TEMPORARY STABILITY IN PARALLEL PROGRAMS

by

Shmuel Katz

Technical Report #403

March 1986
TEMPORARY STABILITY IN PARALLEL PROGRAMS

Shmuel Katz
The Technion, Haifa, Israel

ABSTRACT

A class of safety properties of parallel or distributed programs is defined and investigated. The property is known as temporary stability, and it relates a global property P, a second property B, and a set of processes S to which B is local (in a sense defined in the paper). Intuitively, P is temporarily stable until B in S if P remains true until B becomes true, and only a process in S can cause B. Several examples of the usefulness of this class of properties are given, as well as rules for establishing and manipulating temporary stabilities. The rules can be used both for program analysis, and as part of a systematic development of parallel programs. These properties are also shown to be a natural extension of the class of stable properties.

C. R. Categories: C.2.4 Distributed systems; D.1.3 Concurrent programming
1. Introduction

In this paper a class of safety properties of parallel programs is defined and investigated. The meta-property, known as temporary stability, is valuable for analyzing and developing parallel programs, both for distributed models based on message passing, and for shared memory models. While common in informal arguments, temporary stability has not been previously separated from the considerations of a particular context.

The property will relate to all the computations of a given parallel program. Given predicates \( P \) and \( B \), and a set of processes \( S \), \( P \) is temporarily stable until \( B \) in \( S \) if once \( P \land \neg B \) becomes true, \( P \) may again become false only after a process \( t \in S \) executes some action which causes \( B \) to hold. If \( S \) contains only a single process \( t \), the terminology "\( P \) is temporarily stable until \( B \) at \( t \)" will be used. If \( P \) is temporarily stable until \( B \) at \( t \), then whenever \( P \land \neg B \) is true \( t \) has become "master of its own fate" in the sense that it may either first detect that \( P \) has occurred, and then take any desired action with the knowledge that \( P \) still holds, or initiate a sequence of actions which might lead to \( \neg P \) by causing \( B \). In general, temporary stability allows decisions to be local to the processes of \( S \). Sometimes \( P \) will be referred to as the "global property," and \( B \) as the "local property of \( S \)."

The usefulness of temporary stability will often be due to the following observation: a process \( t \) may perform an action \( A \) which requires a global property \( P \) to be true until or during the action, only if \( P \) is temporarily stable until some \( B \) at \( t \). Otherwise, if there were no such \( B \), then some other process \( r \) could make \( P \) untrue without any action on the part of \( t \), just before the desired \( A \) is executed. This observation is further explained in one of the examples in the following section, where it is also related to recent work on knowledge in distributed programming [CM] [HM] [PR].

On the other hand, if \( P \) is temporarily stable until \( B \) at \( t \), and additionally it has been established that (i) in the past a global state \( s \) occurred in which \( P \land \neg B \) was true and (ii) since that state occurred, \( t \) has taken no action which could have caused \( B \) to be true, then \( P \) must still be true. Establishing (i) and (ii) will usually be far easier than establishing directly that \( P \) holds in the present, which is known to be extremely
difficult in asynchronous distributed programs.

The basic definition of stability was given in [CL]. There a property $P$ is defined to be stable if whenever $P(s)$ is true of a global state $s$, then $P(s')$ is true for all global states $s'$ reachable from $s$. Temporary stability is a generalization of stability, as is made precise in Section 3. The algorithm presented in [CL] for distributed snapshots, as well as other algorithms such as [SF] or [B], can be used to check whether a stable property has occurred. However, as is shown later in the paper, those algorithms can equally well be used to establish whether $P \land \neg B$ has occurred (i.e., (i) above). Thus by using a stable property detection algorithm along with a local check of the processes in $S$, and a static proof of temporary stability, many properties can be checked which are not completely stable.

In addition, the need to guarantee temporary stability can be used as a design principle for distributed programs. Properties which are not temporarily stable may be reduced to properties which are, by using domain dependent knowledge. Then some standard solutions become applicable.

In the next section, some common examples of temporarily stable properties are described. In Section 3, definitions and methods for establishing that a property is temporarily stable are presented, and in Section 4 some applications to the examples and to design methodology are considered.

2. Examples of Temporarily Stable Properties

Below several examples are described of techniques which relate to temporary stability and of properties which are not stable but are temporarily stable for some nontrivial set of processes and local property.

