8	0	0	0.047	0.047
Missile_Guidance.Clock_Utils_Delta_Time	7	1	0	0	1	2	4	2	2	0	0	0	0.000	0.000
Missile_Guidance.Extrapolate_Speed	13	2	0	2	2	7	14	10	6	4	36	16	0.000	0.000
Missile_Guidance.Code_To_State	12	3	0	0	1	7	15	9	14	8	0	0	0.000	0.000
Missile_Guidance.Transition	20	4	0	2	1	9	3527	63	3524	62	4	4	0.156	0.125
Missile_Guidance..Relative_Drag_At_Altitude	8	2	0	0	1	1	7	3	6	2	0	0	0.000	0.000
Missile_Guidance.Drag_cfg.Calc_Drag	21	4	0	1	1	3	37	3	34	0	0	0	0.000	0.000
Missile_Guidance.If_Airspeed_Get_Speed	6	1	0	0	2	3	4	4	2	2	0	0	0.000	0.000
Missile_Guidance.Nav.Handle_Airspeed	18	4	0	4	3	13	117	28	110	25	18	18	0.000	0.000
Missile_Guidance.Nav.Estimate_Height	21	5	0	2	2	11	60	18	57	16	4	4	0.000	0.000

Table 1. Experiment Data (excerpts)

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.

6 Evaluation

The algorithm of Section 5 provides a foundation for both *checking* (conditional and unconditional) `derives` contracts supplied by a developer, and for automatically *inferring* contracts from implementations (for checking contracts supplied by a developer, the algorithm infers a pre-condition Θ from the supplied postcondition and then checks that the supplied precondition entails Θ). There is much merit in a methodology that encourages writing of the contract *before* writing/checking the implementation. However, one of our strategies for injecting our techniques into industrial development groups is to pitch the tools as being able to discover more precise conditional specifications to supplement conventional SPARK `derives` contracts already in the code; thus we focus the experimental studies of this section on the more challenging problem of automatically inferring contracts. For each procedure, the input to the algorithm is a post-

condition $x_o^1 \ltimes_1, \ldots, x_o^k \ltimes_k$ for each $x_o^j \in$ OUT. Since SPARK disallows recursion, we simply move in a bottom-up fashion through the call-graph – guaranteeing that a contract exists for each called procedure. When deployed in actual development, one would probably allow developers to tweak the generated contracts (e.g., by removing unnecessary conditions for establishing end-to-end policies) before proceeding with contract inference for methods in the next level of the call hierarchy. However, in our experiments, we used autogenerated contracts for called methods without modification. All experiments were run under JDK 1.6 on a 2.2 GHz Intel Core2 Duo.

Code bases: Embedded security devices are the initial target domain for our work, and the security-critical sections to be certified from these code bases are often relatively small, *e.g.*, roughly 1000 LOC for the Rockwell Collins high assurance guard mentioned earlier and 3000 LOC for the (undisclosed) device recently certified by Naval Research Labs researchers [17]. For our evaluation, we consider a collection of six small to moderate size applications from the SPARK distribution in addition to an expanded version of the mailbox example of Section 2. Of these, the *Autopilot* and *Missile Control* applications are the most realistic. There are well over 250 procedures in the code bases, but due to space constraints, in Table 1 we list metrics for only the most complex procedures from each application (see [6] for complete data and [31] for the source code of all the examples). Columns **LOC**, **C**, **L**, and **P** report the number of non-comment lines of code, conditional expressions, loops, and procedure calls in each method. Our tool can run in two modes. The first mode (identified as version **1** in Table 1) implements the rules of Figure 4 directly, with just one small optimization: a collection of boolean simplifications are introduced, e.g., simplifying assertions of the form *true* $\wedge\, \phi \Rightarrow E\ltimes$ to $\phi \Rightarrow E\ltimes$. The second mode (version **2** in Table 1) enables a collection of simplifications aimed at compacting and eliminating redundant flows from the generated set of assertions. The first simplification performed is elimination of assertions with *false* in the antecedent (these are trivially true), and elimination of duplicate assertions. Finally, it adds elimination of simple entailed assertions, e.g., it eliminates $\phi \Rightarrow E\ltimes$ when *true* $\Rightarrow E\ltimes$ also appears in the assertion set.

