In E. A. Emerson and A. P. Sistla, editors, Computer-Aided Verification, CAV '2000, volume 1855 of Lecture Notes in Computer Science, pages 508-520, Chicago, IL, July 2000.

Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification*

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
rushby@csl.sri.com

Abstract. I describe a systematic method for deductive verification of safety properties of concurrent programs. The method has much in common with the "verification diagrams" of Manna and Pnueli [17], but derives from different intuitions. It is based on the idea of strengthening a putative safety property into a disjunction of "configurations" that can easily be proved to be inductive. Transitions among the configurations have a natural diagrammatic representation that conveys insight into the operation of the program. The method lends itself to mechanization and is illustrated using a simplified version of an example that had defeated previous attempts at deductive verification.

1 Introduction

In 1997, Shmuel Katz, Patrick Lincoln and I presented an algorithm for Group Membership together with a detailed, but informal proof of its correctness [14]. Shortly thereafter, our colleague Shankar and, independently, Sadie Creese and Bill Roscoe of Oxford University, noted that the algorithm is flawed when the number of nonfaulty processors is three. Model checking a downscaled instance can be effective in finding bugs (that is how Creese and Roscoe found the problem in our algorithm [8]), but true assurance for a potentially infinite-state n-process algorithm such as this seems to require (mechanically checked) deductive methods—either direct proof or justification of an abstraction that can be verified by algorithmic means. Over the next year or so, Katz, Lincoln and I each made several attempts to formalize and mechanically verify a corrected version of the algorithm using the PVS verification system [19]. On each occasion, we were defeated by the number and complexity of the auxiliary invariants needed, and by the "case explosion" that bedevils deductive approaches to formal verification.

Eventually, I stumbled upon the method presented in this paper and completed the verification in April 1999 [23]. This new method made the verification not merely possible, but easy, and it provides a visual representation that conveys considerable insight

* This research was supported by DARPA through USAF Rome Laboratory Contract F30602-96-C-0204 and USAF Electronic Systems Center Contract F19628-96-C-0006, and by the National Science Foundation contract CCR-9509931.

into the operation of the algorithm. Holger Pfeifer of the University of Ulm was subsequently able to use the method to verify a related but much more complicated group membership algorithm [21] used in the Time Triggered Architecture for critical real-time control [15]

I later discovered that my method has much in common with the "verification diagrams" introduced by Manna and Pnueli [17], and subsequently generalized by Manna and several colleagues [5, 7, 10, 16]. However, the intuition that led to my method is rather different than that for verification diagrams, as is the way I approach its mechanization. I hope that by revisiting these methods from a slightly different perspective, I will help others to see their value and to investigate their application to new problems.

I describe my method in the next section and present an example of its application in the one after that. The final section compares the method with verification diagrams and with other techniques and provides conclusions and suggestions for further work.

2 The Method

Concurrent systems are modeled as nondeterministic automata over possibly infinite sets of states. Given set of states S, initiality predicate I on S, and transition relation T on S, a predicate P on S is *inductive* for $\mathcal{S} = (S, I, T)$ if

$$I(s) \supset P(s)^1 \tag{1}$$

and

$$P(s) \wedge T(s,t) \supset P(t). \tag{2}$$

The *reachable states* are those characterized by the smallest (ordered by implication) inductive predicate R on $\mathcal{S}$. A predicate G is an *invariant* or *safety property* if it is larger than R (i.e., includes all reachable states). The focus here is on safety (as opposed to liveness) properties, so we do not need to be concerned with the acceptance criterion on the automaton $\mathcal{S}$.

The deductive method for verifying safety properties attempts to establish that a predicate G is invariant by showing that it is inductive—i.e., we attempt to prove the verification conditions (1) and (2) with G substituted for P. The problem, of course, is that many safety properties are not inductive, and must be strengthened (i.e., replaced by a smaller property) to make them so. Typically, this is done by conjoining additional predicates in an incremental fashion, so that G is replaced by

$$G^i_\wedge = G \wedge G_1 \wedge \cdots \wedge G_i \tag{3}$$

until an inductive $G^m_\wedge$ is found. This process can be made systematic, but is always tedious. In one well-known example, 57 such strengthenings were required to verify a communications protocol [12]; each G_{i+1} was discovered by inspecting a failed proof for inductiveness of $G^i_\wedge$, and the process consumed several weeks.

[1] Formulas are implicitly universally quantified in their free variables; the horseshoe symbol $\supset$ denotes logical implication.