Example 1: A token system, which implements mutual exclusion by passing around a token and guaranteeing that a critical section is entered only by the process holding the token, in fact depends upon temporary stability. It is crucial to the correctness of such algorithms that once a process $t$ has obtained the token, it will remain at $t$ until that process relinquishes the token. In the case of a distributed program, with no shared memory, this will trivially mean that the predicate "the token is at process $t" is
temporarily stable at t until "the token is not at process t". For a shared memory model, the token system could be such as not to actually pass on the token when it is no longer needed at a process, and t could simply "unlock" the token so that another process could take it if necessary. In that case "the token is at process t" is temporarily stable until "unlocked" at t.

Example 2: As mentioned in the Introduction, the (static) temporary stability of P until B in t, is a necessary condition for the process t to be able to depend on the truth of P when some action is to be taken. This is intuitively what is meant for a process t to know a global property P. Several alternative formal definitions of this concept have been suggested [CM], [HM], [PR]. Without explicitly stating the various definitions of what it means for a process to know a fact about the global state, according to all the definitions, only true statements can be known (denoted $K_t(P)$). For this reason, in all the definitions, temporary stability of P until B at t, for some B, is a necessary (but not sufficient) condition for t to know P in a distributed model of computation (without shared memory). In order to see this, consider any state s in which $K_t(P)$ is true. If P were not true until the next atomic action of t, then some other process could perform an atomic action which makes P false, without t having locally changed its state. Thus t cannot know P in that state— a contradiction. A predicate B can always be constructed which is not true in the local projection of the state s but is true in those local states which result from an atomic action of t and in which $K_t(P)$ is no longer true. Then P is temporarily stable until B at t.

Example 3 Several proof methods for parallel programs integrate temporary stability into their considerations. The proof method of [GO] for shared memory parallel programs can be viewed as a particular use of temporary stability. Consider a predicate p which appears in the annotated program as either a precondition or a postcondition of an atomic statement r in a local proof of process t. Showing "interference freedom" is equivalent to demonstrating that each such p is temporarily stable until $\neg p$ at t. That is, no other process can make p untrue. It is crucial to the proof method that p cannot be made false by a statement not in t.
In \([L]\) and \([LS]\) variants of an extension of Hoare logic are proposed. The notation is explained in terms of invariance, but in effect \(P \mathsf{until} \ Q\) is the basic assertion. There is, however, no concept of locality and the application is again to proving partial correctness. In \([MP1]\) the strong until operator (where \(Q\) is guaranteed to occur) is used to prove eventualities for a shared memory model, so that liveness and safety are not separated.

**Example 4:** Temporary stability can be useful for distributed implementations of abstract data structures. In such implementations, user processes send messages to activate implementation processes. Almost always, the properties of the data structure (e.g., which elements it contains, its responses to external requests for data) are temporarily stable until specified new messages from users are received in the implementation. That is, even though the implementation may reorganize the data, it usually will not spontaneously make elements unavailable or add new elements without an instruction from a user process. This property can be valuable in showing the correctness of the implementation by separately demonstrating the temporary stability of the necessary abstract properties, and then showing that each operation is correctly implemented.

**Example 5:** A tremendous effort has been devoted to the discovery of algorithms for the detection of deadlock in distributed programs. Deadlock is not a truly stable property in that the purpose of these algorithms is to impose a detection computation in parallel with the (possibly deadlocked) basic computation. Once the deadlock is detected, it is common to take remedial actions and resume the basic computation. Deadlock is treated by considering each occurrence of deadlock separately and ignoring the restarting phase. In that context it is viewed as a stable property. However, it is more precise to say that deadlock is temporarily stable until an indicator that deadlock has occurred becomes true in a process which can initiate breaking the deadlock. That is, deadlock is a property which will not simply "go away" and must be explicitly broken only after a process has locally concluded that deadlock has occurred by setting a local indicator to true. A proof of correctness of such an algorithm will consist of showing that for repeated activations of the detection algorithm, an
indicator will be set to true iff a deadlock has occurred. The connection between temporary stability and detection algorithms is further explained in Section 4.1.

3. Formalizing temporary stability

The temporary stability of a property $P$ until property $B$ in processes $S$ is a static meta-property of a parallel program. That is, a triple $(P,B,S)$ either is or is not in the proper relation for a given distributed program. In order to express temporary stability more formally, two operators from temporal logic will be used, as well as a new operator written as $S$ causes $B$.

The underlying semantics of a program will be slightly extended from the usual collection of sequences of states, assuming an interleaving model of atomic actions. The only change will be in recording, along with the state, the name of the process containing the atomic statement which corresponds to the transition from the previous state. The semantics will thus be the collection of possible sequences of such pairs.

