21 Oct 1993

Using Temporal Subsumption for Developing Efficient Error-Detecting Distributed Algorithms

Martina Schollmeyer

Bruce M. McMillin
Missouri University of Science and Technology, ff@mst.edu

Using Temporal Subsumption for Developing Efficient Error-Detecting Distributed Algorithms [1]

Martina Schollmeyer and Bruce McMillin
October 21, 1993

CSC 93-28

Abstract

Distributed algorithms can use executable assertions derived from program verification to detect errors at run-time. However, a complete verification proof outline contains a large number of assertions, and embedding all of them into the program to be checked at run-time would make error-detection very inefficient.

The technique of *temporal subsumption* examines the dependencies between the individual assertions along program execution paths. In contrast to classical subsumption, where all logical expressions to be examined are true simultaneously, an assertion need only be true when the corresponding statement in the distributed program has been executed. Thus, temporal subsumption based on the set of assertions derived from a verification proof and in combination with the set of all legal states in the system, allows for the removal of (partial) assertions along execution sequences.

We assume a fault model of Byzantine (malicious) behavior, and therefore an individual process cannot check itself for faults. We assume that a non-faulty process will always perform the correct computation so that once external data (obtained through communication) has been verified, the local computation does not need to be checked. A non-faulty process can thus detect faults produced by a faulty process based on the information it receives from it.

[1]This work was supported in part by the National Science Foundation under Grant Numbers MSS-9216479 and CDA-9222827, and, in part, from the Air Force Office of Scientific Research under contract numbers F49620-92-J-0546 and F49620-93-I-0409 and, in part, by a grant from the University of Missouri Research Board.

1 Introduction

Error-detecting algorithms work by checking assertions, at run-time, to detect hardware, communication [3] and software errors [8]. A properly chosen set of assertions, such as those generated from program verification, guarantees that, when operationally evaluated, the program meets its specification [6].

Mili [9] was the first to notice the relationship between program verification and fault tolerance of a program through software specified executable assertions. However, his approach was designed for the sequential verification environment. The development of executable assertions for a program in the distributed environment is more complex. Since information given in an individual process can only be communicated by message passing, the scope of the tests that may be performed by the local assertions is limited to testing received messages and to testing the local state of a process.

The axiomatic approach to program verification is based on making assertions about program variables before, during, and after program execution. These assertions characterize properties of program variables and relationships between the variables at the different stages of the program execution [2]. Axiomatic proof techniques for distributed systems are described in [10]. They include, besides the sequential proof of a program, a proof of non-interference, a satisfaction proof, and a proof of freedom of deadlock.

In an error-detecting program we embed assertions derived from program verification into the actual program code. In a possibly faulty environment, we require the executable assertions in each individual process to examine the behavior of other, possibly faulty, processes. This means that every process must be suspect of the data received from any other process which it considers to be potentially faulty. The assertions are then operationally evaluated at run-time.

We want to allow for Byzantine faultiness in the individual processes. This fault model allows for malicious behavior such as sending inconsistent messages to different processes. In general, a Byzantine faulty process will never be able to detect its own faultiness and thus its errors can only be detected by some other, non-faulty, process. Therefore, the assertions on the communication and the information received during the communication are vital to detect possible faults in other processes.

A complete verification proof outline on a distributed program contains a large number of assertions. There exist pre- and post-assertions to each statement in a program. However, we do not want every one of these assertions to be evaluated at run-time. Since verification proofs are tedious, often only parts of the program are verified completely. Many times only an incomplete proof outline is available or assertions are weakened.

Thus, turning every assertion into an executable assertion would inevitably slow down execution of the program due to the large overhead imposed by checking each statement by the appropriate assertion. In addition, many redundant checks would be performed. In responsive or safety-critical applications we only have a brief amount of time to perform an operational evaluation and thus we need to select a subset of assertions that provides complete error-coverage with minimal overhead.

The goal of this paper is to introduce a method for selecting this subset of assertions such that the number of assertions is small and such that we can retain the same error coverage in the program with the reduced set as with the complete set.

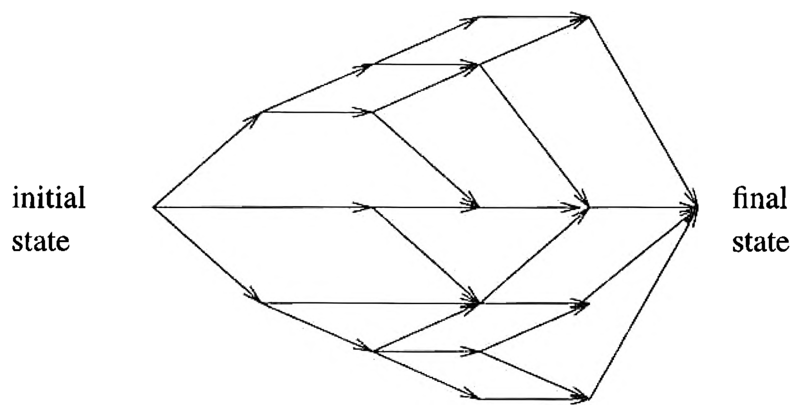

Figure 1: Sequences of states leading from an initial program state to the solution

In the next section we give a brief summary of the method to be used. We then explain the model of the system used in this paper to describe the individual processes and their interactions in a distributed system. We introduce a method for reducing the number of assertions from the original set of assertions to a smaller set of assertions that achieves the goals described above. The method used is based on the theory of subsumption.

2 Reducing the Number of Executable Assertions

In this section we give a brief summary of the model to be developed in more detail in the following sections of this paper. It provides an overview for the reader to be able to follow along with the development of the theory. The respective definitions will be given in the appropriate sections of the paper.

The model to be used here is based on a message-passing, distributed system. We use *states* to describe the status of a distributed program. From a verification proof outline we obtain *assertions* from each program statement, which imply the truth of the current program state with respect to each individual process. The model of the distributed program is based on interleaving semantics, which means that there exist many possible paths through a distributed program, based on the partial orders of the individual process executions. However, verification of the program must show that the processes are non-interfering with respect to their proofs.

Figure 1 shows a set of paths through a distributed program from a shared initial state to a shared final state, where the desired solution to the problem is found. Each path denotes a sequence of states of the overall program.

This global sequence can be divided into sequences of local states for each individual process of the distributed system. The outwardly observable events in such a distributed system are communications only. The individual state-to-state transitions within the distributed program, which form an execution sequence, are not observable. Between communications, there exist strictly sequential execution sequences for each process, and the sequences for all processes can be interleaved based on the individual partial orders and requirements with respect to communication. Figure 2 shows some global states for a distributed program with three processes. This figure gives a general idea of how the global states can be obtained by performing cuts across the complete set of processes.

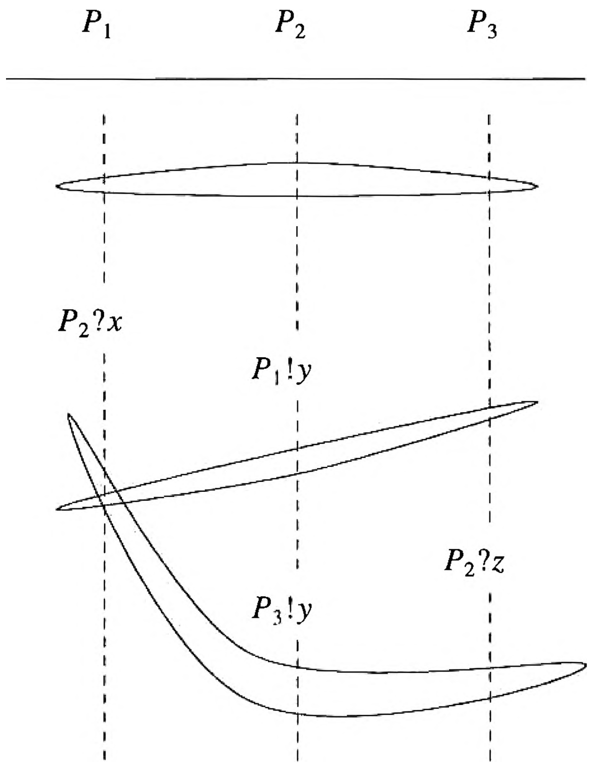

Figure 2: Different possible global states in an interleaving program

Each of the global states described in Figure 2 forms one of the possible states given in Figure 1. Such a global state which describes the overall program status can also be called a *world.* Sequences of worlds make up possible executions of a program.

From a verification proof outline of the program we obtain assertions for each individual process which describe the conditions that have to be met after the execution of each individual statement. We use these assertions for error detection and embed them into each of the individual programs for error detection. However, the number of assertions obtained from such a verification proof is very large.

To reduce the number of executable assertions to be embedded into the program for error detection, we want to remove the ones that are implied by others that have been encountered earlier in the program. It is apparent that the later assertions will be redundant with respect to error detection since we assume that we can only detect faults in other processes. We can *subsume* these assertions, and we will restrict the subsumption to occur between communication points only due to the reasoning above about partial orders.