Some improvements can be made in this process: static analysis [4] and automated calculations of (approximations to) fixpoints of weakest preconditions or strongest post-conditions [5] can discover many useful invariants that can be used to seed the process as $G_1, \ldots, G_i$. Nonetheless, the transformation of a desired safety property into a provably inductive invariant remains the most difficult and costly element in deductive verification, and systematic methods are sorely needed.

The method proposed here is based on strengthening a desired safety property with a *disjunction* of additional predicates, rather than the *conjunction* appearing in (3). That is, we construct

$$G_{\vee}^{m} = G \wedge (G_1 \vee \cdots \vee G_m)$$

instead of $G_{\wedge}^{m}$. Obviously, this can be rewritten as follows

$$G_{\vee}^{m} = (G \wedge G_1) \vee \cdots \vee (G \wedge G_m).$$

Rather than form each disjunct as a conjunction $(G \wedge G_i)$, it is generally preferable to use

$$G_{\vee}^{m} = G'_1 \vee \cdots \vee G'_m \tag{4}$$

and then prove $G'_i \supset G$ for each G'_i. The subexpressions G'_i are referred to as *configurations*, and the indices i as *configuration indices*.

Observe that in the construction of $G_{\wedge}^{m}$, each G_i must be an invariant (the very property we are trying to establish), and that the inadequacy of $G_{\wedge}^{i}$ only becomes apparent through failure of the attempted proof of its inductiveness—and proof of the putative inductiveness of $G_{\wedge}^{i+1}$ must then start over.[2] In contrast, the configurations used in construction of $G_{\vee}^{m}$ need not themselves be invariants, and can be discovered in a rather systematic manner. To see this, first suppose that $G_{\vee}^{m}$ is inductive, and consider the proof obligations needed to establish this fact. Instantiating (2) with $G_{\vee}^{m}$ of (4) and case-splitting across the configurations, we will need to prove a verification condition of the following form for each configuration index i:

$$G'_i(s) \wedge T(s,t) \supset G'_1(t) \vee \cdots \vee G'_m(t).$$

We can further case-split on the right of the implication by introducing predicates $C_{i,j}(s)$ called *transition conditions* such that, for each configuration index i

$$\forall s \in S : \bigvee_j C_{i,j}(s) \tag{5}$$

(here j ranges over the indices of the transition conditions for configuration G'_i) and

$$G'_i(s) \wedge T(s,t) \wedge C_{i,j}(s) \supset G'_j(t) \tag{6}$$

for each transition condition $C_{i,j}$ of each configuration G'_i. Note that some of the $C_{i,j}$ may be identically *false* (so that the proof obligation (6) is vacuously true for this case) and that it is not necessary that the $C_{i,j}$ for different j be disjoint.

[2] PVS attempts to lessen the amount of rework that must be performed in this situation by allowing conjectures to be modified during the course of a proof; such proofs are marked provisional until a final "clean" verification is completed.

This construction can be represented in a diagrammatic form called a *configuration diagram* such as that shown several pages ahead in Figure 1. Here, each vertex represents a configuration and is labeled with the name of the corresponding formula G'_i and each arc represents a non-*false* transition condition and is labeled with a phrase that suggests the corresponding predicate. To verify the diagram, we need to show that the initiality predicate implies some disjunction of configurations

$$I(s) \supset G'_1(s) \vee \cdots \vee G'_m(s) \tag{7}$$

(typically there is just a single *starting configuration*), that each configuration implies the desired safety property

$$G'_1(s) \vee \cdots \vee G'_m(s) \supset G(s), \tag{8}$$

that the disjunction of the transition conditions leaving each configuration is *true* (i.e., (5)), and that the transition relation indeed relates the configurations in the manner shown in the diagram (i.e., the verification conditions (6)). Notice that this is just a new way of organizing a traditional deductive invariance proof (i.e., the proof obligations (5)–(8) imply (1) and (2) with G substituted for P). And although a configuration diagram has some of the character of an abstraction, its verification involves only the original model, and no new verification principles are involved.

