Paper presented at *SAS 2014: Specification, Algebra, and Software, A Festschrift Symposium in Honor of Kokichi Futatsugi.*
Proceedings, S. Iida, J. Meseguer, and K. Ogata (Eds.),
published as Springer LNCS vol. 8373, pp. 110–128, April 2014

The Versatile Synchronous Observer

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA

Abstract. A synchronous observer is an adjunct to a system model that monitors its state variables and raises a signal flag when some condition is satisfied. Synchronous observers provide an alternative to temporal logic as a means to specify safety properties but have the advantage that they are expressed in the same notation as the system model—and thereby lower the mental hurdle to effective use of model checking and other techniques for automated analysis of system models. Model checkers that do use temporal logic can nonetheless employ synchronous observers by checking for properties such as "never(flag raised)."
The use of synchronous observers to specify properties is well-known; rather less well-known is that they can be used to specify assumptions and axioms, to constrain models, and to specify test cases. The idea underlying these applications is that the basic model generates more behaviors than are desired, the synchronous observer recognizes those that are interesting, and the model checker is constrained to just the interesting cases. The efficiency in this approach is that it is usually much easier to write recognizers than generators.
The paper describes and illustrates several applications of synchronous observers.

1 Introduction

Model checkers are called that because, in their basic form, they *check* whether a system defined as a finite state machine is a Kripke *model* of a specification expressed in a temporal logic. The selection of finite state machines for the system description and (branching time) temporal logic for their specification has pragmatic benefits: in this form, the model checking problem can be fully automated and its complexity is linear in the size of both system and specification (although the size of the system is often exponential in the number of its components).

However, the term *model checking* has grown beyond this precise usage and now refers to any highly automated method for formal analysis of systems and their specifications—as contrasted, for example, to methods that use interactive theorem provers. Under this looser usage, specification methods other than temporal logic may be employed, and one example is the *synchronous observer*. Here, the system is described as a state machine, as before, and its specification is likewise described by a state machine that observes the state variables of the

system and sets a Boolean "flag" variable *true* as long as the required properties hold. The "model checker" then verifies that the flag variable is always *true*. Obviously, the flag variable can be used in either "parity": we can choose to set it *true* when the required property is satisfied or, alternatively, if it is violated. For consistency in the examples, we will use state variables whose names are a variant on `ok` for the former case and a variant on `alarm` for the latter.

Both the concept and the term "synchronous observer" were introduced in the context of the *synchronous languages* developed in France and, in particular, by the Lesar model checker for the language Lustre [1, 2]. However, the idea is readily adapted for use with temporal logic model checkers: we simply model check for the temporal property `always(ok)` or `always(NOT alarm)` (this expresses the `never` construct used in the abstract to this paper); the "always" operator may be written as `AG` in a specification language based on the branching time logic CTL (Computation Tree Logic), or as `G` or $\Box$ in one based on Linear Temporal Logic, LTL. If the model checker allows liveness properties (e.g., its specification language provides the operator `eventually`, which may be written `AF`, `F`, or $\Diamond$) then synchronous observers can likewise be extended to liveness properties.

An advantage claimed for synchronous observers over temporal logic specifications is that a single language is used to describe both the system and its required properties (and, as we will see later, also its assumptions). Furthermore, engineers readily understand the method, since it is like adding a runtime check to an executable program. In contrast, when using temporal logic specifications, the engineer is required to learn and use one language and method for describing the system, and another for specifying its properties. As we will see, many simple specification constructs are quite difficult to write as temporal logic formulas and intermediate "pattern languages" have arisen to ease this difficulty [3].

The purpose of this paper is to describe and illustrate the uses and benefits of synchronous observers. We begin with their familiar use in the specification of properties and assumptions, and then proceed to less familiar uses where they enable the specification of relational constraints and of axioms for uninterpreted functions. We then turn from applications in verification to their use in the construction of test cases. These illustrations of the versatility of synchronous observers constitute Section 2 of the paper; brief conclusions are presented in Section 3.

2 Synchronous Observers and their Applications

In the subsections that follow, we introduce several applications for synchronous observers.

2.1 Specification

We begin by describing the standard use of synchronous observers to specify properties and assumptions, as in Lesar [2], but we do so in the framework of

model checkers that ordinarily use temporal logic specifications. To make our illustrations concrete, we use the syntax and tools of the SAL suite of model checkers from SRI [4].

In SAL, systems are specified as synchronous or asynchronous compositions of modules that read and write state variables of various (not necessarily finite) types. Thus, an `observer` module can be written as follows.

```
observer: MODULE =
BEGIN
  INPUT
    <state variables>
  OUTPUT
    ok: BOOLEAN
  INITIALIZATION
    ok = TRUE
  TRANSITION
  [
    <property> --> ok' = TRUE
  []
    ELSE --> ok' = FALSE
  ]
END;
```

Here, `<state variables>` represents declaration of the observed state variables and their types, and `<property>` represents a Boolean expression over these state variables that specifies the desired property. The observer sets the Boolean flag variable `ok` to `TRUE` or `FALSE` according to whether the property is satisfied or not: "primed" variables in SAL represent values in the "new" state, and "unprimed" in the "old" state; the symbol `-->` indicates that the assignments are "guarded" by the Boolean expression appearing to its left.

