DEFINING CONDITIONAL INDEPENDENCE USING COLLAPSES

by

Shmuel Katz and Doron Peled

Technical Report #628

May 1990
Defining Conditional Independence Using Collapses

Shmuel Katz  Doron Peled
Department of Computer Science
The Technion
Haifa 32000, Israel

Abstract

Trace semantics is extended to allow conditional commutativity among operations. Conditional commutativity is obtained by identifying the context (the set of global states) in which operations are commutative using special predicates. These will allow collapsing execution histories into equivalence classes of conditional traces. Using this approach, it is possible that the execution of two operations will be dependent in one context and independent in another. The predicates allow defining a family of possible semantic definitions for each language, where each is an extension of previous standard definitions. Examples are shown when such a semantics is desired. As an example of an application, a proof method for total correctness is introduced.

1 Introduction

The interaction among atomic operations of a concurrent program is an issue essential to implementation [1] and verification [16, 25, 31]. Implementation is often concerned with the restriction that two operations accessing a mutual memory location (either read or write), cannot be executed in parallel when at least one access is a write [6]. However, this demand can be sometimes relaxed [8]. The effect of two independent concurrent operations is typically commutative.

Verification of concurrent programs can benefit from the commutativity among operations by reducing the set of program states which are considered, arranging commutable operations in some convenient order.

Partial order semantics as seen in theories such as traces [20, 21] or interleaving sets [15] gives an appropriate treatment for the independence (commutativity) among operations. However, such formulations have the disadvantage of dealing with a fixed independence relation [16, 25] which cannot fully cover commutativity situations. Trace semantics is extended in this paper to allow conditional dependency (independence) among operations. That is, instead of using a fixed dependency relation among program operations, a set of predicates is used that identify when executions of pairs of operations are considered independent.
Conditional commutativity can be conceived as a direct extension of trace semantics or of interleaving sets. In addition to being useful for verification, it has semantic justification of its own since the execution of two operations might produce two causally dependent events in one occurrence and independent events in another. Some interesting cases are:

- if two operations refer to indexed memory elements such as arrays or pointer expressions. For example, the transitions \( x := A[i] \) and \( A[j] := y \) would be considered dependent iff \( i = j \),

- if operations on communication channels are considered. Putting an element on a communication channel and removing an element from the channel are (directly) dependent iff they concern the same element [10],

- if the truth of an enabling condition is unchanged by an operation. The transitions \( \text{await } p - c < N \) and \( c := c + 1 \) achieve the same result in either order when \( p - c < N \) is already true, because incrementing \( c \) keeps this condition true.

Many definitions for semantics and verification methods suggest adding more structure to the set of interleaving sequences generated by program executions. These methods exploit some observations about the nature of concurrent programs:

**Locality** Some tasks or subtasks in different processes may execute independently without inter-communication. After identifying this independence, there is no reason to consider all the possible interleavings of events of these tasks.

**Sequentiality** Some tasks, constructed from sets of program segments belonging to different processes, can be identified such that the program behaves as if the involved processes collaborate to perform one task, then they (perhaps with some processes joined or retired) commence the second task, and so forth. Once conditions for this behavior are identified, there is no reason to consider states where one process is involved in one task and the other is involved in another.

In both cases, some of the global states are identified as “not interesting” (although they occur in some execution sequences) and special proof methods are devised to handle them. We will mention a few:

- Several works [2, 27, 19, 4, 5] allow defining different granularities of atomic operations. That is, recognizing that various intermediate states of some program segments (which are considered as a single atomic operation) are not interesting for the correctness of the program. This obviously reduces the number of states one has to consider.

- Sequentiality is reflected by the definition of communication closed layers [11]. Segments of different processes are identified to execute together a common task. The program is decomposed into a set of layers (this decomposition is orthogonal to the decomposition into processes) such that there is no interaction (communication) among different layers. Then, the program behaves as if the tasks are executed according to some predefined order. Identifying layers can be done using the proof rules of [12]. Using layers for verification is done in [30].
Reducing the state space of a concurrent program by considering a representative subset of interleavings is suggested in [15, 31, 14, 11]. It is observed that under some conditions of independence among program operations, it is possible to recognize classes of execution sequences that are indistinguishable up to permuting independent operations. For properties which cannot distinguish between sequences of such a class it is possible to use proof rules such as [16, 25]. This obviously can handle locality, because a sequence in which local independent tasks appear entirely one after the other (in any order) is a representative for all the interleavings in which these tasks are merged. As proved in [17], sequentiality can also be captured, choosing representative sequences that commence each layer at the same time.

We propose a semantics which allows exploiting both locality and sequentiality of a concurrent program. This work also provides a semantic framework for the works of [16, 31, 14, 11] which suggest using a reduced state space of global states. Moreover, since the semantics suggested here extends the definition of independence among program operations, a proof method which extends those works is also facilitated. We will show how this can be done. In Section 2, the semantics of conditional traces is introduced. In Section 3, the semantic approach is further described and demonstrated. As an application, it is shown how verification for total correctness can be done within the proposed semantics. Finally, Section 4 discusses when the semantics is most beneficial.