The previous discussion assumed we already had a configuration diagram; in practice, the diagram is constructed incrementally in the course of the proof. To construct a configuration diagram, we start by inventing a starting configuration and checking that it is implied by the initiality predicate and implies the safety property (i.e., proof obligations (7) and (8)). Then, by contemplation of the algorithm (the guard predicates and other case-splits in the specification are good guides here), we invent some transition conditions for the starting configuration and check that their disjunction is true (i.e., proof obligation (5)). For each transition condition, we symbolically simulate a step of the algorithm from the starting configuration, under that condition. The result of symbolic simulation becomes a new configuration (and implicitly discharges proof obligation (6) for that case)—unless we recognize it as a variant of an existing configuration, in which case we must explicitly discharge proof obligation (6) by proving that the result of symbolic simulation implies the existing configuration concerned (sometimes it may be necessary to generalize an existing configuration, in which case we will need to revisit previously-proved proof obligations involving this configuration to ensure that they are preserved by the generalization). We also check that each new or generalized configuration implies the safety property (i.e., proof obligation (8)). This process is repeated for each transition condition and each new configuration until the diagram is closed. The creative steps are the selection of transition conditions, and recognition of new configurations as variants of existing ones. Neither of these is hard, given an informal understanding of the algorithm being verified, and the resulting diagram not only verifies the desired safety property (once all its proof obligations are discharged), but it also serves to explain the operation of the algorithm in a very effective way. Bugs in the algorithm, or unfortunate choices of configurations or of transition conditions, will be manifested as difficulty in closing the diagram (typically, the result of a symbolic simulation step will not imply the expected configuration). As with most deductive methods, it can be tricky to distinguish between these causes of failure.

3 An Example: Group Membership

A simplified version of the group membership algorithm mentioned earlier [14] will serve as an example. There are n processors numbered $0, 1, \ldots, n-1$ connected to a broadcast bus; a distributed clock synchronization algorithm (not discussed here) provides a global clock that ticks off "slots" $0, 1, 2, \ldots$ In slot i it is the turn of processor $i \bmod n$ to broadcast. The broadcast contains a message, not considered here, and the `ack` bit of the broadcasting processor, which is described below. Processors may be faulty or nonfaulty; those that are faulty may be *send-faulty*, *receive-faulty*, or both. A processor that is send-faulty will fail to send its broadcast message in its first slot after it becomes faulty; thereafter it may or may not broadcast in its slots. A processor that is receive-faulty will fail to receive the first broadcast from a nonfaulty processor after it becomes faulty; thereafter it may or may not receive broadcasts. Notice that faults affect only communications: a faulty processor still executes the algorithm correctly; additional elements in the full protocol suite ensure that other kinds of faults are manifested as "fail silence," which appears to the algorithm described here as a combined send- and receive-fault in the processor concerned.

Each processor maintains a *membership set* which contains all and only the processors that it believes to be nonfaulty. Processors broadcast in their slots only if they are in their own membership sets. The goal of the algorithm is to maintain accurate membership sets: all nonfaulty processors should have the same membership sets (this is the *agreement* property) and those membership sets should contain all the nonfaulty processors and at most one faulty one (this is the *validity* property; it is necessary to allow one faulty processor in the membership because it takes time to diagnose a fault). These safety properties must be ensured subject to the *fault arrival hypothesis* that faults do not arrive closer than n slots apart. Initially all processors are nonfaulty, their membership sets contain all processors, and their `ack` bits are *true*.

The algorithm is a synchronous one: in each slot one processor broadcasts and all the other processors expect to receive its message, provided the broadcaster is in their membership sets. Receivers set their `ack` bits to *true* in each slot iff they receive an expected message. In addition, they remove the broadcaster from their membership sets if they fail to receive an expected message (on the interim assumption that the broadcaster must have been send-faulty). A receiver that subsequently receives a message carrying `ack` *false* when its own `ack` is also *false* knows that it made the correct decision in this case (since the current broadcaster also missed the previous expected message), but one that receives `ack` *true* realizes that it must have been receive-faulty (since the current broadcaster did receive the message) and removes itself from its own membership; a receiver that fails to receive an expected message when its `ack` bit is *false* also removes itself from its own membership (because it has missed two expected messages in a row, which is consistent with the fault arrival hypothesis only if that processor is itself receive-faulty); a receiver that receives a message with `ack` *false* when its own `ack` bit is *true* removes the broadcaster from its membership (since the broadcaster must have been receive-faulty on the previous broadcast). Processors that remove themselves from their own membership remain silent when it is their turn to broadcast—thereby communicating their self-diagnosed receive-faultiness to the other processors.

Formally, we let $\texttt{mem}(p)$ and $\texttt{ack}(p)$ denote the membership set and $\texttt{ack}$ bit of processor p. Note that processor p has access to its own $\texttt{mem}$ and $\texttt{ack}$, and can also read the value of $\texttt{ack}(b)$, where $b = i \bmod n$ and i is the current slot number, because this is sent in the message broadcast in that slot.