The common temporal operator used is $[\square]p$, which is interpreted as usual to mean "from now on" $p$ will be true in every state which is reached. More precisely, $[\square]p$ is true in a state $st$ if for all sequences defining the semantics, $p$ is true in all states following $st$ and in $st$ itself. The other temporal operator will be the weak Until operator (also known as Unless) where $pUq$ means $p$ will be at least until $q$ becomes true (but $q$ need never become true). In terms of the semantics, this means that $pUq$ is true in a state $st$ if for all sequences defining the semantics, $st$ is in a (possibly empty) contiguous subsequence of states in which $p$ is true, and the subsequence is either ended by a state in which $q$ is true, or includes all states following $st$. Later in the paper, the following temporal rules, which can be proven from the semantics, will be used. They, or slight variants, are proven in [MP].

TR1: $([\square]A)\land([\square]B) \equiv [\square](A \land B)$

TR2: $(AUB)\land(CUD) \rightarrow (A \land C)U(B \lor D)$

TR3: $(AUB)\land(CUB) \equiv (A \land C)UB$

TR4: $((AUB)\land(\neg BUC)) \rightarrow AUC$
The term \( (S \text{ causes } B) \) means that the transition from a state in which \( \neg B \) to one in which \( B \) is true, must correspond to an atomic statement in a process \( t \in S \). Thus it should be understood as "only a statement in \( S \) can cause \( B \)." In terms of the semantics, this means that for all sequences in the semantics, whenever a state in which \( \neg B \) holds is followed by a state in which \( B \) holds, the name of the process associated with the latter state is one of the processes of \( S \).

The definitions of invariant, stability, and temporary stability may now be reformulated in terms of the above operators. All of the following temporal logic expressions relate to the beginning of the program given (i.e., to the entire sequence which corresponds to its execution).

**Definition:** \( P \) is invariant for a given parallel program if \( [\neg P \rightarrow [\neg P] \) is true.

**Definition:** \( P \) is stable for a given parallel program if \( [\neg (P \rightarrow [\neg P]) \) is true.

**Definition:** \( P \) is temporarily stable until \( B \) in \( S \) if

\[
[P \rightarrow (PUB \land (S \text{ causes } B))
\]

For notational convenience, temporary stability will sometimes be denoted as \([P,B,S]\). In some cases the set \( S \) is either unknown or irrelevant. Then \([P,B,*]\) will be written, and this can be seen as equivalent to \([S,B,N]\), where \( N \) is the set of all processes in the system.

The Basic Theorem for determining whether a triple \((P,B,S)\) is temporarily stable for a parallel program (either distributed or shared memory) assumes the existence of a Hoare-like proof system for partial correctness of sequential atomic statements and an interleaving model of computation. Invariants of the program can be established as usual by showing that each atomic statement maintains the invariant and that it is true initially. It is assumed that such invariants can include assertions of the form \( at(L) \rightarrow \neg P \), and thus can be used to show that \( P \) cannot be true when the statement \( L \) is to be executed. As explained in \([10]\), this serves instead of the interference freedom check of \([GO]\) to indicate when atomic actions from different processes cannot be simultaneously enabled.

**Basic Theorem:** \( P \) is temporarily stable until \( B \) in \( S \) if (possibly given a global invariant \( I \)
proven independently)

(a) For each atomic statement $r$ in a process not in $S$ which changes variables in $P$ or $B$,

$$\{I \land P \land \neg B\} \rightarrow \{P \land \neg B\}$$

and

(b) For each atomic statement $r$ in a process in $S$ which changes variables in $P$ or $B$,

$$\{I \land P \land \neg B\} \rightarrow \{P \lor B\}$$

Whenever the variables of $B$ can only be changed by processes in $S$, then $\neg B$ remains trivially true for statements not in processes of $S$, simplifying part (a).

Proof: By part (a), once $P \land \neg B$ has occurred, it will remain true as long as only statements in processes not in $S$ are executed. By part (b), following a statement in a process in $S$, either $P \land \neg B$ continues to be true, or $B$ holds. In either case, $P \cup B$ is true, and clearly $B$ can be made true only by a statement in a process in $S$, as required by the definition of temporary stability.

The following rules help to clarify the definition, and show the boundary conditions of temporary stability.

**Locality Rule:** For any property $P$, if $P$ contains variables changed only in the set of processes $S$, then $P$ is temporarily stable until $\neg P$ in $S$.

Proof: If $P$ never occurs, the implication in the definition of temporary stability is true. Otherwise, $P \cup \neg P$ is a tautology, and only a process in $S$ can make $P$ become false.

