Model Checking Information Flow

Michael W. Whalen, David A. Greve, and Lucas G. Wagner

Abstract Information flow modeling describes how information can be transferred between different locations within a software and/or hardware system. In this chapter, we define a notion of information flow based on traces that is useful for describing flow relations for synchronous dataflow languages such as Simulink® [11] and SCADE™ [4] that are often used for hardware/software co-design. We then define an automated method for analyzing information flow properties of Simulink models using model checking. This method is based on creating a *flow model* that tracks information flow throughout the model. Often, information flow properties are defined in terms of some form of *noninterference*, which states informally that objects in one security domain cannot perceive the actions of objects within another domain. We demonstrate that this method preserves the GWV functional notion of noninterference. We then describe how this proof relates to the GWV theorem and provide some insight into the relationship of the flow model and the flow graphs used in GWVr1. Finally, we demonstrate our analysis technique by analyzing the architecture of the Turnstile high-assurance cross-domain guard platform using our Gryphon translation framework and the Prover™ model checker.

1 Introduction

In order to describe the secure operation of a computer system, it is useful to study how information propagates through that system. For example, an unintended propagation of information between different components may constitute a *covert channel* that can be used by an attacker to gain access to protected information. We are there-

Michael W. Whalen, David A. Greve, Lucas G. Wagner
Rockwell Collins, Inc., Cedar Rapids, Iowa, USA

fore interested in determining how and when information may be communicated throughout a system. At Rockwell Collins, we have spent several years modeling *information flow* problems to support precise formal analyses of different kinds of software and hardware models.

In this chapter, we describe an analysis procedure that can be used to check a variety of information flow properties of hardware and software systems. One of the properties that can be checked is a form of *noninterference* [5, 20, 19, 21] that is defined over system traces. Informally, it states that a system input does not interfere with a particular output if it is possible to vary the trace of that input without affecting the output in question.

Although great strides have been made in the development of formal analysis tools over the last few years, there have been relatively few instances reported of their successful application to industrial problems outside of the realm of hardware engineering. In fact, software and system engineers are often completely unaware of the opportunities these tools offer. One of the goals of our analysis was that it could be completely automated and directly applicable to the tools and languages used by engineers at Rockwell Collins, such as MATLAB Simulink® [11] and Esterel Technologies SCADE Suite™ [4]. These tools are achieving widespread use in the avionics and automotive industry, and can also be used to describe hardware designs. The graphical models produced by these tools have straightforward formal semantics and are amenable to formal analysis. Furthermore, it is often the case that software and/or hardware implementations are generated directly from these models, so the analysis model is kept synchronized with the actual system artifact.

Our analysis is based on annotations that can be added directly to a Simulink or SCADE model that describe specific sources and sinks of information. After this annotation phase, the translation and model checking tools can be used to automatically demonstrate a variety of information flow properties. In the case of non-interference, they will prove either that there is no information flow between the source and sinks, or demonstrate a source of information flow in the form of a counterexample.

The result returned by the model checker must be justified by a general claim regarding the soundness of the analysis and the annotated model. To justify our analyses, we first define a kind of trace equivalence. This trace equivalence is just a form of the GWVr1 characterization defined earlier in Greve's information flow chapter [6]. We then define syntax and semantics for a synchronous dataflow language and provide an information flow semantics for the language. Next, we demonstrate that this information flow semantics characterizes (i.e. enforces) the trace equivalence, and define non-interference as a dual-property of the information flow characterization. The information flow semantics is then directly reflected into a "flow model" that is emitted as part of the translation and conjoined with the original model. We finally show

that model checking this conjoined model yields the same result as executing the flow model semantics.

The organization of the rest of the chapter is as follows: Section 2 introduces the concepts involved through the use of a motivating shared buffer example. Section 3 describes an abstract formalization of information flow through trace equivalence, presents the syntax and semantics for a simplified dataflow language, and proves an *interference theorem*, i.e., that the information flow semantics preserves the trace equivalence. Section 4 demonstrates how *non-interference* can be defined as a corollary of the interference theorem. Section 5 describes how this formalization is realized in the Gryphon tool suite. Section 6 describes how the tools can be used to analyze *intransitive interference.* Section 7 describes connections between the formalization in this chapter and the GWV formulation from Greve [6]. Section 8 describes applications of the analysis: the shared buffer model and also a large-scale model of the Rockwell Collins Turnstile high-assurance guard. Section 9 presents future directions for the analysis and concludes.

2 A Motivating Example

To motivate our presentation, we use an example of a shared buffer model, shown in Fig. 1. In this model, secret and unclassified information both pass through a shared buffer. In order to prevent leakage of secret information, this buffer is coordinated by a scheduler (bottom of the figure) that mediates access to the buffer. On the left, there are two input processes for secret and unclassified input. On the right, there are two output processes for secret and unclassified output.

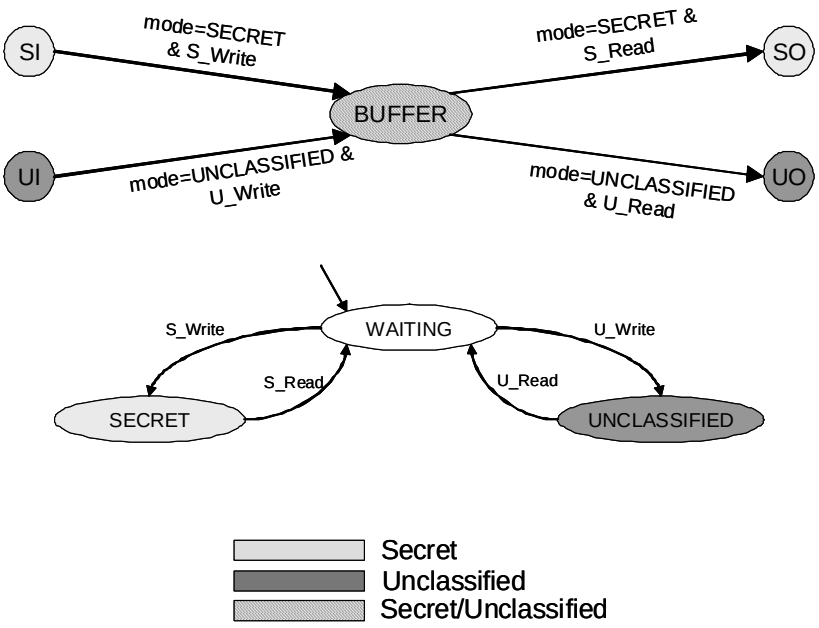

Fig. 1 Shared Buffer Architecture.

When the scheduler is in the WAITING state, a write request from either input process will result in that process obtaining the buffer. The process will continue to control the buffer until a corresponding read from the buffer is completed. The controller is designed to ensure that the secret data is only allowed to be consumed by the secret output, and symmetrically that the unclassified data is only consumed by the unclassified output.

Given this system, we would like to determine whether or not there is information flow between the secret processes and the unclassified processes. In other words, is it possible for the unclassified processes to glean information of any kind from the secret processes and vice versa? This information sharing is usually called *interference*; *non-interference* is the dual idea expressing that no information sharing occurs. In this example, the potential for interference exists via the scheduler. Unclassified processes can perceive the state of the buffer (whether they are able to read and write from it) via the scheduler, which is affected by the secret processes.

If we decide that this interference is allowable, we would like to be able to determine whether there are any other sources of interference between the secret and unclassified processes. An analysis which does not account for the current system state will probably decide that there is the potential for interference, since both kinds of

processes use a shared buffer. We would like a more accurate analysis that accounts for the scheduler state in order to show that there is no interference through the shared buffer.

This example demonstrates important features of the analysis that we will describe in the next sections:

- **Conditional Information Flow:** We would like the analysis to account for enough of the system state to allow an accurate analysis (e.g., that no information flows from a secret input to unclassified output through the shared buffer)
- **"Covert" Information Flow:** The scheduler does not directly convey information from secret processes to unclassified processes, yet its state allows information about the secret processes to be perceived. The analysis should detect this interference.
- **Intransitive Information Flow:** If we are willing to allow information flow through the scheduler, there should be a mechanism to allow us to tag this information path as "allowable" and determine if other sources of flow exist. In the noninterference literature, this is generally described as *intransitive noninterference* [5, 19, 20]. The meaning of *intransitive* has to do with the nature of information flows. Since the scheduler depends on the secret input and the unclassified output depends on the scheduler, a *transitive* analysis would assert that the unclassified output depends on the secret input. However, we would like to be able to tag certain mediation points (e.g., downgraders or encryptors) as "allowed" sources of information flow.

2.1 Shared Buffer Simulink Model

A Simulink model of the shared buffer example is shown in Fig. 2. The inputs to the model are shown on the left: we have the requests to use the buffer from the four processes (the secret input/output process and the unclassified input/output processes) as well as the input buffer data from the secret and unclassified input processes. The scheduler subsystem determines access to the buffer, while the buffer subsystem uses the scheduler state to determine which process writes to the shared buffer.

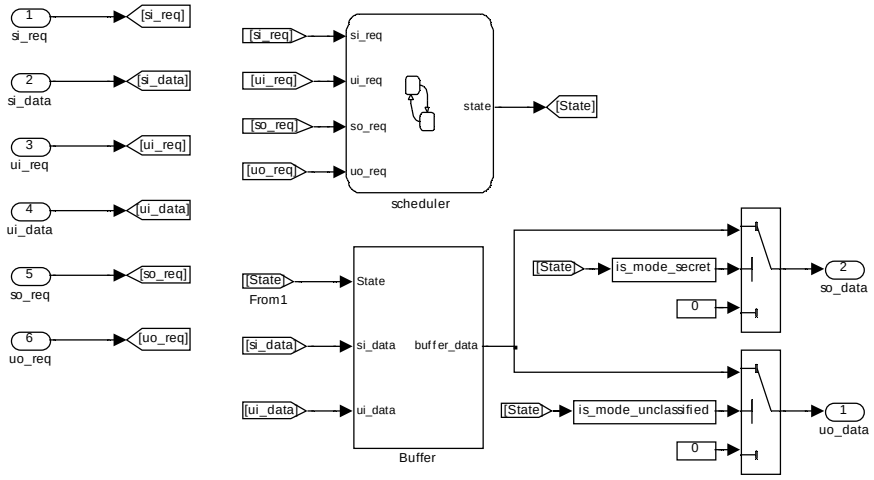

Fig. 2 Shared Buffer Example in Simulink.

The information flow analysis is performed in terms of a set of *principal variables*. These variables are the variables that we are interested in tracking through the model. We always track the input variables to the model, and we sometimes track computed variables internal to the model. To perform the analysis, the Simulink model is annotated to add the principal variables as shown in Fig. 3.

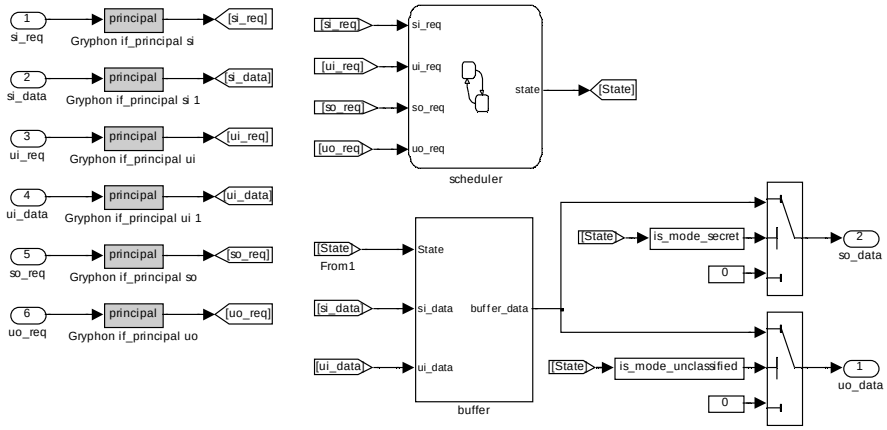

Fig. 3 Annotated Simulink Model.

Once we have annotated the model, we use the *Gryphon* tool set [24] to automatically construct an information flow model that can be model checked on a variety of

model checking tools including NuSMV [8], SAL [23], and Prover [16]. The analysis process extends the original model with a *flow model* that operates over sets of principal variables. Each computed variable in the original model has a *flow variable* in the flow model that tracks its dependencies in terms of the principal variables.

For model checking, sets of principal variables are encoded as bit sets, and checking whether information flow is possible is the same as determining whether it is possible that one of the principal bits is set. For the model above, the translation generates the following bit set for the principals:

```
Principal bit vector: {
   si maps to bit: 0,
   so maps to bit: 1,
   ui maps to bit: 2,
   uo maps to bit: 3 }
```

Now we can write properties over output variables. For example, suppose we want to show that the secret output data is unaffected by the unclassified input or output principal. In this case, we could write:

```
LTLSPEC G(!(gry_IF_so_data[ui_idx] |
gry_IF_so_data[uo_idx]));
```

gry_IF is the prefix used for the flow variables, so the analysis checks whether there is flow to the *so_data* output from the *ui* principal or the *uo* principal. These principals correspond to flow from the *ui_req*, *ui_data*, and *uo_req* input variables.

As described earlier, this property is violated, because there is information flow from the unclassified processes to the secret output through the scheduler. NuSMV generates a counterexample that we can examine to determine how the information leak occurred.