Initiality predicate: $\texttt{mem}(p) = \{0, 1, \ldots, n-1\},\ \ \texttt{ack}(p) = \mathit{true}$.[3]

The algorithm is specified by two lists of guarded commands: one for the broadcaster and one for the receivers. Primes denote the updated values of the state variables. The current slot is i and the current broadcaster is b, where $b = i \bmod n$.

Broadcaster: Processor b executes the appropriate guarded command from the following list.

$$\begin{array}{lll} (a)\ b \in \texttt{mem}(b) & \to \texttt{mem}(b)' = \texttt{mem}(b), & \texttt{ack}(b)' = \mathit{true} \\ \text{otherwise} & \to \text{no change.} & \end{array}$$

Receiver: Each processor $p \neq b$ executes the appropriate guarded command from the following list:

The guards (b)–(g) apply when $b \in \texttt{mem}(p) \wedge p \in \texttt{mem}(p)$

$$\begin{array}{lll} (b)\ \texttt{ack}(p) \wedge \text{no msg rcvd} & \to \texttt{mem}(p)' = \texttt{mem}(p) - \{b\}, & \texttt{ack}(p)' = \mathit{false} \\ (c)\ \texttt{ack}(p) \wedge \texttt{ack}(b) & \to \texttt{mem}(p)' = \texttt{mem}(p), & \texttt{ack}(p)' = \mathit{true}^4 \\ (d)\ \texttt{ack}(p) \wedge \neg\texttt{ack}(b) & \to \texttt{mem}(p)' = \texttt{mem}(p) - \{b\}, & \texttt{ack}(p)' = \mathit{true} \\ (e)\ \neg\texttt{ack}(p) \wedge \text{no msg rcvd} & \to \texttt{mem}(p)' = \texttt{mem}(p) - \{p\} & \\ (f)\ \neg\texttt{ack}(p) \wedge \neg\texttt{ack}(b) & \to \texttt{mem}(p)' = \texttt{mem}(p), & \texttt{ack}(p)' = \mathit{true} \\ (g)\ \neg\texttt{ack}(p) \wedge \texttt{ack}(b) & \to \texttt{mem}(p)' = \texttt{mem}(p) - \{p\} & \\ \text{otherwise} & \to \text{no change.} & \end{array}$$

The environment can perform only a single action: it can cause a new fault to arrive—provided no other fault has arrived "recently." Characterization of "recently" is considered below. We let $\texttt{the\_mem}$ denote the current set of nonfaulty processors, so that the following specifies arrival of a fault in a previously nonfaulty processor x.

Fault Arrival: $\exists x \in \texttt{the\_mem} : \texttt{the\_mem}' = \texttt{the\_mem} - \{x\}$

The desired safety properties are specified as follows.

Agreement: $p \in \texttt{the\_mem} \wedge q \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{mem}(q)$

Validity: $p \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{the\_mem} \vee \exists x : \texttt{mem}(p) = \texttt{the\_mem} \cup \{x\}$

The first says that all nonfaulty processors p and q have the same membership sets; the second says that the membership set of a nonfaulty processor p contains all nonfaulty processors, and possibly one faulty one.

The starting configuration is the following: all nonfaulty processors have their $\texttt{ack}$ bits *true* and their membership sets contain just the nonfaulty processors.

[3] I use the redundant $= \mathit{true}$ because some find that form easier to read.

[4] This case could be absorbed into the "otherwise" clause with no change to the algorithm; however, the structure of the algorithm seems clearer written this way.

Stable: $p \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{the\_mem} \wedge \texttt{ack}(p) = \mathit{true}$

It is natural to consider two transition conditions from this configuration: one where a new fault arrives, and one where it does not. In the latter case, the broadcaster will leave its state unchanged (no matter whether its executes command (a) or its "otherwise" case), and the receivers will execute either their command (c) or their "otherwise" case, and leave their states unchanged. The overall effect is to remain in the *stable* configuration. In the case that a new fault arrives, the same transitions as above will be executed but some previously nonfaulty processor x will become faulty, leading to the following configuration.

Latent(x): $x \notin \texttt{the\_mem}$
$\wedge\, p \in \texttt{the\_mem} \cup \{x\} \supset \texttt{mem}(p) = \texttt{the\_mem} \cup \{x\} \wedge \texttt{ack}(p) = \mathit{true}$

There are two transition conditions from *latent*(x): one where x is the broadcaster in the next slot, and one where it is a receiver.