Note that independently of $P$, $S$ can be taken to be the set of all the processes in the program. On the other hand, if $P$ is a local property of a process $t$, then by the rule, $P$ is temporarily stable until $\neg P$ at $t$. This was seen in the token example for a distributed model of computation.

**Stable Rule:** For any predicate $B$, a stable property $P$ is temporarily stable until $B$ in $S$, for any $S$ which contains all assignments to the variables in $B$. 
Proof: By the definition of a stable property, \([P \Rightarrow ]P\). Clearly, \([P \Rightarrow ]P\) \rightarrow \text{PUB}, for any B (including B = false), since P remains true. Thus P \rightarrow \text{PUB} for each state, as required.

Note that an invariant property, which is true in every state of the program, is obviously stable, as is the negation of an invariant, since it never becomes true.

In the other direction, we have

**Empty Set Rule:** If P is temporarily stable until B in the empty set of processes \(\varphi\) (i.e., [P,B,\(\varphi\)], then P \& \~B is stable.

Proof: If [P,B,\(\varphi\)], then, from the definition, P \rightarrow (\(\varphi\) causes B) must be true for every possible state s. Thus there are no processes in which B can become true once P \& \~B has occurred. It is therefore clear that \~B will remain true, and since P \rightarrow \text{PUB} also follows from the definition, and B never occurs, P also must remain true. Thus P \& \~B is a stable property.

If \~B is an invariant of the program (or B is identically false) and [P,B,\*] holds, then [P,B,\*] is the same as [P,B,\(\varphi\)] since no statement in the program can cause B to be true. The above rule then applies (with \~B=true), and P is a stable property.

**Stable Implication Rule:** If P \rightarrow Q is a stable property of a program, then [Q, \~P,\*] is true for that program.

Proof: By definition, P \rightarrow Q is stable if [[(P \rightarrow Q) \Rightarrow [(P \rightarrow Q)]] is true at the beginning of execution. Consider first the (possibly empty) initial sequence of states in which P \rightarrow Q is not true. In all such states, Q is false, so that Q \rightarrow (Q \& \~P) is true. Now consider any state s in which P \rightarrow Q is true. By the above definition, P \rightarrow Q will remain true in all subsequent states. If Q is false in s, then again Q \rightarrow (Q \& \~P). If Q is true in s, then either Q remains true in all subsequent states (and thus Q \& \~P holds for s), or there is a first state s' in which Q becomes false. However, P must also be false in s', or else P \rightarrow Q would no longer be true. Thus in this case also, Q \& \~P holds, and we have shown [[Q \rightarrow (Q \& \~P)], which is the definition of [Q, \~P,\*].
This rule also holds, of course, when \( P \rightarrow Q \) is an invariant. However, note that in the other direction it is definitely not the case that whenever \([Q, \neg P, \ast]\), then \( P \rightarrow Q \) is stable: there can be (numerous) states in which \( Q \) is false but \( P \) is true, even after \( P \rightarrow Q \) first occurs.

It will often be convenient to establish temporary stability in stages, by using the following rules.

**Generalization Rule:**

\[
[P_1, B_1, S_1], [P_2, B_2, S_2] \\
\overline{[P_1 \land P_2, B_1 \lor B_2, S_1 \cup S_2]}
\]

Proof: From the definition of temporary stability, on the antecedents of the rule, we have

\[
\models [P_1 \rightarrow P_1 UB_1] \land [P_2 \rightarrow P_2 UB_2]
\]

By TR1 and logical manipulation, this implies \(\models [(P_1 \land P_2) \rightarrow ((P_1 UB_1) \land (P_2 UB_2))]\)

By TR2 we obtain

\[
\models [(P_1 \land P_2) \rightarrow ((P_1 \land P_2)U(B_1 \lor B_2))]
\]

Thus we have the temporal logic part of the needed consequent. In order to obtain the locality, note that we have from the antecedents that \( P_1 \rightarrow (S_1 \text{causes } B_1) \) and \( P_2 \rightarrow (S_2 \text{causes } B_2) \). From the definitions, clearly

\[
(P_1 \land P_2) \rightarrow (S_1 \text{causes } B_1 \land S_2 \text{causes } B_2)
\]

and \( B_1 \lor B_2 \) can be caused only by atomic statements in \( S_1 \) or in \( S_2 \). The consequent follows from the definition of temporary stability.

When \( B = B_1 = B_2 \), this rule allows showing \([P_1 \land P_2, B, S]\) incrementally for each \( P_i \).