After analyzing the problem, we decide that the flow of information through the scheduler state is allowable. We would now like to search for additional sources of flow. By adding an additional principal for the scheduler state, as shown in Fig. 4, we can ignore the flows from the *ui* and *uo* principals that occur through the scheduler. After re-running the analysis, the model checker finds no other sources of information flow.

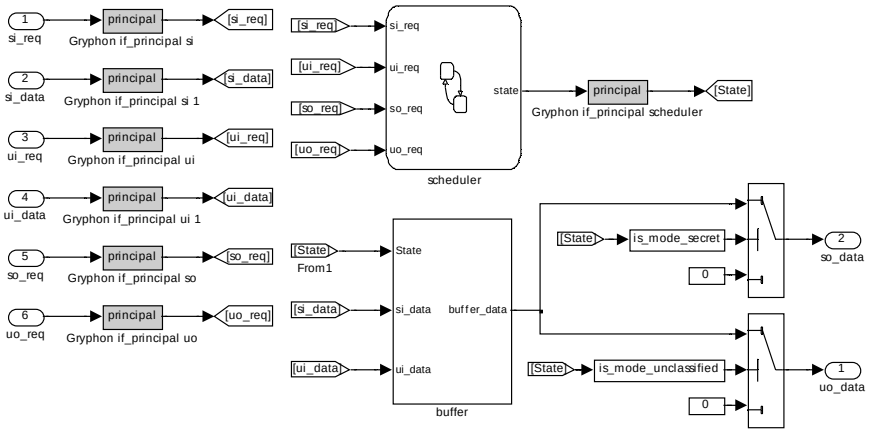

Fig. 4 Annotated Simulink Model with Intransitive Flow.

3 Information Flow Modeling for Synchronous Dataflow Languages

Languages such as Simulink [11] and SCADE [4] are examples of *synchronous dataflow languages.* The languages are *synchronous* because computation proceeds in a sequence of discrete instants. In each instant, inputs are perceived and states and outputs are computed. From the perspective of the formal semantics, the computations are instantaneous. The languages are *dataflow* because they can be understood as a system of assignment equations, where an assignment can be computed as soon as the equations on which it is dependent are computed. The equations can either be represented textually or graphically. As an example, consider a system that computes the values of two variables, X and Y, based on 4 inputs: a, b, c, and d, shown in Fig. 2:

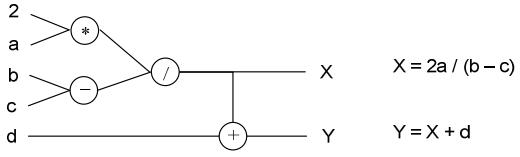

Fig. 5 Graphical and textual presentation of a set of equations.

The variables (often referred to as *signals*) in a dataflow model are used to label a particular computation graph. Therefore, it is incorrect to view the equations as a set of constraints on the model: a set of equations shown in Fig. 6 is not a valid model because X and Y mutually refer to one another. This is shown in Fig. 6, where the bold lines indicate the cyclic dependencies. Such a system may have no solution or infinitely many solutions, so cannot be directly used as a deterministic program. If viewed as a graph, these sets of equations have *data dependency cycles*, and are considered incorrect.

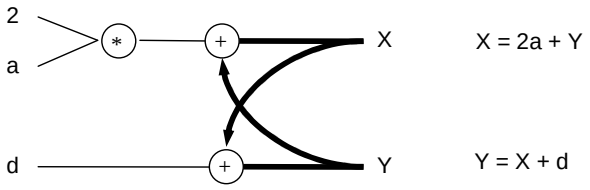

Fig. 6 Cyclic set of equations.

However, in order for the language to be useful, we must be able to have mutual reference between variables. To allow benign cyclic dependencies, we create a step-delay operator (i.e., a latch) using the comma operator. For example: {X = 2a / Y; Y = 1, (X + d))} defines a system where X is equal to 2a divided by the current value of Y, while Y is initially equal to 1, and thereafter equal to the *previous* value of X plus d.

There are several examples of textual dataflow languages, including Lustre [7], Lucid Synchrone [3], and Signal [9] that differ in terms of structuring mechanisms, computational complexity (i.e., whether recursion is allowed), and in terms of *clocks* that define the rates of computation for variables. Our analysis is defined over the Lustre language. Lustre is the kernel language of the SCADE tool suite and also the internal language of the Rockwell Collins Gryphon tool suite. Lustre is also sufficient to model the portions of the Simulink/Stateflow languages that are suitable for hardware/software co-design.

3.1 Modeling Information Flow

When describing information flow, we are often attempting to define a *non-interference* relation of some kind. There have been several formulations of non-interference [5, 20, 21, 19] involving transition systems and process algebras which have focused on non-interference in terms of a trace of actions (inputs) fed into some machine that generates outputs. The idea of non-interference is simple: a security do-

main *u* does not interfere with domain *v* if no action performed by *u* can influence subsequent outputs of *v*.

In the formulation of [20], non-interference is demonstrated by removing actions from the trace T (call it T') and showing that under certain conditions the final output of the machine is the same. However, for synchronous dataflow languages such as Lustre or Simulink, characterizing the "removable" inputs is difficult, as each input variable is assigned a value in each step; one must define predicates over the cross product of the input variables. Characterizing the "action" of a model with potentially tens or hundreds of outputs presents similar difficulties.

Instead, following Greve in the earlier chapter [6], we would like to define a notion of non-interference on individual variables within a model in terms of correspondences between two traces. In our formulation, a trace is a sequence of model states, each state containing the assignments to all variables within the model. We define a set of *principal variables* as a superset of the inputs, and then define an *Interferes* function for any variable *c* that describes the set of principals that could possibly affect the value of *c*. We determine the correctness of the *Interferes* set in terms of trace correspondence. The *Interferes* set is correct if, given any variable *c* and traces π_0 and π_1, if the traces agree on all the variables of *Interferes(c)*, then they will agree on *c*. In other words, the variables in *Interferes(c)* are sufficient to determine the value of *c* at any step. Equivalently, any principal variable outside the *Interferes* set cannot affect the value of *c*.

Formalized in the PVS notation [22], the theorem that we are proving is as follows:

```
InterferenceTheorem: LEMMA
 FORALL (p: Program, gt1:gtrace, st1,st2:strace):
    FORALL (idx:index):
      WFp(p) & St(p,st1) & St(p,st2) &
      IFt(p,st1,gt1) &
      vtraceEquivSet(DepSet(idx,gt1),st1,st2) =>
          liftv(idx,st1) = liftv(idx,st2)
```

This theorem states that if two traces are equivalent (*vtraceEquivSet*) on the dependencies computed for a variable *idx* by our *Interferes* set (*DepSet*(*idx*,*gt1*)), then two traces agree on the value of *idx*. The details of the theorem and steps in the proof will be explained in the following sections.

How this is used in practice is that the user suggests what is believed to be a non-interfering principal variable for some variable *c* and a model checker is used to determine whether or not this variable interferes with (i.e., affects) *c*.

3.2 Using PVS

PVS [22, 15] is a mechanized theorem prover based on classical, typed higher-order-logic. Specifications are organized into (potentially parameterized) theories, which are collections of type and function definitions, assumptions, axioms and theorems. The proof language of PVS is composed of a variety of primitive inference procedures that may be combined to construct more powerful proof strategies.

Normally in PVS the proof process is performed interactively, and the proof script encoding the entire proof is not visible to the user. In our development, we used the *ProofLite* [14] extension to PVS in order to embed the proofs as comments into the PVS theories. To make the theories shorter and easier to understand, we omit the ProofLite scripts in this chapter. However, the interested reader is encouraged to visit the Springer web site to view and run the scripts.

3.3 Traces and Processes

The semantics of synchronous dataflow languages are usually defined in terms of *traces* that describe the behavior of the system over time. These traces are formalized in the language of the PVS theorem prover in Fig. 7. We are interested in two kinds of traces. First, we are interested in the trace of values produced by the execution of the system. We define the set of values that can be assigned to variables using the opaque type *vtype*[1]. The execution traces are mappings from instants in time to states, where states map variables to values, and are defined by the *strace* and *state* types, respectively. The variables in our model correspond to indices in Greve's formulation, and we use the term *index* to identify a variable in a trace.

Second, we are interested in tracing the dependencies of a variable in terms of a set of other variables (in GWV terms, the information flow graph). These traces map instants in time to graph states, where each graph maps an index (i.e., variable) to sets of indices. At each instant, for a given variable *v* the graph captures a set of variables that are necessary for computing *v*. These traces are defined by the *gtrace* and *graphState* types, respectively.

Note that our states are defined over an infinite set of variables *nat*. In a real system, we would have a finite set, but this can be modeled by simply ignoring all variables above some maximum index. This change does not affect the formalization or the proofs.

---

[1] Opaque types in PVS allow one to define a type as an unspecified set of values.

```
Traces: THEORY
BEGIN

  index: TYPE = nat
  time: TYPE = nat
  vtype: TYPE+

  state : TYPE = [ index -> vtype ]
  strace: TYPE = [ time -> state ]

  get(i: index, s: state): vtype = s(i)

  graphState: TYPE = [ index -> set[index] ]
  gtrace: TYPE = [ time -> graphState ]
END Traces
```

Fig. 7 Traces Theory.

Next, we define *processes* that constrain the traces in Fig. 8. The processes are built from expressions: an (unspecified) set of unary and binary operators, constant, variable, and conditional (if/then/else) expressions. We next partition the indices into gates, latches, and inputs. Gates are computed from the current values of other variables, while latches are computed from the previous values of other variables. Latches also have an initial value which is their value in the first step of a trace. Inputs are not computed and assumed to be externally provided.

The processes described in Fig. 8 define a simple *synchronous dataflow language*, such as Simulink or SCADE. For the purposes of this discussion, the structuring mechanisms of these languages (nodes and subsystems) as well as the clocking mechanisms for variables can be thought of as syntactic sugar.

```
ProcessExprTypes: THEORY
BEGIN
  IMPORTING Traces

  BopType: TYPE+
  UopType: TYPE+

  BopEx(Bop: BopType, v1,v2: vtype): vtype
  UopEx(Uop: UopType, v0: vtype): vtype
  isTrue(v0: vtype): bool
END ProcessExprTypes
```

```
ProcessExpr: DATATYPE
BEGIN
  IMPORTING ProcessExprTypes

  Constant(value : vtype): Constant?
  Variable(name  : index):  Variable?
  ITE(test: ProcessExpr, thn: ProcessExpr,
      els: ProcessExpr): ite?
  Bop(OpB: BopType, a1: ProcessExpr,
      a2: ProcessExpr): Bop?
  Uop(OpU: UopType, a0: ProcessExpr): Uop?
END ProcessExpr

ProcessAssignment: DATATYPE
BEGIN
  IMPORTING ProcessExpr
  Gate (gexpr: ProcessExpr): Gate?
  Latch(v0: vtype, lexpr: ProcessExpr): Latch?
  Input: Input?
END ProcessAssignment

Program: THEORY
BEGIN
  IMPORTING ProcessAssignment
  IMPORTING IndexSet[index]

  Program: TYPE = [ index -> ProcessAssignment ]

  StatesP(p: Program): set[index] =
    (LAMBDA (v: index): Latch?(p(v)))

  InputsP(p: Program): set[index] =
    (LAMBDA (v: index): Input?(p(v)))

  GatesP(p: Program): set[index] =
    (LAMBDA (v: index): Gate?(p(v)))

  De(e: ProcessExpr): RECURSIVE set[index] =
    CASES e OF
      Constant(value): Empty,
      Variable(name):  singleton(name),
```

```
      ITE(test,thn,els): De(test) + De(thn) +
        De(els),
      Bop(OpB,a1,a2):  De(a1) + De(a2),
      Uop(OpU,a0): De(a0)
    ENDCASES
  MEASURE e by <<

  belowSet(n: nat, s: set[nat]): bool =
    FORALL (i: nat): member(i,s) => (i < n)

  Ae(v: index, a: ProcessAssignment): ProcessExpr =
    CASES a OF
      Gate (gexpr)   : gexpr,
      Latch(v0,lexpr): lexpr,
      Input          : Variable(v)
    ENDCASES;

  WFp(p: Program) : bool =
    FORALL (v: index):
      belowSet(v, De(Ae(v,p(v))) & GatesP(p))

  WFPrograms : TYPE = { p : Program | WFp(p) }

END Program
```

Fig. 8 Processes and Programs.

In general, a set of simultaneous equations may yield zero or multiple solutions. We want a program to be *functional*, given a particular input trace. In order to ensure that the assignments yield functional traces, we need a strict ordering on gate assignments. Since indices are defined as naturals, it suffices to define an ordering such that the assignment expression for a variable may only refer to gate indices that are strictly smaller than the index being assigned. Note that *only* gate indices are restricted – it is possible to write benign cyclic dependencies involving latches.

The *Ae* function returns the assignment expression associated with a particular index. For inputs, *Ae* just returns a variable expression referring to the input. The *De* predicate defines the dependencies of an expression and *WFp* defines the functional well-formedness constraint on programs. Note that this predicate also forms a basis for inducting over the gates within the program that we will use for several of the proofs.

3.4 Semantic Rule Conventions

We define different kinds of semantics for the values produced by a program and also for information flow. The semantic functions introduced follow a naming convention to make them easier to follow and to relate to one another. The form of the semantics functions is as follows:

<TYPE><syntax><OPTIONAL RESTRICTION>

For example, the *Se* function defines the value-semantic function for expressions, and the *IFsG* function defines the information-flow function for states with respect to gates.