If the system is specified in a module `system` (which may itself be the composition of other modules), then the "observed system" is the synchronous composition of this with the `observer`, which is written as follows (in SAL, the symbol `||` represents synchronous composition).

```
  observed: MODULE = (system || observer);
```

We can then specify the theorem `correctness`, which states that `ok` is always true in the observed system.

```
  correctness: THEOREM observed |- G(ok);
```

Depending on the types of the state variables in `system`, we can examine `correctness` using the symbolic, bounded, or infinite-bounded model checkers of SAL using shell commands such as the following.

```
  sal-smc example.sal correctness

  sal-bmc example.sal correctness -d 17 -it

  sal-inf-bmc example.sal correctness -i -d 2 -ice
```

The first of these invokes SAL's symbolic (BDD-based) model checker on the theorem `correctness` in the file `example.sal`; this will prove the theorem if it is true, or provide a counterexample if it is not (of course, it may also run out of memory or time). The second invokes the bounded (SAT-based) model checker to search for a counterexample up to **d**epth 17, operating **it**eratively (i.e., depth 1, depth 2,...); the third invokes the infinite-bounded (SMT-based) model checker to attempt proof by k-**i**nduction at **d**epth 2 (i.e., 2-induction), and to provide an **i**nductive **c**ounter**e**xample if this fails.

The basic construction illustrated above allows checking of invariants over the state variables of the system; it can be extended to general safety properties, including bounded liveness properties and properties on transitions, by adding new variables to the state of the observer that act as "history variables" to remember the values of system state variables some time in the past.

2.2 Assumptions

Systems are seldom expected to satisfy their specifications in an unconstrained environment; usually there are assumptions about the environment and the system is required to satisfy its specification only in cases where the assumptions are satisfied.

Like properties, assumptions also can be described by synchronous observers and the verification method can be suitably adjusted to ensure that the required properties are satisfied for all those reachable states that satisfy the assumptions.

In SAL, we could use an `assumptions` module, defined in a similar way to the `observer` module in the previous section, but using a flag variable `aok` (for "assumptions OK") whose assignments are guarded by Boolean expressions over the state variables that specify the assumptions. We then form the synchronous composition of the `system`, `assumptions`, and `observer` and state the `requirement` that the correctness property should be true whenever the assumptions are satisfied (the symbol `=>` represents implication in SAL).

```
constrained: MODULE = (system || assumptions || observer);

requirement: THEOREM constrained |- G(aok => ok);
```

Actually, the LTL formula suggested above `G(aok => ok)` raises some interesting issues. Consider the trace of some imaginary system shown below, which displays the values of `aok` and `ok` in the first six steps (where `T` represents *true*, and `F` *false*).

step	1	2	3	4	5	6
aok:	T	T	T	F	T	T
ok:	T	T	T	T	F	T

This fails to satisfy the formula `G(aok => ok)` because `ok` is *false* at step 5 while `aok` is *true*. But `aok` itself was *false* at step 4 and usually we do not care what happens after the assumptions have been violated.

To specify this different requirement in LTL, it is convenient to use the "weak until" operator `W`, which is usually defined in terms of the "strong" variant `U` as follows (where $\vee$ indicates disjunciton):

$$\mathtt{W}(p, q) \stackrel{\text{def}}{=} \mathtt{G}(p) \vee \mathtt{U}(p, q).$$

Intuitively, $\mathtt{U}(p, q)$, which is primitive in most formulations of LTL, requires that q eventually becomes *true*, and that p is *true* until (and possibly beyond) that point; `W` is the same but relaxes the requirement for q to become true if p is invariantly *true*. A subtle point is that LTL formulas are defined only on infinite traces; however, most model checkers extend interpretation of `G` and `W` (but not `F` or `U`) to finite traces. Using the `W` operator, our adjusted requirement can be written in SAL as follows.

```
requirement_alt1: THEOREM constrained |- W(ok, NOT aok);
```

Intuitively, this says that `ok` must be *true* until `aok` is *false*, and it accepts the trace we saw earlier, and also the following one.

step	1	2	3	4	5	6
aok:	T	T	T	F	T	T
ok:	T	T	T	F	F	T

Here, both `aok` and `ok` go *false* at step 4. There are several formulations of compositional reasoning (e.g., [5, 6]) that require the assumptions to fail *before* the property. The trace above does not satisfy this requirement, but the first one does. An LTL formula that specifies the "fails before" requirement uses the strong until operator `U` and is expressed in SAL as follows.

```
requirement_alt2: THEOREM constrained |- NOT U(aok, NOT ok);
```

Most readers will surely agree that it is not easy to see that this formula captures exactly the informal requirement that `aok` fails before `ok`; neither is it straightforward to comprehend the difference between this formula and the earlier one using `W`, nor why the positions of `ok` and `aok` are reversed in the arguments to `W` and `U`.

We are employing a "hybrid" approach here: using synchronous observers to specify properties and assumptions, and temporal logic to combine them. This is rather unnatural and was done to illustrate some of the complexities in writing temporal logic specifications.