Figure 3 gives an overview of the scope of this work. We will perform the reduction of assertions only for each of the local processes and we will not concern us with the global system in general. In addition, we will restrict ourselves to performing this reduction of assertions in an environment only that allows for soundness and (relative) completeness of the system to be used.

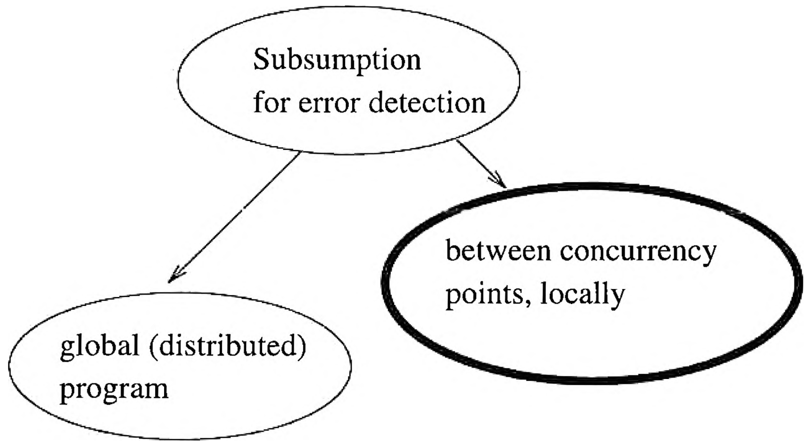
Subsumption for error detection
between concurrency points, locally
global (distributed) program

Figure 3: Scope of the work presented in this paper

In the following section we introduce the foundations of the system model and then continue with a discussion of how to transform the subsumption theory discussed in the field of automated reasoning into a theory used for removing assertions in the proof outlines derived using axiomatic semantics.

3 Temporal Subsumption

In this section we first give a brief overview about other work done in this area and how it relates to the results presented here. The work in subsumption that is described in [1], and which appears to be the only other work in this area, is a very general model which is not directly applicable to program verification and the methods described here.

After the discussion of the general model, we then introduce the terminology used for the remainder of the paper which deals specifically with distributed systems. We will describe the derivation of temporal subsumption from classical subsumption and will then provide the inference rules for the model at hand.

3.1 Subsumption in Modal Logic

In recent years, non-classical modal logics have gained more and more acceptance. Because of the increased use of such logics, automated deduction systems have been developed to make proofs using these logics easier. Subsumption has long been known as a technique to detect redundant clauses in the search space of automated deduction systems for classical first order logics. Because of the need to develop similar techniques for non-classical modal logics, [1] examined how subsumption can be made to work in the context of these modal logic deduction systems.

Modal logics, such as *temporal logic*, reason about *possible worlds*, i.e. not every statement made will be true all the time but only in certain specific worlds. *Paths* can lead from one world to the next, and depending on the path selected, different truths can be shown. Thus, it is easy to see that whatever needs to be proven depends on a specific path, and a logic called *World Path Logic* (WPL) is introduced in [1] as a possible target language in which a proof can be done. [1] then continues to develop a subsumption model which will work in this environment. For more details, the reader is asked to refer directly to the material presented in [1].

It can easily be seen how this this concept of sequences of worlds and paths connecting them could be related to programs and the program statements which are also connected by "paths" through the program. The approach by [1], although more abstract since it deals only with modal logics as such, was developed concurrently at UMR with the approach presented in this paper. In contrast to the work by [1], the model presented in this work deals specifically with distributed programs and describes a very different approach for generating subsumption rules as they are very closely linked to the application, distributed programs.

3.2 Terminology

A distributed program P consists of a set of processes running concurrently on a set of n processors. Each individual process $P_i, 1 \leq i \leq n$, constitutes a sequential program. For each of the individual processes there exists a proof outline $\mathfrak{V}_i$ which contains the set of assertions inducing the program. An assertions ϕ is a logical expression, derived from program verification, which describes the conditions that must be met by the program in connection with a particular program statement. We use the notation ϕ_t to denote the assertion associated with a program statement t. The assertion then describes a state which is expected to be true after the statement t has been executed. The state associated with t is denoted by s_t.

Definition 3.1 *The set of assertions ϕ derived from a verification proof outline is called the set $\mathfrak{V}$, and that the set of all states s that can be true in the corresponding program for all paths taken through the program and all initial conditions is called the set $\mathfrak{S}$. For an individual local program P_i, the respective sets are called $\mathfrak{V}_i$, which corresponds to the proof outline of the local program, and $\mathfrak{S}_i$, the set of local states.*

For the remaining part of this section we will examine only global states which describe the status of all individual processes combined. These global states are not attached to an individual statement but are formed by the local states which may be true simultaneously. All global states permitted in the program, independent of an individual execution sequence, are contained in the set $\mathfrak{S}$.

(Note: We allow for interleaving semantics and we will not describe in detail how the global states can be obtained and attached to the statements and states of the individual local processes. It suffices to say that this transformation can be done but a detailed explanation would add too much bulk to this section. The reader is asked to accept that such global states exist since we will refer to them here only in the abstract sense. In a later section we will refine this model to work with local processes.)

The execution of a distributed program can be decomposed into a relation on the individual local processes P_i, based on the individual proof outlines V_i containing the sets of assertions $\mathfrak{V}_i$. Thus, a program execution E describes a set of snapshots of the global system based on the execution of the individual processes, i.e.

$$E \subseteq \{\mathfrak{V}_1 \times \mathfrak{V}_2 \times \mathfrak{V}_3 \times \cdots \times \mathfrak{V}_n\}$$

and E is only a subset of the cross-product since, due to communication and synchronization, some combinations of assertions in the different individual processes may not be allowed to be true simultaneously. A program execution thus describes the set of consecutive truths with respect to the assertions in the individual programs.

An *accessibility relation* describes which states can be reached directly from a current state by executing the next program statement in any one of the individual processes. Based on the accessibility relation, a distributed program will move from an initial (global) state to a state at which the solution to the problem to be solved is obtained. The accessibility relation can also allow for synchronization of processes at communication points by forcing a process, that has reached a communication or synchronization point, to wait until the other participating process has reached the corresponding state in its program.

An execution of a distributed program, based on an initial state satisfying its precondition, and an accessibility relation, provides a state sequence consisting of a set of global states $\subseteq \mathfrak{S}$ that forms a possible program execution.

Definition 3.2 *The* projection function Π_i *gives the mapping from the global view of the distributed system into the local view of an individual process. Thus, for any specific state* $s_t \in \mathfrak{S}$ *in the global proof,* $\Pi_i(s_t) = s_t^i$ *denotes a local state in process* P_i. s_t^i *can then be associated with an individual program statement and its post-assertion in the local proof.*

The accessibility relation $\mathbf{R}$ guarantees that during the execution of a distributed program at least one of the processes will make progress, i.e. execute the next statement. If a state $s_j \in \mathfrak{S}$ is directly accessible from a state $s_i \in \mathfrak{S}$, i.e., s_j is a possible *next* state of s_i, then we denote this by writing $s_i\mathbf{R}s_j$, where both $s_i, s_j \in \mathfrak{S}$. This can also be written using the notation $s_j = s_i^+$. In a distributed program there may at each step exist several next states, depending on which of the processes make progress.

Definition 3.3 *A general accessibility relation, without special considerations with respect to communication or termination, is a binary relation* $\boldsymbol{R}$ *on the set of all states* $\mathfrak{S}$*, as follows:*

$$s_m\mathbf{R}s_n \Leftrightarrow \forall i[(\Pi_i(s_n) = \Pi_i(s_m)) \vee \ (\Pi_i(s_n) = (\Pi_i(s_m))^+)] \wedge \ \exists j[\Pi_j(s_n) = (\Pi_j(s_m))^+]$$

Definition 3.4 *A transitive relation that gives the set of all (global) states s_j that can be reached from an initial state $s_i \in \mathfrak{S}$, the* set of future states *of s_i, is given as*

$$R_{s_i} = \{s_j \in \mathfrak{S} \,|\, (\exists s_{r1}, s_{r2}, \cdots, s_{r_m} \in \mathfrak{S}) \text{ such that } s_i \mathbf{R} s_{r1}, s_{r1}\mathbf{R}s_{r2}, \cdots, s_{r_m}\mathbf{R}s_j\}$$

so that R_{s_i} denotes the set of states that are possible in the future of s_i.

Properties that depend on the transitivity of the accessibility relation can be defined based on the set R_{s_i}. For example, given any two states $s_i, s_j \in \mathfrak{S}$, if $s_j \in R_{s_i}$, then we say that s_j is *reachable* from s_i. In our definition, a possible interpretation of "reachability" is equivalent to finitely many applications of "accessibility".