The <TYPE>s of semantics that will be used in the following discussion are as follows:

S: value semantics for traces
D: syntactic dependencies
DS: dependencies based on syntax and current state
IF: information flow dependencies

The <syntax>es that will be discussed are the following:

e: expressions
i: indices (assignments)
s: states
t: traces

The <OPTIONAL RESTRICTION>s restrict the semantic functions at a particular syntactic level to:

I: Inputs
G: Gates
L: Latches

3.5 Value Trace Semantics

We next create semantic functions for the expressions and programs in Fig. 8. Following [1] and [12] the semantics are defined in terms of *trace conformance*, as shown in Fig. 9. We state that a trace conforms to a program if the values computed by the assignment expressions for the gates and latches correspond to the values in the trace. The *Se* function computes a value from a Process expression. The *SsG* predicate checks conformance between the gate assignments and a state, and the *SsL* predicates check conformance between the latch assignments and the trace. The *St* predicate defines trace conformance over both gates and latches.

```
ProcessSemantics: THEORY
BEGIN
  IMPORTING Program

  Se(e: ProcessExpr, s: state): RECURSIVE vtype =
    CASES e OF
      Constant(value): value,
      Variable(name): s(name),
      ITE(test,thn,els):
        IF isTrue(Se(test,s)) THEN Se(thn,s)
        ELSE Se(els,s) ENDIF,
      Bop(OpB,a1,a2):  BopEx(OpB,Se(a1,s),Se(a2,s)),
      Uop(OpU,a0): UopEx(OpU,Se(a0,s))
    ENDCASES
  MEASURE e by <<

  Si(p: Program)(i: index, s0: state): vtype =
    CASES p(i) OF
      Gate (gexpr)    : s0(i),
      Latch(v0,lexpr) : Se(lexpr,s0),
      Input           : s0(i)
    ENDCASES

  SsG(p: Program, s0: state): bool =
    FORALL (v: index): Gate?(p(v)) =>
      (s0(v) = Se(Ae(v,p(v)),s0))

  SsL0(p: Program, s0: state): bool =
    FORALL (v: index): Latch?(p(v)) =>
      (s0(v) = v0(p(v)))

  SsLn(p: Program, s0,s1: state): bool =
    FORALL (v: index): Latch?(p(v)) =>
      (get(v,s1) = Si(p)(v,s0))

  St(p: Program, st: strace): bool =
    FORALL (n: nat):
      IF (n = 0) THEN
        SsL0(p,st(0)) & SsG(p,st(0))
      ELSE
        SsLn(p,st(n-1),st(n)) & SsG(p,st(n))
      ENDIF
```

```
END ProcessSemantics
```

Fig. 9 Process Trace Semantics.

3.6 Creating an Accurate Model of Information Flow

Now we can create a semantics that tracks information flow through the model, shown in Fig. 10. This semantics maps indices to the set of indices used when computing the value of the index. For expressions, we create two different semantics; the first tracks the indices that are immediately used within the computation of the expression; the second traces the indices back to *principal variables*, which are the actual concern of the information flow analysis. For the moment, we consider the inputs as the principal variables. We expand this notion when we talk about *intransitive interference* in Section 6.

```
ProcessIndexSets: THEORY
BEGIN
  IMPORTING ProcessSemantics
  IMPORTING MemberRules[index]

  DSe(e: ProcessExpr, s0: state):
      RECURSIVE set[index] =
    CASES e OF
      Constant(value): Empty,
      Variable(name):  singleton(name),
      ITE(test,thn,els):
        IF isTrue(Se(test,s0)) THEN
          SDe(test,s0) + SDe(thn,s0)
        ELSE
          SDe(test,s0) + SDe(els,s0)
        ENDIF,
      Bop(OpB,a1,a2):  SDe(a1,s0) + SDe(a2,s0),
      Uop(OpU,a0): SDe(a0,s0)
    ENDCASES
  MEASURE e by <<

  IFe(e: ProcessExpr, principal: set[index],
      s0: state, g0: graphState): RECURSIVE
      set[index] =
```

```
    CASES e OF
      Constant(value): Empty,
      Variable(name):
         IF principal(name) THEN
            singleton(name)
         ELSE
            g0(name)
         ENDIF,
      ITE(test,thn,els):
        IF isTrue(Se(test,s0)) THEN
          IFe(test,principal,s0,g0) +
          IFe(thn,principal,s0,g0)
        ELSE
          IFe(test,principal,s0,g0) +
          IFe(els,principal,s0,g0)
        ENDIF,
      Bop(OpB,a1,a2):  IFe(a1,principal,s0,g0) +
         IFe(a2,principal,s0,g0),
      Uop(OpU,a0): IFe(a0,principal,s0,g0)
    ENDCASES
  MEASURE e by <<

  IFsI(p: Program, s0: state, g0: graphState): bool=
    FORALL (v: index): Input?(p(v)) =>
      (g0(v) = IFe(Ae(v,p(v)),InputsP(p),s0,g0))

  IFtI(p: Program, st: strace, gt: gtrace): bool =
    FORALL (t: time) : IFsI(p, st(t), gt(t))

  IFsG(p: Program, s: state, g: graphState): bool =
    FORALL (v: index): Gate?(p(v)) =>
      (g(v) = IFe(Ae(v,p(v)),InputsP(p),s,g))

  IFtG(p: Program, st: strace, gt: gtrace): bool =
    FORALL (t: time) : IFsG(p, st(t), gt(t))

  IFsL0(p: Program, g0: graphState): bool =
    FORALL (v: index):
      Latch?(p(v)) => g0(v) = Empty

  IFsLn(p: Program, s0: state,
        g0,g1: graphState): bool =
    FORALL (v: index): Latch?(p(v)) =>
```

```
      (g1(v) = IFe(Ae(v,p(v)),InputsP(p),s0,g0))

  IFtL(p: Program, st: strace, gt: gtrace): bool =
    FORALL (n: nat):
      IF (n = 0) THEN
        IFsL0(p,gt(0))
      ELSE
        IFsLn(p,st(n-1),gt(n-1),gt(n))
      ENDIF

  IFt(p: Program, st: strace, gt: gtrace): bool =
    IFtG(p,st,gt) & IFtL(p,st,gt) & IFtI(p,st,gt)

  tracePair : TYPE = [# s: strace, g: gtrace #];

  tp_ok(p: Program, tp: tracePair) : bool =
     IFt(p, s(tp), g(tp)) AND St(p, s(tp)) ;
```

Fig. 10 Process Index Semantics.

The only difference between the *DSe* and *IFe* semantics in Fig. 10 is in the behavior of the Variable branch. For the *IFe* semantics, a set of *principal variables* are provided. If a referenced variable is a principal variable, then we return it as a dependency; if it is not, then we return the dependencies of that variable. The effect of this rule is to backchain through the intermediate variables so that dependencies are always a subset of the principal variables. The *DSe* semantics, on the other hand, return the immediate dependencies (i.e., the indices of all variables referenced in the assignment expression).

Note that both the *DSe* and *IFe* semantics are state-dependent: For if/then/else expressions, the set of dependencies depends on the if-test; only dependencies for the used branch are returned. This feature allows conditional dependencies to be tracked within the model.

After defining the expression semantics we define the IF semantics on states and programs, matching the structure of the *S* definitions in Fig. 9. At the bottom of Fig. 10, we define trace pairs as a type and define trace pair conformance to a program based on both semantics.

3.7 PVS Proof of Trace Equivalence (InterferenceTheorem)

We can now state the interference theorem that should be proven over the trace pairs. Informally, we'd like to state that for a particular index *idx*, if the inputs referenced in an information flow trace for *idx* (*DepSet*) have the same values in two state traces (*vtraceEquivSet*), then the two traces will have the same values for *idx*. Formally, this obligation is expressed in Fig. 11. Note that there is an asymmetry in the interference theorem: we define two execution traces (*st1* and *st2*) but only one graph trace (*gt1*). The graph trace (*gt1*) corresponding to an execution trace (*st1*) for a given index *idx* characterizes the signals that must match for any other execution trace (in this case *st2*) to match *st1* for signal *idx*. It is equivalent to use a graph trace based on *st2*.

```
vtrace: TYPE = [ time -> vtype ]

liftv(i: index, st: strace): vtrace =
  (LAMBDA (t: time): st(t)(i))

vtraceEquivSet(set: set[index],st1,st2: strace):
  bool =
  FORALL (i: index): member(i,set) =>
    liftv(i,st1) = liftv(i,st2)

DepSet(x: index, gt: gtrace): set[index] =
  (lambda (i: index):
    (EXISTS (t: time): member(i,gt(t)(x))))

InterferenceTheorem: LEMMA
 FORALL (p: Program, gt1:gtrace, st1,st2:strace):
    FORALL (idx:index):
      WFp(p) & St(p,st1) & St(p,st2) &
      IFt(p,st1,gt1) &
      vtraceEquivSet(DepSet(idx,gt1),st1,st2) =>
          liftv(idx,st1) = liftv(idx,st2)
```

Fig. 11 Interference Theorem.

To prove this theorem, we have to build a hierarchy of equivalences shown in Fig. 12. This graph does not show all of the connections between proofs (e.g., which theorems are instantiated in the proofs of other theorems), but it provides a good overview of the structure of the proof. Ultimately, we are interested in proving the final theorem, which defines a relationship between traces as described by the information flow semantics *IF* and the value semantics *S*. In order to prove this theorem, we define an

intermediate flow semantics based on state dependencies (*DS*). Whereas the information flow semantics unwinds the dependencies from outputs to inputs implicitly through the use of the graph state and graph trace, the *DS* flow semantics unwind the graph explicitly and therefore provide an easier basis for inductive proof.

The "rows" of the proof graph correspond to a level in the evaluation hierarchy. Reading from top to bottom, we talk about equivalences in terms of expressions, then in terms of indices (assignments), then states, and finally, traces. The "columns" correspond to the different semantics. On the left is the information flow (*IF*) semantics, in the middle is the *DS* semantics, and on the right is the value (*S*) semantics. One semantics bridges the *IF* and *DS* semantics (*DSiIF*).

There are two different kinds of theorems that are proved between the semantics. The first are equivalences between the different flow semantics (e.g., that two flow semantics yield the same set of dependencies). The second are GWV-style theorems, in the same style as [6]. These state that if the values of the dependent indices for a piece of syntax $\sum$ are equal within two states or traces *s1* and *s2*, then the value produced by evaluating $\sum$ over *s1* and *s2* will be equal.

In our analysis, we prove GWVr1-style theorems. GWVr1 is less expressive than GWVr2 but it is simpler to formulate. The additional expressive power in GWVr2 is necessary to describe dynamic memory, but the synchronous models that we analyze in this chapter do not use dynamic memory, so GWVr1 is sufficiently expressive for our purposes. The connection between the formulation in this chapter and [6] is explored further in Section 7.

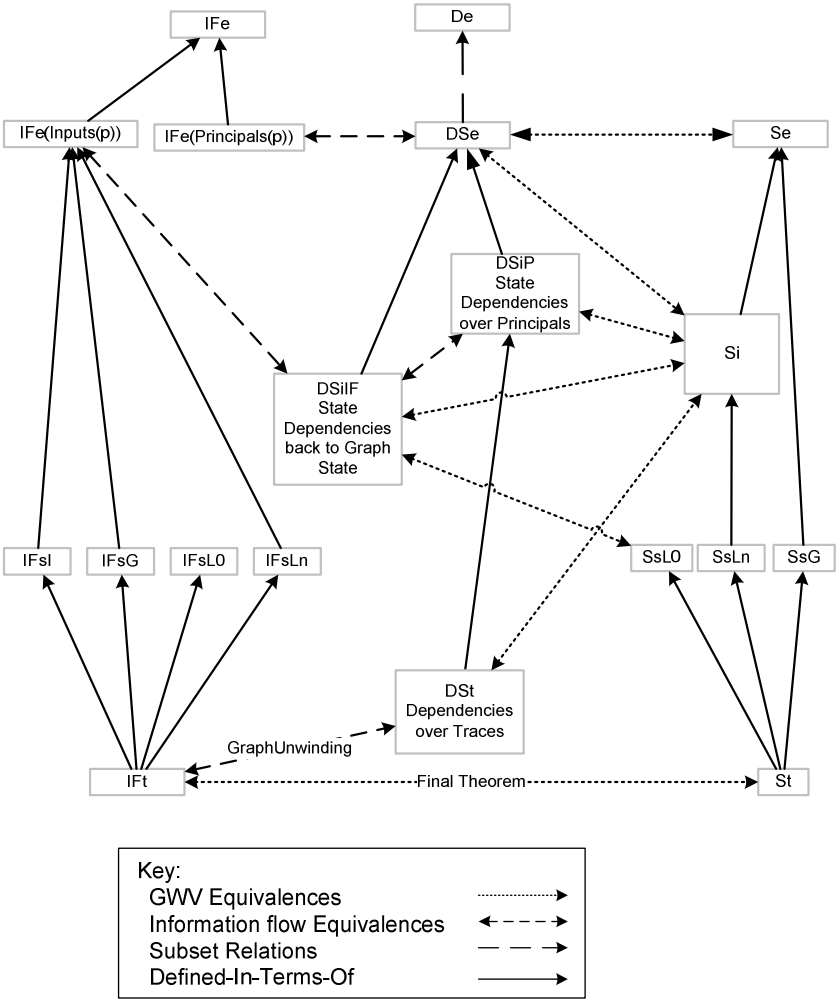

Fig. 12 Proof Graph for final theorem.