Exploiting synchronous observers more fully, it becomes straightforward to say what we mean. First, we adjust the `assumptions` module so that it does nothing if the `<assumption>` is satisfied and "latches" `aok` as soon as it becomes *false*.

```
assumptions: MODULE =
BEGIN
  INPUT
    <state variables>
  OUTPUT
    aok: BOOLEAN
  INITIALIZATION
    aok = TRUE
  TRANSITION
  [
    <assumption> -->
  []
    ELSE --> aok' = FALSE
  ]
END;
```

Then we modify the observer module so that it takes `aok` as an input and latches `ok` to *false* if the desired `<property>` ever goes *false* when `aok` is *true*.

```
observer: MODULE =
BEGIN
  INPUT
    aok: BOOLEAN,
    <state variables>
  OUTPUT
    ok: BOOLEAN
  INITIALIZATION
    OK = TRUE
  TRANSITION
  [
    aok AND NOT <property> --> ok' = FALSE
  []
    ELSE -->
  ]
END;
```

Now we use the model checker simply to verify that `ok` is invariantly *true*.

```
  requirement_simplified: THEOREM constrained |- G(ok);
```

This combination of observers requires the assumptions to fail before the property; if we wish to allow the assumptions to fail at the same time as the property, then we can simply replace the appearance of `aok` in the observer guard by `aok'`.

As always when theorems have the form of an implication, as these implicitly do, it is prudent to check that they are not vacuously *true*: that is, we should check that the antecedent is not invariantly *false*. There is a large literature on the related problem of *vacuity detection* in LTL formulas (e.g., [7]) and the necessary tests become quite difficult. However, this difficulty is a result of the complex LTL formulas used to state the basic property of interest. If we use

synchronous observers of the form described above, then all we need to do is check that each of the positive flag variables can remain *true* and each of the negative ones can remain *false* at least one step beyond its initialization. In the example, this is accomplished by seeking a counterexample to the following formula, which asserts that `aok` is *false* in the second step.

```
check_simple: CLAIM constrained |- X(NOT aok);
```

Of course, an alternative is to prove the positive claim `X(aok)`, but this is computationally more demanding (with a bounded model checker, it requires k-induction).

2.3 Expressivity

Model checkers in which the system is specified by state machines generally provide some way to describe how the values of state variables are updated on a state transition. For example, in SAL, the expression

```
x' = x + y
```

indicates that the "new" value of the state variable `x` is the sum of the current values of itself and the state variable `y`. Nondeterministic assignments are often supported as well, as in the following example, where `x` is nondeterministically assigned a value between 25 and 50, inclusive.

```
x' IN { a: nat | a >= 25 AND a <= 50 }
```

Now, suppose we wish to specify that the new value of `x` can be any value *larger* than its current value. In SAL we could write

```
x' IN { a: nat | a > x }
```

but not all model checkers provide this expressivity. Another option, available in those languages that provide guarded commands is the following.

```
(x' > x) --> x' IN { a: nat | TRUE }
```

But notice this requires a primed variable to appear in the guard, and not all model checker state machine languages allow this.

In this simple case, an alternative would be to use an auxiliary variable that is nondeterministically set to the amount by which `x` should be incremented. However, this does not solve the general problem, which is that of updating (possibly several variables of) the state so that some *constraint* is satisfied—such as to nondeterministically update real variables `x` and `y` so that they lie on a unit circle (i.e., `x*x + y*y = 1`).

However, one method that is almost always feasible uses a synchronous observer. Returning to the simple example of nondeterministically incrementing `x`: in the main system specification, we make an unconstrained nondeterministic assignment to `x` as follows.

```
x' IN { a: nat | TRUE }
```

Then, in a synchronous observer module, we enforce the desired relation using `cok` (for "constraints OK") as our flag variable as follows.

```
TRANSITION
[
  (x' > x) -->
[]
  ELSE --> cok' = FALSE
]
```

If the language does not allow primed variables in the guards, then we will need to introduce a history variable `oldx` to remember the previous value of `x`.

```
TRANSITION
oldx' = x;
[
  (x > oldx) -->
[]
  ELSE --> cok' = FALSE
]
```

Then we model check for whatever property `p` we had in mind, but only in cases where `cok` is *true*.

```
constrained_prop: THEOREM (system || constraints) |- G(cok => p)
```

Or, if the required property is specified by a synchronous observer with flag variable `ok`, we would use the following variant.

```
constrained_req: THEOREM
  (system || observer || constraints) |- G(cok => ok)
```

Rather than the explicit implication `G(cok => ok)`, we can also use the methods of the previous section. Note that if we are using a history variable in the constraints module, then the property `p` or flag `ok` must also be defined in a similar way, or should reference `oldx` rather than `x`, as otherwise the property will be out of step with the constraint.

Sometimes, we may we wish to consider only traces in which the constraints are satisfied globally. For example, in the following trace the constraint is violated at step 6, but it may be that some earlier decisions made this violation inevitable.

step	1	2	3	4	5	6
cok:	T	T	T	T	T	F
ok:	T	T	T	F	F	T

Hence, although the required property is violated at steps 4 and 5, we consider this entire scenario invalid because the constraint is not globally *true*. If desired, although it is seldom appropriate, we can specify this interpretation as follows (and, of course, we can do the same for other kinds of assumptions as well).

```
globally_constrained_req: THEOREM
  (system || observer || constraints) |- G(cok) => G(ok)
```

Note that this is a liveness property and cannot be verified by bounded model checkers, although they can find counterexamples.