Definition 3.5 *A* forward concurrency point *describes a state $s_f \in \mathfrak{S}$ that will occur in every possible execution sequence at some time in the future of the current state s_i.*

$$(\forall s_r \in \mathfrak{S})((s_r \in R_{s_i} \land \; s_r \neq s_f) \rightarrow \; s_f \in R_{s_r}))$$

Definition 3.6 *Correspondingly, a* backward concurrency point *describes a state $s_b \in \mathfrak{S}$ that must have occurred in every possible execution sequence at some point in the past. Thus, $(\forall (s_i \neq s_b \land \; \neg(s_b \in R_{s_i}) \land s_i \in \mathfrak{S})(s_i \in R_{s_b})$.*

The intermediate states between concurrency points can vary dependent on different interleavings of the individual processes. They form an execution sequence which represents a possible path through the distributed program.

We can also define *partial* concurrency points, which are concurrency points that will occur for a subset of processes only. An example for this are communication points during synchronous communication: both participating processes need to reach a state in which a matching communication pair is formed.

Using this model, we can describe how a program can reach a *fixed point* in its computation, i.e., the desired solution. This will be a forward concurrency point with respect to the accessibility relation: all possible execution sequences should lead to the termination point at which the solution will be presented. This requires that a unique solution exists and that it can always be reached. This fixed point corresponds to a state associated with the post-assertion of the distributed program.

4 Subsumption of Assertions: The Full Model

A state s_t is associated with an assertion ϕ_t such that s_t is true after the program statement t has been executed. However, this connection between state and statement is obvious only in a local process but not for the distributed system as a whole since we do not have global statements. As mentioned before, assertions describe properties of program variables and relationships between the variables at the different stages of the program execution. Assertions are usually expressed in predicate logic, and therefore executable assertions are logical expressions which are expected to be true during the execution of the program after a particular program step.

Definition 4.1 *Formally, an assertion on a program statement t, ϕ_t, is associated with a state $s_t \in \mathfrak{S}$, which must be satisfied after the execution of t. Thus,*

$$s_t \models \phi_t$$

where ϕ_t is an assertion which is part of a proof outline, i.e. $\phi_t \in \mathfrak{B}$. At the same time, there exist several states s_{t_i} which could be true after a statement t is executed, depending on which path through the program was selected and which initial state was given. Thus, for each assertion ϕ_t on a statement t,

$$\phi_t \rightarrow \ s_{t_i} \in \mathfrak{S}$$

We now define a model that allows us to decide which of the many assertions that are contained in a complete proof outline must be retained so that all faulty behavior that can be detected by the complete set of assertions can still be detected with the reduced set of assertion. We base our model on the theory of *subsumption.* We remove assertions in a proof outline that are already implied by other assertions since they do not contribute to the error coverage provided by the other assertions. Also, executing these redundant assertions will only cause the execution time of the program to increase. Thus, we remove assertions that can be subsumed by earlier assertions.

Subsumption provides the mathematical justification for assertion-reduction techniques that may seem intuitive to those familiar with the derivation of proof outlines of distributed programs and the selection of subsets of assertions to be embedded into a program as error-detecting executable assertions.

In the (automated reasoning) literature, subsumption is defined as follows [5]:

Definition 4.2 *A clause C subsumes a clause D if and only if $\forall C \rightarrow \ \forall D$ is valid, and $\forall C$ is the notation for the universal closure of the clause C.*

In automated reasoning all clauses to be examined are in skolemized form and thus contain no existential or universal quantifiers. Using assertions on a program or in a proof outline, we do not want to skolemize but rather keep the quantifiers, and thus we will use the following, refined, definition for subsumption.

Definition 4.3 *A clause $C(x)$ subsumes a clause $D(x)$ if and only if $(\forall x)(C(x) \rightarrow D(x))$ is valid.*

The justification for this refined definition can be found in Appendix A.

To define subsumption on a set of assertions in a verification proof outline, we look at current and future program states and the corresponding assertions. For right now we simply consider a set of assertions obtained from a verification proof and the set of states of the corresponding program.

At this point we will neglect the connection between program statements and assertions or states but use abstract states and assertions not attached to any specific program statements to introduce the subsumption model. In a later section we will then refine the general subsumption model obtained here and combine the local program statements with the assertions and the states they describe.

Let s_i be a state of a program P and let s_j be a state in the future of s_i, i.e. $s_j \in R_{s_i}$ and $s_i, s_j \in \mathfrak{S}$. Then there exists a sequence of states $s_i, s_{r1}, s_{r2}, \cdots, s_{r_m}, s_j \in \mathfrak{S}$, starting at s_i and terminating at s_j, which must be expressible using the accessibility relation $\mathbf{R}$, such that $s_i\mathbf{R}s_{r1}, s_{r1}\mathbf{R}s_{r2}, \cdots, s_{r_m}\mathbf{R}s_j$ is a permitted sequence of states and $s_{r_{i+1}} = s_{r_i}^{+}$. Each of the states is associated with an assertion $\phi \in \mathfrak{V}$.

We now need to modify Definition 4.3 to fit a proof outline and the assertions contained in it rather than just a collection of clauses in propositional logic. For this we need to remember that during a program execution not all assertions can be true at the same time but that we evaluate assertions along a path through the program. This means that subsumption is not performed on a set of clauses, which all must be true simultaneously as in automated reasoning, but we perform *temporal subsumption* which describes dependencies between current and future states in the program. We will express these dependencies using the accessibility relation and its transitive closure.

In order to describe this temporal dependency between assertions, we will use a symbol different from the regular implication $\rightarrow$, since we do not want to perform classical subsumption using logical implication, but temporal subsumption with respect to the predicate transformations encountered during the program execution.

Definition 4.4 *An assertion ϕ_i implies a later assertion ϕ_j along a path e through the program with respect to the predicate transformations along this path, if we have an execution sequence e, starting at s_i, terminating at s_j, i.e.,*

$$e = s_i\mathbf{R}s_{i+1}, s_{i+1}\mathbf{R}s_{i+2}, \cdots, s_{j-1}\mathbf{R}s_j$$

such that $s_i \models \phi_i$ and $s_j \models \phi_j$ and $s_j \in R_{s_i}$. We can write this as $(\phi_i \wedge\ e \rightarrow\ \phi_j)$ which can be abbreviated by writing $(\phi_i \overset{e}{\rightarrow} \phi_j)$.

We first consider the basic case of temporal subsumption derived directly from Definition 4.3. We will call this method of subsuming assertions $(\mathfrak{V}, \mathfrak{S})$-*subsumption.*

Definition 4.5 *An assertion ϕ_i associated with the state s_i* $(\mathfrak{V}, \mathfrak{S})$-subsumes *$\phi_j$ associated with s_j along a path e starting at s_i and terminating at s_j, if and only if*

$$(s_j \in R_{s_i}) \wedge\ (\phi_i \overset{e}{\rightarrow} \phi_j)$$

Definition 4.5 provides the temporal connection between sets of assertions by requiring the assertions to be examined for $(\mathfrak{V}, \mathfrak{S})$-subsumption to be assertions on states in the future of an initial state s_i. In addition, it removes the requirement of the universal quantification in Definition 4.3 since the assertions are examined sequentially, one step at a time, instead of concurrently.

As mentioned before, an (executable) assertion that can be $(\mathfrak{V}, \mathfrak{S})$-subsumed by another assertion can be removed from the proof outline (or from the program) without changing the fault coverage of the program, i.e., the program's ability to detect errors. It can be seen that the $(\mathfrak{V}, \mathfrak{S})$-subsumption process retains all assertions containing new information, since these assertions are generally not implied by any previous assertions, and future assertions containing equivalent or dependent information will be subsumed.

However, we need to guarantee that assertions in the future of s_i are $(\mathfrak{V}, \mathfrak{S})$–subsumed and removed only if they are related to ϕ_i. Thus we expand on Definition 4.5 to ensure a direct dependency between the two states s_i and s_j and the corresponding assertions ϕ_i and ϕ_j.

Definition 4.6 *An assertion ϕ_i associated with the state s_i* $(\mathfrak{V}, \mathfrak{S})$-subsumes *$\phi_j$ associated with s_j along a path e starting at s_i and terminating at s_j, if and only if*

$$(s_j \in R_{s_i}) \wedge (\phi_i \overset{e}{\rightarrow} \phi_j) \wedge$$
$$(\forall s_r \in \mathfrak{S})(s_r \in R_{s_i} \wedge s_j \in R_{s_r} \rightarrow ((\phi_i \overset{e}{\rightarrow} \phi_r) \wedge (\phi_r \overset{e}{\rightarrow} \phi_j)))$$

This guarantees that ϕ_i and ϕ_j are directly related since there exists no intermediate assertion ϕ_r that would prohibit ϕ_i to be $(\mathfrak{V}, \mathfrak{S})$-subsumed by ϕ_j.

(Note: at this point in time we still only look at a set of assertions derived from program verification and not at the statements associated with them. For more detail on how assertions can be determined from a program using axiomatic semantics see [2] and [4]).