2 Conditional Traces

The semantics presented in this paper is defined within the framework of traces. However, it may be defined similarly with respect to partial orders among events, where interpreted slices (left closed subsets of events) represent the possible global states of the program (as in [15]). This would give a model known as interleaving sets. Translations between these two models often exist, although they do have some different assumptions in their construction, concerning infinite executions and fairness. Infinite conditional traces can also be defined as an extension of infinite traces [18], and evidently most other partial order semantics could be treated similarly. The programs considered will be given as collections of transitions, although of course it is possible to express the ideas directly in terms of a higher level language.

2.1 Collapses

Traces, as presented in [20, 21] are equivalence classes of strings of program operations \( T \) where each string can be viewed as an interleaved history of an execution. The strings in a single class are indistinguishable in the sense that concurrently executable operations are interleaved in either order. Traces are used to represent partial order semantics. Pairs of operations which can execute concurrently are represented as a symmetric and irreflexive relation \( I \subseteq T \times T \) called the independence. Note that this relation is over operations, and not events (occurrences of operations), and is assumed to be given for each semantic model. We use \( \alpha, \alpha_i, \alpha', \beta \ldots \)
for typical operations. Strings of operations are denoted by \( v, w, x, y, \ldots \) The empty string is denoted by \( \epsilon \).

A program \( Pr \) is a triple \( (T, g, \Theta) \) where \( T \) is a finite set of operations or transitions, \( g \) is a finite set of variables, and \( \Theta \) is a predicate called the initial condition. Each operation \( \alpha \in T \) is associated with a pair \( (en_\alpha, fa) \) where \( en_\alpha \) is a predicate (the enabling condition) and \( fa \) (the transformation) is a tuple of \( |g| \) terms over some fixed first order language. The predicates \( \Theta \) and \( en_\alpha \) for each \( \alpha \in T \) contain no free occurrences other than the variables of \( g \). The same holds for the variables used in the terms \( fa \).

The intuitive meaning of \( en_\alpha \) and \( fa \) is as follows: A (global) state of a program is an assignment function, associating to each of the program variables a value from its domain. In order that an operation \( \alpha \) executes from a global state \( \rho \), \( \rho \) must satisfy \( en_\alpha \). When \( \alpha \) executes, the new program state is \( f_\alpha(\rho) \).

An operation \( \alpha \) can be written as the guarded command \( (en_\alpha(g) \rightarrow g := fa(g)) \). The condition before the arrow controls when the operation may execute and the effect of the operation is to simultaneously assign the \( |g| \) expressions \( fa(g) \) into the set of of variables \( g \) respectively. Notice that in some cases, it is convenient to use partial functions for \( fa \). For example, when \( \alpha = (x \neq 0 \rightarrow y := z \div x) \), the term \( z \div x \) is defined only for a non-zero \( x \). We can always complete \( fa \) into a total function which returns some arbitrary values when \( \neg en_\alpha \) holds.

Since a typical operation changes only a subset of the program variables, we will consider only the part of the assignment which is not the identity. Thus, we write \( (\alpha \rightarrow (x, y, z) := (y, y, z)) \). For CSP-like [13] programs and shared memory programs (with await), a transformation into a set of operations appears in [22]. Petri-nets [26] can be represented as operations as shown in [25]. The notation \( \phi[fa(g)/g] \) means the result of substituting in \( \phi \) for each \( 1 \leq i \leq |g| \), the \( i^{th} \) term of \( fa(g) \) instead of each of the free occurrences of \( g_i \). The predicate transformer \( wp_\alpha(\phi) = en_\alpha(g) \land \phi[fa(g)/g] \) is called the weakest precondition [9]. For a predicate \( \phi \), \( wp_\alpha(\phi) \) returns the predicate that is satisfied exactly by states from which \( \alpha \) can execute and produce a state satisfying \( \phi \).

We now will redefine histories, traces and the independence conditions.

**Definition 2.1** A history of a program \( Pr \) is a pair \( s = (\sigma, v) \) where \( \sigma \) is an interpretation of \( g \) (that is, an assignment) called the initial state of \( s \), and \( v \in T^* \). Let \( n = |v|, v = \alpha_1\alpha_2\ldots\alpha_n \). We require that \( \sigma \models \Theta \) (\( \sigma \) satisfies \( \Theta \)). Furthermore, there exists a sequence of interpretations \( \sigma_0, \sigma_1, \ldots, \sigma_n \) with \( \sigma_0 = \sigma \) such that:

1. For each \( 1 \leq i \leq n \), \( \sigma_{i-1} \models en_{\alpha_i} \).
2. For each \( 1 \leq i \leq n \), \( \sigma_i = f_{\alpha_i}(\sigma_{i-1}) \).