Using observers to specify constraints is especially useful when we wish to update multiple variables in a way that enforces a *relation* on them as, for example, the case mentioned earlier of points constrained to lie on a circle. This finds particular application in specifying *relational abstractions* for hybrid automata [8]. Unlike other methods for abstracting hybrid automata, which typically abstract the state space, relational abstraction retains the state space (i.e., continuous variables continue to range over the reals) but simplifies the transition relation.

Generally, we start with a true hybrid automaton (i.e., a state machine plus differential equations) and calculate a relational abstraction in the manner described by Sankaranarayanan and Tiwari and mechanized in the HybridSAL Relational Abstractor [9]. But another approach, suitable when we have or need only a crude model of the dynamical system, is to *assert* a relational abstraction as the model. An example is described in [10]; there, the basic task is analysis of human-machine interaction and only crude models of the aircraft automation and dynamics are needed, such as "when automation is in a climb mode, the pitch angle must be positive" and "when the pitch angle is positive, the altitude increases." These are specified in a constraints module as follows.

```
INITIALIZATION
   cok = TRUE;
TRANSITION
[  actual_mode = op_des AND pitch > 0 --> cok' = FALSE;
[] actual_mode = op_clb AND pitch < 0 --> cok' = FALSE;
[] actual_mode = vs_fpa AND fcu_fpa <= 0 AND pitch > 0
    --> cok' = FALSE;
[] actual_mode = vs_fpa AND fcu_fpa >= 0 AND pitch < 0
    --> cok' = FALSE;
[] pitch > 0 AND altitude' < altitude --> cok' = FALSE;
[] pitch < 0 AND altitude' > altitude --> cok' = FALSE;
[] pitch=0 AND altitude' /= altitude --> cok' = FALSE;
[] ELSE -->
] END;
```

Observe that each guard is the *negation* of a desired constraint (e.g., the first guard is the negation of the natural constraint `actual_mode = op_des => pitch <= 0`). This is because we generally require *all* assumptions or constraints to be satisfied—i.e., they are conjoined together—whereas guarded commands are disjoined. Hence, we apply De Morgan's rule and disjoin the negations. An alternative is to conjoin all the (unnegated) constraints together in the guard of a single command.

This subsection has shown how synchronous observers provide expressivity that can assist in the construction of *system models*. This is a different topic than their expressivity with respect to the classes of *properties* that can be

specified, which will be considered in Section 3. In the following subsection, we continue our examination of the use of synchronous observers in the construction of system models by considering the case of very abstract models that employ uninterpreted functions.

2.4 Axioms for Uninterpreted Functions

Traditional model checkers require the system model to be totally explicit—effectively, a program or hardware circuit—and this is one of the reasons that interactive theorem provers are preferred for some verification tasks. For reasons of efficiency or generality, we often wish to abstract some parts of a design prior to verification. One way to do this is to replace parts of the design by a nondeterministic component (this is feasible with traditional model checkers) but a more general and attractive method is to use *uninterpreted functions* constrained by suitable axioms.

One standard example in model checking is to verify correctness of the bypass logic in a model of a processor pipeline [11]. The pipeline feeds values into an arithmetic logic unit (ALU) and a standard way to verify its correctness is to prove that the outputs of the ALU are the same in a processor design with and without the pipeline. In traditional model checking we need to provide some implementation for the ALU; if this is fully accurate the verification complexity may be very high, but if it is simplified (e.g., every operation is an addition) some flaws in the bypass logic may be masked by the simplification (e.g., a flaw that transposes arguments will be masked by the commutativity of addition); and if the simplification is excessive (e.g., nondeterministic) then the property may become unverifiable. A much more attractive solution is to model the ALU by an uninterpreted function `ALU(x, y)`: that is, a function about which we know nothing. If we did wish the function to be commutative for some reason, we would add the following axiom.

$$\forall \mathtt{x}, \mathtt{y} : \mathtt{ALU(x,\ y)} = \mathtt{ALU(y,\ x)}$$

First order logic provides uninterpreted functions and, when restricted to the unquantified case (i.e., all variables are implicitly universally quantified), the theory of uninterpreted functions with equality is decidable. Interactive theorem provers provide uninterpreted functions and often a decision procedure to automate the unquantified case and, for this reason (among others), they may be preferred to conventional (i.e., finite-state or explicit-state) model checkers for some verification tasks.

However, the theory of uninterpreted functions is one of the core theories automated in modern solvers for satisfiability modulo theories (SMT) [12] and bounded model checking can be generalized to use these solvers, yielding *infinite* bounded model checkers, abbreviated as inf-BMC [13] ("infinite" because SMT includes theories of infinite cardinality such as the integers and rationals).

Inf-BMC blurs the line between model checking and theorem proving and is widely used for verification of models that use linear arithmetic, arrays, and

other theories decided by SMT. It would also be very attractive to incorporate uninterpreted functions into inf-BMC models, but we must then find a way to convey any axioms about the functions to the underlying SMT solver. By now, readers will not be surprised to learn that synchronous observers provide a way to do this. The method is basically the same as that described in the previous section.