To reduce the number of assertions that need to be examined as candidates for $(\mathfrak{V}, \mathfrak{S})$-subsumption, we want to divide the program into parts that can be examined independently. In general, we want these sections to be enclosed by a forward and a backward concurrency point. As described earlier, non-interference of the sequential proofs allows for arbitrary interleavings of the execution sequences of individual processes, and two processes only synchronize when vital information has to be exchanged. In a proof environment using global auxiliary variables (GAVs) in the assertions, we can delay the communication of the GAVs until an actual communication occurs. Thus, GAVs are treated as local variables in each process and become known to other processes only when they are communicated instead of being communicated and known instantaneously. This is possible due to the non-interference of the sequential proofs [7].

Since this delay of the communication of the 'global' knowledge contained in the GAVs retains the soundness and (relative) completeness of the proof system used for the verification proof, we can use the same boundaries here for the $(\mathfrak{V}, \mathfrak{S})$-subsumption. This means that we subsume between communication points only, which will provide our backward and forward concurrency points.

Definition 4.7 *A* communication point *describes a global state in which two processes have synchronized and are communicating with each other.*

We thus modify Definition 4.6 to allow for subsumption between two concurrency points, associated with the states s_{c1} and $s_{c2} \in \mathfrak{S}$, as follows:

Lemma 4.1 *An assertion ϕ_i associated with the state s_i* $(\mathfrak{V}, \mathfrak{S})$-subsumes *$\phi_j$ associated with s_j along a path e starting at s_i and terminating at s_j, between the concurrency points $s_{c1}, s_{c2} \in \mathfrak{S}$, if and only if*

$$(s_i \in R_{s_{c1}} \cup \{s_{c1}\}) \wedge (s_j \in R_{s_i}) \wedge (s_{c2} \in R_{s_j}) \wedge$$
$$(\phi_i \overset{e}{\rightarrow} \phi_j) \wedge (\forall s_r \in \mathfrak{S})(s_r \in R_{s_i} \wedge s_j \in R_{s_r} \rightarrow ((\phi_i \overset{e}{\rightarrow} \phi_r) \wedge (\phi_r \overset{e}{\rightarrow} \phi_j)))$$

and c1 and c2 are the backward and forward concurrency points, respectively, corresponding to two consecutive communication points in an execution sequence.

This lemma follows directly from the reasoning above and [7].

During program execution, and thus in the proof outline, only few variable values or relationships change from program step to program step. Assertions describe program states and they are therefore often conjunctions of clauses describing individual variables and their relationships with other variables.

Definition 4.8 *An assertion ϕ_i associated with a state $s_i \in \mathfrak{S}$ can be factored into a* set of partial assertions Φ_i, *where*

$$\Phi_i = \{\psi_i \mid (\phi_i \to \psi_i) \land \neg(\psi_i \to \phi_i) \land \\ \phi_i = \psi_{i,1} \land \psi_{i,2} \land \cdots \land \psi_{i,n}\}$$

and each ψ_i is a partial assertion.

When we perform $(\mathfrak{V}, \mathfrak{S})$-subsumption, we want to remove as many assertions, or partial assertions, as possible. Thus, we want to consider all ψ_j of an assertion ϕ_j as we look for subsumable conjuncts of ϕ_j. We can thus expand Definition 4.6 to allow for subsumption of partial assertions (between concurrency points) as follows:

Definition 4.9 *The assertion ϕ_i associated with the state s_i* $(\mathfrak{V}, \mathfrak{S})$-subsumes *the partial assertion ψ_j associated with s_j along a path e starting at s_i and terminating at s_j, between the concurrency points $s_{c1}, s_{c2} \in \mathfrak{S}$, if and only if*

$$(s_{c1}, s_{c2} \in \mathfrak{S}) \land (s_i \in R_{s_{c1}} \cup \{s_{c1}\}) \land (s_j \in R_{s_i}) \land (s_{c2} \in R_{s_j}) \land (\phi_i \xrightarrow{e} \psi_j) \land \\ (\forall s_r \in \mathfrak{S})(s_r \in R_{s_i} \land s_j \in R_{s_r} \to (\exists \psi_r)((\phi_i \xrightarrow{e} \psi_r) \land (\psi_r \xrightarrow{e} \psi_j)) \land \\ \psi_j \in \Phi_j \land \psi_r \in \Phi_r)$$

Subsumption on partial assertions thus reduces the overall number of assertions as well as the size of individual assertions, which means that the individual assertions can be *weakened.*

Definition 4.10 *The set of* critical assertions $\mathfrak{V}_{crit}$ *is the set of partial assertions ψ_i at every state $s_i \in \mathfrak{S}$ such that*

$$\mathfrak{V}_{crit} = \bigcup_{\forall s_i \in \mathfrak{S}} \{\psi_i | \psi_i \textit{ cannot be } (\mathfrak{V}, \mathfrak{S})\textit{-subsumed } \} \Leftrightarrow$$

$$\neg(\exists s_r \in \mathfrak{S})[(s_i \in R_{s_r}) \land (\phi_r \xrightarrow{e} \psi_i) \to (\forall s_t \in \mathfrak{S})(s_t \in R_{s_r} \land s_i \in R_{s_t} \to \\ (\exists \psi_t)((\phi_r \xrightarrow{e} \psi_t) \land (\psi_t \xrightarrow{e} \psi_i)) \land \psi_i \in \Phi_i \land \psi_t \in \Phi_t)]$$

The set of critical assertions provides the set of assertions that is required to detect all errors caused by a faulty process. Thus, it is an important set of assertions to be embedded into the program as executable assertions in an error-detecting program. Since processes are unable to detect their own faults, we need to ensure that error-detection can be performed by other processes through the variables and values that are communicated.

Since we delay the exchange of GAVs to occur at the same time as a regular communication, we can take advantage of this combined communication of variables and limit $(\mathfrak{V}, \mathfrak{S})$-subsumption to occur only between communication points as described in

Lemma 4.1. Thus, we can enforce that after each communication all state information will be verified, since it is new information for each subsumption process, and additionally, after the communication, all variable assignments that involve new information are checked through assertions as well.

Theorem 4.1 *Subsumption between concurrency points provides a set of (partial) assertions that includes the critical set. Thus, the following condition must hold:*

$$\mathfrak{V}_{crit} \subseteq \mathfrak{V}_{conc.points}$$

Proof: The set of critical assertions is a subset of the (partial) assertions such that no assertion can be derived from any preceding assertion for every possible execution sequence (see Definition 4.10). We need to show that if a partial assertion $\psi_{crit} \in \mathfrak{V}_{crit}$ then $\psi_{crit} \in \mathfrak{V}_{conc.points}$ must also hold.

We assume that there exists a partial assertion ψ_{crit} that is a critical assertion and occurs at state s_{crit}. This means that between concurrency points $c1$ and $c2$ the assertion cannot be $(\mathfrak{V}, \mathfrak{S})$-subsumed under the following condition:

$$\psi_{crit} \in \mathfrak{V}_{conc.points} \Leftrightarrow$$

$$\neg(\exists s_r \in \mathfrak{S})[(s_r \in R_{s_{c1}} \cup \{s_{c1}\}) \wedge \ (s_{crit} \in R_{s_r}) \wedge \ (s_{c2} \in R_{s_{crit}}) \wedge \ (\phi_r \overset{e}{\rightarrow} \ \psi_{crit}) \rightarrow$$
$$(\forall s_t \in \mathfrak{S})(s_t \in R_{s_r} \wedge \ s_{crit} \in R_{s_t} \rightarrow \ (\exists \psi_t)((\phi_r \overset{e}{\rightarrow} \ \psi_t) \wedge \ (\psi_t \overset{e}{\rightarrow} \ \psi_{crit})))\wedge$$
$$\psi_{crit} \in \Phi_{crit} \wedge \ \psi_t \in \Phi_t]$$

This means that for concurrency points we restrict the subsumption to the subset of states between s_{c1} and s_{c2}. In Definition 4.10 we $(\mathfrak{V}, \mathfrak{S})$-subsume for the set of states $S_{4.10} = (\forall s_t \in \mathfrak{S})(s_{crit} \in R_{s_t})$, which describes states occurring before ϕ_{crit} is encountered. Between concurrency points we $(\mathfrak{V}, \mathfrak{S})$-subsume for the set of states

$$S_{conc.points} = (\forall s_t \in \mathfrak{S})(s_t \in R_{s_{c1}} \cup \{s_{c1}\} \wedge \ s_{crit} \in R_{s_t})$$

Thus, $S_{conc.points} \subseteq S_{4.10}$ since the set of states to examine is limited. If $\neg(\exists s_i \in S_{4.10})(\phi_i$ $(\mathfrak{V}, \mathfrak{S})$-subsumes $\psi_{crit})$ then it follows that $\neg(\exists s_j \in S_{conc.points})(\phi_j$ $(\mathfrak{V}, \mathfrak{S})$-subsumes $\psi_{crit})$ since $S_{conc.points} \subseteq S_{4.10}$. $\square$

5 Subsumption in the Local Processes

The subsumption model introduced in the previous section is designed for the verification environment of a distributed system where a compositional proof system is used. As mentioned before, we generally have multiple processes that execute in parallel. The non-interference of their sequential proofs is vital for a proper functioning of the program. If we examine the individual assertions in the local processes, we can see that each of them describes part of a global view such that none of them interfere and such that an existing global invariant is never violated.