Expression Equivalence Theorems

In Fig. 13, we begin the process of proving the final theorem by describing some lemmas over expressions. These will form the basis of the later proofs over larger pieces of syntax.

```
DSe_subset_De: LEMMA
    FORALL (e: ProcessExpr, s0: state):
      subset?(DSe(e,s0),De(e))


  Compose(uset: set[index], g0: graphState):
      set[index] =
    (lambda (z: index):
      (EXISTS (m: index):
        member(m,uset) & member(z,g0(m))))

  member_Compose: LEMMA
    FORALL (i: index, uset: set[index],
            g0: graphState):
      member(i,Compose(uset,g0)) =
        (EXISTS (m: index):
          member(m,uset) & member(i,g0(m)))

  IFe_to_DSe_Property(e: ProcessExpr): bool =
    FORALL (principal: set[index], s0: state,
            g0: graphState):
      IFe(e,principal,s0,g0) =
        LET uset: set[index] = DSe(e,s0) IN
          (uset & principal) +
          Compose(uset & (not(principal)),g0)

  IFe_to_DSe_proof: LEMMA
    FORALL (e: ProcessExpr): IFe_to_DSe_Property(e)

  IFe_to_DSe: LEMMA
    FORALL (e: ProcessExpr, principal: set[index],
            s0: state, g0: graphState):
      IFe(e,principal,s0,g0) =
        LET uset: set[index] = DSe(e,s0) IN
          (uset & principal) +
          Compose(uset & (not(principal)),g0)
```

Fig. 13 Expression Equivalence Proofs.

The *DSe_subset_De* lemma states that the state-aware dependency function (*DSe*) returns a subset of the indices referenced by the syntactic dependency function (*De*).

We appeal to this lemma (through another lemma: *WFg_to_WFgDSe*) to establish a basis for induction for some of the proofs involving equivalence of gate assignments.

The *Compose* function is used to look up each of the entries in a set in the graph state. It performs the same function as the Direct Interaction Allowed (*DIA*) function in Greve's formulation [6]. It is used to map from a set of immediate dependencies to their dependencies.

The *IFe_to_DSe_Property* lemma defines the first mapping between the state-based *DS* dependency semantics and the *gtrace*-based *IF* dependency semantics. Remember from Section 3.4 that the *IFe* semantics are defined in terms of a set of *principals*: if a variable is principal, then we look up its dependencies in the graph state. This property creates an equivalence between these semantics by looking up (via *Compose*) the non-principal variables from the *DSe* semantics.

Program Well-Formedness Theorems

In Fig. 14, we define a bridge between the program well-formedness constraint *WFp* and state dependencies (*DSe*). This bridge will allow us to use the *WFp* predicate in reasoning about GWV equivalences involving state dependencies. We define a *WFgDSe* predicate that defines well-formedness in terms of the *DSe*, and show that *WFp* implies the (more accurate) *WFgDSe* predicate.

```
Principals(p: Program): set[index] =
  StatesP(p) + InputsP(p)

WFg(p: Program): bool =
  FORALL (v: index):
    belowSet(v,De(Ae(v,p(v))) - Principals(p))

Principals_Gates_partition : LEMMA
  FORALL (p: Program):
     (GatesP(p) = complement(Principals(p)))

Principals_Gates_subset_equiv : LEMMA
  (FORALL (s: set[index], p: Program) :
    (s & GatesP(p)) = (s - Principals(p)))

WFp_to_WFg : LEMMA
  FORALL (p: Program) :  (WFp(p) = WFg(p))

WFgDSe(p: Program, s0: state): bool =
```

```
    FORALL (v: index):
      belowSet(v,DSe(Ae(v,p(v)),s0) - Principals(p))

  WFg_to_WFgDSe: LEMMA
    FORALL (p: Program, s0: state):
      WFg(p) => WFgDSe(p,s0)

END ProcessIndexSets
```

Fig. 14 Well-Formedness Predicates for Programs.

GWV Equivalence Theorems

Now, we can start proving GWV-style equivalence properties. These state that if the values of the dependent indices for a piece of syntax $\sum$ match within two states or traces *s1* and *s2*, then the value produced by the evaluating $\sum$ over *s1* and *s2* will match. The idea is that we will start from the *immediate* dependencies of an expression and progressively unwind the dependencies toward the inputs. This unwinding occurs in two stages:

- First we unwind to the *principals*, which (for the purposes of the proof) are the states and inputs. Another way of looking at this first unwinding is unwinding back to the "beginning" of the step. This is the definition of the *DSiP* dependencies
- Next, we unwind the dependencies back to the inputs by examining the graph trace over time. This is the definition of the *DSt* dependencies.

We also map these state-based equivalences that are computed via explicit unwindings of dependencies to the *IF* equivalences, which implicitly unwind the dependencies using the graph states. This is accomplished by using the *DSiIF* dependency relation. This will be the key lemma to show the equivalence of the *IF* and *DS* formulations.

Fig. 15 shows the dependency proof for the *DSe* dependencies. There are two equivalences: the first over evaluation of expressions, and the second over evaluation of indices.

```
ProcessInterference: THEORY
BEGIN
  IMPORTING ProcessIndexSets
  IMPORTING GWV_EquivSetRules[index,state,vtype,get]
```

```
StateEquivSet(s:set[index],s1,s2: state): bool =
  equivSet(s,s1,s2)

GWVr1_Se_DSe: LEMMA
  FORALL (e: ProcessExpr):
    FORALL (in1, in2: state):
      StateEquivSet(DSe(e, in1), in1, in2) =>
        (Se(e, in1) = Se(e, in2))

GWVr1_Si_DSe: LEMMA
  FORALL (p: Program):
    FORALL (i: index, in1, in2: state):
      SsG(p,in1) & SsG(p,in2) &
        StateEquivSet(DSe(Ae(i,p(i)), in1),
                      in1, in2) =>
          Si(p)(i,in1) = Si(p)(i,in2)
```

Fig. 15 GWVr1 for DSe Dependencies.

Figure 16 shows the proofs for the next level of unwinding: showing that if the *principal* variables are the same for two states, then the results produced for an index will be the same. This step removes the gates from the dependency calculation.

```
DSiP(p: Program,s0: state)(x: index) :
RECURSIVE set[index] =
  LET uset: set[index] = DSe(Ae(x,p(x)),s0) IN
  LET pri : set[index] = Principals(p) IN
    (uset & pri) +
      (lambda (z: index):
        (EXISTS (m: index):
          m < x &
          member(m,uset & not(pri)) &
          member(z,DSiP(p,s0)(m))))
MEASURE x

DSiP_contains_only_Principals: LEMMA
  FORALL (x: index, p: Program, s0: state):
    subset?(DSiP(p,s0)(x),Principals(p))

DSiP_def: LEMMA
  FORALL (p: Program,s0: state,x: index):
    WFg(p) =>
```

```
        DSiP(p,s0)(x) =
          LET uset: set[index] = DSe(Ae(x,p(x)),s0)
          IN
          LET pri : set[index] = Principals(p)
          IN
            (uset & pri) +
            Compose(uset & not(pri),DSiP(p,s0))

  GWVr1_Si_DSiP: LEMMA
    FORALL (p: Program):
      FORALL (i: index, s1,s2: state):
        WFg(p) & SsG(p,s1) & SsG(p,s2) &
          StateEquivSet(DSiP(p,s1)(i),s1,s2) =>
            Si(p)(i,s1) = Si(p)(i,s2)
```

Fig. 16 GWVr1 for Principal dependencies.

Fig. 17 shows the proofs of the next level of unwinding, to the dependencies of the states. The definition of the *DSiIF* predicate is particularly important as it bridges between the graph-trace-based *IF* semantics and the state-based *DS* semantics. Like the *DSiP* semantics, it backtraces through the gates to reach dependencies based on states and inputs. The distinction is that it then looks up the state dependencies in the graph state. This means that the dependencies computed by *DSiIF* will match the dependencies computed by the IF relation, as demonstrated by the *IFe_to_DSiIF* lemma. This is a key lemma in proving the unwinding theorem over state dependency traces *DSt* and information flow traces *IFt*.

```
GProgram: TYPE = { p : Program | WFg(p) }

DSiIF(p: GProgram, s0: state, g0: graphState)
    (x: index): RECURSIVE set[index] =
  LET uset : set[index] = DSe(Ae(x,p(x)),s0) IN
  LET ins  : set[index] = InputsP(p) IN
  LET dff  : set[index] = StatesP(p) IN
  LET gates : set[index] = GatesP(p)  IN
    (uset & ins) +
      Compose(uset & dff, g0) +
        (lambda (z: index):
      (EXISTS (m: index):
        member(m,uset & gates) &
        member(z,DSiIF(p,s0,g0)(m))))
MEASURE x

DSiIF_to_DSiP: LEMMA
  FORALL (p: Program, s0: state, g0: graphState):
    WFg(p) =>
      FORALL (x: index):
        DSiIF(p,s0,g0)(x) =
          (InputsP(p) & DSiP(p,s0)(x)) +
            Compose(StatesP(p) & DSiP(p,s0)(x),g0)
```

Fig. 17 GWVr1 for State-Input dependencies.

Finally, In Fig. 18, we map dependencies to inputs across a multistep trace. First, we prove a lemma that is sufficient for the proof of latch assignment at step zero (*GWVr1_Si_SsL0*). This lemma will be used to provide the base case for latches in the *GWVr1_Si_DSt* proof.

```
GWVr1_Si_SsL0: LEMMA
  FORALL (p: Program):
    FORALL (i: index, s1,s2: state):
      WFg(p) & SsL0(p,s1) & SsL0(p,s2) &
      SsG(p,s1) & SsG(p,s2) &
      StateEquivSet(InputsP(p) &
      DSiP(p,s1)(i),s1,s2) =>
          Si(p)(i,s1) = Si(p)(i,s2)

DSt(p: Program, st: strace, t: time)(i: index):
RECURSIVE set[index] =
```

```
    IF (t = 0) THEN
      InputsP(p) & DSiP(p,st(t))(i)
    ELSE
      LET uset: set[index] = DSiP(p,st(t))(i) IN
        (uset & InputsP(p)) +
        Compose(not(InputsP(p)) & uset,
                DSt(p,st,t - 1))
    ENDIF
  MEASURE t

  subset_Compose: LEMMA
    FORALL (a: index, x: set[index], g: graphState):
      member(a,x) => subset?(g(a),Compose(x,g))

  vtrace: TYPE = [ time -> vtype ]

  vtrace_extensionality: LEMMA
    FORALL (i: index, s1,s2: vtrace):
      (s1 = s2) =
        FORALL (t: time): s1(t) = s2(t)

  AUTO_REWRITE+ vtrace_extensionality

  liftv(i: index, st: strace): vtrace =
   (LAMBDA (t: time): st(t)(i))

  vtraceEquivSet(set: set[index],st1,st2: strace):
  bool =
    FORALL (i: index): member(i,set) =>
      liftv(i,st1) = liftv(i,st2)

  GWVr1_Si_DSt: LEMMA
    FORALL (p: Program, st1,st2: strace):
      FORALL (t: time, i:index):
        WFg(p) & St(p,st1) & St(p,st2) &
          vtraceEquivSet(DSt(p,st1,t)(i),st1,st2) =>
            Si(p)(i,st1(t)) = Si(p)(i,st2(t))
```

Fig. 18 GWVr1 theorems for trace dependencies.

Next, in Fig. 19, we have to define a graph unwinding theorem, which maps between our state-dependency-based formulation *DSt* and our graph-dependency-based formulation *IFt*. This is performed in two steps. First, we show that the *DSiIF* formu-

lation matches the result returned by *IFe*. Next, we define the unwinding theorem which demonstrates that *DSt* and *IFt* yield the same dependencies.

```
IFe_to_DSiIF: LEMMA
  FORALL (p: GProgram, s0: state, g0: graphState):
    IFsG(p,s0,g0) & WFg(p) =>
      FORALL (x: index):
        IFe(Ae(x,p(x)),InputsP(p),s0,g0) =
          DSiIF(p,s0,g0)(x)

Graph_Unwinding: LEMMA
  FORALL (p: Program, st: strace, gt: gtrace):
    FORALL (t: time, v: index):
      WFg(p) & IFt(p,st,gt) =>
        IFe(Ae(v,p(v)),InputsP(p),st(t),gt(t)) =
        DSt(p,st,t)(v)
```

Fig. 19 The Graph Unwinding Theorem demonstrating equivalence between *IFt* and *DSt* semantics.

Proof of InterferenceTheorem

Now we have finally assembled the pieces necessary to prove the trace theorem that was proposed in Fig. 8 in Section 3.7. The proof is shown in Fig. 20. We state that the information flow *characterizes* the execution of a model if it satisfies the *InterferenceTheorem.*

```
DepSet(x: index, gt: gtrace): set[index] =
  (lambda (i: index): (EXISTS (t: time):
   member(i,gt(t)(x))))
```

```
InterferenceTheorem: LEMMA
 FORALL (p: Program, gt: gtrace, st1,st2: strace):
    FORALL (i:index):
      WFp(p) & St(p,st1) & St(p,st2) &
      IFt(p,st1,gt) &
        vtraceEquivSet(DepSet(i,gt),st1,st2) =>
          liftv(i,st1) = liftv(i,st2)
```

Fig. 20 Proof of the InterferenceTheorem.

