

Specification and Checking of Software Contracts for Conditional Information Flow^{*}

Torben Amtoft¹, John Hatcliff¹, Edwin Rodríguez¹, Robby¹, Jonathan Hoag¹,
David Greve²

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 [30] 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 [26] 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

^{*} 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. The authors gratefully acknowledge the assistance of Rod Chapman in obtaining SPARK examples and running the SPARK tools.

various partitions and between trusted and untrusted segments of a network only when certain *conditions* are satisfied.

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 [7]. 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,

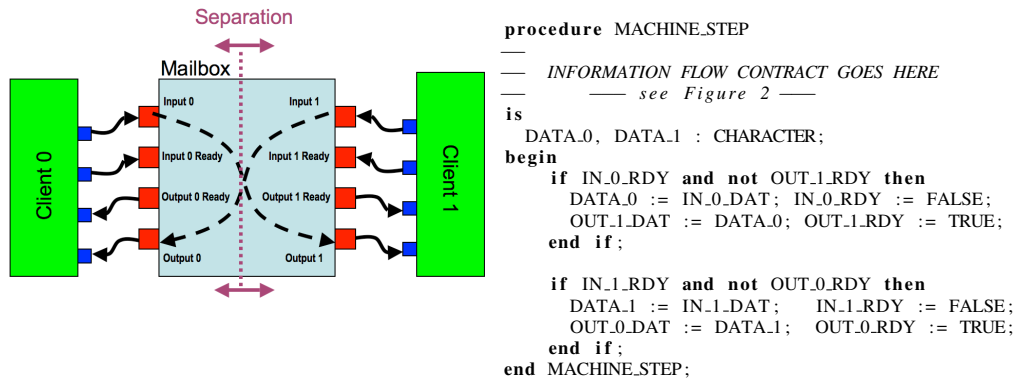


Fig. 1. Simple MILS Guard - mailbox mediates communication between partitions.

- we present a strategy for automatically inferring conditional information flow invariants for while loops, thus significantly reducing developers’ annotation burden,
- 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, 18] used formal models in ACL2 [21] or PVS [24] 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 [20].