Typical refinement power of the algorithm: Column **O** gives the number of OUT variables of a procedure (this is equal to the number of `derives` clauses in the original SPARK contract), and Column **SF** gives the number of *flows* (total number of IN/OUT pairs) appearing in the original contract. Column **Flows** gives the number of flows generated by different versions of our algorithm. This number increases over **SF** as SPARK flows are refined into conditional flows (often creating two or more conditioned flows for a particular IN/OUT variable pair). The data shows that the compacting optimizations often substantially reduce the number of flows; the practical impact of this is to substantially increase the readability/tractability of the contracts. Column **Cond. Flows** indicates the number of flows from **Flows** that are conditional. We expect to see the refining power of our approach in procedures with conditionals (column **C**) primarily, but we also see increases in precision that is due to conditional contracts of called procedures (column **P**). In a few cases we see a blow-up in the number of conditional flows. The worse case is `MissileGuidance.Transition`, which contains a case statement with each branch containing nested conditionals and procedure calls with conditional contracts – leading to an exponential explosion in path conditions. Only a few variables in these conditions lie in what we consider to be the "control logic" of the system. The tractability of this example would improve significantly with the methodology suggested earlier in which developers declare explicitly the guarding variables (such as the

`xx_RDY` variables of Fig. 1) and the algorithm then omits tracking of conditional flows not associated with declared guard variables. Overall, a manual inspection of each inferred contract showed that the algorithm usually produces conditions that an expert would expect. Importantly, we have verified by manual inspection that the algorithm *never* produces a contract that is *less precise* than the original SPARK contract (it is always a formal refinement of the original).

Efficiency of inference algorithm: As can be see in the **Time** columns, the algorithm is quite fast for all the examples, usually taking a little longer in version **2** (all optimizations on). However, for some examples, version **2** is actually faster; these are the cases of procedures with calls to other procedures. Due to the optimizations, the callees now have simpler contracts, simplifying the processing of the caller procedures.

Sources of loss of precision: We would like to determine situations where our treatment of loops or procedure calls leads to abstraction steps that discard conditional information. While this is difficult to determine for loops (one would have to compare to the most precise loop invariant – which would need to be written by hand), Column **Gens.** indicates the number of conditions dropped across processing of procedure calls. The data shows, and our experience confirms, that the loss of precision is not drastic (in some cases, one wants conditions to be discarded), but more experience is needed to determine the practical impact on verification of end-to-end properties.

Threats to validity of experiments: While the applications we consider are representative of small embedded controller systems, only the mailbox example is an information assurance application. While these initial results are encouraging, we are still in the process of negotiating access to the source code of actual products being developed at Rockwell Collins; that will allow us to answer the important question: does our approach provide the precision needed to better verify local and end-to-end MILS policies, without generating large contracts that become unwieldy for developers and certifiers?

7 Related Work

The theoretical framework for the SPARK information flow framework is provided by Bergeretti and Carré [11] who presents a compositional method for inferring and checking dependences [13] among variables. That approach is flow-sensitive, whereas most security type systems [30, 7] are flow-*in*sensitive as they 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 to check conformance of flows to particular security levels. Those ideas could be applied directly to provide security levels of flows in our framework. Rossebo *et al.*[24] 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 properties, notably Jif (Java + information flow) which is based on [21]), and Flow Caml [26].

The seminal work on agreement assertions is [3], whose logic is flow-sensitive, and 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 [2] 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 it is not possible to capture *conditional* information flows. That was what motivated Amtoft & Banerjee [5] to introduce conditional agreement assertions, for a

heap-manipulating language. This paper integrates that approach into the SPARK setting for practical industrial development, adds interprocedural contract-based composition checking, adds an algorithm for computing loop invariants (rather than assuming they are provided by the user), and provides an implementation as well as reports on experiments.

A recently popular approach to information flow analysis is *self-composition*, first proposed by Barthe et al. [10] and later extended by, e.g., Terauchi and Aiken [28] and (for heap-manipulating programs) Naumann [22]. Self-composition works as follows: for a given program S, a copy S' is created with all variables renamed (primed); with the observable variables say x, y, then non-interference holds provided the sequential composition $S; S'$ when given precondition $x = x' \wedge y = y'$ also ensures postcondition $x = x' \wedge y = y'$. This is a property that can be checked using existing verifiers like BLAST [18], Spec# [9], or ESC/Java2 [14]. Darvas et al. [1] use the KeY tool for interactive verification of non-interference; information flow is modeled by a dynamic logic formula, rather than by assertions as in self-composition.

When it comes to *conditional* information flow, the most noteworthy existing tool is the slicer by Snelting et al [27] 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 have recently received the implementation of the approach in [27], and we are currently investigating the deeper technical connections between the two approaches.