**Restriction Rule:**

\[
[P, B_1, S_1], [-B_1 \land R, B_2, S_2] \\
\overline{[P \land -B_1 \land R, B_2, S_2]}
\]

Proof: From the definition of temporary stability with the antecedents of the rule

\[
\models [(P \rightarrow PUB_1)] \land \models [((\neg B_1 \land R) \rightarrow (\neg B_1 \land R) UB_2)]
\]
From TR1 and logical manipulation

\[ \Box((P \land \neg B_1 \land R)) \rightarrow (PUB_1) \land (\neg B_1 \land R)UB_2) \]

By TR3, we have \((\neg B_1 \land R)UB_2 = \neg B_1 UB_2 \land RUB_2 \) and from TR4 we have \((PUB_1 \land (\neg B_1 UB_2)) \rightarrow PUB_2 \). Substituting into the above formula, the result is

\[ \Box(P \land \neg B_1 \land R) \rightarrow (PUB_2) \land ((\neg B_1 \land R)UB_2) \]

Finally, by using TR3,

\[ \Box(P \land \neg B_1 \land R) \rightarrow (P \land \neg B_1 \land R)UB_2 \]

In order to obtain the needed locality, note that from the right antecedent we have \((-B_1 \land R) \rightarrow S_2 \text{ causes } B_2 \).

Clearly, it follows that \((P \land \neg B_1 \land R) \rightarrow S_2 \text{ causes } B_2 \), and the consequent follows from the definition of temporary stability.

This rule allows relating \(P\) to \(B_2\) in stages, rather than directly.

**Intersection Rule:**

\[
\begin{align*}
\circ \quad [P,B_1,S_1], [P,B_2,S_2] & \rightarrow [P,B_1 \land B_2,S_1 \cup S_2] \\
\end{align*}
\]

Proof: Again, from the definitions \(\Box(P \rightarrow PUB_1) \land \Box(P \rightarrow PUB_2)\) and using TR1 and logical manipulations, this is equivalent to

\[ \Box(P \rightarrow PUB_1 \land PUB_2) \]

The above in fact implies the desired result that

\[ \Box(P \rightarrow PU(B_1 \land B_2)) \]. In order to see this, consider the alternative: that for some state \(st\) \(P\) is true and there is a subsequence starting at \(st'\) in which \(\neg P\) occurs before \(B_1 \land B_2\). Say that when \(\neg P\) first occurs, \(\neg B_1\) holds. If in the immediately previous state, \(P \land \neg B_1\) was true, then this contradicts the given \(\Box(P \rightarrow PUB_1)\). Similarly for \(\neg B_2\). Thus either \(B_1 \land B_2\) holds when \(\neg P\) becomes true, or it was true in the immediately preceding state. In either case, the alternative for the state \(st\) is contradicted, and \(PU(B_1 \land B_2)\) is true for that subsequence.

The locality requirement is easily established from the antecedents of the rule, since clearly only an action in \(S_1 \cup S_2\) can cause \(B_1 \land B_2\) to become true.
Note that the rule implies that both \( B_1 \) and \( B_2 \) must be true at the same time before \( P \) may become false.

4. Applications of temporary stability

As already seen, temporary stability is useful in describing a variety of common properties of parallel programs, which can then be checked in a given program by using the rules of the previous section. Thus for the shared memory token system of Example 1, the Basic Theorem shows what must be proven about such a system. If \( \text{att} \) denotes the predicate "the token is at process \( t \)" and \( \text{avail} \) is the predicate "the token is available for other processes to take," then \([\text{att}, \text{avail}, t]\) is needed. This requires showing that for all atomic statements \( r \) not in \( t \),

\[
\{ \text{att} \land \lnot \text{avail} \land I \} \rightarrow \{ \text{att} \land \lnot \text{avail} \}
\]

and for all atomic statements \( r \) in \( t \),

\[
\{ \text{att} \land \lnot \text{avail} \land I \} \rightarrow \{ \text{att} \lor \text{avail} \}
\]

It is also valuable in capturing the common denominator needed both to maintain \( P \), either as a stable property or as an invariant, or to detect the occurrence of \( P \). In the following subsection, the way in which existing detection algorithms may be employed to detect temporarily stable properties is described. Subsection 4.2 demonstrates the use of temporary stability in program development, both for specifications which require detecting a property, and for those which require maintaining a property.