In the former case, x will execute its command (a) while all nonfaulty receivers will note the absence of an expected message and execute their commands (b), leading to the following configuration.

Excluded$_1$(x): $x \notin \texttt{the\_mem} \wedge \texttt{mem}(x) = \texttt{the\_mem} \cup \{x\} \wedge \texttt{ack}(x) = \mathit{true}$
$\wedge\, p \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{the\_mem} \wedge \texttt{ack}(p) = \mathit{false}$

In the latter case, a nonfaulty broadcaster will transmit[5] and its message will be received by all nonfaulty receivers, but missed by x, leading to the following configuration.

Missed_rcv(x): $x \notin \texttt{the\_mem} \wedge \texttt{mem}(x) = \texttt{the\_mem} \cup \{x\} - \{b\} \wedge \texttt{ack}(x) = \mathit{false}$
$\wedge\, p \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{the\_mem} \cup \{x\} \wedge \texttt{ack}(p) = \mathit{true}$

There are four transition conditions from *missed_rcv*(x): one where the next broadcaster is x and it fails to broadcast; one where x does broadcast; one where the next broadcaster is already faulty; and an "otherwise" case. The first of these is similar to the transition from *latent*(x) to *excluded*$_1(x)$ and leads to the following configuration.

Excluded$_2$(x): $x \notin \texttt{the\_mem} \wedge \texttt{mem}(x) = \texttt{the\_mem} \cup \{x\} - \{b\} \wedge \texttt{ack}(x) = \mathit{true}$
$\wedge\, p \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{the\_mem} \wedge \texttt{ack}(p) = \mathit{false}$

We recognize that *excluded*$_1(x)$ and *excluded*$_2(x)$ should each be generalized to yield the following common configuration.

Excluded(x): $p \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{the\_mem} \wedge \texttt{ack}(p) = \mathit{false}$

[5] Treatment of the case that the next broadcaster is an already-faulty one depends on how fault "arrivals" are axiomatized: in one treatment, a fault is not considered to arrive until it can be manifested (thereby excluding this case); the other treatment will produce a self-loop on *latent*(x) in this case. These details are a standard complication in verification of fault-tolerant algorithms and are not significant here.

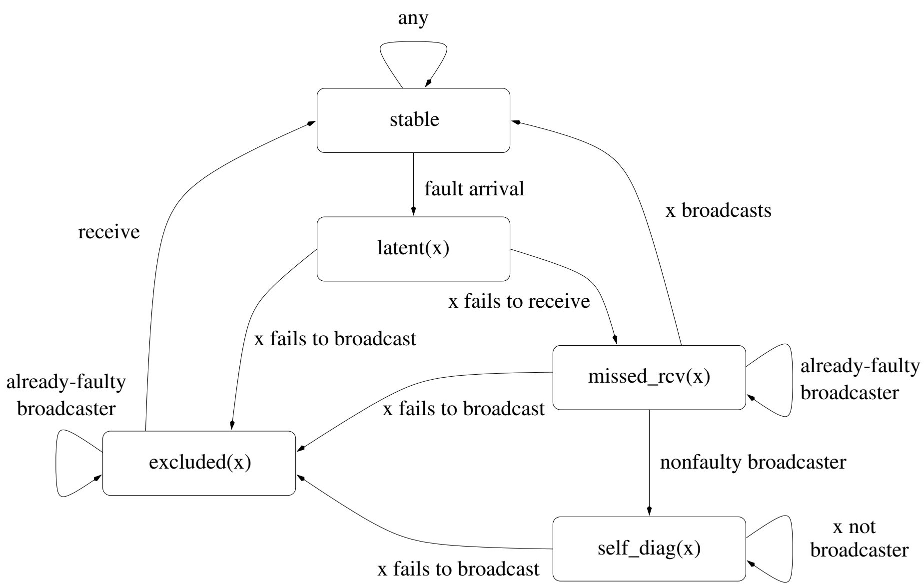

Fig. 1. Configuration Diagram for the Group Membership Example

In the case where x does broadcast, it will do so with `ack` *false*, causing nonfaulty processors to execute their commands (d) and leading directly to the *stable* configuration. The case where the next broadcaster is already faulty causes all nonfaulty processors and processor x to leave their states unchanged (since that broadcaster will not be in their membership sets), thereby producing a loop on *missed_rcv*(x). The remaining case (a broadcast by a nonfaulty processor, executing its command (a)) will cause nonfaulty receivers to execute their commands (c), while x will either miss the broadcast (executing its command (e)), or will discover the *true* `ack` bit on the received message and recognize its previous error (executing its command (g)); in either case, x will exclude itself from its own membership, leading to the following configuration.