Finally, we have already noted how our work has been inspired by and aims to complement previous ground-breaking efforts in certification of MILS infrastructure [16, 17]. 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 Conclusion

We have presented what we believe to be an effective and developer-friendly framework for specification and automatic checking of conditional information flow properties, which are central to verification and certification of information applications built according to the MILS architecture. The directions that we are pursuing are inspired directly by challenge problems presented to us by industry teams using SPARK for MILS component development. The initial prototyping and evaluation of our framework has produced promising results, and we are pressing ahead with evaluating our techniques against actual product codebases developed at Rockwell Colins. A crucial concern in this effort will be to develop design and implementation methodologies for (a) exposing and checking conditional information flows and (b) specifying and checking security levels of data along conditional flows. We believe that our framework will nicely integrate with work on conditional declassification/degrading.

While our framework already supports many of the language features of SPARK and the extension to almost all other features (*e.g.*, records) is straightforward, the primary remaining challenge is the effective treatment of arrays, which are often used in SPARK to implement complex data structures. Rockwell Collins developers are facing significant frustrations because SPARK treats arrays as atomic entities, *i.e.*, it does not

support even unconditional specification and checking of flows in/out of specific array components. We are currently implementing and evaluating a more precise treatment of arrays presented in the extended version of this paper [6].

References

1. Ádám Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *2nd International Conference on Security in Pervasive Computing (SPC 2005)*, volume 3450 of *LNCS*, pages 151–171. Springer, 2005.
2. T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *33rd Principles of Programming Languages (POPL)*, pages 91–102, 2006.
3. T. Amtoft and A. Banerjee. Information flow analysis in logical form. In *11th Static Analysis Symposium (SAS)*, volume 3148 of *LNCS*, pages 100–115. Springer, 2004.
4. T. Amtoft and A. Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Science of Comp. Prog.*, 64(1):3–28, 2007.
5. T. Amtoft and A. Banerjee. Verification condition generation for conditional information flow. In *5th ACM Workshop on Formal Methods in Security Engineering (FMSE)*, 2007. A long version, with proofs, appears as technical report CIS TR 2007-2, Kansas State Univ.
6. T. Amtoft, J. Hatcliff, E. Rodriguez, Robby, J. Hoag, and D. Greve. Specification and checking of software contracts for conditional information flow (extended version). Technical Report SAnToS-TR2007-5, CIS Department, Kansas State University, 2007. Available at `http://www.sireum.org`.
7. A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 2(15):131–177, Mar. 2005.
8. J. Barnes. *High Integrity Software – the SPARK Approach to Safety and Security*. Addison-Wesley, 2003.
9. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, pages 49–69, 2004.
10. G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In R. Foccardi, editor, *CSFW'04*, pages 100–114. IEEE Press, 2004.
11. J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while-programs. *ACM TOPLAS*, 7(1):37–61, Jan. 1985.
12. R. Chapman and A. Hilton. Enforcing security and safety models with an information flow analysis tool. In *SIGAda'04, Atlanta, Georgia*, pages 39–46. ACM, Nov. 2004.
13. E. S. Cohen. Information transmission in sequential programs. In *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
14. D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, pages 108–128, 2004.
15. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
16. D. Greve, M. Wilding, and W. M. Vanfleet. A separation kernel formal security policy. In *4th International Workshop on the ACL2 Prover and its Applications (ACL2-2003)*, 2003.
17. 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 *13th ACM Conference on Computer and Communications Security (CCS'06)*, pages 346–355, 2006.
18. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *10th SPIN Workshop*, volume 2648 of *LNCS*, pages 235–239. Springer, 2003.
19. D. Jackson, M. Thomas, and L. I. Millett, editors. *Software for Dependable Systems: Sufficient Evidence?* National Academies Press, May 2007. Committee on Certifiably Dependable Software Systems, National Research Council.
20. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
21. A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL'99, San Antonio, Texas*, pages 228–241. ACM Press, 1999.
22. D. A. Naumann. From coupling relations to mated invariants for checking information flow. In D. Gollmann, J. Meier, and A. Sabelfeld, editors, *11th European Symposium on Research in Computer Security (ESORICS'06)*, volume 4189 of *LNCS*, pages 279–296. Springer, 2006.
23. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (Lecture Notes in Computer Science 607)*, 1992.
24. 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 International Symposium on Secure Software Engineering*, 2006.
25. J. Rushby. The design and verification of secure systems. In *8th ACM Symposium on Operating Systems Principles*, volume 15(5), pages 12–21, 1981.
26. V. Simonet. Flow Caml in a nutshell. In G. Hutton, editor, *First APPSEM-II workshop*, pages 152–165, Mar. 2003.
27. 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, Oct. 2006.
28. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *12th Static Analysis Symposium*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.
29. M. Vanfleet, J. Luke, R. W. Beckwith, C. Taylor, B. Calloni, and G. Uchenick. MILS: Architecture for high-assurance embedded computing. *CrossTalk: The Journal of Defense Software Engineering*, August 2005.
30. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–188, 1996.
31. Sireum website. `http://www.sireum.org`.