For example, suppose we have a system with one integer-valued state variable called `count` and that the behavior of the system is simply to apply an uninterpreted function `f` to this variable at each step. This can be specified in SAL as follows.

```
f(x: int): int;

system: MODULE =
BEGIN
  OUTPUT
    count: int
  INITIALIZATION
    count = 0
  TRANSITION
    count' = f(count)
END;
```

Now suppose we wish to assert the axiom $\forall$`x` : `f(x)` $\geq$ `x` and then prove that `count` is always non-negative. We introduce a synchronous observer called `constraints` that does nothing as long as the axiom about `f` is satisfied, but sets the flag variable `cok` to *false* when it is violated.

```
constraints: MODULE =
BEGIN
  INPUT
    count: integer
  OUTPUT
    cok: BOOLEAN
  INITIALIZATION
    cok = TRUE
  TRANSITION
  [
    f(count) >= count -->
  []
    ELSE --> cok' = FALSE
  ]
END;
```

We then synchronously compose the system and the observer and state the theorem that `count` is non-negative provided the axiom flagged by `cok` is satisfied.

```
  nonneg: THEOREM (system || constraints) |- G(cok => count >= 0);
```

The SAL inf-BMC can prove this by 1-induction.

```
sal-inf-bmc increments.sal nonneg -i -d 1
```

If we modify the guard encoding the axiom to read as follows, so that `f(x)`

```
f(count) >= count - 375 -->
```

can be less than `x`, then the SAL inf-BMC constructs a 1-step counterexample in which `f(0) = -1`.

This ability of SMT solvers, and hence of inf-BMC, to construct witnesses for uninterpreted functions is very useful: it helps us to discover necessary constraints, as described in the following section.

2.5 Discovering Assumptions

The previous sections have described how synchronous observers can be used to specify assumptions, axioms, and constraints. The descriptions assume we know these beforehand and merely need to formalize them in a way that is feasible and effective for model checking. A variant problem is that of *discovering* suitable assumptions and constraints. Synchronous observers are very convenient for this purpose. We can start with an "empty" assumptions observer of the following form, where `aalarm` is a flag for "assumption alarm" and is set *true* when the assumptions are violated. (It is "empty" because the command with guard `FALSE` can never be taken.) Incidentally, we are using a "negative" flag variable here simply for variety.

```
assumptions: MODULE =
BEGIN
  OUTPUT
    aalarm: BOOLEAN
  INPUT
      <state variables>
  INITIALIZATION
    aalarm = FALSE
  TRANSITION
  [
  assumption_violation:
    FALSE --> aalarm' = TRUE
  assumptions_ok:
    [] ELSE -->
  ]
END;
```

Then we model check for the property of interest `p` when `aalarm` is *false*.

```
learn_assumptions: LEMMA (system||assumptions) |- G((NOT aalarm) => p)
```

If this formula is violated, the counterexample should suggest a missing assumption, which we add to the assumptions module (below the `FALSE` guard).

```
[] NOT <new assumption> --> aalarm' = TRUE
```

We proceed in this way until all necessary assumptions have been discovered. The use of counterexamples to guide discovery of assumptions can be employed no matter how the assumptions are represented. But this form of synchronous observer is a particularly attractive representation because it allows each newly discovered assumption to be added as a new guard: it is truly incremental. By contrast, a more tightly integrated representation of the assumptions might require substantial revision at each step.

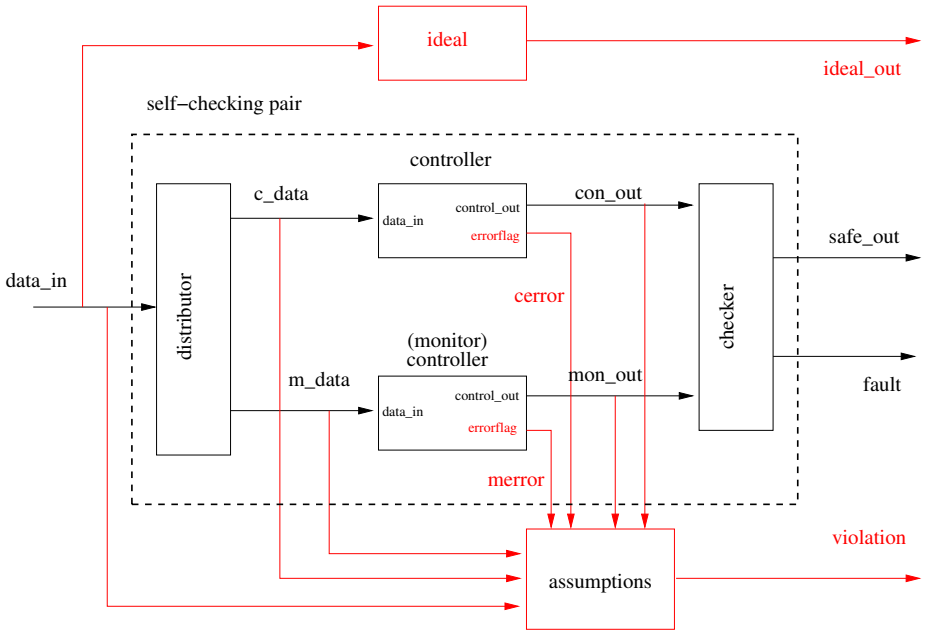

Fig. 1. "Box and Arrow" Diagram of a Self-Checking Pair

This approach combines very well with that of the previous subsection. Uninterpreted functions are very attractive for the highly abstract modeling that is appropriate for the upper levels of system design, where systems are often represented by "box and arrow" diagrams. Flaws at this level of design are a major cause of incidents in aircraft software [14]. Traditionally, it has been difficult to apply any mechanized analysis to this level of description, so often they are prematurely "prototyped" in a simulation environment like Simulink and the prototype then becomes the requirement. However, uninterpreted functions can be used to provide suitably abstract semantics for a box and arrow diagram and the methods of this and the previous subsection can then be used to analyze it and to discover its properties and required assumptions.