4.1 Temporary stability and detection

The fact that temporarily stable properties can sometimes be detected by algorithms designed to detect stable properties can be formalized. As defined in [CL], an algorithm is said to detect a stable property if whenever the algorithm sets an indicator flag to true, the property held upon the termination of the algorithm (and continues to hold in the present), while whenever the flag is set to false, the property did not hold when the algorithm was initiated. As already mentioned, in addition to [CL], algorithms for this task can be found in [SF] and [B], or by generalizing the numerous algo-
It is clear that a global property \( P \) might not be detected by any detection algorithm if it holds only momentarily. Also, if the property is true, but becomes false before the completion of the detection algorithm, all of the mentioned algorithms might nevertheless "detect" the property. Thus, it might seem that such algorithms are not useful for detecting nonstable properties. However, for each of the above algorithms, it is not difficult to show that if a global property \( P \) remains true throughout an entire activation of the algorithm, then it can (and will) be detected. The problem, of course, is in distinguishing this situation from the one described previously, where \( P \) becomes false at the last minute, "fooling" the detection algorithm. In addition, it sometimes is necessary to determine whether \( P \) continues to be true in the present, and not merely whether it occurred in the past. Temporary stability can be used in order to partially overcome these difficulties, and expand the range of properties which can be detected. First we state the following claim:

Claim: An algorithm which detects the occurrence of any stable property can also be used to detect the occurrence of a property \( P \) which is temporarily stable until \( B \) in \( S \), if \(-B\) held throughout the execution of the algorithm.

Proof: The algorithm may depend on the fact that the property to be detected is stable. As already noted, this means that once true, it remains true. If a property \( P \) is only temporarily stable until \( B \) in \( S \), but \(-B\) held throughout the algorithm's execution, then if \( P \) was true when the algorithm began, so was \(-B\). Moreover, for every state during the algorithm's execution, \((PUB \land \neg B) \rightarrow P\). Thus as far as the detection algorithm is concerned, \( P \) was true and stable during its execution, and must be detected. Similarly, if \( P \) was not true when the detection algorithm terminated, and \(-B\) held throughout the algorithm's execution, then \( P \) must not have been true at any time during the execution (otherwise it still would be true), and the detection algorithm must detect that \( P \) is not true. If \( B \) does become true, no statement is made about the result of the detection algorithm.
Using the above result, a detection strategy can be devised for a property \( P \), given an algorithm \( A \) for detecting stable properties. At least one possible scenario would be, assuming that statically (not during execution) it has been proven that \([P,B,t]\) for some process \( t \) and \( B \): (1) Set a local indicator \( \neg B \) to true in \( t \) if \( \neg B \) holds, and modify the computations so that if \( t \) causes \( B \) to become true, \( \neg B \) will be set to false. (2) Activate \( A \) to detect \( P \) at \( t \).

When \( A \) has terminated, if \( P \) has apparently been detected by \( A \), then as long as the \( \neg B \) indicator remains true, \( P \) was true upon the completion of \( A \), and remains true. If algorithm \( A \) did not detect \( P \) and \( \neg B \) is still true (i.e., \( B \) has not been caused by \( t \)), then \( P \) did not hold when \( A \) was begun. If \( B \) has been caused by \( t \), then the result of \( A \) is unreliable: even if \( A \) "detected" \( P \), the property \( P \) may not ever have been true in the computation which actually occurred, and even if \( A \) did not detect \( P \), in the actual computation \( P \) might have been true throughout the execution of the detection algorithm.

In a more general context, where \([P,B,S]\) has been shown for a set of processes, with more than one element, it is still necessary for some process \( t \) to finally detect \( P \), by guaranteeing that \( P \land \neg B \) is temporarily stable. This will require a property \( C \) such that \([\neg B,C,t]\). The simplest such property is established by requiring the processes of \( S \) to "obtain permission" from \( t \) before causing \( B \) at least when detection of \( P \) is going on. The predicate \( C \) then would be "permission has been granted to cause \( B \)". A detection strategy for \( P \), initiated by \( t \), could then be:

Send an announcement \( \text{detect}P \) to the processes of \( S \). In response, each process \( s \in S \) would set a flag to prevent actions which cause \( B \) until a new exchange of \( \text{request/permission} \) is completed between \( s \) and \( t \), or until a \( \text{release} \) announcement is received from \( t \). Then \( s \) would send a \( \text{setup}P \) announcement back to \( t \). When such messages have been received by \( t \) from all processes of \( S \), \( t \) may activate a detection algorithm for \( P \land \neg B \). It is locally clear in \( t \) whether \( B \) was caused during the execution of the detection algorithm, by recording whether permission was granted for such a step since the detection algorithm was begun. If no process of \( S \) caused \( B \), then by the empty set rule, \( P \land \neg B \) is a stable property as far as the detection algorithm is concerned, and must be detected.
4.2. An example of program development