4 Interference to Noninterference

A nearly immediate corollary of the interference theorem is a non-interference theorem, shown in Fig. 21. If a variable *unclass* does not depend on a variable *secret* in any legal trace of the system (as defined by *tp_ok*), then we say that *secret* does not interfere with *unclass.* This is demonstrated by the *Non_Interference* lemma; in this lemma, we state that any two traces whose inputs differ only by *secret* will yield the same values for *unclass.*

```
ProcessNonInterference: THEORY
BEGIN
  IMPORTING ProcessInterference

  Never_Interferes(p: Program, secret: index,
                   unclass: index) : bool =
    FORALL (x: tracePair):
      tp_ok(p, x) =>
        (FORALL (t: time):
          not(member(secret, g(x)(t)(unclass))))

  Inputs_Match_Except_Secret(p: Program,
           st1, st2: strace, secret: index) : bool =
    FORALL (t: time, idx: index):
      ((member(idx, InputsP(p)) AND
       (idx /= secret)) =>
         st1(t)(idx) = st2(t)(idx))
```

```
  Non_Interference : LEMMA
    FORALL (p: Program, secret: index,
            unclass: index, st1,st2: strace):
      (member(secret, InputsP(p)) &
       WFg(p) & St(p, st1) & St(p, st2) &
       Never_Interferes(p, secret, unclass)) &
       Inputs_Match_Except_Secret(p, st1, st2,
            secret)
       =>
         liftv(unclass, st1) = liftv(unclass, st2)

END ProcessNonInterference
```

Fig. 21 Process Non-Interference.

5 Model Checking Information Flow

Up to this point, we have defined formal notions of interference and non-interference over traces for a simple synchronous dataflow language, and shown that an *information flow semantics* can be used to demonstrate noninterference. However, we have not yet proposed a mechanism for computing non-interference relations using the model checker using a temporal logic such as LTL [2].

In order to use a model checker to analyze the notion of non-interference proposed in Section 4, we must do two things. First, we must formalize non-interference in a temporal logic such as LTL that is understood by model checkers. Second, we must encode the model and information flow semantics into the notation of the model checker. The syntax and execution semantics of our language (the *Program* theory in Fig. 7 and *Process* theory in Fig. 8), were chosen in part because they correspond to a subset of the syntax and semantics supported by several popular model checkers including NuSMV [8], SAL [23], and Prover [16]. The translation of the execution model and semantics is therefore immediate.

To support analysis of information flow, however, we have to encode the *IF* semantics in the syntax of the model checker. We call this encoding the *information flow model.* Then we can analyze a *hybrid model*, containing both the original program and the information flow model in order to reason about flow properties.

5.1 Formalizing Noninterference in LTL

We first assume Rushby's formalization of LTL [2] in PVS presented in [6]. We now prove in Fig. 22 that a noninterference assertion over a graph state machine follows from a particular LTL assertion, in the same way as Greve [6].

```
ProcessLTL: THEORY
BEGIN

  IMPORTING ProcessInterference

  GState : TYPE = [# g: graphState, s: state #]

  IMPORTING ltl[GState]

  P : Program
  P_inputs : TYPE = {x: index | Input?(P(x)) }

  split(x: sequence[GState]) : tracePair =
   (# s := LAMBDA (t: time): s(x(t)),
      g := LAMBDA (t: time): g(x(t)) #)

  merge(x: tracePair) : sequence[GState] =
   (LAMBDA (t: time):
     (# s := s(x)(t), g := g(x)(t) #) )

  GSTrace : TYPE =
    { x : sequence[GState] | tp_ok(P, split(x)) }

  Non_Interference(secret: P_inputs, unclass: index)
    (gs: GState) : bool =
      (not (member(secret, g(gs)(unclass))))

  % only consider well-formed models
  reduction: LEMMA
    WFg(P)  =>
      FORALL (secret: P_inputs, unclass: index):
        (FORALL (s: GSTrace):
          (s |=
            G(Holds(
              Non_Interference(secret,unclass)))))
      =>
```

```
        (FORALL (s: tracePair):
          tp_ok(P,s) =>
            (FORALL (t: time):
              (not(member(secret,
                          g(s)(t)(unclass))))))

END ProcessLTL
```

Fig. 22 Connection to LTL.

5.2 Creating the Information Flow Model

Recall that the *IF* semantics correspond to graph traces (*gtrace*) that are composed of a sequence of graph states (*gstate*). Each *gstate* maps program variables to a finite set of *Principal* variables. The information flow semantics from the previous section are then encoded as set manipulations. The information flow model is then the set of assignments to the information flow variables.

The mechanism for creating the information flow variable assignments is a set of transformation rules that are applied to the syntax of *ProcessExpr* and *ProcessAssign* datatypes defined in Fig. 7. The transformation rules generate a slightly richer expression syntax (shown in Fig. 23) that contains two additional variables. The first expression, *IF_Variable*, allows reference variables in the information flow graph state. The second, *SingletonSet*, takes an index and generates a singleton set containing that index.

```
ExprExt: DATATYPE
BEGIN
  IMPORTING ProcessTypes
  Constant(value : vtype): Constant?
  Variable(sname  : index):  Variable?
  ITE(test: ExprExt, thn: ExprExt, els: ExprExt):
      ite?
  Bop(OpB: BopType, a1: ExprExt, a2: ExprExt): Bop?
  Uop(OpU: UopType, a0: ExprExt): Uop?
  IF_Variable(ifname  : index):  IF_Variable?
  SingletonSet(varSet: set[index], prname : index) :
     SingletonSet?
END ExprExt

AssignmentExt: DATATYPE
BEGIN
 IMPORTING ExprExt
  Gate (gexpr: ExprExt): Gate?
  Latch(v0: vtype, lexpr: ExprExt): Latch?
  Input: Input?
END AssignmentExt
```

Fig. 23 Extended Process Syntax.

We can now reflect the information flow semantics into an extended program *ProgramExt* that contains assignments for both the state and graph traces, as shown in Fig. 24.

```
TransformIF : THEORY
BEGIN
  IMPORTING Program, AssignmentExt

  Union : BopType
  EMPTYSET : vtype
  principal_index : [set[index], index -> vtype]

  IDe(e: ProcessExpr): RECURSIVE ExprExt =
  CASES e OF
    Constant(value): Constant(value),
    Variable(name):  Variable(name),
    ITE(test,thn,els):
      ITE(IDe(test), IDe(thn), IDe(els)),
    Bop(OpB,a1,a2):  Bop(OpB, IDe(a1), IDe(a2)),
```

```
    Uop(OpU,a0): Uop(OpU, IDe(a0))
  ENDCASES
  MEASURE e by <<

  IDa(a: ProcessAssignment) : AssignmentExt =
     CASES a OF
       Gate(gexpr) : Gate(IDe(gexpr)),
       Latch(v0, lexpr) : Latch(v0,IDe(lexpr)),
       Input : Input
     ENDCASES

  TRe(e: ProcessExpr, Pr: set[index]):
  RECURSIVE ExprExt =
    CASES e OF
      Constant(value): Constant(EMPTYSET),
      Variable(name):
        IF Pr(name) THEN
          SingletonSet(Pr, name)
        ELSE
          IF_Variable(name)
        ENDIF,
      ITE(test,thn,els):
        Bop(Union,
          ITE(IDe(test),TRe(thn, Pr),TRe(els, Pr)),
          TRe(test, Pr)),
      Bop(OpB,a1,a2):
        Bop(Union, TRe(a1, Pr), TRe(a2, Pr)),
      Uop(OpU,a0): TRe(a0, Pr)
    ENDCASES
  MEASURE e by <<

  TRa(a: ProcessAssignment, Pr: set[index]) :
  AssignmentExt =
    CASES a OF
      Gate(gexpr) : Gate(TRe(gexpr, Pr)),
      Latch(v0, lexpr) :
        Latch(EMPTYSET, TRe(lexpr, Pr)),
      Input : Input
    ENDCASES

  AssignSet: TYPE = [index -> AssignmentExt ]
  ProgramExt: TYPE =
      [# st: AssignSet, gr: AssignSet #]
```

```
  TRp(p: Program) : ProgramExt =
    (# st := (LAMBDA (idx: index) : IDa(p(idx))),
       gr := (LAMBDA (idx: index) :
                TRa(p(idx), InputsP(p))) #)

END TransformIF
```

Fig. 24 Hybrid Model Definitions.

The hybrid model in Fig. 24 contains assignments both for the state variables (*st*) and the graph variables (*gr*). The syntax of the state assignments does not change; however, the strong typing of PVS requires that we define a transformation to map from the *ProcessExpr* and *ProcessAssignment* datatypes into the *ExprExt* and *AssignExt* datatypes, respectively. This is performed by the *IDe* and *IDa* functions, respectively.

The mapping of the information flow *IF* semantics into syntax that can be interpreted is performed by the *TR* functions. These functions create new syntax based on an original program that manipulates index sets. It is instructive to compare the syntax created by the *TRe* function with the definition of the *IFe* semantics originally defined in Fig. 10 and shown again in Fig. 25 below. Note the similarities between the semantic definitions in *IFe* and the syntax generated by the *TRe* function.

```
  IFe(e: ProcessExpr, principal: set[index],
      s0: state, g0: graphState): RECURSIVE
      set[index] =
    CASES e OF
      Constant(value): Empty,
      Variable(name):
         IF principal(name) THEN
            singleton(name)
         ELSE
            g0(name)
         ENDIF,
      ITE(test,thn,els):
        IF isTrue(Se(test,s0)) THEN
          IFe(test,principal,s0,g0) +
          IFe(thn,principal,s0,g0)
        ELSE
          IFe(test,principal,s0,g0) +
          IFe(els,principal,s0,g0)
        ENDIF,
```

```
      Bop(OpB,a1,a2):  IFe(a1,principal,s0,g0) +
         IFe(a2,principal,s0,g0),
      Uop(OpU,a0): IFe(a0,principal,s0,g0)
   ENDCASES
 MEASURE e by <<
```

Fig. 25 Another presentation of the IFe function.

The compositional equivalence between the syntactic rule and the semantic rule can be proven, but we do not demonstrate it in this chapter. To do so would require some further elucidation of sets-as-*vtype* elements as well as an algebraic formulation of the union binary operator over *vtype* elements to show its equivalence to the standard set-union operator. We plan to do this in future work.

The model encoding tool in the Rockwell Collins *Gryphon* tool suite implements the transformation defined by the *TR* rules. It operates over the Lustre language [7]. Lustre includes a superset of the expressions described in the *TR* rules, such as expressions for creating and manipulating composite datatypes including arrays, records, and tuples. It also accounts for Lustre's notion of modularity, called the *node*, which corresponds to Simulink *subsystems*. The complete rules for rewriting Lustre programs are described in a Rockwell Collins technical report that is available at the Springer web site accompanying this text.

For encoding the set of principals for model checking tools, we use bitvectors. The models that we attempt to analyze will always consist of a finite number of variables, and therefore the principal variables form a finite set. We encode this set as a bitvector containing one bit per principal signal. The *Union* and *SingletonSet* operations are encoded as *bit_or* operators and bitvector constants, respectively.

5.3 From Principals to Domains

Our implementation allows multiple variables to be mapped to the same principal identifier (id). This identifier can be thought of as a *security domain* [5,20]. For the purposes of analysis, this can reduce the number of bits necessary for a model checking analysis, which improves performance. It also coarsens the analysis, as it is no longer clear from a counterexample which of the variables mapped to the principal id is responsible for information flow.

5.4 Adding Control Variables

The implementation allows variables to be designated as control variables. The intuition is that an operand of an AND or OR gate sometimes acts as a mask for the other operand (towards FALSE and TRUE, respectively). In this instance, we would like to consider the information flow from the other variable into the gate only if the control variable has the appropriate value. This feature allows for slightly more accurate analysis in some models. It is a conservative extension because the semantics of AND and OR gates are semantically the same as the following if/then/else structure:

```
Y = C and E ⇔ Y = if C then E else false;
Y = C or E ⇔ Y = if C then true else E ;
```

Y is semantically equivalent in both cases, and the soundness of the flow analysis follows from the existing proof of if/then/else expressions in Fig. 12. Note that the condition variable for if/then/else (C) is always used for the information flow analysis, so if *both* variables in a Boolean expression are control variables, the following is generated:

```
Y = C0 and C1 ⇔
Y = if C0 then (if C1 then C0 else false) else
   (if C1 then C0 else false)
```

After applying the syntactic *TRe* transformation to the right hand side of the equivalence and simplifying, this yields the "standard" information flow expression for the original binary expression: *Bop*(*Union, TRe*(*a1, Pr*), *TRe*(*a2, Pr*)).

6 Intransitive Interference and Noninterference

We have defined a considerable amount of infrastructure for determining which variables can *interfere* with a particular computed variable within a model. In the approach we have pursued in the previous sections of this chapter, all interference relations are *transitive.* That is, if variable *A* interferes with variable *B* and *B* interferes with *C*, then *A* interferes with *C.* However, there are several systems in which we are willing to allow certain kinds of interference across security domains, as long as it is mediated in some way. The reasoning for allowing this interference is well explained by Roscoe and Goldsmith [19]:

> It seems intuitively obvious that the relation must be transitive: how can it make sense for A to have lower security level than B, and B to have lower level than C, without A having lower level than C? But this argument misses a crucial possibility, that some high-level users are trusted to *downgrade* material or otherwise influence low-level users. Indeed, it has been argued that no large-scale system for handling classified data would make sense without some mechanism for downgrading information after some review process, interval (e.g., the U.K. 30-year rule) or defined event (the execution of some classified mission plan, for example). Largely to handle this important problem, a variety of extended theories proposing definitions of ''intransitive noninterference'' have appeared, though we observe that this term is not really accurate, as it is in fact the *interference* rather than the *noninterference* relation which is not transitive. Perhaps the best way to read the term is as an abbreviation for ''noninterference under an intransitive security policy''.