For each history \( s = (\sigma, v) \), let the \( n^{th} \) interpretation in the above sequence, namely \( \sigma_{|v|} \) (which is a function of \( s \)) be denoted by \( fin_s \). This is called the final interpretation of \( s \). We use \( s, s', s, \tau \ldots \) for histories. It is obvious from the definition that for a history \( s = (\sigma, v) \), if \( w \) is a prefix of \( v \), then \( s' = (\sigma, w) \) is also a history.
Definition 2.2 A collapse of $Pr$ is a set of predicates $I = \{\Delta_{\alpha\beta} \mid \alpha, \beta \in T, \alpha \neq \beta\}$ such that $\Delta_{\alpha\beta} = \Delta_{\beta\alpha}$ satisfying:

1. $\forall y. \Delta_{\alpha\beta} \land e_{\alpha}(y) \land e_{\beta}(y) \rightarrow f_{\alpha}(f_{\beta}(y)) = f_{\beta}(f_{\alpha}(y))$ [commutativity when operations are enabled].
2. $\forall y. \Delta_{\alpha\beta} \land en_{\alpha}(y) \rightarrow (en_{\beta}(y) \leftrightarrow en_{\beta}(f_{\alpha}(y)/y))$ [if $\alpha$ is enabled, it does not change the enabledness of $\beta$].

A collapse defines the new independence condition. This independence requires both commutativity of the transformations made by both operations and preserving the enabledness (disabledness) conditions. The latter is less obvious and demands a further explanation. Suppose in one execution sequence $\alpha$ is executed immediately prior to $\beta$. Even when $f_{\alpha}$ and $f_{\beta}$ are commutative, it might be the case that if $\beta$ executes first, it disables $\alpha$. Another case is that $\alpha$ can execute before $\beta$ but $\beta$ cannot execute before $\alpha$ because $\alpha$ enables $\beta$, as in the case of the operations $\alpha = (true \rightarrow x := x + 1), \beta = (x > 0 \rightarrow x := 3)$ and a state in which $x = 0$.

The predicate $\Delta_{\alpha\beta}$ is used to identify the context (i.e., the set of global states) where $\alpha$ and $\beta$ are independent. That is, when $\Delta_{\alpha\beta}$ is true, if both the operations are enabled, their execution order can be reversed, obtaining the same result. Moreover, the execution of one operation does not disable nor enable the other. When $\Delta_{\alpha\beta}$ is false, the semantics we will define below does not allow changing the order among $\alpha$ and $\beta$ even if they are commutative. For a program $Pr$, it is sometimes possible to have many collapses. In verification there is no need to find the weakest possible predicates, and we consider below which ones are appropriate for semantic definitions.

Although the predicates in a collapse are symmetric, we shall see that there can be states for which operation $\alpha$ can be switched with $\beta$ when $\alpha \beta$ occurs, but they cannot be switched when $\beta \alpha$ occurs.

Lemma 2.3 If $s = (\sigma, za\beta y)$ is a history of $Pr$, $r = (\sigma, x)$ and $fin_r \models \Delta_{\alpha\beta}$ where $\Delta_{\alpha\beta}$ is in a collapse of $Pr$, then $s' = (\sigma, z\beta y)$ is also a history of $Pr$ and $fin_s = fin_{s'}$.

Proof. Let $r' = (\sigma, x\alpha)$. Since $\beta$ is enabled in $fin_r = f_{\alpha}(fin_{r'})$, by the second clause of definition 2.2 it must also be enabled in $fin_{s'}$. Thus, let $r'' = (\sigma, z\beta)$. Again, using the symmetry in the definition of a collapse, $en_{\alpha}$ holds in $fin_{s'}$. Let now $r_1 = (\sigma, za\beta)$ and $r_2 = (\sigma, z\beta \alpha)$. Again from the definition of a collapse (the first condition) it follows that $fin_{r_1} = fin_{r_2}$. The fact that $s'$ is a history and that $fin_{s'} = fin_{s''}$ follows from Definition 2.1 by a simple induction on the length of $y$.

A collapse induces an equivalence relation on histories of $Pr$ which have the same initial state. Two histories, $s$ and $s'$ are equivalent iff their initial states are the same and one of the strings is obtained from the other by commuting adjacent operations only when the independence among them holds. Thus the semantics of a program is determined by its collection of histories, and by its collapse.

Definition 2.4 The histories $s = (\sigma, v)$ and $s' = (\sigma, w)$ are equivalent (denoted $s \equiv s'$) iff there exists a sequence of histories $(\sigma, v_1), (\sigma, v_2), \ldots, (\sigma, v_n)$ with $v_1 = v$ and $v_n = w$ and for each $1 \leq i < n$ there exist $x, y \in T^*$, $\alpha, \beta \in T$ such that $v_i = za\beta y$, $v_{i+1} = z\beta y$ and for $r = (\sigma, x)$, $fin_r \models \Delta_{\alpha\beta}$. A conditional trace is an equivalence class of histories.
Conditional traces will be called simply traces in the sequel. Denote a trace as $t = [\sigma, v]$ where $\sigma$ is the mutual initial state and $(\sigma, v)$ is a member of the equivalence class. The interpretation $\text{fin}_s$ is generalized to traces by taking the final interpretation of any member of the equivalence class, as justified by Lemma 2.5 below. Traces will be denoted by $t, t', t_1, \ldots$. Using Lemma 2.5, the length of a trace can be defined as the length of any member of it. We will say that a trace $t$ satisfies $\varphi$ if $\text{fin}_t, \models \varphi$.

**Lemma 2.5**: If $s \equiv r$ then $\text{fin}_s = \text{fin}_r$ and the length of the string of $s$ is the same as the string of $r$.