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.

```

—# 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.

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), **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

<p><i>Expressions</i></p> <p><i>arithmetic</i></p> <p>$A ::= x \mid c \mid A \text{ op } A \mid H[A]$</p> <p><i>array</i></p> <p>$H ::= h \mid Z \mid H\{A : A\}$</p> <p><i>boolean</i></p> <p>$B ::= A \text{ bop } A$</p> <p><i>Assertions</i></p> <p>$\phi ::= B \mid \phi \wedge \phi$ $\mid \phi \vee \phi \mid \neg\phi$</p>	<p><i>Commands</i></p> <p>$S ::= \text{skip}$</p> <p>$\mid x := A$ assignment</p> <p>$\mid S ; S$ sequential composition</p> <p>$\mid \text{assert}(\phi)$ programmer assertion</p> <p>$\mid \text{call } p$ procedure call</p> <p>$\mid \text{if } B \text{ then } S \text{ else } S$ conditional</p> <p>$\mid \text{while } B \text{ do } S \text{ od}$ iteration</p> <p>$\mid h := \text{new}$ array creation</p> <p>$\mid h[A] := A$ array update</p>
--	--

Fig. 3. Syntax of a simple imperative language

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. In Figure 3, we present the syntax of a simple imperative language with assertions that one can consider to be an idealized version of SPARK. We omit some features of SPARK that do not present conceptual challenges, such as records, and the package and inheritance structure.

Referring to Figure 3, we consider three kind of expressions ($E \in \mathbf{Exp}$): arithmetic ($A \in \mathbf{AExp}$), boolean ($B \in \mathbf{BExp}$), and array expressions ($H \in \mathbf{HExp}$). We use x, y to range over scalar variables, h to range over array variables, and w, z to range over both kind of variables; actual variables appearing in programs are depicted using typewriter font. We also use c to range over integer constants, p to range over named (parameterless) procedures, op to range over arithmetic operators in $\{+, \times, \text{mod}, \dots\}$, and bop to range over comparison operators in $\{=, <, \dots\}$.

The use of programmer assertions is optional, but often helps to improve the precision of our analysis. For example, a loop **while** B **do** S **od** which is known to have invariant ϕ , may be transformed into **while** B **do** **assert**($\phi \wedge B$) ; S **od**; **assert**($\phi \wedge \neg B$).

Using parameterless procedures simplifies our exposition; our implementation supports procedures with parameters (there are no conceptual challenges in this extended functionality).

The SPARK information flow analysis treats arrays as atomic entities – a conservative approximation that makes analysis significantly easier. In the SPARK analysis, any information flowing to/from a particular array element is treated as flowing to/from *all* elements of the array. Due to the lack of heap-allocated data in SPARK, complex data structures are often implemented in arrays. For information assurance applications, the SPARK treatment of information flow for arrays can significantly impede the ability to verify interesting end-to-end information flow properties. Our logic is designed so as to

$$\begin{array}{l}
\llbracket x \rrbracket_s = s(x) \qquad \llbracket h \rrbracket_s = s(h) \\
\llbracket c \rrbracket_s = c \qquad \llbracket Z \rrbracket_s = \lambda n.0 \\
\llbracket H[A] \rrbracket_s = \llbracket H \rrbracket_s(\llbracket A \rrbracket_s) \qquad \llbracket H\{A_0 : A\} \rrbracket_s = [\llbracket H \rrbracket_s \mid \llbracket A_0 \rrbracket_s \mapsto \llbracket A \rrbracket_s] \\
\llbracket A_1 \text{ op } A_2 \rrbracket_s = \llbracket A_1 \rrbracket_s \text{ op } \llbracket A_2 \rrbracket_s \qquad \llbracket A_1 \text{ bop } A_2 \rrbracket_s = \text{True iff } \llbracket A_1 \rrbracket_s \text{ bop } \llbracket A_2 \rrbracket_s
\end{array}$$

Fig. 4. Semantics of expressions

allow us to reason about individual elements of arrays, thus giving more precision than SPARK. The proofs and technical content in this paper support, for an intraprocedural and loop-free setting, this more precise reasoning about arrays, even though our current implementation (as reported on in Section 6) treats arrays as atomic entities, just as SPARK does. Active ongoing work on reasoning about information flow for individual array elements includes (i) integrating it with procedure calls and while loops; (ii) incorporating it into our implementation; (iii) proposing ways of specifying it as part of enhanced SPARK contracts.

To enable reasoning about individual array elements, in intermediate forms of the assertions we shall need the constructs Z (denoting an initial array as created by the command $h := \mathbf{new}$) and $H\{A_0 : A'\}$ (denoting the array H updated such that index A_0 now has value A'). We shall demand, however, that programs (command) are *pure*, where a syntactic entity is pure if all array expressions that occur are array variables.

For an expression E , we write $\text{fv}(E)$ for the variables occurring free in E which is the union of the free scalar variables $\text{fsv}(E)$ and the free array variables $\text{fav}(E)$, and write $E[A/x]$ (or $E[H/h]$) for the result of substituting in E all occurrences of x by A (or h by H). We use similar notations for assertions ϕ , where as usual we define $\phi_1 \rightarrow \phi_2$ as $\neg\phi_1 \vee \phi_2$, and also define *true* as $0 = 0$, and *false* as $0 = 1$.

Semantics: We model an array as a mapping ($a \in \text{Array}$) from integers to values, where a value ($v \in \text{Val}$) 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 . To keep the exposition simple, we shall ignore bounds and range checks, and assume that an array reference $a(n)$ is always well-defined. A store $s \in \text{Store}$ maps scalar variables to values, and array variables to arrays; we write $\text{dom}(s)$ for the domain of s and write $[s \mid x \mapsto v]$ for the store that is like s except that it maps x into v , write $[s \mid h \mapsto a]$ for the store that is like s except that it maps h into a , and write $[s \mid h(n) \mapsto v]$ for $[s \mid h \mapsto [h \mid n \mapsto v]]$.

The semantics of expressions is defined recursively in Fig. 4. For an arithmetic expression A , the semantics $\llbracket A \rrbracket$ is a function from stores into values; similarly, $\llbracket B \rrbracket_s$ denotes a boolean, and $\llbracket H \rrbracket_s$ denotes an array.

The satisfaction relation for assertions reads $s \models \phi$ and denotes that ϕ holds in s . The definition is inductive in ϕ : $s \models B$ iff $\llbracket B \rrbracket_s = \text{True}$; $s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$, etc. 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 $\phi \triangleright_1 \phi'$, when ϕ logically implies ϕ' . In Appendix (p. 35), we prove the following result:

Lemma 1. *Given ϕ , we can construct pure ϕ' such that $\phi \equiv_1 \phi'$.*

A command transforms the store into another store; hence its semantics is given in relational style, in the form $s \llbracket S \rrbracket s'$. The semantics is given in Fig. 5, and defined inductively on S ; implicitly we assume a global procedure environment P that for each p returns a relation between input and output stores (we expect that if $s P(p) s'$ then, with S the body of p , we have $s \llbracket S \rrbracket s'$). For some S and s , there may not exist any s' such that $s \llbracket S \rrbracket s'$; this can happen if a **while** loop does not terminate, or an **assert** fails.

$$\begin{aligned}
& s \llbracket \text{skip} \rrbracket s' \text{ iff } s' = s \\
& s \llbracket x := A \rrbracket s' \text{ iff } \exists v : v = \llbracket A \rrbracket_s \text{ and } s' = [s \mid x \mapsto v] \\
& s \llbracket S_1 ; S_2 \rrbracket s' \text{ iff } \exists s'' : s \llbracket S_1 \rrbracket s'' \text{ and } s'' \llbracket S_2 \rrbracket s' \\
& s \llbracket \text{assert}(\phi) \rrbracket s' \text{ iff } s \models \phi \text{ and } s' = s \\
& s \llbracket \text{call } p \rrbracket s' \text{ iff } s P(p) s' \\
& s \llbracket \text{if } B \text{ then } S_1 \text{ else } S_2 \rrbracket s' \text{ iff } (\llbracket B \rrbracket_s = \text{True and } s \llbracket S_1 \rrbracket s') \\
& \quad \text{or } (\llbracket B \rrbracket_s = \text{False and } s \llbracket S_2 \rrbracket s') \\
& s \llbracket \text{while } B \text{ do } S \text{ od} \rrbracket s' \text{ iff } \exists i \geq 0 : s f_i s' \text{ where } f_i \text{ is inductively defined by:} \\
& \quad s f_0 s' \text{ iff } \llbracket B \rrbracket_s = \text{False and } s' = s \\
& \quad s f_{i+1} s' \text{ iff } \exists s'' : (\llbracket B \rrbracket_s = \text{True and} \\
& \quad \quad s \llbracket S \rrbracket s'' \text{ and } s'' f_i s') \\
& s \llbracket h[A_0] := A \rrbracket s' \text{ iff } \exists n, v : n = \llbracket A_0 \rrbracket_s, v = \llbracket A \rrbracket_s \text{ and } s' = [s \mid h(n) \mapsto v] \\
& s \llbracket h := \text{new} \rrbracket s' \text{ iff } s' = [s \mid h \mapsto \lambda n.0]
\end{aligned}$$

Fig. 5. Command semantics

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 $x_1 \dots x_n$, the runs agree on variables $y_1 \dots y_m$ at the end of the runs.* This includes non-interference as a special case, as can be seen by letting $x_1 \dots x_n$, and $y_1 \dots y_m$, 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 $\{x_1 \times, \dots, x_n \times\} P \{y_1 \times, \dots, y_m \times\}$, where the *agreement assertion* $x \times$ 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.,

$$\{\text{INP_1_DAT} \times, \text{OUT_0_DAT} \times, \text{INP_1_RDY} \times, \text{OUT_0_RDY} \times\} P \{\text{OUT_0_DAT}\}.$$

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 $\phi \Rightarrow E \times$ 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:

$$s \& s_1 \models \phi \Rightarrow E \times \text{ iff whenever } s \models \phi \text{ and } s_1 \models \phi \text{ then } \llbracket E \rrbracket_s = \llbracket E \rrbracket_{s_1}.$$

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

Fig. 6 depicts 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 `OUT_0_DAT` to hold after execution of P , we must have `DATA_1` before line 3 (since data flows from `DATA_1` to

$$\begin{aligned} & \{ \text{INP_1_RDY} \wedge \neg \text{OUT_0_RDY} \Rightarrow \text{INP_1_DAT} \times, \\ & \quad \neg \text{INP_1_RDY} \vee \text{OUT_0_RDY} \Rightarrow \text{OUT_0_DAT} \times, \\ & \quad \text{INP_1_RDY} \times, \text{OUT_0_RDY} \times \} \\ 1. & \text{ if } \text{INP_1_RDY} \text{ and not } \text{OUT_0_RDY} \text{ then} \\ & \quad \{ \text{INP_1_DAT} \times \} \\ 2. & \text{ DATA_1} := \text{INP_1_DAT}; \text{ INP_1_RDY} := \text{false}; \\ & \quad \{ \text{DATA_1} \times \} \\ 3. & \text{ OUT_0_DAT} := \text{DATA_1}; \text{ OUT_0_RDY} := \text{true}; \\ & \quad \{ \text{OUT_0_DAT} \times \} \\ 4. & \text{ fi} \\ & \quad \{ \text{OUT_0_DAT} \times \} \end{aligned}$$

Fig. 6. A derivation for the mailbox example, illustrating the handling of conditionals.

OUT_0_DAT), $\text{INP_1_DAT} \times$ before line 2 (since data flows from INP_1_DAT to DATA_1), and finally $\text{INP_1_RDY} \times$ and $\text{OUT_0_RDY} \times$ (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 $\Theta \triangleright_2 \Theta'$, 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 $\Theta \triangleright_2 \Theta'$ and $\Theta' \triangleright_2 \Theta$. In development terms, when $\Theta \triangleright_2 \Theta'$ holds we can think of Θ as a *refinement* of Θ' , and Θ' an *abstraction* of Θ . For example, $\{x \times, y \times\}$ refines $x \times$ by adding an (unconditional) agreement assertion, and $y < 10 \Rightarrow x \times$ refines $y < 7 \Rightarrow x \times$ by weakening the antecedent of a 2-assertion. In general, logical implication on **1Assert** conditions in agreement assertions is related in a contravariant manner to logical implication in agreement assertions; this is a special case (with $E = E_0$) of the following result:

Lemma 2. *Assume that (a) $\phi \triangleright_1 \phi_0$, and (b) whenever $s \models \phi$ and $s_1 \models \phi$ and $\llbracket E_0 \rrbracket_s = \llbracket E_0 \rrbracket_{s_1}$ then also $\llbracket E \rrbracket_s = \llbracket E \rrbracket_{s_1}$. Then $(\phi_0 \Rightarrow E_0 \times) \triangleright_2 (\phi \Rightarrow E \times)$.*

Proof: Assuming (i) $s \& s_1 \models \phi_0 \Rightarrow E_0 \times$ and (ii) $s \models \phi$ and $s_1 \models \phi$, we must prove $\llbracket E \rrbracket_s = \llbracket E \rrbracket_{s_1}$. By (a), we infer from (ii) that $s \models \phi_0$ and $s_1 \models \phi_0$, which by (i) implies $\llbracket E_0 \rrbracket_s = \llbracket E_0 \rrbracket_{s_1}$. By (b), we get the desired $\llbracket E \rrbracket_s = \llbracket E \rrbracket_{s_1}$. \square

We define a function *decomp* that converts arbitrary 2-assertions into assertions with only variables as consequents: $\text{decomp}(\Theta) = \{\phi \Rightarrow w \times \mid \phi \Rightarrow E \times \in \Theta, w \in \text{fv}(E)\}$.

Fact 3 *For all Θ , $\text{decomp}(\Theta)$ is a refinement of Θ .*

The converse does not hold; for example we do not have $\{(x + y) \times\} \triangleright_2 \{x \times, y \times\}$ 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 $s(x) = s_1(y) = 3$, $s(y) = s_1(x) = 7$.

The following result is useful to refine 2-assertions while removing non-pure constructs from their consequents (a 2-assertion $\phi \Rightarrow Z[A] \times$ is refined by any Θ).

Lemma 4. *A 2-assertion $\phi \Rightarrow H\{A_0 : A'\}[A] \times$ is refined by $\Theta = \{\theta_1, \theta_2, \theta_3\}$ where*

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

Proof: Assuming $s \& s_1 \models \Theta$ and $s \models \phi$ and $s_1 \models \phi$, we must prove $\llbracket H\{A_0 : A'\}[A] \rrbracket_s = \llbracket H\{A_0 : A'\}[A] \rrbracket_{s_1}$. From $s \& s_1 \models \theta_3$ we have $\llbracket A = A_0 \rrbracket_s = \llbracket A = A_0 \rrbracket_{s_1}$. There are now two cases:

$\llbracket A \rrbracket_s = \llbracket A_0 \rrbracket_s$: Then also $\llbracket A \rrbracket_{s_1} = \llbracket A_0 \rrbracket_{s_1}$, showing $s \models A = A_0$ and $s_1 \models A = A_0$.

From $s \& s_1 \models \theta_1$ we thus infer $\llbracket A' \rrbracket_s = \llbracket A' \rrbracket_{s_1}$, which amounts to the claim.

$\llbracket A \rrbracket_s \neq \llbracket A_0 \rrbracket_s$: Then also $\llbracket A \rrbracket_{s_1} \neq \llbracket A_0 \rrbracket_{s_1}$, showing $s \models A \neq A_0$ and $s_1 \models A \neq A_0$.

From $s \& s_1 \models \theta_2$ we thus infer $\llbracket H[A] \rrbracket_s = \llbracket H[A] \rrbracket_{s_1}$, which amounts to the claim.

□

Note that in Lemma 4, the converse direction does not hold: for example, we might have $s \& s_1 \models h\{x : 7\}[y] \times$ but not $s \& s_1 \models (y = x) \times$, as when $s(y) = 3$ and $s(x) = 3$, and $s_1(y) = 5$ and $s_1(x) = 4$ and $s_1(h) = a_1$ where $a_1(5) = 7$. We *conjecture* that for an impure 2-assertion ϕ , in general there does not exist a pure 2-assertion ϕ' which is 2-equivalent to ϕ .

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 $\text{derives}(z, \bar{w}) \in \mathcal{D}_P$ declares that the final value of output variable z depends on the input values of variables $\bar{w} = w_1, \dots, w_k$. 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 $\{\bar{w} \times\} S \{z \times\}$ where S is the body of procedure P and where $\bar{w} \times$ is a shorthand for $\{w_1 \times, \dots, w_k \times\}$.

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 $\{\text{derives}(z, \bar{w}), \text{derives}(x, \bar{y})\}$ as $\{(\bar{w}\bar{y}) \times\} S \{z \times, x \times\}$ because this would confuse the dependencies associated with z and x , *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* $z \times_c$ 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 $\{\bar{w} \times_z, \bar{y} \times_x\} S \{z \times_z, x \times_x\}$. 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 $s_1 \llbracket S \rrbracket s'_1$ and $s_2 \llbracket S \rrbracket s'_2$ 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 $\phi \Rightarrow E \times$. Here E is an arbitrary expression (not necessarily a variable), whereas SPARK `derives` statements are phrased in terms of IN/OUT variables 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 $\phi \Rightarrow E \times$ only during intermediate (automated) steps of the analysis, and appealing to Fact 3, we have a canonical way of strengthening $\phi \Rightarrow E \times$ to $\phi \Rightarrow w_1 \times, \dots, \phi \Rightarrow w_k \times$ where $w_1, \dots, w_k \in \text{fv}(E)$ 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., $\{\phi_1 \Rightarrow E_1 \times\} P \{\phi_2 \Rightarrow E_2 \times\}$ 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:

$$\text{derives } z \text{ from } w_1 \text{ when } \phi_1, \dots, w_k \text{ when } \phi_k$$

Here $\phi_1 \dots \phi_k$ are boolean expressions on the pre-state of the associated procedure P . Thus, the above specification can be read as “The value of variable z at the conclusion of executing P (for *any* final state s') is derived from w_j when ϕ_j holds in the pre-state s from which s' is computed (if also ϕ_i holds in s , then z depends also on w_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.

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: $\kappa_1 \triangleright_{\kappa} \kappa_2$, pronounced “ κ_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 $\Theta_2 \triangleright_2 \Theta_1$ and $\Theta'_1 \triangleright_2 \Theta'_2$ then $\kappa_1 \triangleright_{\kappa} \kappa_2$. For example, $x \times \rightsquigarrow y \times \triangleright_{\kappa} x \times, y \times \rightsquigarrow y \times$ holds because $x \times, y \times \triangleright_2 x \times$ (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 $x \times \rightsquigarrow y \times$ shows that output y need not depend on input y (the contract before refinement is an *over-approximation* of dependence information). Also, we have $(y < 7 \Rightarrow x \times \rightsquigarrow z \times) \triangleright_{\kappa} (x \times \rightsquigarrow z \times)$ 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. 7 an algorithm Pre for inferring preconditions from postconditions. We write $\{\Theta\} (R) \Leftarrow S \{\Theta'\}$ 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 $\{y_1 y_2 \times_x, y_1 y_3 \times_z\} S \{x \times_x, z \times_z\}$ 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

$$\begin{aligned}
\{\Theta\} (R) &\Leftarrow \mathbf{skip} \{\Theta'\} \text{ iff } R = \{(\theta, u, \theta) \mid \theta \in \Theta'\} \text{ and } \Theta = \Theta' \\
\{\Theta\} (R) &\Leftarrow \mathbf{assert}(\phi_0) \{\Theta'\} \text{ iff } R = \{((\phi \wedge \phi_0) \Rightarrow E \times, u, \phi \Rightarrow E \times) \mid \phi \Rightarrow E \times \in \Theta'\} \text{ and } \Theta = \text{dom}(R) \\
\{\Theta\} (R) &\Leftarrow x := A \{\Theta'\} \text{ iff } R = \{(\phi[A/x] \Rightarrow E[A/x] \times, \gamma, \phi \Rightarrow E \times) \mid \phi \Rightarrow E \times \in \Theta'\}, \\
&\text{ where } \gamma = m \text{ iff } x \in \text{fv}(E) \text{ and } \Theta = \text{dom}(R) \\
\{\Theta\} (R) &\Leftarrow h[A_0] := A \{\Theta'\} \text{ iff } R = \{(\phi[h\{A_0 : A\}/h] \Rightarrow E[h\{A_0 : A\}/h] \times, \gamma, \phi \Rightarrow E \times) \mid \phi \Rightarrow E \times \in \Theta'\} \\
&\text{ where } \gamma = m \text{ iff } h \in \text{fv}(E) \text{ and } \Theta = \text{dom}(R) \\
\{\Theta\} (R) &\Leftarrow h := \mathbf{new} \{\Theta'\} \text{ iff } R = \{(\phi[Z/h] \Rightarrow E[Z/h] \times, \gamma, \phi \Rightarrow E \times) \mid \phi \Rightarrow E \times \in \Theta'\} \\
&\text{ where } \gamma = m \text{ iff } h \in \text{fv}(E) \text{ and } \Theta = \text{dom}(R) \\
\{\Theta\} (R) &\Leftarrow S_1 ; S_2 \{\Theta'\} \text{ iff } \{\Theta''\} (R_2) \Leftarrow S_2 \{\Theta'\} \text{ and } \{\Theta\} (R_1) \Leftarrow S_1 \{\Theta''\} \\
&\text{ and } R = \{(\theta, \gamma, \theta') \mid \exists \theta'', \gamma_1, \gamma_2 : (\theta, \gamma_1, \theta'') \in R_1, (\theta'', \gamma_2, \theta') \in R_2\}, \text{ where } \gamma = m \text{ iff } \gamma_1 = m \text{ or } \gamma_2 = m \\