For a local process P_i we now have statements t^i, corresponding assertions ϕ_t^i and matching states s_t^i, as described in Section 4. The set of permitted states for this process is denoted by $\mathfrak{S}_i$ and the set of assertions is a part of the verification proof outline and thus the assertions are in $\mathfrak{V}_i$.

5.1 Terminology of a Message Passing Environment

In the full model of $(\mathfrak{V}, \mathfrak{S})$-subsumption, as described in the previous section, we can examine a global proof of the system. However, in an actual implementation of a distributed system, we have non-interfering proofs and thus independent processes which exchange information through message passing. Thus, a temporally consistent global view is hardly ever required, and usually not even observable. However, by combining all local views, a complete snapshot of the system can be obtained which does not need to be consistent, i.e., local copies of global variables don't need to have the same values.

For example, the accessibility relation $\mathbf{R}$, which describes the transition from one global state to the next, and the set R_{s_t}, which provides the set of all states that can be reached from an initial state s_t, are not observable at run-time since we do not know at all times the actions of each individual process. In general, we cannot predict which global state will be the next state in an execution sequence since the set of possible next states allows for any one or even all of the local processes to make progress. Only at communication points can synchronization ever be achieved. During consecutive executions of a distributed program, based on the same initial state, there thus exist many different possible execution sequences which nevertheless will arrive at the same final state. This is caused by the non-determinism of the progress of the independent processes which provides a large number of possible interleavings and thus a correspondingly large number of possible states.

For processes whose proofs are non-interfering we thus want to concentrate on the *local* accessibility relation $\mathbf{R}_i$ which provides a possible next state for each individual process P_i.

Definition 5.1 *The* projection function Π_i *provides the local view of a process P_i such that $\Pi_i(\mathbf{R}) = \mathbf{R}_i$, as given in Definition 5.2. From the proof outline $\mathfrak{V}_i$ of the individual process P_i, the projection function obtains the local assertion ϕ_t^i on the statement t^i in P_i, describing the local state s_t^i.*

Definition 5.2 *The* local accessibility relation $\mathbf{R}_i$ *for process P_i is obtained by using the projection function* Π *on the global accessibility relation, i.e., $\mathbf{R}_i = \Pi_i(\mathbf{R})$. It provides the next state $(s_{t_j}^i)^+$ that can be reached from the local state $s_{t_j}^i$. It is defined only between communication points, i.e., concurrency points.*

An execution sequence for a local process P_i can be expressed by using the local accessibility relation $\mathbf{R}_i$, such that $s_{t0}^i \mathbf{R}_i s_{t1}^i, s_{t1}^i \mathbf{R}_i s_{t2}^i, \cdots, s_{t_{n-1}}^i \mathbf{R}_i s_{t_n}^i$ is a permitted sequence of states. Thus, $s_{t_n}^i \in R_{s_{t0}^i}$ means that $s_{t_n}^i$ is an element in the set of future local states of s_{t0}^i. The next local state can thus be determined based on an initial state, the information that was received during communication, and the path that is taken through the program.

Theorem 5.1 *Each process $P_i, 1 \le i \le n$, constitutes a sequential program which forms a* well-ordered sequence T_i *between communication points or points of non-determinism, based on an initial state s_{t0}^i, where*

$$T_i = \{s_{t0}^i, s_{t1}^i, s_{t2}^i, \cdots, s_{t_k}^i, (s_{t_k}^i)^+, \cdots, s_{t_n}^i\}$$

and where s^i_{t0} is a communication point and $s^i_{t_n}$ is the next communication point in the local execution sequence.

(Note that $s^i_{t_j}$ precedes $s^i_{t_k}$ in T_i if $j < k$ and thus $s^i_{t_j}$ is executed before $s^i_{t_k}$. If $k = j+1$ then $s^i_{t_k}$ is the local state immediately following $s^i_{t_j}$ and there exists no other state in T_i that comes in between. This is also denoted by writing $(s^i_{t_j})^+$.)

Proof: A global proof of a distributed system consists of the conjunction of the non-interfering sequential proofs $\mathfrak{V}_i$. Between concurrency points it does not matter which process is making progress as long as at least one process does (Definition 3.3). Without loss of generality we can therefore assume that, between concurrency points, only one process P_i is making progress while all other processes are idle. This means that between concurrency points $c1$ and $c1 \in \mathfrak{S}$

$$(\exists s_{r_k} \in \mathfrak{S})(s_{c1}\mathbf{R}s_{r0}, s_{r0}\mathbf{R}s_{r1}, \cdots, s_{r_n}\mathbf{R}s_{c2})\wedge$$
$$(\exists i)(s^i_{c1}\mathbf{R}_i s^i_{r0}, s^i_{r0}\mathbf{R}_i s^i_{r1}, \cdots, s^i_{r_n}\mathbf{R}_i s^i_{c2} \wedge\ (\forall j \neq i)(\forall k)(\Pi_j(s_{r_k}) = s^j_{r_k} = s^j_{c1}))$$

where the $s^i_{r_k}$ are states local to process P_i which can be obtained through the projection function $\Pi_i(s_{r_k})$. Thus, for each individual step $s_{r_{k+1}} = s^+_{r_k}$ in the set of global states $\mathfrak{S}$ between the concurrency points the following must hold:

$$(\forall j \neq i)(\Pi_j(s_{r_{k+1}}) = \Pi_j(s_{r_k})) \wedge\ (\Pi_i(s_{r_{k+1}}) = (s^i_{r_k})^+)$$

Once P_i reaches its next concurrency point it will wait for the process(es) it needs to synchronize with and it becomes idle. Then another process will proceed. The accessibility relation $\mathbf{R}$ as defined in 3.3 allows for this consecutive sequential execution of the individual processes. Thus, $\Pi_i(E)$ provides the sequential proof $\mathfrak{V}_i$ of an individual process P_i between concurrency points and thus a well-ordered execution sequence between communication points can be obtained and labeled as T_i. □

We now define the local accessibility relation $\mathbf{R}_i$ to obtain this well-ordered execution sequence directly from the sequential proof.

6 Subsumption Refined for a Set of Local Processes

As mentioned before, in a compositional proof systems we do not have assertions on global states. Therefore, we use local assertions on each local execution sequence. For a specific process P_i, a local assertion $\phi^i_{t_j}$ provides an assertion describing the local state $s^i_{t_j}$, and the program state and a statement of the program are related as in Definition 4.1. We now assume a specific process P and we will use no superscript to indicate that we are now looking at local expressions only. Instead we will label each statement within the local program as t_j which is associated with a corresponding post-assertion ϕ_{t_j} without any superscript indicating a particular process.

Using only local processes, we can now refine the subsumption model given earlier into a model designed for the local environments of a compositional proof system.

Theorem 6.1 *For an arbitrary process P, the assertion ϕ_{t1} describing a state s_{t1} local to P* $(\mathfrak{V},\mathfrak{S})$-subsumes *the assertion ϕ_{t2} describing a state s_{t2} local to P, along a path e, if and only if*

$$(s_{t2} \in R_{s_{t1}}) \wedge (\phi_{t1} \overset{e}{\rightarrow} \phi_{t2}) \wedge$$
$$(\forall s_r \in \mathfrak{S}_{local})(s_r \in R_{s_{t1}} \wedge s_{t2} \in R_{s_r} \rightarrow ((\phi_{t1} \overset{e}{\rightarrow} \phi_r) \wedge (\phi_r \overset{e}{\rightarrow} \phi_{t2})))$$

where the r are intermediate statements in the program, ϕ_r are the corresponding assertions and the s_r describe states in the execution sequence which are implied by assertions in the proof outline.

Proof: Instead of arbitrary states s_i, s_j we now use local states s_t associated with program statements t. Thus, based on Definition 4.6, we replace states s_i and s_j by the local states s_{t1} and s_{t2}, and the intermediate state s_r in 4.6 now becomes a local state s_r. From this substitution, immediately the above theorem follows. □

We again introduce concurrency points as subsumption boundaries. This is based on efficiency reasons as well as error-detecting ability and the soundness and completeness of the proof system used [7]. We want to verify the correct execution of each statement, and we also want to ensure that data that was obtained from other processes meets its specifications and that the current states are permitted states. Thus, we restrict the subsumption to be performed between communication points only, i.e., the concurrency points, and we verify the complete state after each communication.