An example, taken from [15] and shown in Figure 1 illustrates this. Here, the goal is to deduce the assumptions under which a "self-checking pair" works correctly. Self-checking pairs are used quite widely in safety-critical systems to provide protection against random hardware faults: two identical controllers perform the same calculations and their results are compared; if they differ the pair shuts down (thereby becoming a "fail-stop" processor [16]) and some higher-level fault management activity takes over. Obviously, this does not work if both controllers become faulty and compute the *same* wrong result. We would like to learn if there are any other scenarios that can cause a self-checking pair to deliver the wrong result; we can then assess their likelihood (for example, the double fault scenario just described may be considered extremely improbable) and calculate the overall reliability of this architecture.

In the SAL model corresponding to the box and arrow diagram shown in Figure 1, the `controller`s simply compute some uninterpreted function `f` of their inputs, unless they are faulty—in which case the produce some nondeterministic (but incorrect) output. The components shown in red are synchronous observers (the arrows represent the state variables that they observe). The `ideal` box serves as a correctness specification: it computes the same function `f` as the real controllers but never fails. The requirement is that if the (as yet undetermined) assumptions are not violated, and if the checker component does not signal a fault, then the output of the self-checking pair should be the same as that of the ideal controller.

Among the assumptions discovered (see [15] or [17] for fuller descriptions) is one that says a faulty distributor component must not relay different, incorrect values `x` and `y` to the two controllers such that `f(x) = f(y)` (`x` and `y` correspond to `c_data` and `m_data` in the diagram). This would be a Byzantine fault on the part of the distributor (this can occur—even when the implementation of the distributor is as simple as a solder joint—if voltages or timing are close to their boundaries) and is unlikely to be discovered in simulation experiments. This is because we would first have to anticipate the possibility of Byzantine faults and build this into the simulation model for the distributor, and would also have to supply some concrete instantiation for `f(x)` (e.g., `x+1`) and are unlikely to choose one that can produce the same output for different inputs. In contrast, the SMT solver underneath inf-BMC synthesizes whatever behavior of the distributor and whatever instantiation of `f` are needed to construct a counterexample.

2.6 Test Cases

All the applications of synchronous observers that we have seen so far concern their use in verification. A quite different application is their use in test generation. It is well-known that model checkers can be used to construct test cases: if we seek a test characterized by a property `p` then we model check for `G(NOT p)` and the counterexample provides a test case.

One difficulty in exploitation of this idea is to decide what properties `p` correspond to good test cases, and how to specify such `p`. Often, tests are based on structural coverage of the source program or specification: that is, we aim

to find a suite of tests that exercises each statement or branch (or visits each state or transition). This can be accomplished by adding "trap variables" that are set *true* when some coverage target is encountered, and then using these variables in definition of `p` [18]. Unfortunately, model checkers are so good at finding counterexamples that the tests produced by this method are often short and very similar to each other [19].

A better approach uses a synchronous observer to indicate whether the trace seen so far satisfies some "test purpose." The observer merely has to *recognize* tests satisfying its purpose, then raise a flag `tok` (for "test OK"). We then model check for `G(NOT tok)` and the model checker effectively performs constraint satisfaction to generate tests satisfying that purpose. This approach is very effective. Several examples are given in the manual for the SAL test generator `sal-atg` [20]. One concerns the "shift scheduler" for the automatic gearbox of a car.

The inputs to this component are `torque`, `velocity`, and `gear`; its outputs drive actuators that change clutch pressures and thereby influence the gearbox to select a different gear. The goal of test generation in this example is to find sequences of inputs that drive the state machine of the shift scheduler through all its transitions and this is easily accomplished by `sal-atg`. However, the test cases have many "discontinuities" in the `gear` input: that is, the currently selected gear may go from 2 to 4 to 1 in successive inputs. We might suppose that a more realistic test sequence would not have these discontinuities, and therefore propose a test purpose in which the `gear` input changes by at most one at each step. We can implement this purpose by adding the following observer to the SAL specification of the shift scheduler.

```
purpose: MODULE =
BEGIN
  INPUT
   gear: [1..4]
  OUTPUT
   continuous: BOOLEAN
  INITIALIZATION
   continuous = (gear=1);
  TRANSITION
   continuous' = continuous AND (gear - gear' <= 1)
    AND (gear' - gear <= 1);
END;

monitored_system: MODULE = (scheduler || purpose);
```

Here, the `purpose` module takes `gear` as input and produces the Boolean output `continuous`: this output remains `TRUE` as long as the sequence of inputs changes by at most 1 at each step (and starts at 1). The `purpose` module is then synchronously composed with the existing `scheduler` to yield the `monitored_system`. We then repeat test generation, but indicate to `sal-atg` that the flag `continuous`, representing the test purpose, must remain *true*. It

turns out that the test generated holds the `gear` input constant for long periods (e.g., the first ten inputs that it generates are 1, 1, 1, 1, 1, 1, 2, 3, 3, 3) so we might adjust the test purpose to additionally require that the `gear` input *always* changes value from one step to the next. It is easy to add this to the `purpose` module and we then obtain a single test of length 51 that discharges all the coverage goals while satisfying the enlarged test purpose (the first ten `gear` inputs are 1, 2, 3, 2, 3, 2, 3, 2, 3, 2).