\{\Theta\} (R) &\Leftarrow \mathbf{if } B \mathbf{ then } S_1 \mathbf{ else } S_2 \{\Theta'\} \\
&\text{ iff } \{\Theta_1\} (R_1) \Leftarrow S_1 \{\Theta'\}, \{\Theta_2\} (R_2) \Leftarrow S_2 \{\Theta'\}, R = R'_1 \cup R'_2 \cup R'_0 \cup R_0, \text{ and } \Theta = \text{dom}(R), \\
&\text{ where } R'_1 = \{((\phi_1 \wedge B) \Rightarrow E_1 \times, m, \theta') \mid \theta' \in \Theta'_m, (\phi_1 \Rightarrow E_1 \times, -, \theta') \in R_1\} \\
&\text{ and } R'_2 = \{((\phi_2 \wedge \neg B) \Rightarrow E_2 \times, m, \theta') \mid \theta' \in \Theta'_m, (\phi_2 \Rightarrow E_2 \times, -, \theta') \in R_2\} \\
&\text{ and } R'_0 = \{(((\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)) \Rightarrow B \times, m, \theta') \\
&\quad \mid \theta' \in \Theta'_m, (\phi_1 \Rightarrow E_1 \times, -, \theta') \in R_1, (\phi_2 \Rightarrow E_2 \times, -, \theta') \in R_2\} \\
&\text{ and } R_0 = \{(((\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)) \Rightarrow E \times, u, \theta') \\
&\quad \mid \theta' \in \Theta'_u, (\phi_1 \Rightarrow E \times, u, \theta') \in R_1, (\phi_2 \Rightarrow E \times, u, \theta') \in R_2\} \\
&\text{ and } \Theta'_m = \{\theta' \in \Theta' \mid \exists(-, m, \theta') \in R_1 \cup R_2\} \text{ and } \Theta'_u = \Theta' \setminus \Theta'_m \\
\{\Theta\} (R) &\Leftarrow \mathbf{call } p \{\Theta'\} \\
&\text{ iff } R = R_u \cup R_0 \cup R_m \text{ and } \Theta = \text{dom}(R), \\
&\text{ where } R_u = \{(rm_{\text{OUT}_p}^+(\phi) \Rightarrow E \times, u, \phi \Rightarrow E \times) \mid (\phi \Rightarrow E \times) \in \Theta' \wedge \text{fv}(E) \cap \text{OUT}_p = \emptyset\} \\
&\text{ and } R_0 = \{(rm_{\text{OUT}_p}^+(\phi) \Rightarrow w \times, m, \phi \Rightarrow E \times) \\
&\quad \mid (\phi \Rightarrow E \times) \in \Theta' \wedge \text{fv}(E) \cap \text{OUT}_p \neq \emptyset \wedge w \in \text{fv}(E) \wedge w \notin \text{OUT}_p\} \\
&\text{ and } R_m = \{(rm_{\text{OUT}_p}^+(\phi) \wedge \phi_w^z \Rightarrow z \times, m, \phi \Rightarrow E \times) \\
&\quad \mid (\phi \Rightarrow E \times) \in \Theta' \wedge w \in \text{fv}(E) \cap \text{OUT}_p \wedge \phi_w^z \Rightarrow z \times \text{ among preconditions for } w \times \text{ in } p\text{'s summary}\} \\
\{\Theta\} (R) &\Leftarrow \mathbf{while } B \mathbf{ do } S_0 \mathbf{ od } \{\Theta'\} \\
&\text{ iff } R = R_u \cup R_m \text{ and } \Theta = \text{dom}(R), \\
&\text{ where for each } w \in X \text{ (with } X \text{ the variables "involved") we inductively in } i \text{ define } \phi_w^i, \Theta^i, R^i, \psi_w^i \text{ by} \\
&\quad \phi_w^0 = \bigvee \{\phi \mid \exists E : (\phi \Rightarrow E \times) \in \Theta' \wedge w \in \text{fv}(E)\}, \Theta^i = \{\phi_w^i \Rightarrow w \times \mid w \in X\}, \{-\} (R^i) \Leftarrow S_0 \{\Theta^i\} \\
&\quad \psi_w^i = \bigvee \{\phi \mid \exists(\phi \Rightarrow E \times, -, -) \in R^i \text{ with } w \in \text{fv}(E) \\
&\quad \text{ or } w \in \text{fv}(B) \text{ and } \exists(\theta, m, \theta') \in R^i \text{ with } \phi = \text{ant}(\theta) \text{ or } \phi = \text{ant}(\theta')\} \\
&\quad \phi_w^{i+1} = \text{if } \psi_w^i \triangleright_1 \phi_w^i \text{ then } \phi_w^i \text{ else } \phi_w^i \nabla \psi_w^i \\
&\text{ and } j \text{ is the least } i \text{ such that } \Theta^i = \Theta^{i+1} \\
&\text{ and } R_u = \{(\phi \Rightarrow E \times, u, \theta') \mid \theta' \in \Theta'_u, E = \text{con}(\theta'), (\text{fv}(E) = \emptyset, \phi = \text{true}) \vee (\text{fv}(E) \neq \emptyset, \phi = \bigvee_{w \in \text{fv}(E)} (\phi_w^j))\} \\
&\text{ and } R_m = \{(\theta, m, \theta') \mid \theta' \in \Theta'_m \wedge \theta \in \Theta^j \cup \{\text{true} \Rightarrow 0 \times\}\} \\
&\text{ and } \Theta'_m = \{\theta' \in \Theta' \mid \exists w \in \text{fv}(\text{con}(\theta')) : \exists(-, m, - \Rightarrow w \times) \in R^j\} \text{ and } \Theta'_u = \Theta' \setminus \Theta'_m
\end{aligned}$$

Fig. 7. The Precondition Generator

is one occurring in the consequent of θ' . We use γ to range over $\{m, u\}$, and write $\text{dom}(R) = \{\theta \mid \exists(\theta, -, -) \in R\}$ and $\text{ran}(R) = \{\theta' \mid \exists(-, -, \theta') \in R\}$.

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

Theorem 1 (Correctness). *Assume $\{\Theta\} (-) \Leftarrow S \{\Theta'\}$. Then $S \models \Theta \rightsquigarrow \Theta'$. That is, if $s \& s_1 \models \Theta$, and s', s'_1 are such that $s \llbracket S \rrbracket s'$ and $s_1 \llbracket S \rrbracket s'_1$, then $s' \& s'_1 \models \Theta'$.*

Note that Theorem 1 is termination-insensitive; 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 1 is proved in Appendix (p. 41), 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 that have largely determined the design of Pre. The first such property is a variant of the “*-property” by Bell and La Padula [10], also called “write confinement” [6], which is used to preclude, e.g., “low writes under high guards”. In our setting, it captures the role of the u tag and reads as follows:

Lemma 5. *Assume $\{\Theta\} (R) \Leftarrow S \{\Theta'\}$. Then $\text{dom}(R) = \Theta$ and $\text{ran}(R) = \Theta'$. Given $\theta' \in \Theta'$, there exists at most one θ such that $(\theta, u, \theta') \in R$. If there exists such θ , then $\text{con}(\theta) = \text{con}(\theta')$, and with $E = \text{con}(\theta)$ we have that if $s \llbracket S \rrbracket s'$ then s agrees with s' on $\text{fv}(E)$.*

Lemma 5, proved in Appendix (p. 38), is needed in the proof of Theorem 1 to handle the case where the two runs in question follow *different branches* in a conditional, as we must then ensure that neither run modifies a variable on which we want the two runs to agree afterwards. We shall also use a lemma, proved in Appendix (p. 39), which expresses that there will always be one applicable condition in the precondition:

Lemma 6. *Assume $\{\Theta\} (R) \Leftarrow S \{\Theta'\}$. Given $\theta' \in \Theta'$, there exists $(\theta, _, \theta') \in R$ such that whenever $s \llbracket S \rrbracket s'$ and $s' \models \text{ant}(\theta')$ then $s \models \text{ant}(\theta)$.*

Intraprocedural analysis: We now explain the various clauses of Pre in Fig. 7, where the clause for **skip** is trivial. For an assignment $x := A$, each 2-assertion $\phi \Rightarrow E \times$ 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 $(y > 4 \Rightarrow w \times, m, y > 4 \Rightarrow x \times)$ and $(w > 3 \Rightarrow z \times, u, x > 3 \Rightarrow z \times)$.

The rule for $h[A_0] := A$ captures that semantically, such an update is really an assignment $h := h\{A_0 : A\}$ and can therefore be treated just as an assignment [17]: for each 2-assertion $\phi \Rightarrow E \times$ in Θ' , we substitute $h\{A_0 : A\}$ for h , and tag the connection m only when h occurs in E . (One could imagine a more precise rule which, e.g., for command $h[1] := 7$ would allow postcondition $\text{true} \Rightarrow h[2] \times$ to be tagged u ; to prove the soundness of such a rule, we would need a more complex statement of Lemma 5.) The rule for $h := \text{new}$ similarly captures that this is really an assignment $h := Z$.

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. 6 where, given postcondition $\text{OUT_0_DAT} \times$, the **then** branch generates (as the domain of R_1) precondition $\text{INF_1_DAT} \times$ 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 $\text{OUT_0_DAT} \times$ 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

Summary information for p :

$\text{OUT}_p = \{x\}$

derives x from y, z when $y > 0$, w when $y \leq 0$

Procedure call

$$\{z > 7 \Rightarrow v \times, z > 5 \Rightarrow u \times, z > 5 \Rightarrow y \times, z > 5 \wedge y > 0 \Rightarrow z \times, z > 5 \wedge y \leq 0 \Rightarrow w \times\}$$

call p

$$\{x > 5 \wedge z > 7 \Rightarrow v \times, x > 7 \wedge z > 5 \Rightarrow (x + u) \times\}$$

Fig. 8. An example illustrating the handling of procedure calls.

by R'_0 which generates the precondition $(\text{true} \wedge B) \vee (\text{true} \wedge \neg B) \Rightarrow B \times$; optimizations (not shown) in our algorithm simplify this to $B \times$ and then use Fact. 3 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 $\phi \Rightarrow E \times$ where E is not modified by either branch: if also ϕ is not modified then $\phi \Rightarrow E \times$ 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 $E \times$.

Interprocedural analysis: A procedure summary for p must satisfy:

1. if $s P(p) s'$ then $s(w) = s'(w)$ for all $w \notin \text{OUT}_p$;
2. a postcondition is of the form $w \times$ with $w \in \text{OUT}_p$, and for each $w \in \text{OUT}_p$ there is exactly one postcondition $w \times$;
3. a precondition is of the form $\phi \Rightarrow z \times$;
4. for each postcondition $w \times$ there is a precondition of the form $\text{true} \Rightarrow z \times$;
5. for each postcondition $w \times$, with Θ_w its preconditions: if $s \& s_1 \models \Theta_w$ and $s P(p) s'$ and $s_1 P(p) s'_1$ then $s'(w) = s'_1(w)$.

Here Requirement 1 expresses that OUT_p does indeed contain all output variables. Requirements 2 and 3 were motivated in Sect. 4.2. Requirement 4 is needed for Lemma 6 to hold; it might seem restrictive but can always be established without losing precision, as by adding $\text{true} \Rightarrow \text{q} \times$ where q is a variable not occurring in the program. Requirement 5 expresses that the summary computes correct information flow.

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 $rm^+_{\{x\}}(x > 7 \wedge z > 5) = (z > 5)$ as is done by the definition of rm^+ given in Appendix (p.36).

Equipped with rm^+ , we can now define the analysis of procedure call, as done in Fig. 7 and illustrated in Fig. 8. Here R_u deals with assertions (such as $x > 5 \wedge z > 7 \Rightarrow v \times$ 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 $x > 7 \wedge z > 5 \Rightarrow (x + u) \times$), 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

<pre> while i < 7 do if odd(i) then r := r + v; v := v + h else v := x; i := i + 1 {r×} </pre>	<table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="border-bottom: 1px solid black; padding: 2px 10px;">Iteration</th> <th style="border-bottom: 1px solid black; padding: 2px 10px;">0</th> <th style="border-bottom: 1px solid black; padding: 2px 10px;">1</th> <th style="border-bottom: 1px solid black; padding: 2px 10px;">2</th> <th style="border-bottom: 1px solid black; padding: 2px 10px;">3</th> <th style="border-bottom: 1px solid black; padding: 2px 10px;"></th> </tr> </thead> <tbody> <tr> <td style="padding: 2px 10px;"><i>false</i></td> <td style="padding: 2px 10px;"><i>false</i></td> <td style="padding: 2px 10px;"><i>false</i></td> <td style="padding: 2px 10px;"><i>false</i></td> <td style="padding: 2px 10px;"><i>false</i></td> <td style="padding: 2px 10px;">$\Rightarrow h \times$</td> </tr> <tr> <td style="padding: 2px 10px;"><i>false</i></td> <td style="padding: 2px 10px;"><i>true</i></td> <td style="padding: 2px 10px;"><i>true</i></td> <td style="padding: 2px 10px;"><i>true</i></td> <td style="padding: 2px 10px;"><i>true</i></td> <td style="padding: 2px 10px;">$\Rightarrow i \times$</td> </tr> <tr> <td style="padding: 2px 10px;"><i>true</i></td> <td style="padding: 2px 10px;"><i>true</i></td> <td style="padding: 2px 10px;"><i>true</i></td> <td style="padding: 2px 10px;"><i>true</i></td> <td style="padding: 2px 10px;"><i>true</i></td> <td style="padding: 2px 10px;">$\Rightarrow r \times$</td> </tr> <tr> <td style="padding: 2px 10px;"><i>false</i></td> <td style="padding: 2px 10px;"><i>odd(i)</i></td> <td style="padding: 2px 10px;"><i>odd(i)</i></td> <td style="padding: 2px 10px;"><i>odd(i)</i></td> <td style="padding: 2px 10px;"><i>odd(i)</i></td> <td style="padding: 2px 10px;">$\Rightarrow v \times$</td> </tr> <tr> <td style="padding: 2px 10px;"><i>false</i></td> <td style="padding: 2px 10px;"><i>false</i></td> <td style="padding: 2px 10px;">$\neg \text{odd}(i)$</td> <td style="padding: 2px 10px;"><i>true</i></td> <td style="padding: 2px 10px;"><i>true</i></td> <td style="padding: 2px 10px;">$\Rightarrow x \times$</td> </tr> </tbody> </table>	Iteration	0	1	2	3		<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	$\Rightarrow h \times$	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	$\Rightarrow i \times$	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	$\Rightarrow r \times$	<i>false</i>	<i>odd(i)</i>	<i>odd(i)</i>	<i>odd(i)</i>	<i>odd(i)</i>	$\Rightarrow v \times$	<i>false</i>	<i>false</i>	$\neg \text{odd}(i)$	<i>true</i>	<i>true</i>	$\Rightarrow x \times$
Iteration	0	1	2	3																																	
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	$\Rightarrow h \times$																																
<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	$\Rightarrow i \times$																																
<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	$\Rightarrow r \times$																																
<i>false</i>	<i>odd(i)</i>	<i>odd(i)</i>	<i>odd(i)</i>	<i>odd(i)</i>	$\Rightarrow v \times$																																
<i>false</i>	<i>false</i>	$\neg \text{odd}(i)$	<i>true</i>	<i>true</i>	$\Rightarrow x \times$																																

Fig. 9. Iterative analysis of while loop. (We use $\text{odd}(i)$ as a shorthand for $i \bmod 2 = 1$.)

conditions”); for those in OUT_p (such as \times), 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 $\phi_x \Rightarrow x \times$ and then repeatedly analyze the loop body so as to iteratively weaken the antecedents until a fixed point is reached. To illustrate the overall behavior, consider the example in Fig. 9 where we are given $r \times$ 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 $\text{odd}(i)$, since v is used to compute r when i is odd, and also updates i ’s antecedent to *true*, since (the parity of) i is used to decide whether r is updated or not. The second iteration updates x ’s antecedent to $\neg \text{odd}(i)$, 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 $r \times$, 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” ∇ on 1-assertions, with the following properties:

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

A trivial widening operator is the one that always returns *true*, in effect converting conditional agreement assertions into unconditional. A less trivial option will utilize a number of assertions, say $\psi_1 \dots \psi_k$, and allow $\psi \nabla \phi = \psi_j$ 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 $\psi_1 \dots \psi_k$ as antecedent.

We can now explain the various lines in the clause for while loops in Fig. 7. The iteration starts with antecedents ϕ_w^0 that are computed such that the corresponding 2-assertion, Θ^i , implies the postcondition Θ' . The i ’th iteration updates the antecedents ϕ_w^i into antecedents ϕ_w^{i+1} that are potentially weaker in that for each $w \in X$, each disjunct of ψ_w^i must imply ϕ_w^{i+1} ; here ψ_w^i captures the “business logic” of the while loop:

1. if the precondition computed for the iteration contains an assertion $\phi \Rightarrow E \times$ with $w \in \text{fv}(E)$, then ϕ is an element of ψ_w^i .

2. if a consequent has been modified by the loop body, then the antecedent must belong to ψ_w^i for all $w \in \text{fv}(B)$.

Here (2) ensures that if one run stays in the loop and updates a variable on which the two runs must agree, then also the other run stays in the loop, cf. the role of R'_0 did in the clause for conditionals, whereas (1) caters for the soundness when both runs stay in the loop, cf. the role of R'_1 and R'_2 in the case for conditionals. Alternatively, to more closely follow the rule for conditionals, for (1) we could instead demand that $\phi \wedge B$ belongs to ψ_w^i ; our current choice reflects that we expect the bodies of **while** loops to be prefixed by **assert** statements (which will automatically add B to the antecedents), but do not expect such transformations for branches of a conditional.

With the iteration stabilizing after j steps (thanks to the widening operator), the while loop's precondition Θ and its R component can now be computed; the former is given as the domain of the latter which is made up from two parts.

- First, R_u deals with those assertions in Θ' whose consequents have not been modified (a kind of “frame condition” for the while loop); each such assertion is connected to an assertion with the same consequent (so as to establish Lemma 5) but with an antecedent that is designed to be so weak that we can establish Lemma 6.
- Next, R_m deals with those assertions in Θ' whose consequents have been modified; each such assertion is connected to *all* other assertions in Θ^j so as to express that the subsequent iterations of the while loop may give rise to chains of variable dependences. (It would be possible to give a definition that in most cases produces only a subset of those connections, but this would increase the conceptual complexity of Pre, without – we conjecture – any improvement in the overall precision of the algorithm.) In addition, again to establish Lemma 6, we introduce a trivial assertion $\text{true} \Rightarrow 0 \times$.

6 Evaluation

6.1 Summary of performance

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 $w_o^1 \times_1, \dots, w_o^k \times_k$ for each $w_o^j \in \text{OUT}$. 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

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)

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 [18]. 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 [32] 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 7 directly, with just one small optimization: a collection of boolean simplifications are introduced, e.g., simplifying assertions of the form $true \wedge \phi \Rightarrow E \times$ to

$\phi \Rightarrow E \times$. 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 $\phi \Rightarrow E \times$ when $true \Rightarrow E \times$ 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 seen 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?

6.2 Detailed discussion of selected examples

In this section we give a detailed discussion of two case studies: the Mailbox example (briefly discussed in Sec. 2), and part of the control code for the Autopilot code base (for which number figures were given in Tab. 1). For the Mailbox, we will simply discuss in detail the `MACHINE_STEP` procedure, previously introduced, comparing the results of running our tool with the original SPARK specification. On the other hand, in the Autopilot case study we will discuss 4 procedures and 2 functions, spanning a whole call chain in the package, starting at the Main procedure, and going through the code that controls the altitude in this simplified example of an aircraft autopilot.

Mailbox Example This example was discussed in detail in Sec. 2, so here we will focus in comparing the resulting information flow specifications obtained from running our tool on the code with the original SPARK specification. Figure 10 shows the procedure `MACHINE_STEP` with the original SPARK information flow specification. On the other hand, Fig. 11 shows the information flow specification obtained by running our tool on the same procedure (using the slightly modified version of the SPARK language described in Sec. 4.2). For simplicity, in Fig. 11 we have omitted the body of the procedure, as well as the `global` annotations. In addition to using unabbreviated variable names, the code of Figure 10 differs from that of Figure 1 in its use of procedures to manipulate both the control variables (e.g., `Mailbox.CHARACTER_INPUT_0_READY`) as well as the data variables of the system. For example, the procedure `NOTIFY_INPUT_0_CONSUMED` clears the `Mailbox.CHARACTER_INPUT_0_READY` flag where as `NOTIFY_OUTPUT_1_READY` sets the `Mailbox.CHARACTER_OUTPUT_1_READY` flag.

Upon close examination of Fig. 11 we can see the usage of the symbol `{}`. These *empty braces* are used to represent flow from a constant value. For example, in the following information flow declaration from Fig. 11:

```
derives ... Mailbox.CHARACTER_INPUT_0_READY from
... {} when (Mailbox.CHARACTER_INPUT_0_READY
            and not Mailbox.CHARACTER_OUTPUT_1_READY)
```

indicates that the variable `Mailbox.CHARACTER_INPUT_0_READY`, in the case when the condition specified holds, has its post-condition value derived from a constant instead of another variable. By examining the code in Fig. 10 we can see that this is the case when `Mailbox.CHARACTER_INPUT_0_READY` is assigned the literal `false`.

The results displayed in Fig. 11 show that the information flow specifications for every variable in this example have been refined with at least one conditional flow. Now, what we really are interested in is determine what are the benefits gained from having such a refined information flow specification. That is, what do we gain from having information flow specifications split into cases denoted by particular conditions? We must keep in mind what the objective of research and engineering effort is: we want to build a foundation for an information assurance specification and verification framework.

From our point of view, an adequate information flow assurance framework must capture and describe the following information about an information-critical system:

```

procedure MACHINE.STEP
—# global in out Mailbox.CHARACTER.INPUT_0_READY,
—# Mailbox.CHARACTER.INPUT_1_READY,
—# Mailbox.CHARACTER.OUTPUT_0_READY,
—# Mailbox.CHARACTER.OUTPUT_1_READY,
—# Mailbox.CHARACTER.OUTPUT_0_DATA,
—# Mailbox.CHARACTER.OUTPUT_1_DATA;
—# in Mailbox.CHARACTER.INPUT_0_DATA,
—# Mailbox.CHARACTER.INPUT_1_DATA;
—# derives Mailbox.CHARACTER.OUTPUT_0_DATA from Mailbox.CHARACTER.INPUT_1_DATA,
—# Mailbox.CHARACTER.OUTPUT_0_READY,
—# Mailbox.CHARACTER.OUTPUT_0_DATA,
—# Mailbox.CHARACTER.INPUT_1_READY &
—# Mailbox.CHARACTER.OUTPUT_1_DATA from Mailbox.CHARACTER.INPUT_0_DATA,
—# Mailbox.CHARACTER.INPUT_0_READY,
—# Mailbox.CHARACTER.OUTPUT_1_DATA,
—# Mailbox.CHARACTER.OUTPUT_1_READY &
—# Mailbox.CHARACTER.INPUT_0_READY from Mailbox.CHARACTER.INPUT_0_READY,
—# Mailbox.CHARACTER.OUTPUT_1_READY &
—# Mailbox.CHARACTER.INPUT_1_READY from Mailbox.CHARACTER.INPUT_0_READY,
—# Mailbox.CHARACTER.OUTPUT_0_READY &
—# Mailbox.CHARACTER.OUTPUT_0_READY from Mailbox.CHARACTER.OUTPUT_0_READY,
—# Mailbox.CHARACTER.INPUT_1_READY &
—# Mailbox.CHARACTER.OUTPUT_1_READY from Mailbox.CHARACTER.OUTPUT_0_READY,
—# Mailbox.CHARACTER.INPUT_0_READY;
is
DATA_0 : CHARACTER;
DATA_1 : CHARACTER;
begin
if Mailbox.INPUT_0_READY and Mailbox.OUTPUT_1_CONSUMED then
DATA_0 := Mailbox.READ_INPUT_0;
Mailbox.NOTIFY_INPUT_0_CONSUMED;
Mailbox.WRITE_OUTPUT_1(DATA_0);
Mailbox.NOTIFY_OUTPUT_1_READY;
end if;

if Mailbox.INPUT_1_READY and Mailbox.OUTPUT_0_CONSUMED then
DATA_1 := Mailbox.READ_INPUT_1;
Mailbox.NOTIFY_INPUT_1_CONSUMED;
Mailbox.WRITE_OUTPUT_0(DATA_1);
Mailbox.NOTIFY_OUTPUT_0_READY;
end if;
end MACHINE.STEP;

```

Fig. 10. Original SPARK specification for Mailbox example.