Theorem 6.2 *The assertion ϕ_{t1} describing the local state s_{t1}* $(\mathfrak{V},\mathfrak{S})$-subsumes *the assertion ϕ_{t2} describing s_{t2} between two statements c1 and c2 along a path e, where $s_{c2} \in R_{s_{c1}}$, if and only if*

$$(s_{t1} \in R_{s_{c1}} \cup \{s_{c1}\}) \wedge (s_{t2} \in R_{s_{t1}}) \wedge (s_{c2} \in R_{s_{t2}}) \wedge (\phi_{t1} \overset{e}{\rightarrow} \phi_{t2}) \wedge$$
$$(\forall s_r \in \mathfrak{S}_{local})(s_r \in R_{s_{t1}} \wedge s_{t2} \in R_{s_r} \rightarrow ((\phi_{t1} \overset{e}{\rightarrow} \phi_r) \wedge (\phi_r \overset{e}{\rightarrow} \phi_{t2})))$$

where c1 and c2 are the backward and forward concurrency points, respectively, corresponding to two communication points in the proof outline.

Proof: As in Theorem 6.1, we use the local states $t1$ and $t2$ instead of the states s_i and s_j, and we also use the local state s_r instead of the global state s_r as the intermediate state. The theorem then follows immediately from Lemma 4.1. □

The $(\mathfrak{V},\mathfrak{S})$-subsumption for the local processes can also be extended for partial assertions. This refined version can easily be obtained through a similar derivation from Definition 4.9.

Theorem 6.3 *The assertion ϕ_{t1} describing the local state s_{t1}* $(\mathfrak{V},\mathfrak{S})$-subsumes *the partial assertion ψ_{t2} on state s_{t2} between two communication points c1 and c2 along a path e, where $s_{c2} \in R_{s_{c1}}$, if and only if*

$$(s_{t1} \in R_{s_{c1}} \cup \{s_{c1}\}) \wedge (s_{t2} \in R_{s_{t1}}) \wedge (s_{c2} \in R_{s_{t2}}) \wedge (\phi_{t1} \overset{e}{\rightarrow} \psi_{t2}) \wedge$$
$$(\forall s_r \in \mathfrak{S}_{local})(s_r \in R_{s_{t1}} \wedge s_{t2} \in R_{s_r} \rightarrow ((\phi_{t1} \overset{e}{\rightarrow} \psi_r) \wedge (\psi_r \overset{e}{\rightarrow} \psi_{t2})) \wedge$$
$$\psi_{t2} \in \Phi_{t2} \wedge \psi_r \in \Phi_r)$$

Proof: We use the same reasoning as for Theorem 6.2 and Definition 4.9. The theorem then follows immediately. □

7 Subsumption in a Program

The following discussion provides a model for subsumption for each sequential process in a distributed system. The general subsumption rule given in Definition 4.9 can be refined for the programming constructs such as branching, looping, and non-branching execution.

The computation and program counters are part of each local state. To simplify the relationship between states and statements, the computation counter is added at run-time and incremented by one for each statement execution. It thus provides the execution sequence for any program execution. This allows us to express relationships between states which are now firmly associated with a statement. The value of the communication counter for a specific statement may vary from execution to execution. Also, all program execution in a loop is unrolled so that each statement in the program will have a unique label.

Definition 7.1 *The binary relation $\prec$ on the set of statements in an execution sequence indicates, based on the computation counter, which statement is executed before another. Thus, $t_1 \prec t_2$ means that statement t_1 is executed before statement t_2. Therefore, in an execution sequence the state s_{t_2}, which is true after executing statement t_2, must be reachable from state s_{t_1}, at statement t_1, i.e. $s_{t_2} \in R_{s_{t_1}}$.*

Theorem 7.1 *The binary relation $t_1 \prec t_2$ is anti-symmetric and transitive.*

Proof: The anti-symmetry of the relation can be shown by examining an execution sequence and the set of future states. If $t_1 \prec t_2$ then s_{t_2} must be reachable from s_{t_1} and thus $s_{t_2} \in R_{s_{t_1}}$. This also means that the computation counter at s_{t_1} is less than the computation counter at s_{t_2}. Because of a resulting conflict with the computation counters, it can never be true that $s_{t_1} \in R_{s_{t_2}}$. Therefore, the binary relation $\prec$ is anti-symmetric.

For transitivity we need to prove that $((t_1 \prec t_2) \wedge (t_2 \prec t_3)) \rightarrow (t_1 \prec t_3)$. Based on the computation counter, if t_1 is executed before t_2 then $s_{t_2} \in R_{s_{t_1}}$. Similarly, for $t_2 \prec t_3$, $s_{t_3} \in R_{s_{t_2}}$. Due to the transitivity of the reachable set R, we can conclude that $s_{t_3} \in R_{s_{t_1}}$ and thus $t_1 \prec t_3$. Thus, the binary relation $\prec$ is transitive. $\Box$

Definition 7.2 *The binary relation $=$ on the set of program statements in an execution sequence indicates, that the program statements examined have the same computation and program counters, and that they have the same state associated with them. Thus $t_1 = t_2$ means that the two states associated with t_1 and t_2, s_{t_1} and s_{t_2}, are identical. This relation is reflexive.*

Theorem 7.2 *For any execution sequence, the binary relation on the set of statements $(t_1 \leq t_2) \Leftrightarrow (t_1 \prec t_2 \vee t_1 = t_2)$ describes a total order based on the computation counter.*

Proof: The statements t_1 and t_2 provide the value of the computation counter at which the respective states s_{t_1} and s_{t_2} are or become true. Since the relation $=$ is reflexive and it is combined with the relation $\prec$, which is anti-symmetric and transitive, it follows that $\leq$ must be a partial order, i.e.,

$$(\forall(t_1,t_2))[((t_1,t_2) \in S_= \vee \ (t_1,t_2) \in S_\prec) \Leftrightarrow ((t_1,t_2) \in S_\leq)]$$

where the sets S indicate the sets of all ordered pairs in the corresponding relations.

To prove a total order, we need to show that any two states within an execution sequence are related by the relation $\leq$. For an execution sequence $s_{t_0}\mathbf{R}_i s_{t_1}, s_{t_1}\mathbf{R}_i s_{t_2}, \cdots, s_{t_{n-1}}\mathbf{R}_i s_{t_n}$ and two arbitrary states s_{t_i} and s_{t_j} within this sequence, we can determine $R_{s_{t_i}}$ and $R_{s_{t_j}}$ and therefore whether $t_i \prec t_j$ or $t_j \prec t_i$ as described in Definition 7.1. Thus, for any pair of states s_{t_i} and s_{t_j} if $\neg(t_i = t_j)$ then $(t_i \prec t_j) \vee \ (t_j \prec t_i)$ must hold, i.e., one of the statements must be executed before the other. Thus, $\leq$ forms a total order. □

In contrast to Definition 4.9 where we "attached" the execution sequence between two states and performed $(\mathfrak{V},\mathfrak{S})$-subsumption, we now look at program statements in a program with unrolled loops and subsume between them. Thus, we examine the set of all statements $\mathfrak{T}$ in an execution sequence and subsume between them for the different possible types of programming constructs. We call this revised model, based on the set of assertions and program statements, $(\mathfrak{V},\mathfrak{T})$-subsumption.

In order to determine the status of the current computation, we define a function $at(t_k)$ which compares the value of the current program counter with the value of the program counter for statement t_k.

Definition 7.3 *The function $at(t_k)$, where t_k is a branching or looping statement of the program, checks if the current program counter corresponds to the program counter value for statement t_k. If $at(t_k)$ is satisfied then the guard is satisfied and we have an execution sequence $e = s_{t_j}\mathbf{R}_i s_{t_{j+1}}, \cdots, s_{t_{k-1}}\mathbf{R}_i s_{t_k}$ where ϕ_{t_k} must be true at s_{t_k}. Otherwise, if $at(t_k)$ evaluates to false, s_{t_k} is not the next state in e.*

Definition 7.4 *A* tag point *is a program statement that denotes either the start or the end of a particular program construct such as a branching, looping, or strictly non-branching construct, or a communication point.*

7.1 Non-Branching Programs

A program segment without any branching can follow only one path. Thus, the subsumption process evaluates, starting at an initial state, if there are any (partial) assertions along this path which are implied by the current assertion according to Definition 4.9 with respect to the predicate transformations. These assertions can then be $(\mathfrak{V},\mathfrak{T})$-subsumed.

We can describe a non-branching program segment with no embedded communication points using the local accessibility relation $\mathbf{R}_i$. Starting at an initial state s_{t_i} there exists only one possible state sequence to the end of the non-branching segment.

Definition 7.5 *The* local accessibility relation $\mathbf{R}_i$ *for a non-branching program gives the (unique) next state based on the current state in process P_i. $s_{t_j}\mathbf{R}_i s_{t_{j+1}}$ means that t_{j+1} is the statement to be executed immediately after statement t_j and there exists no state between s_{t_j} and $s_{t_{j+1}}$.*

An execution sequence for a strictly non-branching program between two tag points k_1 and k_2 and no intermediate communication points is

$$s_{k_1}\mathbf{R}_i s_{t_1}, s_{t_1}\mathbf{R}_i s_{t_2}, \cdots, s_{t_n}\mathbf{R}_i s_{k_2} \wedge (\forall s_r \in \mathfrak{S}_i)(s_r \in R_{s_{k_1}} \wedge s_{k_2} \in R_{s_r} \rightarrow \exists j(s_r = s_{t_j}))$$

and $\mathfrak{S}_i$ denotes the set of local states of process P_i.