**Proof.** Simple induction on the number of times an operation is commuted when transforming $s$ to $r$ and using Lemma 2.3.

Concatenation between two traces $t_1 = [\sigma, v]$ and $t_2 = [\rho, w]$, denoted $t_1t_2$, is defined when $\text{fin}_{t_1} = \rho$ as $[\sigma, vw]$. The prefix relation $\sqsubseteq$ between conditional traces is defined as $t_1 \sqsubseteq t_2$ iff there exists some $t_3$ such that $t_1t_2 = t_2$. It is said that $t_1$ is subsumed by $t_2$. If in addition, the length of $t_1$ is shorter than the length of $t_2$ by exactly one, it is said that $t_2$ is an immediate successor of $t_1$.

Two traces $t_1$ and $t_2$ of a program $Pr$ are consistent iff there exists some $t_3$ of $Pr$ such that $t_1 \sqsubseteq t_3$ and $t_2 \sqsubseteq t_3$. A run of $Pr$ is a maximal set of traces which are pairwise consistent. Denote runs by $\Pi, \Pi', \ldots$. The set of traces of a run $\Pi$ will be denoted as $\text{traces}(\Pi)$. An execution sequence of $\Pi$ is a maximal sequence of traces $t_0 t_1 t_2 \ldots$ of $\text{traces}(\Pi)$ such that for each $i \geq 0$, $t_{i+1}$ is an immediate successor of $t_i$.

By strengthening or weakening the independence conditions (i.e., the collapse), the traces can shrink or grow, respectively. For example, by choosing $\Delta_{\sigma, \beta} = \text{false}$ for each distinct pair of operations, each trace contains exactly a single execution sequence. This is exactly as in linear interleaving semantics, where equivalences among sequences are not explicit on a semantic level. Partial order semantics of traces [20, 21] allows each $\Delta_{\sigma, \beta}$ to be fixed as either true or false. For such a semantics, $\Delta_{\sigma, \beta}$ will be identically true for the pairs of operations generating events that are always commutative and always do not affect each others' enabledness, and will be identically false otherwise. (Sometimes operations in the same process are arbitrarily assumed to be ordered, so that the condition for such operations will be false even when otherwise it could be true.)

Lamport and Schneider [19] also utilize independence to simplify reasoning on concurrent programs. In their formalism, it is said that $\alpha$ commutes to the right with $\beta$ if whenever $\beta$ is executed immediately after $\alpha$, their order can be interchanged, producing the same net effect. This definition is obviously non-symmetric with respect to this pair of operations. Although the definition of a collapse requires that the predicates $\Delta_{\alpha, \beta}$ and $\Delta_{\beta, \alpha}$ are the same, symmetry can be broken by using a predicate which will allow permuting the operations only when they can be executed from this state one after the other in some given order. Furthermore, we show below how to expand commutativity to allow conditional right commutativity.

Assume that it is given that $\alpha$ commutes to the right with $\beta$, provided that $p_1$ holds. Then, $\Delta_{\alpha, \beta} = \varphi \land wp_1(\text{en}_\beta)$ allows $\alpha$ and $\beta$ to be commuted exactly in states when $\varphi$ holds and $\alpha$ can be executed followed by $\beta$.
Now, to see why the above formulation of (part of) a collapse captures the property of commuting to the right, observe that histories whose final interpretation satisfies $wp_a(ena)$ are those from which $a$ is enabled and, after executing $a$, the operation $\beta$ is enabled. That is, if $s = (\sigma, \nu)$ is a history of $Pr$ and $fin_s \models wp_a(ena)$, then $(\sigma, \nu \beta \alpha)$ is also a history of $Pr$.

Assume that $\Delta_{a, \beta}$ satisfies the conditions of Definition 2.2. Let $s = (\sigma, \nu)$ be a history of $Pr$ satisfying $\Delta_{a, \beta}$. Then, $s' = (\sigma, \nu \beta \alpha)$ is also a history of $Pr$. By Lemma 2.3, it holds that $s'' = (\sigma, \nu \beta a)$ is also a history of $Pr$ as required. Conversely, assume that $a$ commutes to the right with $\beta$ from all the histories which can be extended with $a \beta$ and whose final interpretation satisfies $\varphi_1$. Let $r = (\sigma, \nu)$ be a history of $Pr$ where $fin_r \models \varphi_1$ and $r' = (\sigma, \nu \beta \alpha)$ is another history of $Pr$ (hence, $fin_r \models \varphi_1 \land wp_a(ena)$). Thus, from right commutativity, $r'' = (\sigma, \nu \beta a)$ is also a history of $Pr$, with $fin_{r''} = fin_{r'}$. The conditions of Definition 2.2 hold: Condition 2 (for both pairs $\alpha, \beta$ and $\beta, \alpha$) stems from the fact that both $r'$ and $r''$ exist, and hence $a$ and $\beta$ are enabled in $fin_r$, $\alpha$ is enabled after the execution of $\beta$, and $\beta$ is enabled after the execution of $\alpha$. Condition 1 holds since $fin_{r''} = fin_{r'}$ and $\alpha$ and $\beta$ are enabled in $fin_r$.