```

procedure MACHINE.STEP;
—# derives Mailbox.CHARACTER.OUTPUT_0_DATA
—# from Mailbox.CHARACTER.INPUT_1_DATA
—# when (Mailbox.CHARACTER.INPUT_1_READY and not Mailbox.CHARACTER.OUTPUT_0_READY),
—# Mailbox.CHARACTER.OUTPUT_0_READY,
—# Mailbox.CHARACTER.OUTPUT_0_DATA
—# when (not (Mailbox.CHARACTER.INPUT_1_READY and not Mailbox.CHARACTER.OUTPUT_0_READY)),
—# Mailbox.CHARACTER.INPUT_1_READY &
—# Mailbox.CHARACTER.OUTPUT_1_DATA
—# from Mailbox.CHARACTER.INPUT_0_DATA
—# when (Mailbox.CHARACTER.INPUT_0_READY and not Mailbox.CHARACTER.OUTPUT_1_READY),
—# Mailbox.CHARACTER.INPUT_0_READY,
—# Mailbox.CHARACTER.OUTPUT_1_DATA
—# when (not (Mailbox.CHARACTER.INPUT_0_READY and not Mailbox.CHARACTER.OUTPUT_1_READY)),
—# Mailbox.CHARACTER.OUTPUT_1_READY &
—# Mailbox.CHARACTER.INPUT_0_READY
—# from Mailbox.CHARACTER.INPUT_0_READY,
—# {} when (Mailbox.CHARACTER.INPUT_0_READY and not Mailbox.CHARACTER.OUTPUT_1_READY),
—# Mailbox.CHARACTER.OUTPUT_1_READY &
—# Mailbox.CHARACTER.INPUT_1_READY
—# from Mailbox.CHARACTER.INPUT_1_READY,
—# {} when (Mailbox.CHARACTER.INPUT_1_READY and not Mailbox.CHARACTER.OUTPUT_0_READY),
—# Mailbox.CHARACTER.OUTPUT_0_READY &
—# Mailbox.CHARACTER.OUTPUT_0_READY
—# from Mailbox.CHARACTER.OUTPUT_0_READY,
—# {} when (Mailbox.CHARACTER.OUTPUT_0_READY and not Mailbox.CHARACTER.INPUT_1_READY),
—# Mailbox.CHARACTER.INPUT_1_READY &
—# Mailbox.CHARACTER.OUTPUT_1_READY
—# from Mailbox.CHARACTER.OUTPUT_1_READY,
—# {} when (Mailbox.CHARACTER.OUTPUT_1_READY and not Mailbox.CHARACTER.INPUT_0_READY),
—# Mailbox.CHARACTER.INPUT_0_READY;

```

Fig. 11. Results of running tool on Mailbox example.

- **Admissible Channels of Information Flow:** the framework must provide mechanisms to appropriately specify when a flow of information from one part of the system to another (or from one variable to another) is acceptable. The original `derives` annotations from SPARK, and its corresponding checking mechanism,

can already be used for this purpose (although they were not originally intended to fulfill this functionality).

- **Enabling Conditions for Information Flow Channels:** the framework must provide mechanisms to specify under what conditions a particular information flow channel is active. In information flow assurance applications, information flow channels are often controlled by system conditions. However, as it is, SPARK does not possess any mechanism that allows specifying under what conditions a particular information flow channel is active.

In the case of the mailbox example, we have a device intended to serve as a communication channel between two entities. If we were to try to describe the information flow policy requirements for the mailbox, we could write something like:

The mailbox will guarantee that information produced at the by Client 0 will be forwarded to Client 1, and the information produced by Client 1 will be forwarded to Client 0.

However, when we look at the information flow specification for the Client 0's output on Fig. 10, we have:

```
derives Mailbox.CHARACTER_OUTPUT_0.DATA from Mailbox.CHARACTER_INPUT_1.DATA,
                                             Mailbox.CHARACTER_OUTPUT_0.READY,
                                             Mailbox.CHARACTER_OUTPUT_0.DATA,
                                             Mailbox.CHARACTER_INPUT_1.READY
```

The output of Client 0 is not derived only from Client 1's input, but from other 3 variables. It is not necessarily obvious where these other dependences are coming from, and they certainly do not match with our first attempt at describing the mailbox's behavior. As it turns out, what happens here is that this specification describes *more than one* information flow channel, and the conditions on which they are active, but all this information has been merged into a single annotation. Let us look at the equivalent annotation from Fig. 11 to see what is going on:

```
derives Mailbox.CHARACTER_OUTPUT_0.DATA
  from Mailbox.CHARACTER_INPUT_1.DATA
      when (Mailbox.CHARACTER_INPUT_1.READY
            and not Mailbox.CHARACTER_OUTPUT_0.READY),
  Mailbox.CHARACTER_OUTPUT_0.READY,
  Mailbox.CHARACTER_OUTPUT_0.DATA
      when (not (Mailbox.CHARACTER_INPUT_1.READY
            and not Mailbox.CHARACTER_OUTPUT_0.READY)),
  Mailbox.CHARACTER_INPUT_1.READY
```

In the original SPARK specification, we cannot tell whether there are several information channels, or that the target variable is derived from a combination of the source variables, because there are no conditions. However, by looking at the specification produced by our tool, we can see that there are actually **2 information flow channels** acting on this variable, controlled by two different conditions. We can also see that the dependence on the extra 2 variables is produced from *control dependence* on the variables that are used to compute the conditions.

It is clear now that there are 2 information flow channels working on this variable: (1) when information is available from Client 1 and Client 0 is ready to receive this information, then the output read by Client 0 is derived from the input produced by Client 1, and (2) if either there is not input from Client 1 or Client 0 is not ready to receive, then the output read by Client 0 keeps its old value. And clearly, which of these two channels is active depends on the aforementioned conditions, which in turn produced a control dependence on the variables that keep track of whether Client 1 has produced any information, and whether Client 0 is ready to receive.

After the previous discussion, the benefits of having conditional information flow specifications are immediately clear. We have a more precise description of the behavior of the system, and are able to check both aspects of the information assurance behavior of a system that we described before: the channels of information flow and the conditions under which those channels are active.

Another improvement that could be done is differentiate the parts of the specification that deal with the control logic from those that deal exclusively with information flow. For example, in the case of the mailbox annotation for output 0, we get dependences on a couple of extra variables that arise from control dependence. Perhaps one could mark these flows with a special annotation to explicitly state that they arise from the control logic. Similarly, we can see in Fig. 10 and Fig. 11 that, besides those for the output variables, we have flow annotations for each of the control variables. These annotations are needed because these variables may be reset by the procedure. However, these modifications of control variables is also part of the control logic, and maybe these flows could also be annotated in a special way. Furthermore, one could imagine a tool that would use these annotations to filter views and show all annotations, or hide flows corresponding to the control logic, etc.

Autopilot Example The Autopilot system is one of the examples included in the SPARK distribution (discussed in detail in [7, Chapter 14]). This is a control system controlling both the altitude and heading of an aircraft. The altitude is controlled by manipulating the elevators and the heading is controlled by manipulating the ailerons and rudder. The autopilot has a control panel with three switches each of which has two positions – on and off.

- The master switch – the autopilot is completely inactive if this is off.
- The altitude switch – the autopilot controls the altitude if this is on.
- The heading switch – the autopilot controls the heading if this is on.

Desired autopilot heading values are entered in a console by the pilot, whereas desired altitude values are determined by the current altitude (similar to how an automobile cruise control takes its target speed from the current speed when the cruise control is activated). For this example we will take a look at a total of 4 procedures and 2 functions.

The procedure in Figure 16 is interesting for conditional information flow analysis for multiple reasons:

- it contains nested case statements with a call at the lowest level of nesting to procedure `Pitch.Pitch_AP` that updates global variables,
- the actual updates to global variables occur several levels down the call chain from `Pitch.Pitch_AP`, and
- the call chain includes several procedures with conditional flows – some of the conditions propagate up through the call chain whereas others do not.

We discuss in detail the conditional information flow along the following call path.

- `Main(main.adb)` – contains an infinite loop that does nothing but call `AP.Control` on each iteration
- `AP.Control(ap.adb)` – reads values for the three switches above from the environment. If `Master_Switch` is on, then it uses the values read for `Altitude_Switch` and `Heading_Switch` and to set switch variables `Altitude_Selected` and `Heading_Selected`,

otherwise `Altitude_Selected` and `Heading_Selected` are set to “off”. Instruments needed to calculate altitude and heading are read, then `Altitude.Maintain` (with `Altitude_Selected` as the actual parameter for `Switch_Pressed`) and `Heading.Maintain` are called to update the autopilot state.

- `AP.Altitude.Maintain` (`ap-altitude.adb`) – If `Altitude_Switch` has transitioned from off to on, the `Present_Altitude` is used as value for `Target_Altitude`. Otherwise, the previous value of `Target_Altitude` is used for value of `Target_Altitude`. `Pitch.Pitch_AP` is called to calculate the value of `Surfaces.Elevators` based on the parameter values of `Pitch.Pitch_AP` and the pitch history.
- `Pitch.Pitch_AP` (`ap-altitude-pitch.ads`) – calls a series of helper functions which update the local variables `Present_Pitchrate`, `Target_Pitchrate`, and `Elevator_Movement` and these are used in `Surfaces.Move.Elevators` to calculate the value of the global output variable `Surfaces.Elevators`. The behavior of `Surfaces.Move.Elevators` lies outside the SPARK boundary and thus the interface to `Surfaces.Move.Elevators` represents the leaf of the call tree path under consideration.
- We will also take a look at a couple of functions called from `Pitch.Pitch_AP`: `Altitude.Target_Rate` and `Altitude.Target_ROC`. We do this to discuss some interesting aspects of computing information flow specifications for SPARK functions.

The first procedure we look at is the main procedure. This is, like in most languages, the top most procedure and the point of access for the whole system. Figure 12 shows the original SPARK specifications and the code, and Fig. 13 shows the corresponding information flow specifications computed by our tool. The first thing we notice is that there are more *derived* variables in the annotations derived by our tool than in the original annotations. This is not a mistake. The reason for this is that we still have not incorporated SPARK’s abstraction mechanism in our tool. All the variables in the `derives` annotations from Fig. 13 that start with `AP.` are abstracted into the variable `AP.State` in Fig. 12. As a consequence, we get more flow specifications because they are refined from those in the original annotations.

An interesting effect of not having abstraction in our annotations is that some of the false flows introduced by the abstraction process are not present in our annotations. For instance, in Fig. 12 one of the annotations suggests that `Surfaces.Ailerons` may be derived from `Instruments.Altitude`. However, as we can see in Fig. 13 this is not the case, such flow is absent from the specification. The reason this false flow appears in the abstracted version is that, when all the `AP.` variables are abstracted into `AP.State`, then now `Surfaces.Ailerons` depends on this new abstract variable (because it depends on some `AP.` variables), and since some `AP.` variables depend on `Instruments.Altitude`, then `Surfaces.Ailerons` gets a possible dependence on `Instruments.Altitude`, even though this dependence is really non-existent.

Other than the differences described above, the specifications obtained by our tool are basically the same as those in the original annotations. Also, we see `{}` annotations (derivations from constant values). In the original SPARK, these *constant derivations* are simply ignored, however we leave them explicit for the sake of completeness. In fact, these annotations become more interesting when they are associated by themselves with a condition, as they might actually represent “reset” conditions (as in the case of the mailbox example).

```

procedure Main
  —# global in out AP.State;
  —# out Surfaces.Elevators,
  —# Surfaces.Ailerons,
  —# Surfaces.Rudder;
  —# in Instruments.Altitude,
  —# Instruments.Bank,
  —# Instruments.Heading,
  —# Instruments.Heading_Bug,
  —# Instruments.Mach,
  —# Instruments.Pitch,
  —# Instruments.Rate_Of_Climb,
  —# Instruments.Slip;
  —# derives AP.State
  —# from *
  —# Instruments.Altitude,
  —# Instruments.Bank,
  —# Instruments.Pitch,
  —# Instruments.Slip &
  —# Surfaces.Elevators
  —# from
  —# AP.State,
  —# Instruments.Altitude,
  —# Instruments.Bank,
  —# Instruments.Mach,
  —# Instruments.Pitch,
  —# Instruments.Rate_Of_Climb,
  —# Instruments.Slip
  —# &
  —# Surfaces.Ailerons
  —# from
  —# AP.State,
  —# Instruments.Altitude,
  —# Instruments.Bank,
  —# Instruments.Heading,
  —# Instruments.Heading_Bug,
  —# Instruments.Mach,
  —# Instruments.Pitch,
  —# Instruments.Slip &
  —# Surfaces.Rudder
  —# from AP.State,
  —# Instruments.Altitude,
  —# Instruments.Bank,
  —# Instruments.Mach,
  —# Instruments.Pitch,
  —# Instruments.Slip
  —# ;
is
begin
  loop
    AP.Control;
  end loop;
end Main;

```

Fig. 12. Original SPARK specification for procedure main from the autopilot code base.

Next we look at the procedure `AP.Control`. The original SPARK annotations, as well as the code for the procedure are given in Fig. 14, and the result from our tool is presented in Fig. 15. Just as with main procedure explained before, we get basically the same annotations, except for getting extra variables due to differences in abstraction, and the presence of `{}` annotations in our tool's results. However, it is rather disappointing that we don't get conditional information flow specifications in this example. On close look at the code, we see there is `case` statement in the body that should give rise to conditions. Furthermore, as we will see later, most of the procedures called from this procedure generate conditional specifications. So, why don't we get any conditional specifications here?

The bottom line is that we are getting hurt by the generalization rules triggered by the procedure call rule discussed in Sec. 5. The procedure does generate conditions, however, all this conditions are dropped once the top 3 procedure calls are analyzed: the procedures that read the value of the switches (the guard variables). What happens is that the conditions generated are basically predicates in terms of the guard variables (the value of the switches) and since the top 3 procedures set these switch variables, we have to drop the conditions, and turn the annotation into an un-conditional one.

```

procedure Main;
-- derives AP.Heading.Yaw.Rate.Yaw.History
--   from *,
--   {}.
--   AP.Controls.Master.Switch,
--   AP.Controls.Heading.Switch,
--   Instruments.Slip &
-- AP.Altitude.Pitch.Rate.Pitch.History
--   from *,
--   {}.
--   AP.Controls.Master.Switch,
--   AP.Controls.Altitude.Switch,
--   Instruments.Pitch &
-- AP.Heading.Roll.Rate.Roll.History
--   from *,
--   {}.
--   AP.Controls.Master.Switch,
--   AP.Controls.Heading.Switch,
--   Instruments.Bank &
-- AP.Altitude.Target.Altitude
--   from *,
--   {}.
--   AP.Controls.Master.Switch,
--   AP.Controls.Altitude.Switch,
--   AP.Altitude.Switch.Pressed.Before,
--   Instruments.Altitude &
-- AP.Altitude.Switch.Pressed.Before
--   from *,
--   {}.
--   AP.Controls.Master.Switch,
--   AP.Controls.Altitude.Switch &
-- Surfaces.Elevators
--   from {}.
--   AP.Controls.Master.Switch,
--   AP.Controls.Altitude.Switch
--   AP.Altitude.Pitch.Rate.Pitch.History,
--   AP.Altitude.Switch.Pressed.Before,
--   AP.Altitude.Target.Altitude,
--   Instruments.Altitude,
--   Instruments.Mach,
--   Instruments.Pitch,
--   Instruments.Rate.Of.Climb &
-- Surfaces.Ailerons
--   from {}.
--   AP.Controls.Master.Switch,
--   AP.Controls.Heading.Switch,
--   AP.Heading.Roll.Rate.Roll.History,
--   Instruments.Bank,
--   Instruments.Heading,
--   Instruments.Heading.Bug,
--   Instruments.Mach &
-- Surfaces.Rudder
--   from {}.
--   AP.Controls.Master.Switch,
--   AP.Controls.Heading.Switch,
--   AP.Heading.Yaw.Rate.Yaw.History,
--   Instruments.Mach,
--   Instruments.Slip
-- ;

```

Fig. 13. Results of running tool on procedure main from the autopilot code base.

To see this with more detail, let us take a look at what happens at the return point of the procedure call to `Controls.ReadHeading.Switch`. Recall that our algorithm is a weakest pre-condition algorithm, and as such, it works bottom-up. So, when we reach the point right before the call to this procedure, the algorithm has, among all of the derivations generated, the following flow specification:

```

derives Surfaces.Ailerons from Instruments.Heading
  when Heading_Switch = Controls.On and Master_Switch = Controls.On

```

This specification basically tells us that if the heading and master switches are in the ON position, then `Surfaces.Ailerons` derives its value from the current heading reading (`Instruments.Heading`). Furthermore, another similar specification tells us that if the heading switch is OFF, then `Surfaces.Ailerons` maintains its current value. However, when we process the call to `Controls.ReadHeading.Switch`, all this information is lost, because this procedure sets the value of `Heading_Switch` with the actual state of the switch. In doing this, since our analysis is modular (*i.e.*, we do not look at the body of `Controls.ReadHeading.Switch`), we do not know