There have been several formulations of intransitive interference based on state machines [20], process algebras [19], and event traces [10].

6.1 Formulating Intransitive Interference

Our model is entirely defined in terms of variables. Operations such as encryption or downgrading are implemented as subsystems (sets of variables) within a larger model whose output is another variable within the model. Therefore, it is natural to think of extending the set of principal variables P from only the inputs to include internal variables that define the mediation points of interest. Since the definition of *Noninterference* requires only that the principal variables agree, these intermediaries are easily incorporated into our definition.

For example, in the shared buffer model, we are willing to allow information to flow through the scheduler. By adding the scheduler state to P, we restrict ourselves to reasoning over traces in which the scheduler states match. From the perspective of reasoning, it is straightforward to parameterize the proofs over a superset of the inputs and reprove the *InterferenceTheorem* and *NoninterferenceTheorem* defined in Sections 3 and 4.

The Problem of Implicit Functional Dependencies

Unfortunately, this formulation of "correctness" allows unintended covert information flows around the mediation point as long as they can be *functionally derived* from an input variable.

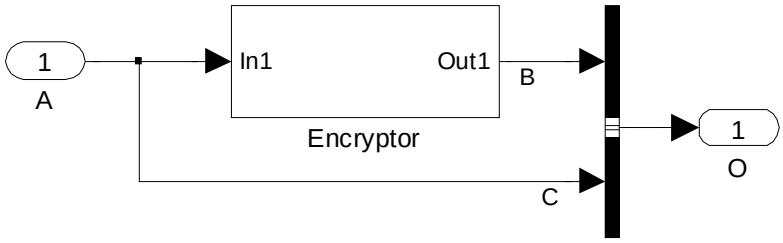

Fig. 26 Simulink model containing a bypass.

Figure 26 presents a Simulink model that illustrates the problem. Output O is a record type that contains two fields *B* and *C*. Field *B* is the output of a subsystem that encrypts the input variable (A); field *C* is a simple pass-through of *A*. Suppose that the output of the encryptor *B* is functionally derived from input *A*. That is, two traces on B agree *only when* the traces on *A* also agree. In this case, according to the interference theorem, we can adjudge output *O* to be dependent only on *B*, even though there is clearly a flow that bypasses *B*. The problem is that the encryptor variable is *functionally derived* from a single input *A*, so the equivalence on *B* forces a corresponding equivalence on the input *A*. In other words, requiring a trace equivalence on a computed principal variable may cause an implicit equivalence on another principal variable. These implicit equivalences allow an attacker to bypass the desired mediation variable.

An Overly Conservative Formulation

An approach that could be considered for intransitive interference reframes the problem: given a program *P* involving a computed principal variable *c* we construct a program *P'* in which *c* is an input, and assert that all traces must agree on *P'*. *P'* has at least as many traces as *P*, as the value of *c* is unconstrained with respect to the other variables in *P'*. The additional traces distinguish variables that bypass the computed principal as there is no longer a functional connection between the computed variable and the inputs.

Unfortunately, treating states as inputs leads to overly conservative analyses involving traces that are impossible in the original program. Consider the shared buffer model from Section 2. If a new model is created in which the scheduler output is instead a system input, then the scheduler can no longer correctly mediate access to the shared buffer and so information flow occurs through the buffer. The flow analysis will (correctly) state that there is information flow through the buffer, but the flagged traces are not possible in the original model.

6.2 Modeling Intransitive Interference using Graph Cuts

The analysis approach that is used in Gryphon is to model intransitive information flow through *cuts* in the information flow graph. That is, we define a new principal variable in the information flow graph by cutting the edges that define the dependencies of that computed variable. To implement the change in the information flow (IF) semantics defined in Section 3, we add the internal variable indices to the set of inputs that are used in the *IFe, IFi, IFs,* and *IFt* relations. The definition of the program *P* is left unchanged.

This modified graph model is sufficient to correctly characterize both a program *P* and a modified program *P'* in which a principal variable *c* is treated as an input. In other words, this formulation is sensitive to the *structure* of the computation of the system execution traces as well as the *functional result*. The original program *P* is analyzed, so there are no problems introduced by the additional traces of *P'*, but we (correctly) characterize models such as the one described by Fig. 26 as containing direct information flows from input variable *A* to output *O*.

Illustrations of a transitive flow model and an intransitive model using graph cuts are shown in Fig. 27. Recall that the *hybrid model* that is generated for model checking is composed of both a *functional model* (the original system) and an *information flow model* which is an encoding of the *IF* semantics as described in Section 5. In Fig. 27, the functional model is shown at the top of the figure. In the middle is a *transitive* information flow model. At the bottom is an *intransitive* information flow model[2]. Each model is presented both graphically on the left and in terms of equations on the right. In this figure, the principal bitvector for a variable *X* is notated $\underline{X}$.

Suppose variable *Y* (the switch gate) acts as a downgrader for variable *D*. We would like to state that the output (*Z*) depends on input *D* only when mediated through the downgrader. Given the transitive formulation of information flow in the middle of Fig. 27, it is not possible to make this claim. However, the intransitive graph at the bottom of Fig. 27 breaks the information flow graph for each use of variable *Y*, replacing the input flows through the computed definition of *Y* with a new principal signal $\underline{Y}$. Given this new graph, it is possible to prove that no information flows from *D* to *Z* that is not mediated by *Y*. On the other hand, note that with this intransitive graph, a non-interference proof would still not be possible for variable *C* as it has a flow to *Z* that bypasses *Y*.

---

[2] For model-checking analysis only *one* of the two information flow models would be generated, depending on the set of principal signals provided. However, Fig. 24 is designed to illustrate the differences between the transitive and intransitive analysis.

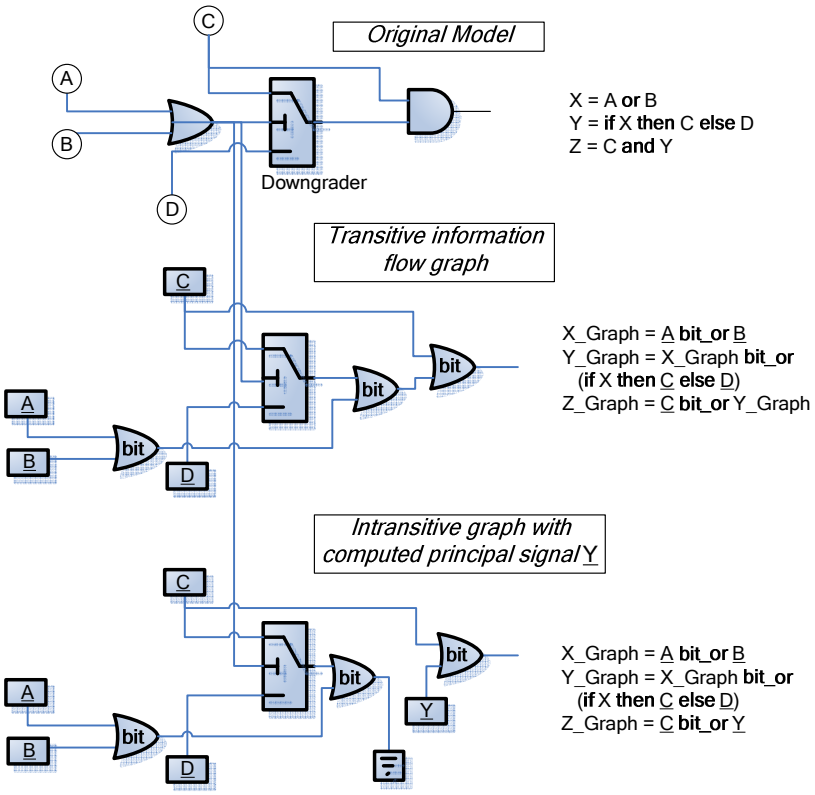

Fig. 27 Transitive vs. Intransitive flow graphs.

We currently do not have a strong theorem (such as the *InterferenceTheorem*) that we can prove about intransitive dependencies. Further, we conjecture that it is not possible to functionally characterize such dependencies using trace semantics. Instead, the structure of the computation function must be examined – the property is *intrinsic* to the structure.

7 Connections to GWV

In the current chapter and the previous chapter by Greve [6], we have presented two quite similar formulations of information flow modeling. The formulation in Greve's

chapter is more abstract and describes information flow over arbitrary functions using flow graphs. It then describes how these functions can be composed and how multi-step state transition systems can be encoded. Two different formulations (GWVr1 and GWVr2) are presented. The GWVr2 formulation is capable of modeling dynamic information flows, in which storage locations are created and released during the computation of the function, but this additional capability comes at a cost of some additional complexity.

In this chapter, we have modeled information flow specifically for synchronous dataflow languages. The basis for this approach was modeling GWV-style equivalences using a model checker. However, the approach was originally justified by manual proofs over trace equivalences due to the first author's familiarity with this style of formalization for synchronous dataflow languages. The mechanized proofs in this chapter reflect the manual proofs.

As a basis for formalization, the trace equivalence allows a very natural style of presentation. It provides a nice abstraction of the computation and information flow analysis in that a total computation order for the assignments of the semantic and flow analyses is not required. Instead, we can talk about *conformance* to some existing trace. Also, since the entire trace is provided, we can describe latch conformance by examining the previous state in the trace.

7.1 From InterferenceTheorem to GWVr1

From the *InterferenceTheorem*, it is straightforward to map directly into the GWVr1 theorem presented in Greve's chapter [6], as shown in Figure 28.

```
GWVr1_Connection[
    (importing ProcessInterference)
    P: WFPrograms]: THEORY
BEGIN
  IMPORTING ProcessInterference

  valid_tp : TYPE = {tp: tracePair | tp_ok(P, tp)}

  st_liftv(i: index, tp: tracePair) : vtrace =
    liftv (i, s(tp))

  IMPORTING GWVr1[index, valid_tp, vtrace, st_liftv,
                  index, valid_tp, vtrace, st_liftv]
```

```
  step_id(tp: valid_tp) : valid_tp = tp;

  gtrace_graph(tp: valid_tp)(idx: index) :
  GraphEdge[index] =
     Compute(DepSet(idx, g(tp)))

  precondition(tp: valid_tp) : bool = true ;

  inputEquivSet_to_vtraceEquivSet : LEMMA
    (FORALL (is: set[index], tp1, tp2: tracePair) :
       Input.equivSet(is, tp1, tp2) =>
       vtraceEquivSet(is, s(tp1), s(tp2)))

  GraphIsGWVr1 : LEMMA
     GWVr1(step_id)(precondition, gtrace_graph);

END GWVr1_Connection
```

Fig. 28 Connection to GWVr1 theorem.

GWVr1 is defined as a proof obligation over a transition function from an input state to an output state. The fragment of the GWVr1 theory required for the proof is shown in Fig. 29.

```
GWVr1 [INindex, INState, INvalue: TYPE,
       getIN: [[INindex, INState] -> INvalue],
       OUTindex, OUTState, OUTvalue: TYPE,
       getOUT: [[OUTindex, OUTState] -> OUTvalue]

GWVr1 [INindex, INState, INvalue: TYPE,
       getIN: [[INindex, INState] -> INvalue],
       OUTindex, OUTState, OUTvalue: TYPE,
       getOUT: [[OUTindex, OUTState] -> OUTvalue]
       ]: THEORY

BEGIN

  IMPORTING GWV_Graph[INindex,OUTindex]
  IMPORTING GWV_Equiv[INindex,INState,INvalue,getIN]
    AS Input
  IMPORTING GWV_Equiv[OUTindex,OUTState,OUTvalue,
                      getOUT] AS Output

  StepFunction: TYPE = [ INState -> OUTState ]
  GraphFunction: TYPE = [ INState -> graph ]
  PreCondition: TYPE  = [ INState -> bool ]

  GWVr1(Next: StepFunction)
   (Hyps: PreCondition, Graph: GraphFunction): bool=
    FORALL (x: OUTindex, in1,in2: INState):
      Input.equivSet(DIA(x,Graph(in1)),in1,in2) &
        Hyps(in1) & Hyps(in2) =>
          Output.equiv(x,Next(in1),Next(in2))
```

Fig. 29 Fragment of GWVr1 theory.

The *index*, *state*, *value*, and *get* parameters to the theory define the indices of discourse, the state, the values that can be stored at indices, and the "getter" function to look up a value for the inputs and outputs of the transition function. In our case, the types of inputs and outputs are the same: we are looking at traces. To format our trace equivalences as a GWVr1 theorem, we create a theory parameterized by an arbitrary well-formed program. The GWV index values are simply our index type, the state is the trace pair containing both the execution state and the information flow state, values map to our *vtype*, and the *get* function returns a variable trace from the state trace.

The proof to GWVr1 merely involves re-shaping the *InterferenceTheorem* into the expected arguments for GWVr1. Our *StepFunction* is simply the identity; we already

have the entire trace. The *GraphFunction* returns the trace dependency set for a variable of interest; this is the same set used by the *InterferenceTheorem.* No hypotheses are necessary, so we create a trivial precondition function. We introduce a lemma *inputEquivSet_to_vtraceEquivSet* to map between the set equivalence functions used by *InterferenceTheorem* and *GWVr1,* then can establish the *GraphIsGWVr1* lemma with very little difficulty using the *InterferenceTheorem* as a lemma.