In a similar manner, if in addition to the conditional right commutativity of $\alpha$ and $\beta$ discussed above it is also known that $\beta$ commutes to the right with $\alpha$ when $\varphi_2$ holds, then $\Delta_{a, \beta} = (\varphi_1 \land wp_a(ena)) \lor (\varphi_2 \land wp_\beta(ena))$ can be used. Notice that nothing is explicitly said about a state $s$ in which $\varphi_1$ holds but $\beta$ can be executed before $\alpha$ (or similarly, a state in which $\varphi_2$ holds and $\alpha$ can be executed before $\beta$). It is possible that $\alpha$ is not enabled in $s$ and $\beta$ makes it enabled.

Another definition of independence, by Best and Lengauer [8], is titled semantic independence. They show that every pair of events from a pair of operations can be commutative even when the same variables are used in both, because the property needed from each mutual variable is always invariant to events of the other operation. Their definition is however stronger than commutativity (that is, if two operations are semantically independent, then they are commutative) and concerns the possibility of distributed implementation. Their approach might also be generalized to a conditional invariance.

The semantics of a language can either be determined by picking a fixed collapse, as above, and combining it with the basic collection of traces to obtain the equivalence classes above, or by using the weakest collapse as the most general default. This collapse is true for $\alpha$ and $\beta$ exactly in those states for which the two conditions in the definition of a collapse are true. Thus it allows exploiting the independence of events whenever possible. The weakest collapse can be formulated as

$$\Delta_{a, \beta} = (ena(g) \land en_\beta(g) \land en_a[f_\beta(g)/g] \land en_\beta[f_\alpha(g)/g] \land f_\alpha(f_\beta(g)) = f_\beta(f_\alpha(g)))$$

The first disjunct gives the positive conditions in Definition 2.2, while the others treat cases where operations are not enabled.

When specific collapses are used, e.g., for verification (as will be seen in Section 3), they must be shown to imply the collapse given for the semantics of the language. That is, whenever the specific collapse is true, so is the collapse of the language semantics.
2.2 Conditional Traces as Partial Orders

A construction that transforms (ordinary) traces into partial order semantics and its opposite companion exist [20]. The existence of these transformations justify the consideration of traces as representing partial order semantics. In this subsection we investigate the conditions under which these transformations may be applied in the context of conditional traces. In the following discussion, for notational convenience we ignore the initial state element of histories and traces, as its explicit occurrence is orthogonal to the transformations. Thus, we denote by \([w]\) a trace where the history \((\sigma, w)\) is one of its histories with some initial state \(\sigma\). Similarly, we may simply write \(w\) instead of denoting the above history.

A partial order is a pair \(E = (E, \prec)\), where \(E\) is a set and \(\prec \subseteq E \times E\) is a transitive, irreflexive relation on \(E\). A linearization \(L\) of \(E\) is a total order \(L = (E, \prec)\) where \(\prec \subseteq\). Let \(#_aw\) be the number of times the symbol \(\alpha\) occurs in the string \(w\). The \(i^{th}\) symbol in \(w\) will be denoted by \(\alpha_i\). Let \(T_w = \{\alpha \mid \alpha \in T \land \#_aw > 0\}\) (the set of operations occurring in \(w\)). An occurrence of \(w\) (or, equivalently of \([w]\), because each operation appears the same number of times in each member of the equivalence class) is any pair \((\alpha, n)\) where \(\alpha \in T_w\) and \(1 \leq n \leq \#_aw\). Let \(\xi_w\) be the isomorphism from \(1 \ldots |w|\) to the set of occurrences of \(w\) defined as \(\xi_w(i) = (\alpha_i, |\{j \mid j \leq i \land \alpha_j = \alpha_i\}|)\). For example, \(\xi_{a\beta\beta\alpha\lambda}\) will map the integers \(1 \ldots 5\) to \((\alpha, 1), (\beta, 1), (\beta, 2), (\gamma, 1), (\alpha, 2)\) respectively. We identify histories with total orders by denoting a history as a sequence of occurrences rather than a simple string (so that no element occurs twice on such a sequence). For each string \(w\), define a total order among its occurrences \(\prec_w\) such that \(\xi_w(i) \prec_w \xi_w(j)\) iff \(i < j\). Define now a partial order relation \(\preceq_t = \bigcap_{w \neq t} \prec_w\) among the set of occurrences of a trace \(t\).

The opposite construction is to form a trace by taking the set of all the linearizations of the partially ordered events. Then, a string is constructed from each total order by ignoring the number part of each occurrence. For ordinary traces, this transformation is the inverse of the former transformation. Namely, by taking the linearizations of a partially ordered set of occurrences constructed from a trace \(t\), it is guaranteed that \(t\) is obtained back. (Obtaining the linearizations of a finite partial order and then applying the transformation which generates a partial order by taking the intersection always returns the original partial order.)

However, for conditional traces, it might happen that by applying the first transformation, and then applying the second on the result of the first, a larger set of histories is obtained. For example, it might happen that an operation \(\gamma\) which can be commuted with both \(\alpha\) and \(\beta\) changes the truth value of \(\Delta_{\alpha\beta}\). That is, \(\alpha\beta\gamma \equiv \beta\alpha\gamma\) which are also equivalent to \(\gamma\alpha\beta\), but these histories are not equivalent to \(\gamma\beta\alpha\).