Self_diag(x): $x \notin \texttt{the\_mem} \wedge x \notin \texttt{mem}(x)$
$\wedge\, p \in \texttt{the\_mem} \supset \texttt{mem}(p) = \texttt{the\_mem} \cup \{x\} \wedge \texttt{ack}(p) = \mathit{true}$

The transition conditions from this new configuration are those where x is the broadcaster, and those where it is not. In the former case, x will fail to broadcast (since it is not in its own membership), causing nonfaulty processors to execute their commands (b) and leading to the configuration *excluded*(x). The other case will cause them to execute their commands (c), or their "otherwise" cases, producing a self-loop on the configuration *self_diag*(x).

The only transitions that remain to be considered are those from configuration *excluded*(x). The transition conditions here are the case where the next broadcaster is already faulty, and that where it is not. The former produces a self-loop on this configuration, while the latter causes all nonfaulty receivers to execute their commands (f) while the broadcaster executes its command (a), leading to a transition to configuration *stable*.

It is easy to see that the initiality predicate implies the *stable* configuration and that all configurations imply the desired safety properties, and so we have now completed construction and verification of the diagram shown in Figure 1. The labels in the vertices of this diagram indicate the corresponding configuration, while the labels on the arcs are intended to suggest the corresponding transition condition. One detail has been glossed over in this construction, however: what about the cases where a new fault arrives while we are still dealing with a previous fault? In fact, this possibility is excluded in the full axiomatization of the fault arrival hypothesis, which states that faults may only arrive when the configuration is *stable* (we then need to discharge trivial proof obligations that all the other configurations are disjoint from this one). We connect this axiomatization of the fault arrival hypothesis with the "real" one that faults must arrive more than n slots apart by proving a bounded liveness property that establishes that the system always returns to a *stable* configuration within n slots of leaving it. This proof requires that configurations are embellished with additional parameters and clauses that remember the slots on which certain events occurred and count the numbers of self-loop iterations. The details are glossed because they are peripheral to the main concern of this paper; they are present in the mechanized verification of this example using PVS, which is available at `http://www.csl.sri.com/~rushby/cav00.html` and in a paper that describes verification of the full membership protocol [23]. (The full algorithm differs from the simplified version given here in that all faulty processors eventually diagnose their faults and exclude themselves from their own membership; its proof is about four times as long as that presented here).[6]

4 Discussion, Comparison, and Conclusion

The flawed verification of the full membership algorithm in [14] strengthens the desired safety properties, *agreement* and *validity*, with six additional invariants in an attempt to obtain a conjunction that is inductive. Five of these additional invariants are quite complicated, such as the following.

> "If a receive fault occurred to processor p less than n steps ago, then either p is not the broadcaster or $\mathtt{ack}(p)$ is false while all nonfaulty q have $\mathtt{ack}(q) = \mathit{true}$, or p is not in its own membership set."

The informal proof of inductiveness of the conjoined invariants is long and arduous, and it must be flawed because the algorithm has a bug in the $n = 3$ case. This proof resisted several determined attempts to correct and formalize it in PVS. In contrast, the approach presented here led to a straightforward mechanized verification of a corrected version of the algorithm.[7] Furthermore, as I hope the example has demonstrated, this

[6] The algorithm presented here is fairly obvious; there is a similarly obvious solution to the full problem (with self-diagnosis) that uses two `ack` bits per message; this clarifies the contribution of [14], which is to achieve full self-diagnosis with only one `ack` bit per message.

[7] The verification was completed on a Toshiba Libretto palmtop computer of decidedly modest performance (75 MHz Pentium with 32 MB of memory).

approach is naturally incremental, develops understanding of the target algorithm, and yields a diagram that helps convey that understanding to others. In fact, the diagram (or at least its outline) can usually be constructed quite easily using informal reasoning, and then serves as a guide for the mechanized proof.