Since we cannot subsume across communication boundaries, there exist three different cases for strictly non-branching segments: (1) The initial state is a tag point k_1 and the terminal state is a tag point k_2, and neither k_1 nor k_2 are communication points; (2) The initial state is a communication point c_1 and the terminal state is a tag point, k_2. In this case, c_1 becomes a tag point; (3) The initial state is a tag point k_1 and the terminal state is a communication point, c_2. In this case, c_2 becomes a tag point.

From now on we will divide non-branching segments containing communication points into separate entities where the communication points become tag points and where the other tag point in the sequence is set by either another communication point or by the beginning or end of a new programming construct.

Theorem 7.3 *In a program segment enclosed by tag points k_1 and k_2, either of which may be a communication point, with no communication points contained in the execution sequence, the post-assertion on a statement t_1, ϕ_{t_1},* $(\mathfrak{V}, \mathfrak{T})$-subsumes *the partial assertion ψ_{t_2} on t_2, along a path e, if and only if*

$$(k_1 \leq t_1 \prec t_2 \prec k_2) \wedge (\phi_{t_1} \xrightarrow{e} \psi_{t_2}) \wedge$$
$$(\forall r)(t_1 \prec r \prec t_2 \rightarrow (\exists \psi_r)((\phi_{t_1} \xrightarrow{e} \psi_r) \wedge (\psi_r \xrightarrow{e} \psi_{t_2}) \wedge \psi_{t_2} \in \Phi_{t_2} \wedge \psi_r \in \Phi_r))$$

where t_1, t_2, and r indicate the individual statements and their labels along e, and k_1 and k_2 limit the scope of the subsumption in the program segment.

Proof: Consider Definition 4.9 which defines $(\mathfrak{V}, \mathfrak{S})$-subsumption for local states. We re-write the term $(s_{t_1} \in R_{s_{c_1}} \cup \{s_{c_1}\}) \wedge (s_{t_2} \in R_{s_{t_1}}) \wedge (s_{c_2} \in R_{s_{t_2}})$ in 4.9 as an execution sequence using tag points (k_1, k_2) rather than communication points (c_1, c_2) since tag points can be used to describe any arbitrary subsequence not including communication points:

$$s_{k_1}\mathbf{R}_i s_{t_n}, s_{t_n}\mathbf{R}_i s_{t_{n+1}}, \cdots, s_{t_{m-1}}\mathbf{R}_i s_{t_m}, s_{t_m}\mathbf{R}_i s_{k_2} \wedge$$
$$(\exists r1)(s_{t_{r1}} \models \phi_{t_1}) \wedge (\exists r2)(s_{t_{r2}} \models \phi_{t_2}) \wedge s_{t_{r2}} \in R_{s_{t_{r1}}}$$

from which we can obtain the expression $(k_1 \prec t_1 \prec t_2 \prec k_2)$ by assigning the respective statement labels to each state. However, due to $(s_{t_1} \in R_{s_{c_1}} \underline{\cup \{s_{c_1}\}})$, we need to allow for t_1 to be equal to c_1. However, since we substitute k_1 and k_2 for c_1 and c_2, $(k_1 \leq t_1 \prec t_2 \prec k_2)$ follows.

Similarly, we can rewrite $(s_r \in R_{s_{t_1}} \wedge s_{t_2} \in R_{s_r})$ from 4.9 as an execution sequence

$$s_{t_1}\mathbf{R}_i s_{t_k}, \cdots, s_{t_r}\mathbf{R}_i s_{t_{r+1}}, \cdots, s_{t_l}\mathbf{R}_i s_{t_2} \wedge (s_{t_1} \models \phi_{t_1}) \wedge (s_{t_2} \models \phi_{t_2}) \wedge s_{t_2} \in R_{s_{t_1}}$$

Again, $(t_1 \prec r \prec t_2)$ follows immediately. Forming the conjunction of all terms and combining them with the corresponding assertions in Definition 4.9, the above theorem immediately follows. □

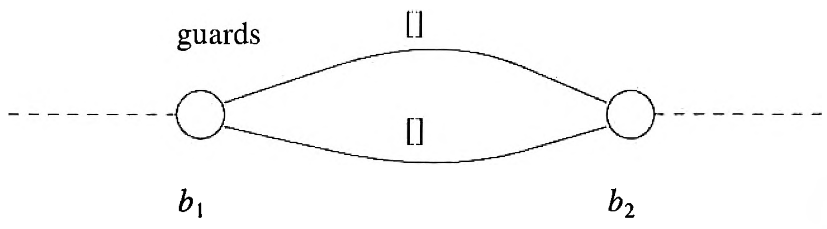

Figure 4: Two possible execution sequences between tag points b_1 and b_2.

7.2 Branching Programs

Branching occurs when we use an alternative statement. We now define a local accessibility relation $\mathbf{R}_i$ which allows multiple paths between two tag points b_1 and b_2, where the tag points are the backward and forward concurrency point for each sequence, respectively, i.e. the delimiters of the branching construct. Figure 4 shows a branching construct with tag points b_1 and b_2.

Definition 7.6 *The* local accessibility relation $\mathbf{R}_i$ *for a branching program gives a next state based on the current state in process P_i. $s_{t_j}\mathbf{R}_i s_{t_{j+1}}$ means that state $s_{t_{j+1}}$ is the state immediately following s_{t_j}, and there exists no state in between. It is possible for multiple next states to exist, i.e., $s_{t_j}\mathbf{R}_i s_{t_{j1}}$ or $s_{t_j}\mathbf{R}_i s_{t_{j2}}$ and $s_{t_{j1}} \neq s_{t_{j2}}$.*

A set of execution sequences for a branching program between the tag points b_1 and b_2 displays the following properties:

$$s_{b_1}\mathbf{R}_i s_{t_k}, s_{t_k}\mathbf{R}_i s_{t_{k+1}}, \cdots, s_{t_{m-1}}\mathbf{R}_i s_{t_m}, s_{t_m}\mathbf{R}_i s_{b_2} \wedge$$
$$(\exists s_{t_{k1}}, s_{t_{k2}} \in R_{s_{b_1}})(s_{b_2} \in R_{s_{t_{k1}}} \wedge s_{b_2} \in R_{s_{t_{k2}}} \wedge s_{t_{k1}} \notin R_{s_{t_{k2}}} \wedge s_{t_{k2}} \notin R_{s_{t_{k1}}})$$

which indicates that there exist at least two possible, disjoint paths between the concurrency points b_1 and b_2, one containing $s_{t_{k1}}$ and the other containing $s_{t_{k2}}$.

When a branch is selected, its branching condition has to be true. This means that at statement t_b, which contains the branching condition for a particular branch, $at(t_b)$ implies the branching condition B, i.e, $at(t_b) \xrightarrow{e} B$ as in Definition 7.3.

In a branching program segment enclosed by tag points b_1 and b_2, the subsumption rule for non-branching programs segments will be used for the non-branching segments in each individual branch. In order to subsume across the tag point b_2 that terminates the branching constructs, all branches need to agree on the (partial) assertions to be subsumed from the post-assertion on b_2. For this we will use the following rule:

Theorem 7.4 *The partial assertion ψ_{b_2} on b_2 can be $(\mathfrak{V}, \mathfrak{T})$-subsumed if and only if all assertions associated with states s_{t_j} for which s_{b_2} is the next state, i.e. $s_{t_j}\mathbf{R}_i s_{b_2}$, can subsume ψ_{b_2}. Thus,*

$$(\forall s_{t_j})((t_j \prec b_2 \wedge s_{t_j}\mathbf{R}_i s_{b_2}) \rightarrow \phi_{t_j} \xrightarrow{e} \psi_{b_2})$$

Proof: The proof follows directly from Definition 4.9. We can safely assume that there are no communication statements located between t_j and b_2 since t_j is the last statement at the end of a branch and b_2 is the statement immediately following. Thus, since there are no intermediate statements, we consider only the requirement $(\phi_{t_1} \xrightarrow{e} \psi_{t_2})$, which for our case turns into $(\phi_{t_j} \xrightarrow{e} \psi_{b_2})$.

If we want to allow subsumption that is valid for an arbitrary execution sequence (or path) e through the program, we need to account for every possible sequence. Thus, although it may be possible to subsume a (partial) assertion for a particular path e_1, it may not be possible for a different path e_2. Therefore, it is necessary for all assertions that immediately precede the execution of b_2 to subsume the same partial post-assertion on b_2. Only then the subsumption can be performed independent of the path taken. By adding the quantifier on the above condition, the theorem follows. □

This theorem shows that,when we subsume across the tag point indicating the end of a branching construct, we need to ensure that the subsumable conjuncts can be derived for all paths. If this is not the case then subsumption cannot be performed across this boundary.