```

procedure Control
  —# global in    Controls.Master_Switch ,
  —#            Controls.Altitude_Switch ,
  —#            Controls.Heading_Switch ;
  —# in out      Altitude.State ,
  —#            Heading.State ;
  —# out         Surfaces.Elevators ,
  —#            Surfaces.Ailerons ,
  —#            Surfaces.Rudder ;
  —# in          Instruments.Altitude ,
  —#            Instruments.Bank ,
  —#            Instruments.Heading ,
  —#            Instruments.Heading_Bug ,
  —#            Instruments.Mach ,
  —#            Instruments.Pitch ,
  —#            Instruments.Rate_Of_Climb ,
  —#            Instruments.Slip ;
  —# derives    Altitude.State
  —#            from *,
  —#            Controls.Master_Switch ,
  —#            Controls.Altitude_Switch ,
  —#            Instruments.Altitude ,
  —#            Instruments.Pitch &
  —#            Heading.State
  —#            from *,
  —#            Controls.Master_Switch ,
  —#            Controls.Heading_Switch ,
  —#            Instruments.Bank ,
  —#            Instruments.Slip &
  —#            Surfaces.Elevators
  —#            from Controls.Master_Switch ,
  —#            Controls.Altitude_Switch ,
  —#            Altitude.State ,
  —#            Instruments.Altitude ,
  —#            Instruments.Mach ,
  —#            Instruments.Pitch ,
  —#            Instruments.Rate_Of_Climb &
  —#            Surfaces.Ailerons
  —#            from Controls.Master_Switch ,
  —#            Controls.Heading_Switch ,
  —#            Heading.State ,
  —#            Instruments.Bank ,
  —#            Instruments.Heading ,
  —#            Instruments.Heading_Bug ,
  —#            Instruments.Mach &
  —#            Surfaces.Rudder
  —#            from Controls.Master_Switch ,
  —#            Controls.Heading_Switch ,
  —#            Heading.State ,
  —#            Instruments.Mach ,
  —#            Instruments.Slip
  —# ;

is
  Master_Switch , Altitude_Switch , Heading_Switch ,
  Altitude_Selected , Heading_Selected : Controls.Switch ;
  Present_Altitude : Instruments.Feet ;
  Bank : Instruments.Bankangle ;
  Present_Heading : Instruments.Headdegree ;
  Target_Heading : Instruments.Headdegree ;
  Mach : Instruments.Machnumber ;
  Pitch : Instruments.Pitchangle ;
  Rate_Of_Climb : Instruments.Feetpermin ;
  Slip : Instruments.Slipangle ;

begin
  Controls.Read_Master_Switch(Master_Switch);
  Controls.Read_Altitude_Switch(Altitude_Switch);
  Controls.Read_Heading_Switch(Heading_Switch);
  case Master_Switch is
    when Controls.On =>
      Altitude_Selected := Altitude_Switch;
      Heading_Selected := Heading_Switch;
    when Controls.Off =>
      Altitude_Selected := Controls.Off;
      Heading_Selected := Controls.Off;
    end case ;
  Instruments.Read_Altimeter(Present_Altitude);
  Instruments.Read_Bank_Indicator(Bank);
  Instruments.Read_Compass(Present_Heading);
  Instruments.Read_Heading_Bug(Target_Heading);
  Instruments.Read_Mach_Indicator(Mach);
  Instruments.Read_Pitch_Indicator(Pitch);
  Instruments.Read_VSI(Rate_Of_Climb);
  Instruments.Read_Slip_Indicator(Slip);
  Altitude.Maintain(Altitude_Selected , Present_Altitude , Mach , Rate_Of_Climb , Pitch);
  Heading.Maintain(Heading_Selected , Mach , Present_Heading , Target_Heading , Bank , Slip);
end Control;

```

Fig. 14. Original specification for procedure AP.Control from the autopilot code base.

in general what this modification to Heading_Switch did, and so we have to drop the condition. At the very least we would have to drop the part of the condition that

```

procedure Control ;
--# derives Altitude.Switch.Pressed.Before
--#   from *,
--#       {},
--#       Controls.Master_Switch ,
--#       Controls.Altitude_Switch &
--# Altitude.Pitch.Rate.Pitch_History
--#   from *,
--#       {},
--#       Controls.Master_Switch ,
--#       Controls.Altitude_Switch ,
--#       Instruments.Pitch &
--# Altitude.Target_Altitude
--#   from *,
--#       {},
--#       Controls.Master_Switch ,
--#       Controls.Altitude_Switch ,
--#       Altitude.Switch_Pressed.Before ,
--#       Instruments.Altitude &
--# Heading.Yaw.Rate.Yaw_History
--#   from *,
--#       {},
--#       Controls.Master_Switch ,
--#       Controls.Heading_Switch ,
--#       Instruments.Slip &
--# Heading.Roll.Rate.Roll_History
--#   from *,
--#       {},
--#       Controls.Master_Switch ,
--#       Controls.Heading_Switch ,
--#       Instruments.Bank &
--# Surfaces.Elevators
--#   from {},
--#       Controls.Master_Switch ,
--#       Controls.Altitude_Switch ,
--#       Altitude.Target_Altitude ,
--#       Altitude.Switch_Pressed.Before ,
--#       Altitude.Pitch.Rate.Pitch_History ,
--#       Instruments.Altitude ,
--#       Instruments.Mach ,
--#       Instruments.Pitch ,
--#       Instruments.Rate_Of_Climb &
--# Surfaces.Ailerons
--#   from {},
--#       Controls.Master_Switch ,
--#       Controls.Heading_Switch ,
--#       Heading.Roll.Rate.Roll_History ,
--#       Instruments.Bank ,
--#       Instruments.Heading ,
--#       Instruments.Heading_Bug ,
--#       Instruments.Mach &
--# Surfaces.Rudder
--#   from {},
--#       Controls.Master_Switch ,
--#       Controls.Heading_Switch ,
--#       Heading.Yaw.Rate.Yaw_History ,
--#       Instruments.Mach ,
--#       Instruments.Slip
--# ;

```

Fig. 15. Results of running tool on procedure AP.Control from the autopilot code base.

refers to `Heading_Switch`, however in this case it does not matter, because as soon as `Controls.Read_Master_Switch` was analyzed, the rest of the condition would be dropped.

As it turns out, in this particular case, it would be safe not to drop the condition and simply substitute `Heading_Switch` with `Controls.Heading_Switch`, but this cannot be determined without looking at the procedure's body and breaking modularity. One way this situation could ameliorated would be to refactor the procedure into two different procedures: one that reads in the value of the switches, and another that implements the rest of the logic. For example,

```

procedure Control
is
  ...
begin
  Read_Controls_Switches (Master_Switch , Altitude_Switch , Heading_Switch);
  Execute_Control_Logic (Master_Switch , Altitude_Switch , Heading_Switch);
end Control ;

```

```

procedure Maintain( Switch.Pressed : in Controls.Switch;
                  Present.Altitude : in Instruments.Feet;
                  Mach : in Instruments.Machnumber;
                  Climb.Rate : in Instruments.Feetpermin;
                  The.Pitch : in Instruments.Pitchangle)
—# global in out Target.Altitude ,
—# Switch.Pressed.Before ,
—# Pitch.State ;
—# out Surfaces.Elevators ;
—# derives Target.Altitude
—# from *,
—# Switch.Pressed ,
—# Switch.Pressed.Before ,
—# Present.Altitude &
—# Pitch.State
—# from *,
—# Switch.Pressed ,
—# The.Pitch &
—# Switch.Pressed.Before
—# from
—# Switch.Pressed &
—# Surfaces.Elevators
—# from Switch.Pressed ,
—# Switch.Pressed.Before ,
—# Target.Altitude ,
—# Present.Altitude ,
—# Mach ,
—# Climb.Rate ,
—# The.Pitch ,
—# Pitch.State
—# ;
is
begin
  case Switch.Pressed is
    when Controls.On =>
      case Switch.Pressed.Before is
        when Controls.Off =>
          Target.Altitude := Present.Altitude ;
        when Controls.On =>
          null ;
      end case ;
      Pitch.Pitch.AP(Present.Altitude ,Target.Altitude ,Mach,Climb.Rate ,The.Pitch);
    when Controls.Off =>
      null ;
  end case ;
  Switch.Pressed.Before := Switch.Pressed ;
end Maintain ;

```

Fig. 16. Original SPARK specification for procedure Altitude.Maintain from the autopilot code base.

where `Read.Controls.Switches` simply performs the top three procedure calls in `Control`, and the rest of the functionality is implemented in `Execute.Control.Logic`. Then the conditional information flow specifications of `Control` would be exposed in the procedure `Execute.Control.Logic`.

Another option would be to exploit other annotations in the code (like post-conditions and/or assertions) to avoid un-necessary generalizations. For example, if the procedure `Controls.Read.Heading.Switch` had the following annotation:

```

procedure Read_Heading_Switch (Heading_Switch)
—# post: Heading_Switch = Controls.Heading_Switch ;

```

then from this annotation we could determine exactly what the value of `Heading_Switch` is in the post-condition (`Controls.Heading_Switch`), and perform a direct substitution in the condition expressions instead of having to drop them. These are all options that we are considering for future versions of the tool.

Next procedure to discuss is `Altitude.Maintain`, which is called from `AP.Control`. This is the first procedure in our study of the Autopilot that has generates conditional specifications. The original SPARK annotations as well as the code are displayed in Fig. 16, and the annotations generated by our tool are presented in Fig. 17. The purpose of this procedure is to maintain the altitude of the airplane depending on the current configuration of the autopilot, so there are quite a few cases this procedure has to handle, which is why we get several conditional information flow specifications.

```

procedure Maintain(Switch.Pressed : in Controls.Switch;
                   Present.Altitude : in Instruments.Feet;
                   Mach : in Instruments.Machnumber;
                   Climb.Rate : in Instruments.Feetpermin;
                   The.Pitch : in Instruments.Pitchangle);

-- derives Target.Altitude
-- from * when (not Switch.Pressed = Controls.On),
-- Switch.Pressed,
-- Switch.Pressed.Before when (Switch.Pressed = Controls.On),
-- Present.Altitude
--   when (Switch.Pressed = Controls.On and Switch.Pressed.Before = Controls.Off),
-- * when (Switch.Pressed = Controls.On and (not Switch.Pressed.Before = Controls.Off)) &
-- Pitch.Rate.Pitch.History
-- from {*, The.Pitch} when (Switch.Pressed = Controls.On),
-- Switch.Pressed,
-- * when (not Switch.Pressed = Controls.On) &
-- Switch.Pressed.Before
-- from
--   Switch.Pressed &
-- Surfaces.Elevators
-- from Switch.Pressed,
-- {Switch.Pressed.Before,
-- Present.Altitude,
-- Mach,
-- Climb.Rate,
-- The.Pitch,
-- Pitch.Rate.Pitch.History} when (Switch.Pressed = Controls.On),
-- Target.Altitude
--   when (Switch.Pressed = Controls.On and (not Switch.Pressed.Before = Controls.Off)),
-- Present.Altitude
--   when (Switch.Pressed = Controls.On and Switch.Pressed.Before = Controls.Off),
-- Surfaces.Elevators when (not Switch.Pressed = Controls.On)
-- ;

```

Fig. 17. Results of running tool on procedure Altitude.Maintain from the autopilot code base.

Here again we have a case in which it is extremely beneficial to have conditional information flow contracts. The information flow contracts in this example not only describe the information flow channel presents, but also the conditions under which they are active, revealing details of the control logic of the system. Not only that, in this particular example, the details of the control logic revealed actually give a good insight of the actual functionality of the procedure.

Let us start by looking at the flow specifications for Target.Altitude. We can see that Target.Altitude derives either derives its new value from Present.Altitude, or it keeps from its own previous value. We can see that the dependences on Switch.Pressed and Switch.Pressed.Before are simply control dependences, as these are the variables that appear in the conditions. Switch.Pressed contains the state of the altitude switch, passed as an argument, and Switch.Pressed.Before is a global used to store the state of Switch.Pressed in the previous state.

So, under what conditions is the value of Target.Altitude modified using Present.Altitude? Looking at the first specification, we can see that whenever Switch.Pressed is OFF, Target.Altitude's new value is derived from itself. Now we have two interesting cases. If Switch.Pressed is ON, but Switch.Pressed.Before is OFF (*i.e.*, the value of Switch.Pressed in the previous state was OFF) then Target.Altitude derives gets its new value derived from Present.Altitude. On the other hand, if Switch.Pressed is ON and Switch.Pressed.Before is ON, then Target.Altitude gets its new value derived from itself. So basically here we have exposed the logic of the altitude control system: when the altitude switch transitions from OFF to ON, the target altitude (the altitude at which the plane will be automatically maintained) is set to the present altitude, after that initial transition the target altitude does not change (transition from ON to ON), unless the system transitions again from OFF to ON. So Switch.Pressed.Before is basically used to detect the transitions from OFF to ON and set the target altitude. A similar analysis applied to the information flow specifications for Surfaces.Elevators.

```

procedure Pitch_AP( Present.Altitude : in Instruments . Feet ;
                    Target.Altitude : in Instruments . Feet ;
                    Mach              : in Instruments . Machnumber ;
                    Climb.Rate       : in Instruments . Feetpermin ;
                    The.Pitch        : in Instruments . Pitchangle )
-- global in out Rate . Pitch_History ;
-- out Surfaces . Elevators ;
-- derives Rate . Pitch_History
-- from * ,
-- The.Pitch &
-- Surfaces . Elevators
-- from Rate . Pitch_History ,
-- Present.Altitude ,
-- Target.Altitude ,
-- Mach ,
-- Climb.Rate ,
-- The.Pitch
-- ;
is
    Present.Pitchrate : Degreespersec ;
    Target.Pitchrate  : Degreespersec ;
    Elevator.Movement : Surfaces . Controlangle ;
begin
    Calc.Pitchrate( The.Pitch , Present.Pitchrate ) ;
    Target.Pitchrate := Target.Rate( Present.Altitude , Target.Altitude , Climb.Rate ) ;
    Elevator.Movement := Calc.Elevator_Move( Present.Pitchrate , Target.Pitchrate , Mach ) ;
    Surfaces . Move_Elevators( Elevator.Movement ) ;
end Pitch_AP ;

```

Fig. 18. Original SPARK specification for procedure AP.Altitude.Pitch_AP from the autopilot code base.

```

procedure Pitch_AP( Present.Altitude : in Instruments . Feet ;
                    Target.Altitude : in Instruments . Feet ;
                    Mach              : in Instruments . Machnumber ;
                    Climb.Rate       : in Instruments . Feetpermin ;
                    The.Pitch        : in Instruments . Pitchangle )
-- derives Rate . Pitch_History
-- from * ,
-- The.Pitch &
-- Surfaces . Elevators
-- from Rate . Pitch_History ,
-- Present.Altitude ,
-- Target.Altitude ,
-- Mach ,
-- Climb.Rate ,
-- The.Pitch
-- ;
is
    Present.Pitchrate : Degreespersec ;
    Target.Pitchrate  : Degreespersec ;
    Elevator.Movement : Surfaces . Controlangle ;
begin
    Calc.Pitchrate( The.Pitch , Present.Pitchrate ) ;
    Target.Pitchrate := Target.Rate( Present.Altitude , Target.Altitude , Climb.Rate ) ;
    Elevator.Movement := Calc.Elevator_Move( Present.Pitchrate , Target.Pitchrate , Mach ) ;
    Surfaces . Move_Elevators( Elevator.Movement ) ;
end Pitch_AP ;

```

Fig. 19. Results of running tool on procedure AP.Altitude.Pitch_AP from the autopilot code base.

Now let us examine procedure `Altitude.Pitch.Pitch_AP` which is called from `Altitude.Maintain`. The original SPARK annotations as well as the code can be seen in Fig. 18, and the results of our tool are presented in Fig. 19. This example is actually relatively simple, and as seen by looking at the figures the results of our tool are exactly the same as the original SPARK annotations. As `Pitch_AP`'s purpose is just to update a set of variables, depending on its input, there is really no conditional information flow behavior in this procedure. This is the procedure sets the pitch, depending on the values of the present and target altitude. What is interesting about this procedure is that it calls a SPARK function, which is the one we look at next.

To conclude we look at a couple of SPARK functions, which are the bottom of this call chain in Autopilot. The reason we look at this functions is to discuss a couple of relevant concepts in the computation of the annotations relevant to functions, which are not issues in the original SPARK. The functions are `Target_Rate`, which is called

```

function Target_Rate(Present.Altitude : Instruments.Feet;
                    Target.Altitude : Instruments.Feet;
                    Climb.Rate : Instruments.Feetpermin)
return Degreespersec
—# derives @result
—# from {},
—# Climb.Rate ,
—# Present.Altitude ,
—# Target.Altitude
—# ;
is
Target.Climb.Rate : Floorfpm;
Floor.Climb.Rate : Floorfpm;
Result : Degreespersec;
begin
Target.Climb.Rate := Target.ROC(Present.Altitude , Target.Altitude );
if Climb.Rate > Floorfpm'Last then
Floor.Climb.Rate := Floorfpm'Last;
elsif Climb.Rate < Floorfpm'First then
Floor.Climb.Rate := Floorfpm'First;
else
Floor.Climb.Rate := Climb.Rate;
end if;
—# assert Floor.Climb.Rate in Floorfpm and
—# Target.Climb.Rate in Floorfpm;
Result := Degreespersec( (Target.Climb.Rate - Floor.Climb.Rate) / 12);
if (Result > 10) then
Result := 10;
elsif (Result < -10) then
Result := -10;
end if;
return Result;
end Target_Rate;

```

Fig. 20. Results of running tool on function AP.Altitude.Target_Rate from the autopilot code base.