Consider two processes $P$ and $Q$ which communicate by message passing with a FIFO queue of messages. Each process maintains a set of values $SP$ and $SQ$, respectively. Based on local computations, from time to time a value is to be added to one of the sets, or a value is to be deleted. One of the processes, say $Q$, can perform certain actions only when the partition property

$$\forall v,w (v \in SP \land w \in SQ) \rightarrow v \neq w$$

holds. For example, the values could represent priorities of requests for some global service, and the property PAR means that $Q$ must have the right to service, no matter which request of $SQ$ is chosen to be answered. Another interpretation might be that $Q$ may need to change the values in $SQ$ without changing the order relations with values in $SP$, and PAR guarantees that this can be done.

As already seen, $Q$ can take actions which require the condition to be true up to and during the action, only if for some predicate $B$, $[\text{PAR},B,Q]$. Globally, it is clear that the partition property will be violated if either $Q$ adds a value to $SQ$ which decreases $\min(SQ)$ below the value of $\max(SP)$, or $P$ adds a value which increases $\max(SP)$ above $\min(SQ)$. In order that $Q$ be able to check whether either of these events have occurred, it will have a local variable, $mzP$, which contains an upper bound on the value of $\max(SP)$. If indeed $\max(SP) \leq mzP$ can be established as a (global) invariant, then by the Stable Rule, $[\max(SP) \leq mzP, \text{false}, Q]$ is true. Since both $mzP$ and $SQ$ are local to $Q$, by the Locality Rule $[mzP \leq \min(SQ), -(mzP \leq \min(SQ)), Q]$. Using the Combination rule with the above temporary stabilities, the relation

$$[\max(SP) \leq mzP \leq \min(SQ), mzP > \min(SQ), Q]$$

is obtained.

As explained in the previous section, one way in which the needed invariant $\max(SP) \leq mzP$ can be guaranteed is to have $P$ 'request permission' from $Q$ before inserting a value $v$ into $SP$ whenever the insertion would increase $\max(SP)$. That is, $P$ will send a message \text{request}(v) to $Q$, and wait to receive a message \text{granted}(v) before adding $v$. $Q$ will update $mzP$ whenever such a request is received, and send back a \text{granted}(v) message.
Clearly, if mere detection is required, Q will always grant such a request, and will simply note whether the new \( mxP \) is greater than \( \min(SQ) \). If so, then PAR is no longer guaranteed to hold. Similarly, if \( \min(SQ) < mxP \) is caused by locally adding a new value to \( SQ \). On the other hand, if PAR is to be kept an invariant property, then, since it is trivially true initially for empty \( SQ \) and \( SP \), it will be necessary to sometimes refuse requested additions. The simplest strategy might be to have Q refuse to allow P to add to \( SP \) any value larger than the present \( \min(SQ) \), and for Q to locally refuse to add to \( SQ \) any value smaller than \( mxP \).

An intermediate possibility is to allow free addition of local values to the respective sets and not maintain \( mxP \) in Q, until Q would like to take some action which requires PAR to be true. Q can detect this situation by testing a local flag \( wantaction \) and when it is true, obtain a value for \( mxP \) from P, through an exchange of messages. P would only then begin 'requesting permission' from Q to insert values which increase \( \max(SP) \), as above. Only when both \( wantaction \) and \( mxP < \min(SQ) \) are true for a newly obtained value of \( mxP \), Q may temporarily maintain PAR by refusing some of P's and its own requests to add values. While maintaining PAR, Q can perform the desired action (possibly interleaved with allowable updates of \( SP \) and \( SQ \)), and then reset \( wantaction \) to false, allow PAR to be violated, and cease maintaining \( mxP \) until it is again needed. The motivation for this solution is that the desired action in Q may be a rare occurrence, and there is no need to impede the free operation of P most of the time. If this assumption is unjustified, it is clearly possible to maintain \( mxP \) always, and leave it to Q to either always grant P's request (when \( wantaction \) is false), or to possibly refuse as above.

In terms of temporary stability, instead of the invariant \( \max(SP) < mxP \), the weaker temporary stability \( \max(SP) < mxP, \neg wantaction, Q \) will hold. Since locally it is easy to establish \( mxP < \min(SQ), \neg wantaction, Q \), the same result as previously can be obtained, guaranteeing that once \( \max(SP) < mxP < \min(SQ) \) occurs, it will remain true until \( \neg wantaction \) becomes true in Q.