If a communication point is located inside any of the branches, we can simply divide the program into branching and non-branching segments and apply Theorems 7.3 and 7.4 where appropriate, respectively.

7.3 Programs containing Loops

Looping constructs allow subsumption within the whole loop structure when we unroll it. Looping involves temporal dependencies between assertions for the individual iterations of the loop. For example, an assertion in the loop which can be subsumed during the first iteration may not be subsumed during later iterations. Thus, we will always unroll any loops.

A loop terminates if none of the guards can be evaluated to true, and for each program execution the number of actual loop iterations may vary. In this paper we only consider terminating loops. Figure 5 shows how a loop can be unrolled and examined. l_1 indicates the start of the looping construct, and l_2 marks the end. The intermediate circles denote the re-evaluation of the loop guard. Thus, l_1 and l_2 are tag points that determine the scope of the looping construct.

If a loop is entered, the looping condition must be true. If t_{l_1} is the statement where the looping condition is verified, then $at(t_{l_1})$ must imply the looping condition B, i.e., $at(t_{l_1}) \xrightarrow{e} B$. Similarly, the negation of the looping condition must be true when the loop ends, i.e., $at(t_{l_2}) \xrightarrow{e} \neg B$ where t_{l_2} is the terminating statement of the loop.

In a looping program segment enclosed by tag points l_1 and l_2, the subsumption rule for non-branching programs segments will be used for the non-branching segments within the loop. In order to subsume across the tag point l_2 that terminates a looping construct, all branches that combine at l_2 need to agree on the (partial) assertions to be subsumed. It is important to note that, if the looping condition is not true, the program execution will skip directly to the end of the loop and thus the pre-assertion on l_1 needs to subsume the same partial conjuncts as all branches.

We will use the following rule for looping constructs:

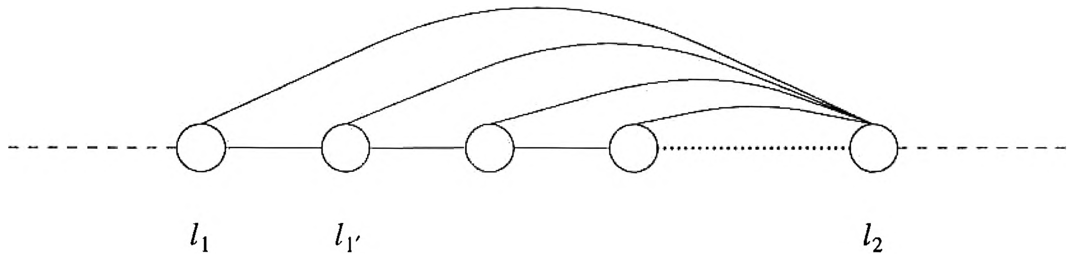

Figure 5: An unrolled loop between two tag points l_1 and l_2 bounding the looping construct.

Theorem 7.5 *The partial assertion ψ_{l_2} on l_2 can be $(\mathfrak{V},\mathfrak{T})$-subsumed if and only if all assertions associated with states s_{t_j} for which s_{l_2} is the next state, i.e. $s_{t_j}\mathbf{R}_i s_{l_2}$, can subsume ψ_{l_2}. Thus,*

$$(\forall s_{t_j})((t_j \prec l_2 \wedge s_{t_j}\mathbf{R}_i s_{l_2}) \rightarrow \phi_{t_j} \xrightarrow{e} \psi_{l_2})$$

Proof: This theorem is identical to Theorem 7.4 which describes multiple branching, except that it uses the label l_2 instead of b_2. Since loop constructs simply describe multiple consecutive branching, the proof immediately follows from the proof of Theorem 7.4. □

Note that subsumption can be performed independent of the number of iterations in the loop as long as there are at least two iterations. This is due to the fact that for the first entry into the loop, the pre-assertion to the loop will be used to subsume partial assertions for the first statement in the loop. For consecutive iterations, the last assertion inside the loop body is used to subsume partial assertions on the first, repeated, statement in the loop as the loop is unrolled ($l_{1'}$. The subsumption will then be the same for all consecutive iterations, i.e. $l_{1''}$, etc.

The differentiation between these two cases also allows for an optimized static analysis since we can evaluate the obtained assertion for one or the other, depending on the current iteration number.

The different rules for branching, looping, and non-branching program segments can be combined to obtain all other possible combinations of statements and program flows with and without intermediate communication points.

8 Conclusion

In this paper we introduced an abstract model describing the relationship between program states, statements, and assertions. This model was used as the foundation for the development of a set of subsumption rules that would allow for the removal of (partial) assertions to be embedded into the program as executable assertions for error detection at run-time, based on redundancy.

We can infer that the reduced set of assertions, after the subsumption has been performed, will allow for the detection of the same set of errors as the complete set of assertions. The goal will be to show that temporal subsumption can make error detection more efficient in distributed algorithms, in addition to maintaining the same fault latency as the complete set of assertion. This is especially important for fault-tolerant responsive systems, where we want to detect errors as soon as possible with very little computational overhead.

Continued development of this model as well as experimental results of a prototype subsumption model will be discussed in future reports.

References

[1] D. Heydtmann. Subsumption in modal logic. *M.S. Thesis*, Computer Science Department, University of Missouri-Rolla, 1993.

[2] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[3] J. Jou and J. Abraham. Fault-tolerant matrix arithmetic and signal processing on highly parallel computing structures. *Proceedings of the IEEE*, 74(5):732–741, May 1986.

[4] G.M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.

[5] D.W. Loveland. *Automated Theorem Proving*, chapter 4. Number 6 in Fundamental Studies in Computer Science. North-Holland, New York, 1978.

[6] H. Lutfiyya, M. Schollmeyer, and B. McMillin. Fault-tolerant distributed sort generated from a verification proof outline. *2nd Responsive Systems Symposium*, 1992. Springer Verlag.

[7] H. Lutfiyya, M. Schollmeyer, and B. McMillin. Formal generation of executable assertions for application-oriented fault tolerance. Technical Report CSC 92-15, UMR Department of Computer Science, 1992.

[8] B. McMillin and L. Ni. Executable assertion development for the distributed parallel environment. *Proceedings of the 12th International COMPSAC*, pages 284–291, October 1988.

[9] A. Mili. Self-checking programs: An axiomatisation of program validation by executable assertions. *Proceedings of the 11th International Symposium on Fault-Tolerant Computing*, pages 118–120, 1981.

[10] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

A Justification of the General Subsumption Rule

In Definitions 4.2 and 4.3 we introduce two different notations for subsumption. The first one,

$$(\forall x)C(x) \rightarrow (\forall x)D(x) \tag{1}$$

uses the universal closure on each clause individually, whereas the second one requires both clauses to be quantified together:

$$(\forall x)(C(x) \rightarrow D(x)) \tag{2}$$

From predicate calculus it follows that if 2 is valid then 1 must be valid as well. Thus 2 implies 1. However, we will now show that for our case, where we retain quantifiers in the assertions, 1 does not always hold, and we will justify the use of 2 instead as the basic rule for subsumption for the remaining sections of the paper.

Let $\mathcal{M}$ be a model for which the following condition holds: $\mathcal{M} \not\models (\forall x)(C(x) \rightarrow D(x))$. Suppose that $(\forall x)C(x) \rightarrow (\forall x)D(x)$ is valid. Let $\mathcal{N}$ be another model such that $\mathcal{N} = \{a \in \mathcal{M} | \mathcal{M} \models C(a)\}$, so $\mathcal{N} \models (\forall x)C(x)$. Let $b \in \mathcal{M}$ with $\mathcal{M} \not\models (C(b) \rightarrow D(b))$, i.e., $\mathcal{M} \models C(b)$ but $\mathcal{M} \not\models D(b)$. Then $b \in \mathcal{N}$ and thus $\mathcal{N} \models D(b)$, which leads to a contradiction.

However, we can show that this is an invalid argument and thus it is not true that 1 implies 2.

The first problem arises when $C(x)$ contains an existential quantifier such as $C(x) = (\exists y)p(x, y)$. It is possible that the particular y that makes this condition true exists in $\mathcal{M}$ but not in the subset of it, $\mathcal{N}$. In that case, $\mathcal{N} \models (\forall x)C(x)$ is false.

Another problem is encountered if $D(x)$ is universally quantified, such as in an expression $D(x) = \neg(\exists y)q(x, y)$. It is possible that y does not exist in $\mathcal{N}$ and thus $D(x)$ will be true, but it may exist in $\mathcal{M}$ which then makes $D(x)$ false in $\mathcal{M}$.

These problems do not occur in the field of automated reasoning where Definition 4.2 was obtained from. There, all clauses are skolemized and the quantifiers removed. The *Axiom of Choice* is applied to select an arbitrary variable instantiation to make each clause true. However, during the execution of a program we only know at run-time which variable values are required to make each assertion true. Thus, the quantifiers in the assertions that describe possible ranges of values cannot be removed and the Axiom of Choice cannot be applied. Therefore, the stronger definition of subsumption has to be used, the Definition given in 4.3.