This approach is strongly related to the verification diagrams and their associated methods introduced by Manna and Pnueli [17]. These were subsequently extended and generalized by Manna with Bjørner, Browne, de Alfaro, Sipma, and Uribe [5, 7, 10, 16]. However, these later methods mostly concern fairness and liveness properties, or extensions for deductive model checking and hybrid systems, and so I prefer to compare my approach with the original verification diagrams. These comprise a set of vertices labeled with formulas and a set of arcs labeled with transitions that correspond to the configurations and transition conditions, respectively, of my method. However, there are small differences between the corresponding notions. First, it appears that verification diagrams have a finite number of vertices, whereas configurations can be finite or infinite in number. The example presented in the previous section is a parameterized system with an unbounded parameter n, and most of the configurations are parameterized by an individual x selected from the set $\{0, 1, \ldots, n\}$, yielding an arbitrarily large number of configurations; Skolemization (selection of an arbitrary representative) reduces the number of proof obligations to a finite number. Second, the arcs in verification diagrams are associated with transitions, whereas those in my approach are associated with predicates. It is quite possible that this difference is a natural manifestation of the different examples we have undertaken: those performed with verification diagrams have been asynchronous systems (where each system transition corresponds to a transition by *some* component), whereas I have been concerned with synchronous systems (where each system transition corresponds to simultaneous transitions by *all* components). Thus, in asynchronous systems the transitions suggest a natural analysis by cases, whereas in synchronous systems (especially those, as here, without explicit control) the case analysis must be consciously imposed by selection of suitable transition conditions.

Mechanized support for verification diagrams is provided in STeP [18]: the user proposes a diagram and the system generates the necessary verification conditions. PVS provides no special support for my approach, but its standard mechanisms are adequate because the approach ultimately yields a conventional inductive invariance proof that is checked by PVS in the usual way. As illustrated in the example, the configuration diagram can be constructed incrementally: starting from an existing configuration, the user proposes a transition condition and then symbolically simulates a step of the algorithm (mechanized in PVS by rewriting and simplification); the result either suggests a new configuration or corresponds to (possibly a generalization of) an existing one. Enhancements to PVS that would better support this activity are primarily improvements in symbolic simulation (e.g., faster rewriting and better simplification).

The key to any inductive invariance proof is to find a partitioning of the state space and a way to organize the case analysis so that the overall proof effort is manageable. The method of disjunctive invariants is a systematic way to do this that seems effective for some problem domains. Other recent methods provide comparably systematic constructions for verifications based on simulation arguments: the *aggregation method* of Park and Dill [20] and the *completion functions* of Hosabettu, Gopalakrishnan and Srivas [13] greatly simplify construction of the abstraction functions used in verifying cache protocols and processor pipelines, respectively.

Other methods with some similarity to the approach proposed here are those based on abstractions: typically the idea is to construct an abstraction of the original system that preserves the properties of interest and that has some special form (e.g., finite state) that allows very efficient analysis (e.g., model checking). Methods based on *predicate abstraction* [24] seem very promising [1, 3, 9, 25]. A configuration diagram can be considered an abstraction of the original state machine and it is plausible that it could be generated automatically by predicate abstraction on the predicates that characterize its configurations and transition conditions. However, it is difficult to see how the user could obtain sufficient insight to propose these predicates without constructing most of the configuration diagram beforehand, and it is also questionable whether fully automated theorem proving can construct sufficiently precise abstractions of these fairly difficult examples using current technology.

Such an abstracted system would still have n processes and further reduction would be needed to obtain a finite-state system that could be model checked. Creese and Roscoe [8] do exactly this for the algorithm of [14] using a technique based on a suitable notion of data independence [22]. They use a clever generalization to make the processes of algorithm independent of how they are numbered and are thereby able to establish the abstracted n-process case by an induction whose cases can be discharged by model checking with FDR. This is an attractive approach with much promise, but formal and mechanized justification for the abstraction of the original algorithm still seems quite difficult (Creese and Roscoe provide a rigorous but informal argument).[8]

In summary, the approach presented here is one of a growing number of methods for verifying properties of certain classes of algorithms in a systematic manner. Circumstances in which this approach seems most effective are those where the algorithm concerned naturally progresses through different phases: these give rise to distinct disjuncts G'_i in a disjunctive invariant $G^m_\vee$ but are correspondingly hard to unify within a conjunctive invariant $G^m_\wedge$. Besides those examples already mentioned, the approach has been used successfully by Holger Pfeifer to verify another group membership algorithm [21]: the very tricky and industrially significant algorithm used in the Time Triggered Architecture for safety-critical distributed real-time control [15].