Although the trace formulation provides a nice level of abstraction for describing synchronous dataflow languages, in this chapter we have duplicated some of the infrastructure that had already been established in [6] with respect to function composition, mapping from interference to noninterference, and justifying LTL theorems in terms of trace equivalence. It would be possible to re-formalize the synchronous language semantics defined in Section 3 in order to better utilize the GWV infrastructure, but we leave this for future work.

8 Using Gryphon For Information Flow Analysis

We now demonstrate the information flow analysis in the Rockwell Collins *Gryphon* tool suite. *Gryphon* is an analysis framework designed to support model-based development tools such as Simulink/Stateflow and SCADE. Model-based development (MBD) refers to the use of domain-specific, graphical modeling languages that can be executed and analyzed before the actual system is built. The use of such modeling languages allows the developers to create a model of the system, execute it on their desktop, analyze it with automated tools, and use it to automatically generate code and test cases.

As MBD established itself as a reliable technique for software development, an effort was made to develop a set of tools to enable the practitioners of MBD to formally reason about the models they created. Fig. 30 illustrates MBD development process flow.

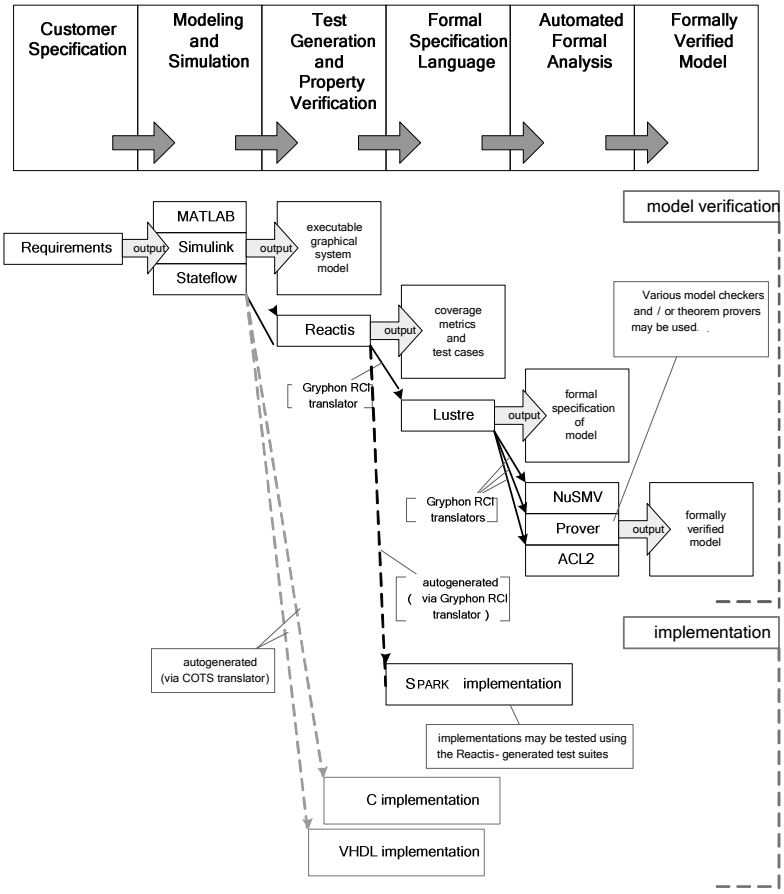

Fig. 30 Model-Based Developmment Process Flow.

8.1 Model-Based Development Toolchain

The following sections briefly describe each component of the MBD tool-chain.

Simulink, Stateflow, MATLAB

Simulink, Stateflow, and MATLAB are products of The MathWorks, Inc. [11] Simulink is an interactive graphical environment for use in the design, simulation, implementation, and testing of dynamic systems. The environment provides a customizable set of block libraries from which the user assembles a system model by selecting and connecting blocks. Blocks may be hierarchically composed from predefined blocks.

Reactis

Reactis® [17], a product of Reactive Systems, Inc., is an automated test generation tool that uses a Simulink/Stateflow model as input and auto-generates test code for the verification of the model. The generated test suites target specific levels of coverage, including state, condition, branch, boundary, and modified condition/decision coverage (MC/DC). Each test case in the generated test suite consists of a sequence of inputs to the model and the generated outputs from the model. Hence, the test suites may be used in testing of the implementation for behavioral conformance to the model, as well as for model testing and debugging.

Gryphon

Gryphon [24] refers to the Rockwell Collins tool suite that automatically translates from two popular commercial modeling languages, Simulink/Stateflow and SCADE [4], into several back-end analysis tools, including model-checkers and theorem provers. Gryphon also supports code generation into Spark/Ada and C. An overview of the Gryphon framework is shown in Fig. 31. Gryphon uses the Lustre [7] formal specification language (the kernel language of SCADE) as its internal representation. This allows for the reuse of many of the RCI proprietary optimizations.

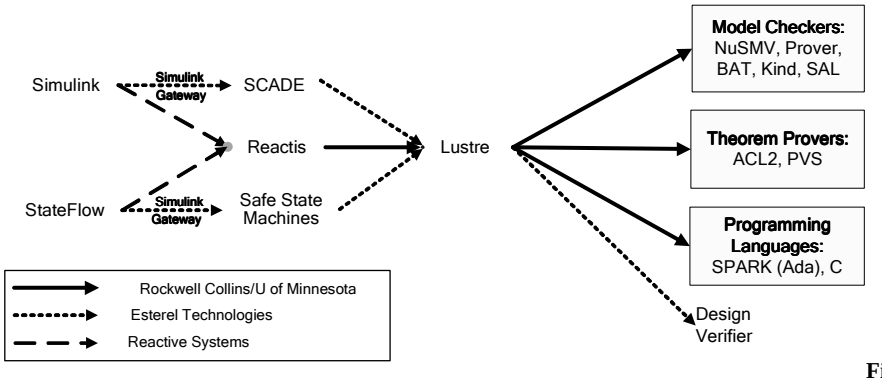

Fig. 31 Gryphon Translator Framework.

Prover

Prover [16] is a best-of-breed commercial model checking tool for analysis of the behavior of software and hardware models. Prover can analyze both finite state models and infinite-state models, that is, models with unbounded integers and real numbers, through the use of integrated decision procedures for real and integer arithmetic. Prover supports several proof strategies that offer high performance for a number of different analysis tasks including functional verification, test-case generation, and bounded model checking (exhaustive verification to a certain maximum number of execution steps).

Custom Code Generation

By leveraging its existing Gryphon translator framework, Rockwell Collins designed and implemented a tool-chain capable of autotomatically generating SPARK-compliant Ada95 source code from Simulink/Stateflow models.

8.2 Modeling and Analyzing the Turnstile High-Assurance Guard Architecture

A large scale use of the Gryphon analysis was performed on the Rockwell Collins Turnstile high-assurance cross-domain guard [18]. A high-level view of the architecture is shown in Fig. 32. The offload engines (OEs) provide the external interface to Turnstile. The Guard Engine (GE) is responsible for enforcing the desired security policy for message transport. The Guard Data Movers (GDMs) provide a high-speed mechanism to transfer messages under the direction of the GE. The GE is implemented on the EAL-7 AAMP7 microprocessor [25] and uses the partitioning guarantees provided by the AAMP to ensure secure operation.

In its initial implementation, Turnstile provides a "one way" guard. It has a *high side* OE (OE1 in Fig. 32) that submits messages (generates input) for the guard, a *low side* OE (OE3 in Fig. 32) that emits messages if they are allowed to pass through the guard, and an *audit* OE (OE2 in Fig. 32) that provides audit functionality for the system.

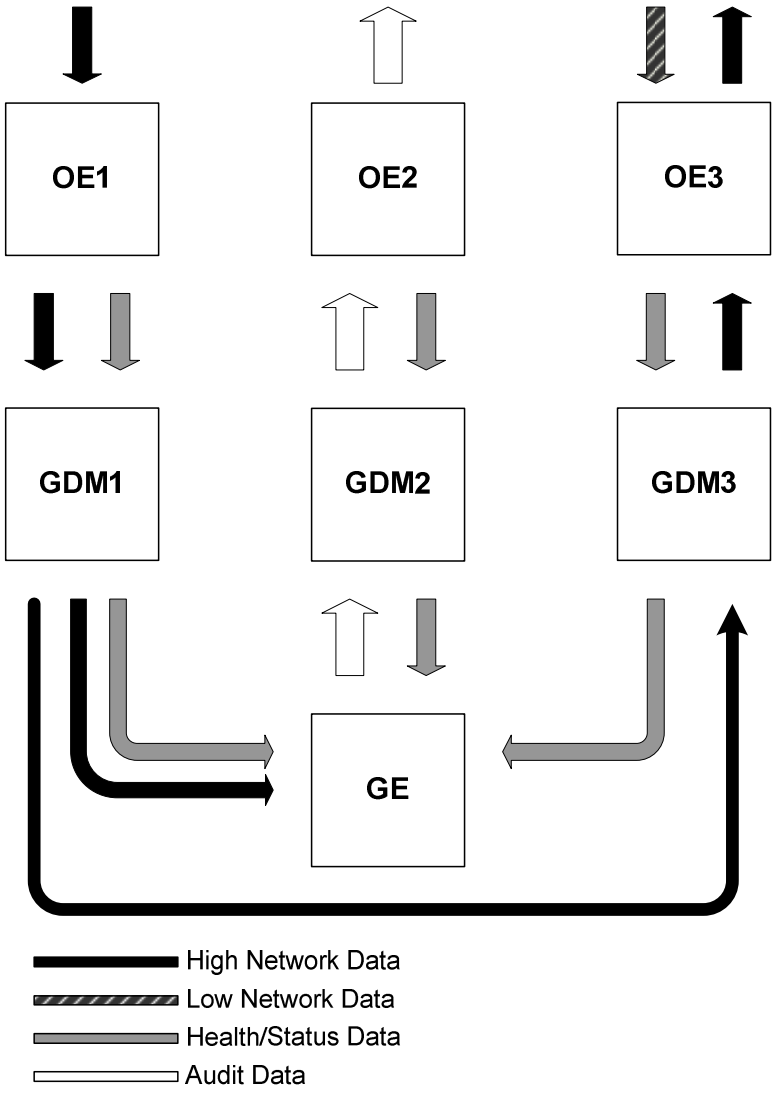

Fig. 32 Turnstile System Architecture.

The architectural analysis focused on the interaction between the GDMs, GE, and OEs. The OEs, GDMs and GE do not share a common clock and both execute and communicate asynchronously. In the model, we *clock* each of the subsystems using a

system input. This input is allowed to vary non-deterministically, allowing us to model all possible interleavings of system execution.

Representing the Turnstile Architecture in Simulink

The Simulink model of the Turnstile system architecture is shown in Fig. 33. The components were modeled at various levels of fidelity, depending on their relevance to the information flow problem:

- The GDMs are responsible for most of the data routing and were modeled to a high level of fidelity. All of the GDM channels (transmit, receive, audit, control, and health monitor) are modeled as well as the GDM-to-GDM and GDM-to-GE transfer protocols.
- The data routing portions of the GE were accurately modeled. The policy enforcement portions (the guard evaluator) were modeled non-deterministically: the GE component randomly chooses whether messages are dropped or propagated.
- The OEs were modeled at a fairly low level of fidelity. As the OEs are not trusted by the Turnstile architecture, we allow them to non-deterministically submit requests on all of the interfaces between OE and GDM. This approach allows us to model situations in which the OE violates the Turnstile communications protocols (which should cause the system to enter a fail-safe mode).

The principals of interest are those processes on the Offload Engines that interact with the outside world (the low and high networks): the reading and writing processes on OE1 and the reading and writing processes on OE3. To represent the arbitrary interleavings of the Turnstile processes, we used enabled (clocked) subsystems in Simulink. The GDMs run in synchrony at the basic rate of the model while the OEs and GE run at arbitrary intervals of the basic rate.