3 Discussion and Conclusion

I hope the examples in this paper serve to alert readers to the utility and versatility of synchronous observers. The first examples that we considered were focused on the use of observers to specify properties and assumptions. The main advantage that we claim for this application of synchronous observers is convenience: it is generally easier and less error-prone to specify properties and assumptions in this way than to write temporal logic formulas. But convenience aside, do we give up any expressiveness in using synchronous observers rather than temporal logic?

In the "hybrid" case where we use observers to define flag variables and temporal logic to combine them, the answer is obviously "no," because we have the full resources of both methods at our disposal. So let us focus on the case where flag variables are used only in formulas of the form `G(ok)` and `G(aok => ok)`, and compare these to general temporal logic formulas over "natural" state variables (i.e., state variables that are intrinsically part of the system model, not those introduced as observers). The main loss with synchronous observers is that they are restricted to safety properties and therefore cannot specify general liveness properties. However, most applications are not concerned with possibilities in the indefinite future (e.g., "every request eventually receives a response"), but with explicit bounds (e.g., "every request receives a response within 8 steps, or returns an error"), and these are safety properties.

On the other hand, synchronous observers can specify properties that LTL cannot (an example is "p is true on every alternate state"). Industrial specifications languages such as the Accellera/IEEE Property Specification Language (PSL) [21] and SystemVerilog Assertions (SVA) [22] extend LTL with regular expressions and thereby achieve approximate expressive parity with synchronous observers.[1] CTL and LTL are mutually incomparable and some properties that are in CTL but not LTL may also be beyond the reach of synchronous observers (because CTL can specify nondeterministic possibilities whereas synchronous observers monitor single threads).

In general, it is safe to assume that synchronous observers have approximately the same expressive power as industrial assertion languages based on

[1] We say "approximate" because, although regular expressions are equivalent to finite automata, the comparison here is complicated by the presence of state variables ranging over possibly infinite types and constrained by theories and, in the case of SVA, local variables.

LTL, when the latter are restricted to safety properties. Hence, selecting between the two approaches can be based on user preferences, tool capabilities, and overall workflows, rather than fundamental limitations.

The later examples that we considered focussed on the use of observers to enhance the class of system models that can be defined conveniently. These include models where updates to state variables must maintain some constraint, and those where axioms are applied to uninterpreted functions. The methods used in these examples were later used to facilitate the discovery of suitable constraints, and the specification of test purposes.

The idea underlying these applications is that the basic system model generates more behaviors than are desired, the synchronous observer recognizes those that are "good" (or "bad," depending on the parity) and raises a flag appropriately, and the model checker is constrained to scenarios where the flag is raised. The value in this approach derives from the fact that it is usually much easier to specify systems that recognize desired behavior than those that generate it. Of course, the approach is "inefficient" in that the basic system model generates many scenarios that are rejected and "thrown away" by the observer and this may make it unsuitable for explicit-state model checkers, which really do have to enumerate all behaviors. But there is no comparable penalty when using symbolic model checkers, whether based on BDDs, SAT, or SMT: the observer simply adds to the constraints that must be solved by the underlying symbolic method.

Despite their versatility, synchronous observers are intuitive and easy to use: the system, its requirements, assumptions, axioms, and test plans are all written using the same state-machine notation, and this reduces the learning burden for verification and test engineers. Furthermore, it eliminates the need to use a property specification language (apart from "canned" formulas such as `G(p)`): in traditional model checking, the need to learn a property language based on temporal logic is often a major obstacle to adoption and effective use.

Synchronous observers closely correspond to runtime monitors for executable programs, and this also is a familiar concept to most engineers. Furthermore, assumptions for high-level models, possibly discovered in the manner described in Section 2.5, can provide the basis for runtime monitors that deliver significant benefit in system reliability [23]. These monitors could even be formally synthesized directly from the model and this is an attractive direction for future research.

Acknowledgments. Some of this material was developed for a talk at JAIST in 2007 and I am grateful to Kokichi Futatsugi for facilitating that visit and for the many interesting discussions that we have enjoyed since first meeting as visitors to SRI in the early 1980s.

This paper is based on an invited talk presented at the 15th Brazilian Symposium on Formal Methods (SBMF) in Natal on 25 September 2012. The abstract was published in the proceedings of the symposium [24].

I am grateful to Ashish Tiwari for showing me several ways to represent the constraints for relational abstractions in a model checker, to N. Shankar for helpful discussions on observers and LTL, and to Moshe Vardi for information on industrial specification languages. Comments by the anonymous referees helped improve the presentation.