Note that deleting a value from \( SQ \) has no effect on the considerations above, except to make it more likely that PAR will be true. Similarly, deleting a value from \( SP \)
could possibly decrease $\max(SP)$. However, in this case $mxP$ does not reflect the actual maximum of $SP$. Although all safety properties are maintained, unnecessary refusals could result, depending on Q's strategy. In order to reduce this risk, P can send a message $\text{newmax}(v)$ to Q if it is in the mode of requesting permission and if the maximum value of $SP$ has been reduced due to a deletion. Upon receiving such a message, Q would update $mxP$ to $v$. This particular solution will maintain $\max(SP) \leq mxP$ during the needed periods due to the FIFO ordering on the queue of messages. Any subsequent increase in $\max(SP)$ would have to first be permitted by Q.

For the intermediate possibility described above, a partial description of the programs for P and for Q is given in Figure 1. The programs are presented as a collection of responses to messages or conditions. Only when a response is completed, will another message or condition be chosen. The choice of which available message or true condition to respond to is arbitrary, except that messages between P and Q must be treated in FIFO order. Initially SP and SQ are empty, $\text{getpermit}$ and $\text{wantaction}$ are false, the max of an empty set is $-\infty$ (arbitrarily small), and the min of an empty set is $\infty$ (arbitrarily large).

Depending on the precise specification, 'refuse to add' in the program for Q could mean to ignore the request, or to merely delay it, by enqueueing the request. In the latter case, additional code would be needed to treat the enqueued requests, both in the response to $\text{newmax}(v)$ whenever $mxP$ is reduced, and in the response to $\text{delete}(v)$ whenever $\min(SQ)$ is increased.

As implied by the previous discussion, the fact crucial to the correctness is that $[\max(SP) \leq mxP \leq \min(SQ), \neg \text{wantaction},Q]$ and $\max(SP) \leq mxP \leq \min(SQ) \Rightarrow \text{PAR}$. However, note that this solution does not guarantee that PAR actually occurs from time to time. If such a requirement is needed, either it would follow from additional information about the pattern of insertions and deletions into SQ and SP, or further modifications to this solution would be required.
P::
[setup() --
  {a message from Q to begin requesting permission}
  getpermit:=true;
  send newmax(max(SP)) to Q]

alldone() -- {a message from Q to cease requesting permission}
  getpermit:=false

addrequest(v) -- {an internally generated request to add v to SP}
  if ~getpermit V v=max(SP) then SP:= SP \cup \{v\}
  else send request(v) to Q

granted(v) -- {a response message from Q to a previous request}
  SP:= SP \cup \{v\}

delete(v) -- {an internally generated request to delete v from SP}
  if getpermit A max(SP-\{v\}) < max(SP) then send newmax(max(SP-\{v\})) to Q;
  SP:= SP - \{v\}

Q::
[addrequest(v) -- {an internally generated request to add v to SQ}
  if v=maxP V \text{ wantaction}
    then SQ:= SQ \cup \{v\}
    else "refuse to add"

request(v) -- {a request from P to add v to SP}
  if v=min(SQ) V \text{ wantaction}
    then begin mxP:= v;
        send granted(v) to P
    end
  else "refuse to add"

delete(v) -- {an internally generated request to delete v from SQ}
  SQ:= SQ - \{v\}

newmax(v) -- {an informative message that max(SP) has become v}
  mxP:= v

want() -- {an internally generated request to perform the desired action}
  mxP:=\infty; \text{ to prevent misunderstanding until a new value is received from P}
  wantaction:= true;
  send setup() to P;

mxP<\text{min(SQ)} A wantaction --
  take (part of) the desired action; the steps here can be interleaved with the other responses. When the action is completed: wantaction:= false; send alldone() to P

Figure 1. Outline of the programs for P and Q
5. Conclusions

In addition to standardizing the various uses of temporary stability in verification, its usefulness has been demonstrated in expanding the range of applicability of detection algorithms, and in naturally expressing design requirements when invariants are awkward. Even when it is clear that a program is required to maintain an invariant throughout execution, temporary stability can be useful as an intermediate stage in the design process, separating global and local considerations. Thus to establish $P$ as an invariant, it is often useful to design the program first so as to establish $[P, B, S]$, initialize so that $P$ holds, and then locally in $S$ guarantee that $B$ does not occur.

The use of temporary stability is widespread in the design and verification of both shared memory and distributed parallel programs. By identifying the property explicitly, and providing uniform tools for establishing and exploiting temporary stability, its use in new applications should be facilitated, providing one of the design principles so badly needed in parallel programming.

References