Concretely, let \(\alpha : (\text{true} \rightarrow z := z + 2), \beta : (z > 0 \rightarrow z := z - 1), \gamma : (\text{true} \rightarrow z := z - 2), \Delta_{\alpha\beta} = (z \neq 0 \land z \neq -1), \Delta_{\alpha\gamma} = \text{true}, \Delta_{\beta\alpha} = (z \neq 1 \land z \neq 2).\) (This is the weakest collapse, obtained by substituting the appropriate terms and conditions to the formula at the end of the previous subsection.) Starting from an initial state in which \(z = 1\), there exists a trace which consists of the strings \(\alpha\beta\gamma, \beta\alpha\gamma, \alpha\gamma\beta, \gamma\alpha\beta, \beta\gamma\alpha\). When transforming this set of equivalent histories into a partial order, the three occurrences \((\alpha, 1), (\beta, 1)\) and \((\gamma, 1)\) become pairwise unordered, because for each pair of these occurrences there exists a trace in which the first precedes the second, and the other way around. Transforming the partial order back into
a set of histories results in $\gamma \beta \alpha$ in addition to the original histories.

To pin down the source of this phenomena, observe that in ordinary traces, each pair of operations is either constantly dependent or constantly independent, while in conditional traces, this is not the case. The key to identifying when conditional traces can be viewed as representations of partial order is to treat histories as total orders (using the mappings $\xi$ defined above) and observe the relative order among occurrences (rather than operations) in different histories of a trace.

Two occurrences $o_1 : (\alpha, i)$ and $o_2 : (\beta, j)$ of a trace $t$ can appear in the same order along each of $t$'s histories, or occur in different orders (other occurrences of $t$ involving $\alpha$ and $\beta$ can have a totally different relative order than $o_1$ and $o_2$). The pair of transformations above are inverses precisely if whenever $o_1$ and $o_2$ occur adjacent either they never can appear in the opposite order or they always can appear in either order, and commute. That is, the independence of occurrences is fixed. This condition, which can be proven sufficient and necessary for the transformations to be inverses, provides an intermediate level between the fixed independence for operations seen in the original traces, and the full generality of collapses. Whether or not the transformations are considered essential is a matter of taste. For purposes of verification and optimization, where the final results are the important consideration, they seem to be extraneous. Note, of course, that global invariants of a system are not affected by the choice of a collapse, since the same collection of states exists in the system, and only the grouping of histories is determined by the collapse chosen.

3 Examples

3.1 Two Programs

The following program (which is a slight modification of a program in [23]) computes the number of possible combinations when choosing $k$ out of $n$ distinct elements using the formula

$$\binom{n}{k} = \frac{n \times (n-1) \times \cdots \times (n-k+1)}{1 \times 2 \times \cdots \times k}$$

The left process repeatedly multiplies the numerator as the values of $y_1$ range between $n$ and $n - k + 1$, while the right process repeatedly divides the denominator as the values of $y_2$ range between $1$ and $k$. The operation $m_3$ allows $m_4$ to be executed only when the number of values multiplied is greater than the number of values divided. This guarantees that $m_4$ will always produce an integer result (and thus can be implemented as an integer division). The initial condition is

$$\Theta \equiv y_1 = n \land y_2 = 0 \land y_3 = 1 \land l = l_1 \land m = m_1$$

$$l_1 : \text{if } y_1 = (n - k) \text{ then halt}$$

$$l_2 : y_3 := y_3 \times y_1$$

$$l_3 : y_1 := y_1 - 1$$

$$l_4 : \text{goto } l_1$$

$$m_1 : \text{if } y_2 = 5 \text{ then halt}$$

$$m_2 : y_2 := y_2 + 1$$

$$m_3 : \text{await } y_2 \leq n - y_1$$

$$m_4 : y_3 := y_3 / y_2$$

$$m_5 : \text{goto } m_1$$
Following is a translation of the program into a set of operations. We have used the function
\[
\text{if}(\text{cond}, a, b) = \begin{cases} 
    a & \text{if } \text{cond} = \text{true} \\
    b & \text{if } \text{cond} = \text{false} 
\end{cases}
\]
which returns either its second or third argument, depending on the boolean value of its first argument. The variables \(l\) and \(m\) represent the program counters. Executing the command \text{halt} terminates the execution of a process. This is translated as assigning a special value to the process's program counter which disables all its enabledness conditions.

\[
(l = l_1 \rightarrow l := \text{if}(y_1 = (n - k), \text{halt}, l_2)) \\
(l = l_2 \rightarrow (l, y_2) := (l_1, y_2 \times y_1)) \\
(l = l_3 \rightarrow (l, y_1) := (l_4, y_1 - 1)) \\
(l = l_4 \rightarrow l := l_1) \\
(m = m_1 \rightarrow m := \text{if}(y_2 = k, \text{halt}, m_2)) \\
(m = m_2 \rightarrow (m, y_2) := (m_3, y_2 + 1)) \\
(m = m_3 \land y_2 \leq n - y_1 \rightarrow m := m_4) \\
(m = m_4 \rightarrow (m, y_3) := (m_5, y_3/y_2)) \\
(m = m_5 \rightarrow m := m_1)
\]