```

function Target_ROC(Present.Altitude : Instruments.Feet;
                   Target.Altitude : Instruments.Feet)
return Floorfpm
—# derives @result
—# from {}
—# when ((Target.Altitude - Present.Altitude) / 10 < Floorfpm'First
—# and not ((Target.Altitude - Present.Altitude) / 10 > Floorfpm'Last)),
—# {}
—# when ((Target.Altitude - Present.Altitude) / 10 > Floorfpm'Last),
—# Target.Altitude ,
—# Present.Altitude
—# ;
is
Result : Instruments.Feetpermin;
begin
Result := Instruments.Feetpermin( Integer(Target.Altitude - Present.Altitude) / 10);
if (Result > Floorfpm'Last) then
Result := Floorfpm'Last;
elsif Result < Floorfpm'First then
Result := Floorfpm'First;
end if;
return Result;
end Target_ROC;

```

Fig. 21. Results of running tool on function AP.Altitude.Target_ROC from the autopilot code base.

from Pitch_AP, and Target_ROC, which is called from Target_Rate. The code for these functions, and the annotations obtained with our tool are presented in Fig. 20 and Fig. 21, respectively.

The main thing to observe in these functions is the annotations. In SPARK, functions do not have `derives` annotations. In fact, information flow is not computed for functions. The information flow for a function in SPARK is straightforward: whatever value is computed by the function depends on its inputs. However, in our case, we compute conditional information flow contracts. Some functions might actually give rise to conditions, *e.g.*, different values might be computed depending on conditions calculated from the inputs. So, we need to be able to compute conditional information flows on functions. Moreover, because we insist in modular analysis, we need to be able to attach conditional `derives` annotations to functions, as shown in Fig. 20 and Fig. 21.

In order to be able to specify conditional information flow for functions, we need to be able to talk about the value computed by the function. We define a special variable `@result`, which denotes the value returned by the function, and we compute dependences for this special variable. In the case of `Target_Rate` we do not have conditional information flow specifications, but in the case of `Target_ROC` we do have a couple of conditional flows. However, these two conditional flows are *constant flows*, and as such they disappear in `Target_Rate`. The main point we want to present here is the need for information flow specifications for SPARK functions, and the way we implement those in our adaptation of SPARK.

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 [31, 6] are flow-*insensitive* 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.* [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 properties, notably Jif (Java + information flow) which is based on [22]), and Flow Caml [27].

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.* [9] and later extended by, e.g., Terauchi and Aiken [29] and (for heap-manipulating programs) Naumann [23]. 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 [19], Spec# [8], 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.

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. A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 2(15):131–177, Mar. 2005.
7. J. Barnes. *High Integrity Software – the SPARK Approach to Safety and Security*. Addison-Wesley, 2003.
8. 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.
9. 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.
10. D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, MITRE Corp., 1973.
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. D. Gries. *The Science of programming*. Springer-Verlag, New York, 1981.
18. 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.
19. 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.
20. 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.
21. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
22. A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL’99, San Antonio, Texas*, pages 228–241. ACM Press, 1999.
23. 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.
24. 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.
25. B. Rossebo, P. Oman, J. Alves-Foss, R. Blue, and P. Jaskowiak. Using SPARK-Ada to model and verify a MILS message router. In *Proceedings of the International Symposium on Secure Software Engineering*, 2006.
26. J. Rushby. The design and verification of secure systems. In *8th ACM Symposium on Operating Systems Principles*, volume 15(5), pages 12–21, 1981.

27. V. Simonet. Flow Caml in a nutshell. In G. Hutton, editor, *First APPSEM-II workshop*, pages 152–165, Mar. 2003.
28. 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.
29. 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.
30. 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.
31. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–188, 1996.
32. Sireum website. <http://www.sireum.org>.

A Proofs of Technical Results

Proof of Lemma 1

We first state some auxiliary facts that allow us to remove the nonpure part of array expressions, replacing them by case analyses of the actual index. To formalize the notion “part of”, we introduce evaluation contexts C by the syntax

$$C ::= [] \mid C \text{ op } A \mid A \text{ op } C \mid h[C]$$

We write $C[A] \in \mathbf{AExp}$ for the result of plugging A into context C .

Fact 7 *If A is not pure, then A is of the form $C[H[A_0]]$ with H not a variable.*

Fact 8 *For all C, A, A_2 , $\text{bop}: C[Z[A]] \text{ bop } A_2$ is 1-equivalent to $C[0] \text{ bop } A_2$.*

Fact 9 *For all C, H, A_0, A', A, A_2 , $\text{bop}: C[H\{A_0 : A'\}[A]] \text{ bop } A_2$ is 1-equivalent to $(A = A_0 \rightarrow C[A'] \text{ bop } A_2) \wedge (A \neq A_0 \rightarrow C[H[A]] \text{ bop } A_2)$*

Lemma 1 *Given ϕ , we can construct pure ϕ' such that $\phi \equiv_1 \phi'$.*

Proof: We define a measure Q on $\mathbf{1Assert}$ by stipulating that $Q(\phi) = (j, k_1, k_2)$ if either ϕ is pure and $j = k_1 = k_2 = 0$, or ϕ contains at least one impure arithmetic expression of size j but no impure arithmetic expression of size greater than j , and ϕ contains k_1 boolean expressions $A_1 \text{ bop } A_2$ with A_1 impure of size j , and ϕ contains k_2 boolean expressions $A_1 \text{ bop } A_2$ with A_2 impure of size j . We now impose an order on $\mathbf{1Assert}$ by stipulating that $\phi \prec \phi'$ iff, with $Q(\phi) = (j, k_1, k_2)$ and $Q(\phi') = (j', k'_1, k'_2)$, either $j < j'$, or $j = j'$ and $k_1 < k'_1$, or $j = j'$ and $k_1 = k'_1$ and $k_2 < k'_2$. Given impure ϕ , with $Q(\phi) = (j, k_1, k_2)$, ϕ contains $A_1 \text{ bop } A_2$ such that either A_1 or A_2 is impure and of size j . We shall now show how to transform ϕ , by replacing $A_1 \text{ bop } A_2$ with an equivalent assertion, into an 1-equivalent assertion ϕ' with $\phi' \prec \phi$. Since \prec is clearly a well-founded ordering, this will yield the claim.

If A_1 is impure and of size j , it is (Fact 7) of the form $C[H[A]]$ with H not an array variable. If H is Z , we can apply Fact 8 to replace $A_1 \text{ bop } A_2$ in ϕ . If H is of the form $H\{A_0 : A'\}$, we can apply Fact 9 to replace $A_1 \text{ bop } A_2$ in ϕ . In both cases, we end up with a ϕ' which is 1-equivalent to ϕ and which satisfies $\phi' \prec \phi$, since with $Q(\phi') = (j', k'_1, k'_2)$ we have either $j' < j$ (if A_1 was the only impure arithmetic expression of size j) or $k'_1 < k_1$.

If A_2 is impure and of size j , but A_1 is not, then A_2 is of the form $C[H[A]]$ with H not an array variable. If H is Z we can apply (the symmetric version of) Fact 8 to replace $A_1 \text{ bop } A_2$ in ϕ . If H is of the form $H\{A_0 : A'\}$, we can apply (the symmetric version of) Fact 9 to replace $A_1 \text{ bop } A_2$ in ϕ . In both cases, we end up with a ϕ' which is 1-equivalent to ϕ and which clearly satisfies $\phi' \prec \phi$, since with $Q(\phi') = (j', k'_1, k'_2)$ we have either $j' < j$ (if A_2 was the only impure arithmetic expression of size j) or $j' = j$ and $k'_1 = k_1$ and $k'_2 < k_2$. \square

Definition of rm^+

We define rm^+ , having the property that if $\phi' = rm^+_X(\phi)$ then ϕ' does not contain any variables from X , and is logically implied by ϕ , simultaneously with its dual rm^- which has the property that if $\phi' = rm^-_X(\phi)$ then ϕ' does not contain any variables from X , and logically implies ϕ .

$$\begin{array}{l|l} rm^+_X(B) = true & \text{if } fv(B) \cap X \neq \emptyset \\ rm^+_X(B) = B & \text{if } fv(B) \cap X = \emptyset \\ rm^+_X(\phi_1 \wedge \phi_2) = rm^+_X(\phi_1) \wedge rm^+_X(\phi_2) \\ rm^+_X(\phi_1 \vee \phi_2) = rm^+_X(\phi_1) \vee rm^+_X(\phi_2) \\ rm^+_X(\neg\phi_0) = \neg rm^+_X(\phi_0) \end{array} \quad \left| \quad \begin{array}{l} rm^-_X(B) = false & \text{if } fv(B) \cap X \neq \emptyset \\ rm^-_X(B) = B & \text{if } fv(B) \cap X = \emptyset \\ rm^-_X(\phi_1 \wedge \phi_2) = rm^-_X(\phi_1) \wedge rm^-_X(\phi_2) \\ rm^-_X(\phi_1 \vee \phi_2) = rm^-_X(\phi_1) \vee rm^-_X(\phi_2) \\ rm^-_X(\neg\phi_0) = \neg rm^-_X(\phi_0) \end{array} \right.$$

Some results about substitution

Lemma 10. For all E, A, s, x , with $v = \llbracket A \rrbracket_s$ and with $s' = [s \mid x \mapsto v]$, we have

$$\llbracket E[A/x] \rrbracket_s = \llbracket E \rrbracket_{s'}.$$

Proof: Structural induction on E . If E is a constant c , then both sides evaluate to c . If E is the array constant Z , then both sides evaluate to $\lambda n.0$. If E equals x , both sides evaluate to v . If E is a scalar variable y with $y \neq x$, then both sides evaluate to $s(y)$. If E is an array variable h , then both sides evaluate to $s(h)$. If E is of the form $A_1 \text{ op } A_2$ or $A_1 \text{ bop } A_2$, the claim follows easily from the induction hypothesis. If E is of the form $H[A_0]$, the left hand side evaluates to $\llbracket H[A/x] \rrbracket_s(\llbracket A_0[A/x] \rrbracket_s)$ while the right hand side evaluates to $\llbracket H \rrbracket_{s'}(\llbracket A_0 \rrbracket_{s'})$; the equality now follows from the induction hypothesis. If E is of the form $H\{A_0 : A'\}$, the left hand side evaluates to $\llbracket H[A/x] \rrbracket_s \mid \llbracket A_0[A/x] \rrbracket_s \mapsto \llbracket A'[A/x] \rrbracket_s$ whereas the right hand side evaluates to $\llbracket H \rrbracket_{s'} \mid \llbracket A_0 \rrbracket_{s'} \mapsto \llbracket A' \rrbracket_{s'}$; the equality now follows from the induction hypothesis. \square

Lemma 11. For all E, H, s, h , with $a = \llbracket H \rrbracket_s$ and with $s' = [s \mid h \mapsto a]$, we have

$$\llbracket E[H/h] \rrbracket_s = \llbracket E \rrbracket_{s'}.$$

Proof: Structural induction on E . If E is a constant c , then both sides evaluate to c . If E is the array constant Z , then both sides evaluate to $\lambda n.0$. If E is a scalar variable x , then both sides evaluate to $s(x)$. If E equals h , then both sides evaluate to a . If E is an array variable h_1 with $h_1 \neq h$, then both sides evaluate to $s(h_1)$. If E is of the form $A_1 \text{ op } A_2$ or $A_1 \text{ bop } A_2$, the claim follows easily from the induction hypothesis. If E is of the form $H_0[A]$, the left hand side evaluates to $\llbracket H_0[H/h] \rrbracket_s(\llbracket A[H/h] \rrbracket_s)$ while the right hand side evaluates to $\llbracket H_0 \rrbracket_{s'}(\llbracket A \rrbracket_{s'})$; the equality now follows from the

induction hypothesis. If E is of the form $H_0\{A_0 : A'\}$, the left hand side evaluates to $\llbracket [H_0[H/h]]_s \mid [A_0[H/h]]_s \mapsto [A'[H/h]]_s \rrbracket$ whereas the right hand side evaluates to $\llbracket [H_0]_{s'} \mid [A_0]_{s'} \mapsto [A']_{s'} \rrbracket$; the equality now follows from the induction hypothesis. \square

Lemma 12. *For all ϕ, A, s, x , with $v = \llbracket A \rrbracket_s$ and with $s' = [s \mid x \mapsto v]$, we have*

$$s \models \phi[A/x] \text{ iff } s' \models \phi.$$

Proof: Structural induction in ϕ , with a case analysis on the form of ϕ . If ϕ is some boolean expression B , the claim follows directly from Lemma 10. Otherwise, for instance when ϕ is of the form $\phi_1 \wedge \phi_2$, the claim follows easily from the induction hypothesis. \square

Lemma 13. *For all ϕ, H, s, h , with $a = \llbracket H \rrbracket_s$ and with $s' = [s \mid h \mapsto a]$, we have*

$$s \models \phi[H/h] \text{ iff } s' \models \phi.$$

Proof: As the proof of Lemma 12, using Lemma 11 rather than Lemma 10. \square

Some results about the analysis of while loops

We consider the algorithm in Fig. 7, the clause for while loops.

Lemma 14. *For all w , and for all i, i' with $i \leq i'$, we have $\phi_w^i \triangleright_1 \phi_w^{i'}$.*

Proof: Follows from property (a) of the ∇ operator. \square

Lemma 15. *For all i, i' with $i \leq i'$, we have $\Theta^{i'} \triangleright_2 \Theta^i$.*

Proof: Follows from Lemma 14, by the contravariance result in Lemma 2. \square

Lemma 16. *The number j is well-defined, and for all $i \geq j$ and all $w \in X$, $\phi_w^i = \phi_w^j$.*

Proof: Due to property (b) of the ∇ operator, for each $w \in X$ the chain $\{\phi_w^i \mid i \geq 0\}$ contains only a finite number of elements; the claim now follows since X is finite. (Note that if we would allow individual array elements to appear as consequents of loop invariants, we would operate with a set X that is essentially infinite, and we might need widening to eventually replace, say, $h[x]$ by h .) \square

Lemma 17. *In Fig. 7, the clause for while loops, we have $\Theta_0 \triangleright_2 \Theta'$.*

Proof: Assume that $s \& s_1 \models \Theta_0$, and let $(\phi \Rightarrow E \times) \in \Theta'$ be given; assuming $s \models \phi$ and $s_1 \models \phi$, our task is to show $\llbracket E \rrbracket_s = \llbracket E \rrbracket_{s_1}$ where it suffices to prove that for an arbitrary $w \in \text{fv}(E)$, $s(w) = s_1(w)$. By construction of ϕ_w^0 , ϕ logically implies ϕ_w^0 , and thus $s \models \phi_w^0$ and $s_1 \models \phi_w^0$. From $(\phi_w^0 \Rightarrow w \times) \in \Theta_0$ and $s \& s_1 \models \Theta_0$ we now infer the desired $s(w) = s_1(w)$. \square

Lemma 18. *In Fig. 7, the clause for while loops, if $\Theta'_m \neq \emptyset$ then $\Theta \triangleright_2 \Theta^j$.*

Proof: If $\Theta'_m \neq \emptyset$ then $\Theta^j \subseteq \text{dom}(R_m) \subseteq \Theta$ which yields the claim. \square

Lemma 19. *In Fig. 7, the clause for while loops, for all i we have $\Theta^{i+1} \triangleright_2 \text{dom}(R^i)$.*

Proof: Assume that $s \& s_1 \models \Theta^{i+1}$, and let $(\phi \Rightarrow E \times) \in \text{dom}(R^i)$ be given; assuming $s \models \phi$ and $s_1 \models \phi$, our task is to show $\llbracket E \rrbracket_s = \llbracket E \rrbracket_{s_1}$ where it suffices to prove that for an arbitrary $w \in \text{fv}(E)$, $s(w) = s_1(w)$. Since $(\phi \Rightarrow E \times, -, -) \in R^i$, we see from the construction of ψ_w^i that ϕ logically implies ψ_w^i , which by construction logically implies ϕ_w^{i+1} . Thus $s \models \phi_w^{i+1}$ and $s_1 \models \phi_w^{i+1}$, so from $(\phi_w^{i+1} \Rightarrow w \times) \in \Theta^{i+1}$ and $s \& s_1 \models \Theta^{i+1}$ we infer the desired $s(w) = s_1(w)$. \square

Proof of Lemma 5

Lemma 5 Assume $\{\Theta\} (R) \Leftarrow S \{\Theta'\}$. Then

Totality $dom(R) = \Theta$ (left totality) and $ran(R) = \Theta'$ (right totality).

Uniqueness Given $\theta' \in \Theta'$, there exists at most one θ such that $(\theta, u, \theta') \in R$.

Write Confinement If $(\theta, u, \theta') \in R$, then $con(\theta) = con(\theta')$, and with $E = con(\theta)$ we have that if $s \llbracket S \rrbracket s'$ then s agrees with s' on $fv(E)$.

Proof: The proof is by induction in S , with a case analysis on the form of S where we shall use the terminology from Fig. 7.

$S = \text{skip}$ or $S = \text{assert}(\phi_0)$. The claims are all trivial.

$S = x := A$ or $S = h := \text{new}$ or $S = h[A_0] := A'$. Left Totality follows from the construction of Θ , while Right Totality follows from the construction of R . Uniqueness is trivial since for each $\theta' \in \Theta'$ there exists exactly one θ with $(\theta, -, \theta') \in R$.

For Write Confinement, assume $(\theta, u, \theta') \in R$ with $E = con(\theta')$.

- If S is of the form $x := A$, then $x \notin fv(E)$. Therefore $con(\theta) = E$, and if $s \llbracket S \rrbracket s'$ then s agrees with s' except on x , in particular they agree on $fv(E)$.
- Otherwise, $h \notin fv(E)$. Therefore $con(\theta) = E$, and if $s \llbracket S \rrbracket s'$ then s agrees with s' except on h , in particular they agree on $fv(E)$.

$S = S_1 ; S_2$. The induction hypothesis obviously gives us Totality and Uniqueness. For Write Confinement, assume that $(\theta, u, \theta') \in R$; this happens because there exists θ'' with $(\theta, u, \theta'') \in R_1$ and $(\theta'', u, \theta') \in R_2$. With $E = con(\theta')$, inductively we infer first $E = con(\theta'')$ and next $E = con(\theta)$. Finally, assume that $s \llbracket S \rrbracket s'$; then there exists s'' such that $s \llbracket S_1 \rrbracket s''$ and $s'' \llbracket S_2 \rrbracket s'$. Given $w \in fv(E)$, we must show that $s(w) = s'(w)$, but this follows since inductively we have $s(w) = s''(w)$ and $s''(w) = s'(w)$.

$S = \text{if } B \text{ then } S_1 \text{ else } S_2$. Uniqueness follows easily from the induction hypothesis, and Left Totality follows from the construction of Θ . For Right Totality, consider $\theta' \in \Theta'$. If $\theta' \in \Theta'_m$ then the claim follows, due to R'_1 , since inductively we can assume that R_1 is total. If $\theta' \in \Theta'_u$ then we infer inductively that there exists θ_1, θ_2 such that $(\theta_1, u, \theta') \in R_1$ and $(\theta_2, u, \theta') \in R_2$; we also infer inductively that $con(\theta_1) = con(\theta') = con(\theta_2)$. But this shows, due to R_0 , that there exists θ with $(\theta, u, \theta') \in R$.

We now address Write Confinement, and consider $(\theta, u, \theta') \in R$, that is $(\theta, u, \theta') \in R_0$ with $\theta' \in \Theta'_u$. Thus there exists $(\theta_1, u, \theta') \in R_1$ and $(\theta_2, u, \theta') \in R_2$ with $con(\theta) = con(\theta_1) = con(\theta_2)$. Inductively, we infer that $con(\theta_1) = con(\theta')$ and hence $con(\theta) = con(\theta')$. Finally, assume that $s \llbracket S \rrbracket s'$ where we can assume, wlog., that $s \models B$ and $s \llbracket S_1 \rrbracket s'$. Given $w \in fv(E)$, with $E = con(\theta)$, we must show that $s(w) = s'(w)$, but this follows from the induction hypothesis.

$S = \text{call } p$. Uniqueness is obvious from the construction of R , and Left Totality from the construction of Θ . To show Right Totality, assume that $\theta' = (\phi \Rightarrow E \times) \in \Theta'$: if $fv(E) \cap \text{out}_p = \emptyset$ then $\theta' \in \text{ran}(R_u) \subseteq \text{ran}(R)$; if $fv(E) \cap \text{out}_p \neq \emptyset$ then $\theta' \in \text{ran}(R_m) \subseteq \text{ran}(R)$ since each postcondition has at least one precondition, cf. Requirement 4 on p. 14.

We now address Write Confinement, and consider $(\theta, u, \theta') \in R$, that is $(\theta, u, \theta') \in R_u$. We see that there exists E such that $con(\theta') = E = con(\theta)$. Finally, assume

that $s \llbracket S \rrbracket s'$, that is $s \models P(p) \ s'$, and let $w \in \text{fv}(E)$ be given; we must show that $s(w) = s'(w)$. But from the construction of R_u we infer that $w \notin \text{OUT}_p$, and the claim now follows from Requirement 1 for summaries (p. 14).

$S = \text{while } B \text{ do } S_0 \text{ od}$. Left Totality follows by construction of Θ . For Right Totality, note that if $\theta' \in \Theta'$ belongs to Θ'_m there exists at least one θ such that $(\theta, m, \theta') \in R_m$; otherwise, if $\theta' \in \Theta'_u$, there exists exactly one θ (determined by the form of $\text{con}(\theta)$) such that $(\theta, u, \theta') \in R_u$. The latter observation also shows Uniqueness.

For Write Confinement, assume that $(\theta, u, \theta') \in R$, that is $\theta' \in \Theta'_u$ and $\text{con}(\theta) = E$ where $E = \text{con}(\theta')$. Now consider $w \in \text{fv}(E)$, and assume that $s \llbracket S \rrbracket s'$; we must prove that $s(w) = s'(w)$. By the definition of Θ'_u we infer that R^j does not contain any element of the form $(-, m, - \Rightarrow w \times)$. We now apply the induction hypothesis to S_0 , and the call $\{ \cdot \} (R^j) \Leftarrow S_0 \{ \Theta^j \}$, first inferring from Right Totality that R^j contains an element of the form $(-, u, - \Rightarrow w \times)$, and next inferring from Write Confinement that if $s_0 \llbracket S_0 \rrbracket s'_0$ then $s_0(w) = s'_0(w)$. With f_i as defined in Fig. 5, it is now easy to show by induction in i that if $s \models f_i \ s'$ then $s(w) = s'(w)$. But this establishes Write Confinement. \square

Proof of Lemma 6

Lemma 6 Assume $\{ \Theta \} (R) \Leftarrow S \{ \Theta' \}$. Given $\theta' \in \Theta'$, there exists $(\theta, -, \theta') \in R$ such that whenever $s \llbracket S \rrbracket s'$ and $s' \models \text{ant}(\theta')$ then $s \models \text{ant}(\theta)$.