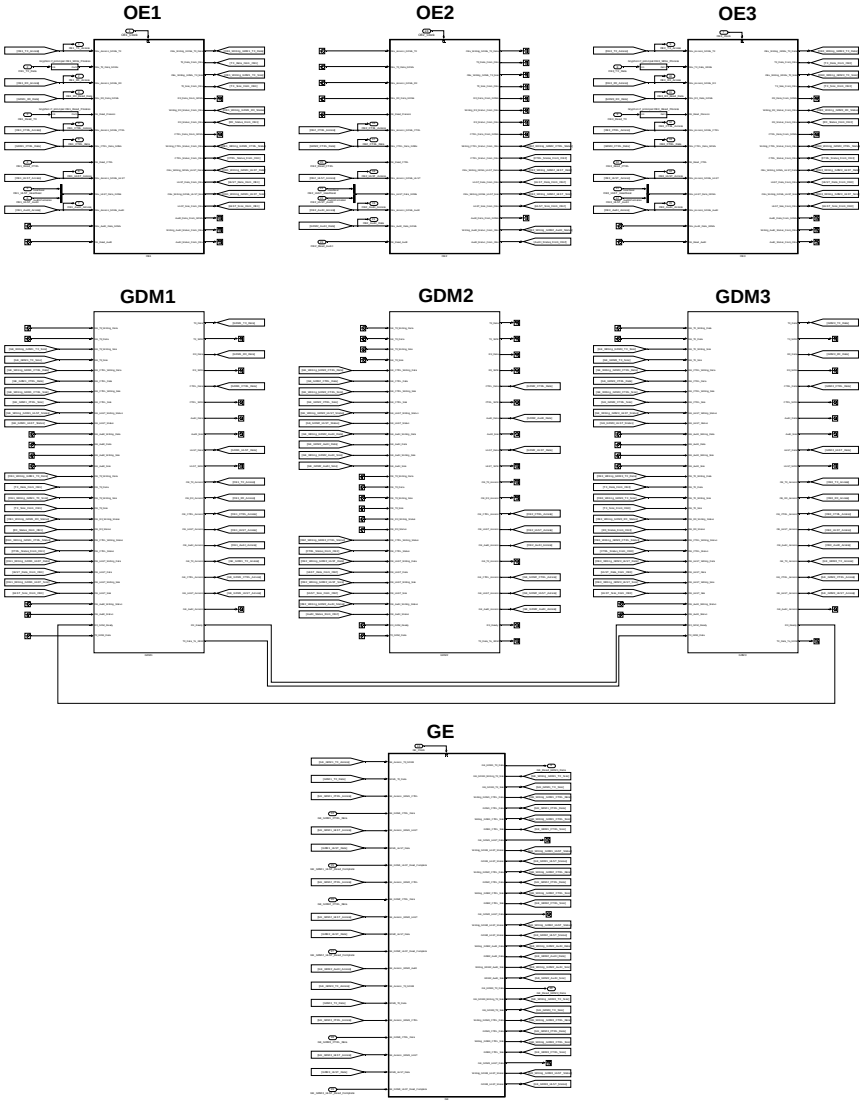

Fig. 33 Simulink Turnstile Model.

The model in Fig. 33 was translated via Gryphon into the model checkers NuSMV [8] and Prover [16]. With these tools we analyzed several of the information flows through the model. Since the OE has multiple inputs in our model (and in real life) we

analyzed every input into the OEs for the possible presence of information from an unwanted source. In a one-way guard configuration, we are interested in determining whether there is backflow of information to the high-side network, that is, whether any GDM input into OE1 is influenced by the low-side (OE3) reading or writing principals. These properties can be encoded as shown in Fig. 34.

```
l2h_tx1 = not gry_IF_OE1_TX_Access[p_oe3_writer] ;
l2h_tx2 = not gry_IF_OE1_TX_Access[p_oe3_reader] ;
l2h_tx3 = not gry_IF_OE1_RX_Access[p_oe3_writer] ;
l2h_tx4 = not gry_IF_OE1_RX_Access[p_oe3_reader] ;
l2h_tx5 = not gry_IF_OE1_RX_Read_Data[p_oe3_writer] ;
l2h_tx6 = not gry_IF_OE1_RX_Read_Data[p_oe3_reader] ;
l2h_ctrl1 = not gry_IF_OE1_CTRL_Data[p_oe3_writer] ;
l2h_ctrl2 = not gry_IF_OE1_CTRL_Data[p_oe3_reader] ;
l2h_ctrl3 = not gry_IF_OE1_CTRL_Access[p_oe3_writer] ;
l2h_ctrl4 = not gry_IF_OE1_CTRL_Access[p_oe3_reader] ;
l2h_hlst1 = not gry_IF_OE1_HLST_Access[p_oe3_writer] ;
l2h_hlst2 = not gry_IF_OE1_HLST_Access[p_oe3_reader] ;
```

Fig. 34 Backflow Properties from “Low Side” OE3 to “High Side” OE1.

One of the back flow properties (shown in bold font) was violated in the architectural model. However, this was already a known source of back flow because of the implementation of the GDM transfer protocol that resulted from a quality of service requirement levied on the Turnstile implementation. This requirement stated that a new message cannot be accepted until the previous message had been delivered. In the Turnstile architecture, the high-side writer is unable to transmit to the GDM until the low side reader has finished consuming the last message. The low-side reader could potentially use this mechanism to transmit information (interfere) with the high side network. The verification of the other properties demonstrates that the high-side OE is not, for example, influenced by the low-side writer.

Also, because the Audit OE may also be connected to the high network we wanted to verify that no information from OE3 leaks out to the Audit network from any of the GDM inputs to OE2. These properties, which are all proven correct by the Prover model checker are shown in Fig. 35.

```
oe2_audit1 = not gry_IF_OE2_Audit_Access[p_oe3_writer] ;
oe2_audit2 = not gry_IF_OE2_Audit_Access[p_oe3_reader] ;
oe2_audit3 = not gry_IF_OE2_Audit_Data[p_oe3_writer] ;
oe2_audit4 = not gry_IF_OE2_Audit_Data[p_oe3_reader] ;
oe2_audit5 = not gry_IF_OE2_CTRL_Access[p_oe3_writer] ;
oe2_audit6 = not gry_IF_OE2_CTRL_Access[p_oe3_reader] ;
oe2_audit7 = not gry_IF_OE2_CTRL_Data[p_oe3_writer] ;
oe2_audit8 = not gry_IF_OE2_CTRL_Data[p_oe3_reader] ;
oe2_audit9 = not gry_IF_OE2_HLST_Access[p_oe3_writer] ;
oe2_audit10 = not gry_IF_OE2_HLST_Access[p_oe3_reader] ;
```

Fig. 35 Backflow Properties from "Low Side" OE3 to Audit OE2.

Though much more complex, the Turnstile architectural model is conceptually similar to the shared buffer example. The GE acts as the scheduler between the GDMs, which are physically connected together and can be thought of as defining a "shared" resource. It is crucial to note that accurate *conditional* information flow is necessary to successfully analyze the Turnstile system architecture and many other industrial systems of interest. Since the GDMs are directly connected, an unconditional analysis of the architecture would not be able to demonstrate non-interference properties between the high and low side OEs. Only by considering the state of the system (especially the GE) can one demonstrate the security of the architecture.

9 Conclusion and Future Work

In this chapter, we have described an analysis procedure that can be used to check a variety of information flow properties of hardware and software systems, including *noninterference* over system traces. This procedure is an instantiation of the GWV-style flow analysis specialized for synchronous dataflow languages such as SCADE [4] and Simulink [11]. Our analysis is based on annotations that can be added directly to a Simulink or SCADE model that describe specific sources and sinks of information. After this annotation phase, the translation and model checking tools can be used to automatically demonstrate a variety of information flow properties. In the case of non-interference, they will prove either that there is no information flow between the source and a variable of interest, or demonstrate a source of information flow in the form of a counterexample.

In order to justify the model checking analysis, we have presented a formalization of our approach in PVS and demonstrated a *NoninterferenceTheorem.* This theorem

states that if our model-checking analysis determines a system input does not interfere with a particular output, then it is possible to vary the trace of that input without affecting the output in question. The analysis is both scalable and accurate, and can be used to describe:

- **Conditional Information Flow:** The analysis is sensitive to the state of the model, and can be used in situations in which multiple domains "share" a resource, such as the shared buffer model.
- **"Covert" Information Flow:** The analysis can detect flows due to (for example) contention for resources. These flows are ultimately manifest in the test expressions for conditionals, which are propagated to the output of the conditional.
- **Intransitive Information Flow:** The analysis can be used to define intransitive information flows, in which we are willing to allow information flows between domains as long as they occur through well-defined mediation points.

Our analysis is implemented in the *Gryphon* tool suite that supports several kinds of formal analysis of Simulink and Stateflow models. *Gryphon* has been used in several large-scale formal verification efforts [24], including a flow analysis of the Turnstile high-assurance cross-domain guard.

9.1 Future Work

There are several directions for future work given the framework that has been created. First, there are a variety of interesting properties beyond non-interference that can be formalized using temporal logic. For example, it is possible to begin talking about *rates* of information flow through a system by creating more interesting temporal logic formulations of flow properties. For example, one can state that flow occurs at most every ten cycles of evaluation (say), with the following Real-Time CTL (RTCTL) [2] property:

```
SPEC AG(gry_IF_output[P1] -> ABF[1,10] (!gry_IF_output[P1]));
```

where 'ABF' is the bounded future operator of RTCTL. This formula states that if flow occurs from principal *P1* to variable *output* in the current steps, that no flow occurs from *P1* to *output* over the next 10 steps. In order to be informative, this obligation would have to be paired with some notion of *how much* information was being transmitted by a particular flow in an instant when flow occurs. It should be possible to annotate (manually or automatically) an information flow model with the flow rates along particular edges within the graph. Such an annotation could be used to over-

approximate “acceptable” levels of information loss when strict non-interference is not possible (such as with the scheduler in the shared buffer example from Section 2.2).

Similarly, we may want to describe modal information flow properties. For example: *as long as the system is not in the self-test mode, then no information flows from A to B.* These properties are straightforward to specify in temporal logic, but precisely defining the meaning of these kinds of properties in a more general *InterferenceTheorem* would be a useful exercise.

It should be possible to partition the model checking analyses using compositional reasoning techniques such as those described in [12, 13] for very large models. Determining the obligations over both the functional state and also the information flow graph should be an interesting exercise, and may yield further insights into the relationship between a functional model and information flow graph.

There are several directions in which to extend the full formalization of the approach in PVS. First, we should formalize the proof of equivalence between the *IFe* semantics and the *information flow model* that is generated by the translation rules in Section 5. A more ambitious step would be to formalize the entire Lustre language in PVS including the clock operators and modularity constructs and demonstrate the correctness of the complete translation provided in the Gryphon toolsuite.

Finally, we would like to be able to compose the model checking results with results from theorem proving GWV-style theorems using a theorem prover such as PVS or ACL2. This would allow partitioning of very large problems into portions that can be analyzed with “the right tool for the job”, using theorem proving where required (e.g., when complex dynamic data structures are involved) but using automated analysis using model checking where possible.

Acknowledgments

We would like to thank the reviewers of early drafts of this paper, especially Matt Staats, Andrew Gacek, and Kimberly Whalen, for their many helpful comments and suggestions.

References

[1] S. Bensalem, P. Caspi, C. Parent-Vigouroux, and C. Dumas. A methodology for proving control systems with Lustre and PVS. In Proceedings of the Seventh Working Conference on Dependable Computing for Critical Applications (DCCA 7), San Jose, January 1999.

[2] E.M Clarke, O. Grumberg, and D. A. Peled. Model Checking. MIT Press, Cambridge, Massachusetts, 1999.

[3] J.L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, September 2004.

[4] Esterel Technologies, Inc., SCADE Suite product description. http://www.esterel-technologies.com/products/scade-suite

[5] J. A. Goguen and J. Meseguer. Security policies and security models, *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pp 1 1-20. IEEE Computer Society Press, 1982.

[6] D. Greve. Information Security Modeling and Analysis. In *Design and Verification of Microprocessor Systems for High-Assurance Applications.* D. Hardin, ed. Springer, 2010.

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. In *Proceedings of the IEEE*, Volume 79, #9, pp. 1305-20, September 1991.

[8] IRST: http://nusmv.irst.itc.it/ The NuSMV Model Checker, IRST, Trento Italy

[9] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, Claude Le Maire. *Programming real-time applications with Signal.* Proceedings of the IEEE, v. 79, pp. 1321-1336, Sept 1991.

[10] J. MacLean Proving noninterference and functional correctness using traces, *Journal of Computer Security* 1: 37-57 (1992).

[11] The Mathworks, Inc., Simulink and Stateflow product description. http://www.mathworks.com/Simulink, http://www.mathworks.com/products/stateflow/

[12] K. McMillan, Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking, *Proceedings of the 10th International Conference on Computer Aided Verification (CAV '98)*, Vancouver, Canada, June, 1998.

[13] K. McMillan. Circular Compositional Reasoning about Liveness. *Advances in Hardware Design and Verification: IFIP WG10.5 International Conference on Correct Hardware Design and Verification Methods (CHARME '99)*, pp. 342-45, 1999.

[14] Cesar Munoz. ProofLite product description http://research.nianet.org/~munoz/ProofLite

[15] S. Owre, J. M. Rushby, and N. Shankar, PVS: A Prototype Verification System, *11th International Conference on Automated Deduction (CADE)*, v. 607 pp. 748-752, Saratoga, NY, June 1992.

[16] Prover Technologies, Inc. Prover SL/DE plug-in product description. http://www.prover.com/products/prover_plugin

[17] Reactive Systems, Inc. Reactis product description. http://www.reactive-systems.com

[18] Rockwell Collins Turnstile Product Page. http://www.rockwellcollins.com/products/gov/airborne/cross-platform/information-assurance/cross-domain-solutions/index.html

[19] A.W. Roscoe and M.H. Goldsmith. What is intransitive noninterference? In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pp. 228-38, 1999.

[20] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report csl-92-2, SRI, 1992.

[21] P.Y.A. Ryan, A CSP formulation of non-interference, *Cipher*, pp 19-27.IEEE Computer Society Press, 1991.

[22] SRI, Incorporated. PVS Specification and Verification System, http://pvs.csl.sri.com

[23] SRI, SAL Home Page, http://www.csl.sri.com/projects/sal/.

[24] M. Whalen, D. Cofer, S. Miller, B. Krogh, and W. Storm. Integration of Formal Analysis into a Model-Based Software Development Process. *12th International Workshop on Industrial Critical Systems (FMICS 2007),* Berlin, Germany, July, 2007.

[25] M. Wilding, D. Greve, R. Richards, and D. Hardin. Formal Verification of Partition Management for the AAMP7G Microprocessor. In *Design and Verification of Microprocessor Systems for High-Assurance Applications.* D. Hardin, ed. Springer, 2010.