Divide the program operations into two sets (the name of an operation and the value of the program counter when the operation is available are the same):

\[
L = \{l_1, l_2, l_3, l_4\}, M = \{m_1, m_2, m_3, m_4, m_5\}
\]

In every history with operations from \(L\) and \(M\), one can commute adjacent pairs of operations from \(L\) and \(M\) unless the pair is \((l_3, m_3)\) and \(y_3 = n - y_1 + 1\). In that case, \(m_3\) cannot be done first, but if \(l_3\) is done first, \(y_3 = n - y_1\) and the \text{await} statement succeeds. We define

\[
\Delta_{a, \beta} = \begin{cases} 
    \text{true} & \text{if } (a, \beta) \in L \times M \setminus \{(l_3, m_3)\} \\
    \text{false} & \text{if } (a, \beta) \in L \times L \cup M \times M \\
    y_1 \neq n - y_1 + 1 & \text{if } a = l_3 \land \beta = m_3
\end{cases}
\]

It is always possible to commute an operation from \(L\) with a previous operation from \(M\). The only case which is not obvious is that it is possible to have the operation \(l_3\) precede \(m_3\). That is, if \(m_3\) is executed immediately before \(l_3\), then its enabledness condition \(y_2 \leq n - y_1\) implies \(y_2 \leq n - y_1 + 1\) or, equivalently, \(y_2 \leq n - (y_1 - 1)\). Thus, in that same state, if we now want to execute \(l_3\) first, decrementing \(y_1\) will leave the condition for executing \(m_3\) still true, so that the same result is obtained as previously. Since each execution sequence is equivalent to the sequence in which all the operations from \(L\) are executed before all the operations from \(M\), there is a single maximal trace for the program (and hence, a single run).

As another example, consider the following producer/consumer program \[10\] with a bounded buffer. The program counters of the two processes are \(p\) and \(c\), respectively, while \(n_p\) and \(n_c\) are used to count the number of values produced and consumed, respectively. In this program, the left process \(P(\text{producer})\) generates \(M\) values using the function \text{inp}(n_p)\ where \(0 \leq n_p < M\). It uses the buffer \(bf[0..N - 1]\) to communicate the values to the right process \(C(\text{consumer})\). Process \(C\) consumes the values from the buffer by executing the procedure \text{out}(y)\ for each \(y\) obtained from the buffer.

The initial condition is \(\Theta \equiv p = p_1 \land c = c_1 \land n_p = 0 \land n_c = 0\).
Since the buffer is bounded, both processes must synchronize so that exactly the values which are produced are finally consumed. Since the producer uses \( n_p \) to count the number of values produced, while the consumer uses \( n_c \) to count the number of values consumed, when \( n_p - n_c = N \), the buffer is full. Hence, the producer has to wait for the consumer to consume some values. On the other hand, when \( n_p = n_c \), no new values are ready in the buffer, and the consumer has to wait for the producer to produce new values.

Let \( P \) be the producer's operations and \( C \) be the consumer operations. Independence predicates can be defined as follows:

\[
\Delta_{a,b} = \begin{cases} 
\text{true} & \text{if } (a, \beta) \in P \times C \setminus \{ (P_3, C_2), (P_5, C_2), (P_4, C_3) \} \\
\text{false} & \text{if } (a, \beta) \in P \times P \setminus C \times C \\
\end{cases}
\]

Using these predicates, it can be shown that each execution sequence of the program is equivalent to the sequence of operations in which the the elements are inserted and removed one at a time, i.e., the sequence \( P_1 P_2 P_3 P_4 P_6 C_1 C_2 C_4 C_5 C_6 \) appears \( M \) times and then \( P_1 \) and \( C_1 \) appear once. It should be noted that it is not always the case that a single representative sequence can be found, even though this was the case for these two examples.

Although in the above example we followed the convention of having the events from operations in the same process be totally ordered (as seen in the second line of both definitions of \( \Delta_{a,b} \)), this restricts the generality. In the second example \( \Delta_{C_4,C_5} \) could be \textit{false}. This would require changing the translation from the program to a set of transitions, so that the update of the program counter could be ignored in determining independence. Other possibilities are left as an exercise for the reader.

### 3.2 An Application: Verifying Total Correctness

The interaction between concurrent segments of a program (usually called processes) poses considerable difficulty in verification that does not exist in sequential programs. The basic method is to use assertions that cover all the possible states of the program generated by interleaving independent (concurrent) operations [3]. The assertions are global (referring at the same time to variables of different processes). Considering only the number of possible combinations of values for program counters, this number can grow to the product of the sizes of the different processes (by "size" we mean the number of operations).
It is evident from the works mentioned in the introduction that the following pattern of program verification is appealing: In a first stage, identify the structuring of a concurrent program. That is, independence among various parts. Then, at a second stage, use proof rules which can exploit this independence. Identifying independence does not mean that actual extra structuring is marked or constructed on programs or program models. Merely, some conditions are satisfied that guarantee the soundness of the rules used in the second step. We present a set of proof rules which exploits collapses. These rules are shown to be sound and complete for proving total correctness.