This work was supported by NASA under contracts NNA13AB02C with Drexel University and NNL13AA00B with the Boeing Company, and by SRI International. The content is solely the responsibility of the author and does not necessarily represent the official views of NASA.

References

1. Halbwachs, N., Lagnier, F., Ratel, C.: Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. IEEE Transactions on Software Engineering **18** (1992) 785–793 2
2. Halbwachs, N., Lagnier, F., Raymond, P.: Synchronous observers and the verification of reactive systems. In: Algebraic Methodology and Software Technology (AMAST93). Workshops in Computing. Springer-Verlag, Enschede, The Netherlands (1994) 83–96 2
3. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: International Conference on Software Engineering, Los Angeles, CA, IEEE Computer Society (1999) 411–420 2
4. SAL home page: http://sal.csl.sri.com/ 3
5. McMillan, K.L.: Circular compositional reasoning about liveness. In Pierre, L., Kropf, T., eds.: Advances in Hardware Design and Verification: IFIP WG10.5 International Conference on Correct Hardware Design and Verification Methods (CHARME '99). Volume 1703 of Lecture Notes in Computer Science., Bad Herrenalb, Germany, Springer-Verlag (1999) 342–345 5
6. Rushby, J.: Formal verification of McMillan's compositional assume-guarantee rule. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA (2001) 5
7. Kupferman, O., Vardi, M.Y.: Vacuity detection in temporal model checking. International Journal on Software Tools for Technology Transfer **4** (2003) 224–233 6
8. Sankaranarayanan, S., Tiwari, A.: Relational abstractions for continuous and hybrid systems. In: Computer-Aided Verification, CAV '2011. Volume 6806 of Lecture Notes in Computer Science., Snowbird, UT, Springer-Verlag (2011) 686–702 9
9. Tiwari, A.: HybridSAL relational abstracter. In: Computer-Aided Verification, CAV '2012. Volume 7358 of Lecture Notes in Computer Science., Berkeley, CA, Springer-Verlag (2012) 725–731 9
10. Bass, E.J., Feigh, K.M., Gunter, E., Rushby, J.: Formal modeling and analysis for interactive hybrid systems. In: Fourth International Workshop on Formal Methods for Interactive Systems: FMIS 2011. Volume 45 of Electronic Communications of the EASST, Limerick, Ireland (2011) 9
11. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation **98** (1992) 142–170 10

12. de Moura, L., Rueß, H.: Lemmas on demand for satisfiability solvers. In: Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02), Cincinnati, OH (2002) 10
13. de Moura, L., Rueß, H., Sorea, M.: Lazy theorem proving for bounded model checking over infinite domains. In Voronkov, A., ed.: 18th International Conference on Automated Deduction (CADE). Volume 2392 of Lecture Notes in Computer Science., Copenhagen, Denmark, Springer-Verlag (2002) 438–455 10
14. Rushby, J.: New challenges in certification for aircraft software. In Baruah, S., Fischmeister, S., eds.: Proceedings of the Ninth ACM International Conference On Embedded Software: EMSOFT, Taipei, Taiwan, Association for Computing Machinery (2011) 211–218 13
15. Rushby, J.: A safety-case approach for certifying adaptive systems. In: AIAA Infotech@Aerospace Conference, Seattle, WA, American Institute of Aeronautics and Astronautics (2009) AIAA paper 2009-1992 14
16. Schlichting, R.D., Schneider, F.B.: Fail-stop processors: An approach to designing fault-tolerant computing systems. ACM Transactions on Computer Systems **1** (1983) 222–238 14
17. Rushby, J.: Composing safe systems. In: 8th International Symposium on Formal Aspects of Component Software (FACS'11). Volume 7253 of Lecture Notes in Computer Science., Oslo, Norway, Springer-Verlag (2011) 14
18. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. In Nierstrasz, O., Lemoine, M., eds.: Software Engineering—ESEC/FSE '99: Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering. Volume 1687 of Lecture Notes in Computer Science., Toulouse, France, Springer-Verlag (1999) 146–162 15
19. Hamon, G., de Moura, L., Rushby, J.: Generating efficient test sets with a model checker. In: 2nd International Conference on Software Engineering and Formal Methods (SEFM), Beijing, China, IEEE Computer Society (2004) 261–270 15
20. Hamon, G., de Moura, L., Rushby, J.: Automated test generation with SAL. Technical note, Computer Science Laboratory, SRI International, Menlo Park, CA (2005); http://www.csl.sri.com/users/rushby/abstracts/sal-atg. 15
21. IEEE Stanadard 1850–2010: Property Specification Language (PSL) (2010) 16
22. IEEE Standard 1800–2012: SystemVerilog—Unified Hardware Design, Specification, and Verification Language. (2012) 16
23. Littlewood, B., Rushby, J.: Reasoning about the reliability of diverse two-channel systems in which one channel is "possibly perfect". IEEE Transactions on Software Engineering **38** (2012) 1178–1194 17
24. Rushby, J.: The versatile synchronous observer (abstract only). In Gheyi, R., Naumann, D., eds.: 15th Brazilian Symposium on Formal Methods: Foundations and Applications (SBMF). Volume 7498 of Lecture Notes in Computer Science., Natal, Brazil, Springer-Verlag (2012) 1 17