Proof: The proof is by induction in S , with a case analysis on the form of S where we shall use the terminology from Fig. 7.

$S = \text{skip}$. Trivial.

$S = \text{assert}(\phi_0)$. Given $\theta' \in \Theta'$, there exists $(\theta, -, \theta') \in R$ such that $\text{ant}(\theta) = \text{ant}(\theta') \wedge \phi_0$. This does the job: assume $s \llbracket S \rrbracket s'$, which amounts to $s \models \phi_0$ and $s' = s$, and that $s' \models \text{ant}(\theta')$; then clearly $s \models \text{ant}(\theta)$.

$S = x := A$. Given $\theta' \in \Theta'$, there exists $(\theta, -, \theta') \in R$ such that with $\phi = \text{ant}(\theta)$ and $\phi' = \text{ant}(\theta')$ we have $\phi = \phi'[A/x]$. This does the job: assume $s \llbracket S \rrbracket s'$, which amounts to $s' = [s \mid x \mapsto v]$ where $v = \llbracket A \rrbracket_s$, and that $s' \models \phi'$; then by Lemma 12, we have the desired $s \models \phi$.

$S = h := \text{new}$. Given $\theta' \in \Theta'$, there exists $(\theta, -, \theta') \in R$ such that with $\phi = \text{ant}(\theta)$ and $\phi' = \text{ant}(\theta')$ we have $\phi = \phi'[Z/h]$. This does the job: assume $s \llbracket S \rrbracket s'$, which amounts to $s' = [s \mid h \mapsto \lambda n.0]$, that is $s' = [s \mid h \mapsto \llbracket Z \rrbracket_s]$. If also $s' \models \phi'$, then by Lemma 13 we have the desired $s \models \phi$.

$S = h[A_0] := A'$. Given $\theta' \in \Theta'$, there exists $(\theta, -, \theta') \in R$ such that with $\phi = \text{ant}(\theta)$ and $\phi' = \text{ant}(\theta')$ we have $\phi = \phi'[h\{A_0 : A'\}/h]$. This does the job: assume $s \llbracket S \rrbracket s'$, which amounts to $s' = [s \mid h(n) \mapsto v]$ with $n = \llbracket A_0 \rrbracket_s$ and $v = \llbracket A' \rrbracket_s$, that is $s' = [s \mid h \mapsto a]$ where $a = \llbracket h\{A_0 : A'\} \rrbracket_s$. If also $s' \models \phi'$, then by Lemma 13 we have the desired $s \models \phi$.

$S = S_1 ; S_2$. Given $\theta' \in \Theta'$, inductively on S_2 we can find $(\theta'', -, \theta') \in R_2$ with the property that if $s'' \llbracket S_2 \rrbracket s'$ and $s' \models \text{ant}(\theta')$ then $s'' \models \text{ant}(\theta'')$, and inductively on S_1 we can next find $(\theta, -, \theta'') \in R_1$ with the property that if $s \llbracket S_1 \rrbracket s''$ and $s'' \models \text{ant}(\theta'')$ then $s \models \text{ant}(\theta)$. Since $(\theta, -, \theta') \in R$, our task can be accomplished by showing $s \models$

$ant(\theta)$ from the assumptions $s' \models ant(\theta')$ and $s \llbracket S \rrbracket s'$. The latter assumption implies the existence of s'' with $s'' \llbracket S_2 \rrbracket s'$ and $s \llbracket S_1 \rrbracket s''$; from $s' \models ant(\theta')$ and the property of S_2 and θ'' we now infer $s'' \models ant(\theta'')$, and from the property of S_1 and θ we next infer the desired $s \models ant(\theta)$.

$S = \text{if } B \text{ then } S_1 \text{ else } S_2$. Given $\theta' \in \Theta'$, inductively on S_1 we can find $(\theta_1, _, \theta') \in R_1$ with the property that if $s \llbracket S_1 \rrbracket s'$ and $s' \models ant(\theta')$ then $s \models ant(\theta)$, and inductively on S_2 we can find $(\theta_2, _, \theta') \in R_2$ with the property that if $s \llbracket S_2 \rrbracket s'$ and $s' \models ant(\theta')$ then $s \models ant(\theta)$. Let $\theta_1 = (\phi_1 \Rightarrow E_1 \times)$, let $\theta_2 = (\phi_2 \Rightarrow E_2 \times)$, and let $\theta' = (\phi' \Rightarrow E \times)$. We now first define ϕ as $(\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)$, and next define θ as follows: if $\theta' \in \Theta'_m$ then we let $\theta = (\phi \Rightarrow B \times)$ so that $(\theta, _, \theta') \in R'_0 \subseteq R$; if $\theta' \in \Theta'_u$, in which case Lemma 5 tells us that $E = E_1 = E_2$, we let $\theta = (\phi \Rightarrow E \times)$ so that $(\theta, _, \theta') \in R_0 \subseteq R$. We must prove that θ thus constructed has the desired property, so assume that $s \llbracket S \rrbracket s'$ and $s' \models \phi'$ so as to prove $s \models \phi$. Wlog. we may assume that $s \models B$ and thus $s \llbracket S_1 \rrbracket s'$, in which case the property of S_1 and θ_1 tells us that $s \models \phi_1$ which together with $s \models B$ implies the desired $s \models \phi$.

$S = \text{call } p$. Given $\theta' \in \Theta'$, and let $\phi' = ant(\theta')$. We define $\phi = rm_{out_p}^+(\phi')$, and assuming $s \llbracket S \rrbracket s'$ and $s' \models \phi'$ we shall prove $s \models \phi$: by the requirements to rm^+ we have $s' \models \phi$, and since $fv(\phi) \cap out_p = \emptyset$ holds by construction of ϕ , we infer from Requirement 1 for summaries (p. 14) that $s(w) = s'(w)$ for all $w \in fv(\phi)$, yielding the desired $s \models \phi$.

It is thus sufficient to show that R always contains a triple of the form $(\phi \Rightarrow _, _, \theta')$. But if $fv(E) \cap out_p = \emptyset$, this follows by the definition of R_u ; otherwise, this follows by the definition of R_m , together with Requirement 4 for summaries (p. 14).

$S = \text{while } B \text{ do } S_0 \text{ od}$. Let $\theta' \in \Theta'$ be given, with θ' of the form $\phi' \Rightarrow E \times$. We can assume $\theta' \in \Theta'_u$, since otherwise we have $(true \Rightarrow 0 \times, m, \theta') \in R_m \subseteq R$ and the claim is trivial. We can also assume $fv(E) \neq \emptyset$, since otherwise we have $(true \Rightarrow E \times, u, \theta') \in R_u \subseteq R$ and the claim is again trivial. With $\phi = \bigvee_{w \in fv(E)} \phi_w^j$, it thus holds that $(\phi \Rightarrow E \times, u, \theta') \in R$.

Let w belong to $fv(E)$. By construction of ϕ_w^0 , ϕ' logically implies ϕ_w^0 which (by Lemma 14) logically implies ϕ_w^j which logically implies ϕ . Therefore, it is sufficient to prove that if $s \llbracket S \rrbracket s'$ and $s' \models \phi$ then $s \models \phi$. A simple inductive argument, on f_i as defined in Fig. 5, now shows that it is sufficient to prove that if $s_0 \llbracket S_0 \rrbracket s'_0$ and $s'_0 \models \phi$ then $s_0 \models \phi$.

So assume $s_0 \llbracket S_0 \rrbracket s'_0$, and assume that $s'_0 \models \phi$, implying that there exists $w \in fv(E)$ such that $s'_0 \models \phi_w^j$. We now apply the induction hypothesis on S_0 , and infer from $\{-\} (R^j) \Leftarrow S_0 \{\Theta^j\}$ that there exists $(\phi_0 \Rightarrow E_0 \times, \gamma, \phi_w^j \Rightarrow w \times) \in R^j$ such that $s_0 \models \phi_0$. From $\theta' \notin \Theta'_m$ we infer that $\gamma = u$, which by Lemma 5 implies $E_0 = w$. By construction of ψ_w^j , we see that ϕ_0 logically implies ψ_w^j , and thus $s_0 \models \psi_w^j$. By definition of j , we have $\phi_w^j = \phi_w^{j+1}$, so from the definition of ϕ_w^{j+1} , we infer that ψ_w^j logically implies ϕ_w^j . Thus $s_0 \models \phi_w^j$ and hence the desired $s_0 \models \phi$. \square

Proof of Theorem 1

We need one more auxiliary result:

Lemma 20. *In Fig. 7, the clause for while loops: if $s \& s_1 \models \Theta^j$ and $\llbracket B \rrbracket_s \neq \llbracket B \rrbracket_{s_1}$ and $s \llbracket S_0 \rrbracket s'$ then $s' \& s_1 \models \Theta^j$.*

Proof: To prove the conclusion, let $(\phi_w^j \Rightarrow w \times) \in \Theta^j$ be given; assuming $s' \models \phi_w^j$ and $s_1 \models \phi_w^j$, we must prove $s'(w) = s_1(w)$. By Lemma 6 applied to $\{-\}$ (R^j) $\Leftarrow S_0 \{\Theta^j\}$ and our premise $s \llbracket S_0 \rrbracket s'$, we see that R^j contains an element $(\phi \Rightarrow E \times, \gamma, \phi_w^j \Rightarrow w \times)$ where $s \models \phi$. Here $\gamma = u$, as we shall show below.

Assume, to get a contradiction, that $\gamma = m$. It suffices to show that for all $z \in \text{fv}(B)$ we have $s(z) = s_1(z)$, since then $\llbracket B \rrbracket_s = \llbracket B \rrbracket_{s_1}$ which contradicts one of our premises. So let $z \in \text{fv}(B)$ be given; by construction of ψ_z^j we see from $(\phi \Rightarrow E \times, \gamma, \phi_w^j \Rightarrow w \times) \in R^j$ that $\phi \triangleright_1 \psi_z^j$ and $\phi_w^j \triangleright_1 \psi_z^j$; since $\psi_z^j \triangleright_1 \phi_z^{j+1}$ and $\phi_z^{j+1} = \phi_z^j$ this implies $s \models \phi_z^j$ and $s_1 \models \phi_z^j$. From $(\phi_z^j \Rightarrow z \times) \in \Theta^j$ and $s \& s_1 \models \Theta^j$ we thus infer the desired $s(z) = s_1(z)$.

We have showed that $\gamma = u$, so by Lemma 5 we infer that $E = w$, and that $s(w) = s'(w)$. By construction of ψ_w^j we see that $\phi \triangleright_1 \psi_w^j$, and hence also $\phi \triangleright_1 \phi_w^j$. Thus we have not just $s_1 \models \phi_w^j$ but also $s \models \phi_w^j$, so since $s \& s_1 \models \Theta^j$ with $(\phi_w^j \Rightarrow w \times) \in \Theta^j$, we infer $s(w) = s_1(w)$ and hence the desired $s'(w) = s_1(w)$. \square

Theorem 1 Assume that

1. $\{\Theta\} (-) \Leftarrow S \{\Theta'\}$
2. $s \llbracket S \rrbracket s'$ and $s_1 \llbracket S \rrbracket s'_1$
3. $s \& s_1 \models \Theta$

Then $s' \& s'_1 \models \Theta'$.

Proof: The proof is by induction in S , with a case analysis on the form of S where we shall use the terminology from Fig. 7.

$S = \text{skip}$. Trivial.

$S = \text{assert}(\phi_0)$. We have $s \models \phi_0$, $s_1 \models \phi_0$, $s' = s$, and $s'_1 = s_1$. Let $(\phi \Rightarrow E \times) \in \Theta'$ be given, and assume that $s \models \phi$ and $s_1 \models \phi$; we must prove $\llbracket E \rrbracket_s = \llbracket E \rrbracket_{s_1}$. But $(\phi \wedge \phi_0) \Rightarrow E \times \in \Theta$ so from $s \& s_1 \models \Theta$ we have $s \& s_1 \models (\phi \wedge \phi_0) \Rightarrow E \times$, and since $s \models \phi \wedge \phi_0$ and $s_1 \models \phi \wedge \phi_0$ this implies the desired $\llbracket E \rrbracket_s = \llbracket E \rrbracket_{s_1}$.

$S = S_1 ; S_2$. There exists s'' and s'_1 such that $s \llbracket S_1 \rrbracket s''$ and $s'' \llbracket S_2 \rrbracket s'$ and $s_1 \llbracket S_1 \rrbracket s'_1$ and $s'_1 \llbracket S_2 \rrbracket s'_1$. Given $s \& s_1 \models \Theta$, we can apply the induction hypothesis to S_1 to give $s'' \& s'_1 \models \Theta''$, and next apply the induction hypothesis to S_2 to give the desired $s' \& s'_1 \models \Theta'$.

$S = x := A$. Given $\theta' = (\phi' \Rightarrow E' \times) \in \Theta'$; assuming $s' \models \phi'$ and $s'_1 \models \phi'$, we must prove $\llbracket E' \rrbracket_{s'} = \llbracket E' \rrbracket_{s'_1}$. Here $s' = [s \mid x \mapsto v]$ with $v = \llbracket A \rrbracket_s$, and $s'_1 = [s_1 \mid x \mapsto v_1]$ with $v_1 = \llbracket A \rrbracket_{s_1}$. With $\phi = \phi'[A/x]$ and $E = E'[A/x]$ we have $\phi \Rightarrow E \times \in \Theta$ and thus $s \& s_1 \models \phi \Rightarrow E \times$. From $s' \models \phi'$ and $s'_1 \models \phi'$ we infer by Lemma 12 that $s \models \phi$ and $s_1 \models \phi$, implying $\llbracket E \rrbracket_s = \llbracket E \rrbracket_{s_1}$. Since by Lemma 10 we have $\llbracket E' \rrbracket_{s'} = \llbracket E \rrbracket_s$ and $\llbracket E' \rrbracket_{s'_1} = \llbracket E \rrbracket_{s_1}$, this amounts to the desired $\llbracket E' \rrbracket_{s'} = \llbracket E' \rrbracket_{s'_1}$.

$S = h := \text{new}$. Given $\theta' = (\phi' \Rightarrow E' \times) \in \Theta'$; assuming $s' \models \phi'$ and $s'_1 \models \phi'$, we must prove $\llbracket E' \rrbracket_{s'} = \llbracket E' \rrbracket_{s'_1}$. Here $s' = [s \mid h \mapsto a]$ with $a = \llbracket Z \rrbracket_s$, and $s'_1 = [s_1 \mid h \mapsto a_1]$ with $a_1 = \llbracket Z \rrbracket_{s_1}$. With $\phi = \phi'[Z/h]$ and $E = E'[Z/h]$ we have $\phi \Rightarrow E \times \in \Theta$ and thus $s \& s_1 \models \phi \Rightarrow E \times$. From $s' \models \phi'$ and $s'_1 \models \phi'$ we infer by Lemma 13 that $s \models \phi$ and $s_1 \models \phi$, implying $\llbracket E \rrbracket_s = \llbracket E \rrbracket_{s_1}$. Since by Lemma 11 we have $\llbracket E' \rrbracket_{s'} = \llbracket E \rrbracket_s$ and $\llbracket E' \rrbracket_{s'_1} = \llbracket E \rrbracket_{s_1}$, this amounts to the desired $\llbracket E' \rrbracket_{s'} = \llbracket E' \rrbracket_{s'_1}$.

$S = h[A_0] := A'$. Given $\theta' = (\phi' \Rightarrow E' \times) \in \Theta'$; assuming $s' \models \phi'$ and $s'_1 \models \phi'$, we must prove $\llbracket E' \rrbracket_{s'} = \llbracket E' \rrbracket_{s'_1}$. Here $s' = [s \mid h(n) \mapsto v]$ with $n = \llbracket A_0 \rrbracket_s$ and $v = \llbracket A' \rrbracket_s$, and thus $s' = [s \mid h \mapsto a]$ with $a = \llbracket h\{A_0 : A'\} \rrbracket_s$. Similarly, $s'_1 = [s_1 \mid h \mapsto a_1]$ with $a_1 = \llbracket h\{A_0 : A'\} \rrbracket_{s_1}$. With $\phi = \phi'[h\{A_0 : A'\}/h]$ and $E = E'[h\{A_0 : A'\}/h]$ we have $\phi \Rightarrow E \times \in \Theta$ and thus $s \& s_1 \models \phi \Rightarrow E \times$. From $s' \models \phi'$ and $s'_1 \models \phi'$ we infer by Lemma 13 that $s \models \phi$ and $s_1 \models \phi$, implying $\llbracket E \rrbracket_s = \llbracket E \rrbracket_{s_1}$. Since by Lemma 11 we have $\llbracket E' \rrbracket_{s'} = \llbracket E \rrbracket_s$ and $\llbracket E' \rrbracket_{s'_1} = \llbracket E \rrbracket_{s_1}$, this amounts to the desired $\llbracket E' \rrbracket_{s'} = \llbracket E' \rrbracket_{s'_1}$.

$S = \text{if } B \text{ then } S_1 \text{ else } S_2$. Except for symmetry, there are two cases. The first case is when $\llbracket B \rrbracket_s = \llbracket B \rrbracket_{s_1} = \text{True}$, in which case we have $s \llbracket S_1 \rrbracket s'$ and $s_1 \llbracket S_1 \rrbracket s'_1$. It is sufficient to establish $s \& s_1 \models \Theta_1$, since then the desired $s' \& s'_1 \models \Theta'$ will follow by induction on S_1 . So given $(\phi_1 \Rightarrow E_1 \times) \in \Theta_1$, and assuming $s \models \phi_1$ and $s_1 \models \phi_1$, our obligation is to show $\llbracket E_1 \rrbracket_s = \llbracket E_1 \rrbracket_{s_1}$. By Lemma 5 (Totality) applied to S_1 , there exists $\theta' \in \Theta'$ such that $(\phi_1 \Rightarrow E_1 \times, \neg, \theta') \in R_1$. Two cases:

- if $\theta' \in \Theta'_m$ then, by R'_1 , $(\phi_1 \wedge B \Rightarrow E_1 \times) \in \Theta$.
- if $\theta' \in \Theta'_u$ then, by R_0 and Lemma 5, there exists ϕ_2 such that $(\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B) \Rightarrow E_1 \times \in \Theta$.

Since $s \models \phi_1 \wedge B$ and $s_1 \models \phi_1 \wedge B$, and since $s \& s_1 \models \Theta$, in both cases we can infer the desired $\llbracket E_1 \rrbracket_s = \llbracket E_1 \rrbracket_{s_1}$.

The second case to be considered is when $\llbracket B \rrbracket_s = \text{False}$ but $\llbracket B \rrbracket_{s_1} = \text{True}$, in which case we have $s \llbracket S_2 \rrbracket s'$ and $s_1 \llbracket S_1 \rrbracket s'_1$. Given $\theta' = (\phi' \Rightarrow E' \times) \in \Theta'$, and assuming $s' \models \phi'$ and $s'_1 \models \phi'$, our proof obligation is to show $\llbracket E' \rrbracket_{s'} = \llbracket E' \rrbracket_{s'_1}$.

We shall establish that $\theta' \in \Theta'_u$, by showing that $\theta' \in \Theta'_m$ leads to a contradiction: by Lemma 6 applied to S_1 and S_2 , there exists $(\phi_1 \Rightarrow \neg \times, \neg, \theta') \in R_1$ and $(\phi_2 \Rightarrow \neg \times, \neg, \theta') \in R_2$ such that $s_1 \models \phi_1$ and $s \models \phi_2$. Due to R'_0 we see that Θ contains $(\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B) \Rightarrow B \times$. Since $s \models (\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)$ and $s_1 \models (\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)$, from $s \& s_1 \models \Theta$ we infer $\llbracket B \rrbracket_s = \llbracket B \rrbracket_{s_1}$. But this contradicts our case assumption.

We have established $\theta' \in \Theta'_u$. By Lemma 5 we now infer that there exists unique $\theta \in \Theta$ such that $(\theta, \neg, \theta') \in R$ and also that $E = \text{con}(\theta)$. By Lemma 6, with $\phi = \text{ant}(\theta)$, we infer that $s \models \phi$ and $s_1 \models \phi$. From $s \& s_1 \models \Theta$ we thus infer $\llbracket E \rrbracket_s = \llbracket E \rrbracket_{s_1}$. But since Lemma 5 also states that s, s' agree on $\text{fv}(E)$, and that s_1, s'_1 agree on $\text{fv}(E)$, this amounts to the desired $\llbracket E' \rrbracket_{s'} = \llbracket E' \rrbracket_{s'_1}$.

$S = \text{call } p$. We consider $\theta' \in \Theta'$ of the form $\phi' \Rightarrow E' \times$, and must prove $s' \& s'_1 \models \phi' \Rightarrow E' \times$. So assume $s' \models \phi'$ and $s'_1 \models \phi'$, in order to prove $\llbracket E' \rrbracket_{s'} = \llbracket E' \rrbracket_{s'_1}$. Define $\phi = \text{rm}_{\text{out}_p}^+(\phi')$ and observe, as in the proof of Lemma 6, that by the properties of rm^+ we have first $s' \models \phi$ and $s'_1 \models \phi$, and next (due to Requirement 1 for summaries) also $s \models \phi$ and $s_1 \models \phi$.

We now have two cases, the first one being when $\text{fv}(E) \cap \text{out}_p = \emptyset$, implying (by Requirement 1 for summaries) that s, s' , and also s_1, s'_1 , agree on $\text{fv}(E)$. By R_u we see that $\phi \Rightarrow E \times \in \Theta$, so from $s \& s_1 \models \Theta$ we infer $\llbracket E \rrbracket_s = \llbracket E \rrbracket_{s_1}$ which amounts to the desired $\llbracket E' \rrbracket_{s'} = \llbracket E' \rrbracket_{s'_1}$.

Next assume that $\text{fv}(E) \cap \text{out}_p \neq \emptyset$. Note that it is sufficient to prove that for all $w \in \text{fv}(E)$, $s'(w) = s'_1(w)$. So let $w \in \text{fv}(E)$ be given, with two subcases: if $w \notin \text{out}_p$ we see by R_0 that $\phi \Rightarrow w \times \in \Theta$, so from $s \& s_1 \models \Theta$ we infer $s(w) = s_1(w)$ which (by Requirement 1 for summaries) amounts to the desired $s'(w) = s'_1(w)$.

The last subcase is when $w \in \text{out}_p$. Let Θ_w be the preconditions for $w \times$ in the summary for p . It is sufficient to prove $s \& s_1 \models \Theta_w$, since the correctness of the sum-

mary (requirement 5 on p.14) will then imply the desired $s'(w) = s'_1(w)$. So consider $\theta \in \Theta_w$, of the form $\phi_w^z \Rightarrow z \times$, and assume $s \models \phi_w^z$ and $s_1 \models \phi_w^z$ so as to prove $s(z) = s_1(z)$. From R_m we see that $(\phi_w^z \wedge \phi) \Rightarrow z \times \in \Theta$. But since $s \& s_1 \models \Theta$ is part of our overall assumption, and since we have assumed $s \models \phi_w^z$ and $s_1 \models \phi_w^z$ and have already showed $s \models \phi$ and $s_1 \models \phi$, we can infer the desired $s(z) = s_1(z)$.

$S = \text{while } B \text{ do } S_0 \text{ od}$. First we consider the case when $\Theta'_m = \emptyset$. Let $(\phi' \Rightarrow E \times) \in \overline{\Theta'}$ be given; assuming $s' \models \phi'$ and $s'_1 \models \phi'$, we must prove $\llbracket E \rrbracket_{s'} = \llbracket E \rrbracket_{s'_1}$. There exists exactly one element in R of the form $(\theta, \gamma, \phi' \Rightarrow E \times)$; here $\gamma = u$ and θ is of the form $\phi \Rightarrow E \times$. By Lemma 5, applied to S , we know that $\llbracket E \rrbracket_s = \llbracket E \rrbracket_{s'}$ and $\llbracket E \rrbracket_{s_1} = \llbracket E \rrbracket_{s'_1}$. By Lemma 6, applied to S , we have $s \models \phi$ and $s_1 \models \phi$. Since $(\phi \Rightarrow E \times) \in \Theta$, we from $s \& s_1 \models \Theta$ infer $\llbracket E \rrbracket_s = \llbracket E \rrbracket_{s_1}$ which amounts to the desired $\llbracket E \rrbracket_{s'} = \llbracket E \rrbracket_{s'_1}$.

Next we consider the case when $\Theta'_m \neq \emptyset$. By Lemma 18 we then have $\Theta \triangleright_2 \Theta^j$. By Lemma 17 and Lemma 15 we also have $\Theta^j \triangleright_2 \Theta'$. It is therefore enough to show that, with f_i as defined in Fig. 5, that

$$\text{if } s f_k s' \text{ and } s_1 f_{k_1} s'_1 \text{ and } s \& s_1 \models \Theta^j \text{ then } s' \& s'_1 \models \Theta^j.$$

We shall do so by induction in $k + k_1$, performing a case analysis.

- If $k = k_1 = 0$, the claim is obvious as then $s = s'$, $s_1 = s'_1$.
- If $k > 0$ and $k_1 > 0$, then there exists s'', s''_1 such that $s \llbracket S_0 \rrbracket s''$, $s_1 \llbracket S_0 \rrbracket s''_1$, $s'' f_{k-1} s'$, $s''_1 f_{k_1-1} s'_1$. By Lemma 19 we have $\Theta^j \triangleright_2 \text{dom}(R^j)$, and therefore $s \& s_1 \models \text{dom}(R^j)$. The overall induction hypothesis, applied to S_0 and the call $\{\Theta_0\} (R^j) \Leftarrow S_0 \{\Theta^j\}$ where (by Lemma 5) we have $\Theta_0 = \text{dom}(R^j)$, now tells us $s'' \& s''_1 \models \Theta^j$. We can then apply the inner induction hypothesis to get the desired $s' \& s'_1 \models \Theta^j$.
- Apart from symmetry, the only case left is when $k > 0$ but $k_1 = 0$. Then $\llbracket B \rrbracket_s = \text{True}$ but $\llbracket B \rrbracket_{s_1} = \text{False}$, $s'_1 = s_1$, and there exists s'' such that $s \llbracket S_0 \rrbracket s''$ and $s'' f_{k-1} s'$. By Lemma 20, we have $s'' \& s_1 \models \Theta^j$. We can now apply the inner induction hypothesis to get the desired $s' \& s'_1 \models \Theta^j$.

□