The most immediate targets for further research are empirical and, perhaps, theoretical investigations into the general utility of these approaches. The targets of my approach have all been synchronous group membership algorithms, while the verification diagrams of Manna et al. seem not to have been applied to any hard examples (the verification in STeP of an interesting Leader Election algorithm [6] did not use diagrammatic methods). If practical experience with a variety of different problem types shows the approach to have sufficient utility, then it will be worth investigating provision of direct mechanical support.

[8] Verification by abstraction of the communications protocol example mentioned earlier required 45 of the 57 auxiliary invariants used in the direct proof [12].

Acknowledgments

I am grateful for useful criticisms and suggestions made by the anonymous referees and by my colleagues Jean-Christophe Filliâtre, Patrick Lincoln, Ursula Martin, Holger Pfeifer, N. Shankar, and M. Srivas, and also for feedback received from talks on this material at NASA Langley, SRI, and Stanford.

References

Papers on formal methods and automated verification by SRI authors can generally be found at `http://www.csl.sri.com/fm-papers.html`.

[1] Parosh Aziz Abdulla, Aurore Annichini, Saddek Bensalem, Ahmed Bouajjani, Peter Habermehl, and Yassine Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In Halbwachs and Peled [11], pages 146–159.

[2] Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, Volume 1102 of Springer-Verlag *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996.

[3] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, Volume 1427 of Springer-Verlag *Lecture Notes in Computer Science*, pages 319–331, Vancouver, Canada, June 1998.

[4] Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [2], pages 323–335.

[5] Nikolaj Bjørner, I. Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[6] Nikolaj Bjørner, Uri Lerner, and Zohar Manna. Deductive verification of parameterized fault-tolerant systems: A case study. In *Second International Conference on Temporal Logic, ICTL'97*, Manchester, England, July 1997.

[7] I. Anca Browne, Zohar Manna, and Henny Sipma. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, Volume 1026 of Springer-Verlag *Lecture Notes in Computer Science*, pages 484–498, Bangalore, India, December 1995.

[8] S. J. Creese and A. W. Roscoe. TTP: A case study in combining induction and data independence. Technical Report PRG-TR-1-99, Oxford University Computing Laboratory, Oxford, England, 1999.

[9] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Halbwachs and Peled [11], pages 160–171.

[10] Luca de Alfaro, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Visual verification of reactive systems. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, Volume 1217 of Springer-Verlag *Lecture Notes in Computer Science*, pages 334–350, Enschede, The Netherlands, April 1997.

[11] Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, Volume 1633 of Springer-Verlag *Lecture Notes in Computer Science*, Trento, Italy, July 1999.

[12] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, Volume 1051 of Springer-Verlag *Lecture Notes in Computer Science*, pages 662–681, Oxford, UK, March 1996.

[13] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In Halbwachs and Peled [11], pages 47–59.

[14] Shmuel Katz, Pat Lincoln, and John Rushby. Low-overhead time-triggered group membership. In Marios Mavronicolas and Philippas Tsigas, editors, *11th International Workshop on Distributed Algorithms (WDAG '97)*, Volume 1320 of Springer-Verlag *Lecture Notes in Computer Science*, pages 155–169, Saarbrücken Germany, September 1997.

[15] Hermann Kopetz and Günter Grünsteidl. TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

[16] Zohar Manna, Anca Browne, Henny B. Sipma, and Tomás E. Uribe. Visual abstractions for temporal verification. *Algebraic Methodology and Software Technology, AMAST'98*, Volume 1548 of Springer-Verlag *Lecture Notes in Computer Science*, pages 28–41, Amazonia, Brazil, January 1999.

[17] Zohar Manna and Amir Pnueli. Temporal verification diagrams. In M. Hagiya and J.C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software: TACS'94*, Volume 789 of Springer-Verlag *Lecture Notes in Computer Science*, pages 726–765, Sendai, Japan, April 1994.

[18] Zohar Manna and The STeP Group. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Alur and Henzinger [2], pages 415–418.

[19] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[20] Seungjoon Park and David L. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory of Computing Systems*, 31(4):355–376, 1998.

[21] Holger Pfeifer. Formal verification of the TTA group membership algorithm. In *Formal Description Techniques and Protocol Specification, Testing and Verification FORTE XIII/PSTV XX 2000*, Pisa, Italy, October 2000. To appear.

[22] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1998.

[23] John Rushby. Formal verification of a low-overhead group membership algorithm, 2000. In preparation.

[24] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, Volume 1254 of Springer-Verlag *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997.

[25] Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [11], pages 443–454.