Total correctness of a program $Pr$ with respect to an assertion $\psi$ demands that if $Pr$ started executing from a global state satisfying $\Theta$, then it will eventually terminate in some state which satisfies $\psi$.

Methods based on linear execution sequences use well-founded induction to show that each sequence is finite and terminates with a correct global state. A parametrized inductive assertion $\varphi(n)$, where $n$ belongs to some domain $W$ is used to guarantee that from all the intermediate states of the execution satisfying it, each successor state satisfies $\varphi(m)$ for some $m < n$ according to some well-founded order '<' on $W$. Here, instead of taking care of all the successors of each state, we demand that $\varphi(m)$ with $m < n$ will be satisfied by at least one successor of each state satisfying $\varphi(n)$ in every run. By repeatedly choosing successors according to this rule, a set of representative execution sequences (at least one for each run) satisfy the well-founded induction.

Below we prove two lemmas that connect the intuitive notions of termination and total correctness to the conditional trace model.

**Lemma 3.1** If a program does not terminate, there exists an infinite run.

**Proof.** If the program does not terminate, there exists an infinite sequence of histories $s_0, s_1, s_2, \ldots$ where $s_0 = (\sigma, e)$, $\sigma \vdash \Theta$ and for each $i \geq 0$, for some $a \in T$, $\text{fin}_{s_i} = e_{m_0}$ and $v_{i+1} = v_i a$. For each $i \geq 0$, let $t_i$ be the set of histories equivalent to $s_i$. Thus, $t_0, t_1, t_2, \ldots$ is a sequence of traces. The set of traces in this sequence is pairwise consistent. It might be the case that this set is not maximal, but then it is contained in a maximal infinite run.

**Lemma 3.2** If a run $II$ is finite, it has a single maximal trace (according to the trace order '<=' ) and its final interpretation satisfies $\text{Term} = \land_{a \in T} e_{m_0}$. Otherwise, no trace of $II$ satisfies $\text{Term}$.

**Proof.** Since by definition any two traces of a run must be consistent, there is only one maximal $t$, since the run is finite. Obviously, $\text{fin}_t \models \text{Term}$.

Now, assume that an infinite run $II'$ has a trace $t$ satisfying $\text{Term}$. It must also have a trace $t'$ whose length exceeds that of $t$ (because the number of traces with length not more than $|t|$ is finite). Thus, there must exist a trace subsuming both $t$ and $t'$. However, this is impossible, since $t$ satisfies $\text{Term}$.

Thus, total correctness of a program $Pr$ with respect to $\Theta$ and $\psi$ holds iff each run $II$ of $Pr$ is finite and the final interpretations of its maximal trace satisfies $\psi$.
A notation is needed to describe assertions which use intermediate states, residing on representatives from each equivalence class. Linear Temporal Logic [22] is inadequate here, because it implicitly asserts about all the sequences. The logic ISTL [15] is appropriate. Here, only the subset which is needed for total correctness proofs is used. The rules presented generalize those seen in [16].

**Definition 3.3** Denote \( \varphi \rightarrow EX\psi \) iff for each run \( \Pi \) of \( Pr \) having a trace \( t \) with \( \text{fin}_t = \varphi \), there exists an immediate successor \( t' \) with \( \text{fin}_{t'} = \psi \). Denote \( \varphi \rightarrow EF\psi \) iff for each run \( \Pi \) of \( Pr \) having a trace \( t \) with \( \text{fin}_t = \varphi \), there exists a successor \( t' \) with \( \text{fin}_{t'} = \psi \).

**Rule 1 SIMP:** The following rules are simple properties of 'EX' and 'EF'. They reflect semantic properties of runs.

\[
\begin{align*}
\text{SIMP1} & \quad \frac{\varphi \rightarrow EX\psi}{\varphi \rightarrow EF\psi} \\
\text{SIMP2} & \quad \frac{\varphi \rightarrow \psi}{\varphi \rightarrow EF\psi} \\
\text{SIMP3} & \quad \frac{\varphi \rightarrow \psi_1 \lor \psi_2}{\varphi \rightarrow EF\psi} \\
\text{SIMP4} & \quad \frac{\psi_1 \rightarrow EF\psi_1}{\varphi \rightarrow EF\psi}
\end{align*}
\]

**Rule 2 IND:** Well founded induction can be proved using the following rule. Let \((W, \prec)\) be a well-founded domain (no infinitely decreasing chains). Let \(\varphi(n)\) be a first order formula with a parameter \( n \) from the domain of \( W \).

\[
\begin{align*}
\text{IND} & \quad \frac{\varphi \rightarrow \exists n \varphi(n)}{\varphi(n) \rightarrow EF(\psi \lor \exists m < n \varphi(m))} \\
& \quad \frac{\varphi \rightarrow EF\psi}{\varphi \rightarrow EF\psi}
\end{align*}
\]

**Rule 3 STEP:** Proving \( \varphi \rightarrow EX\psi \) is the kernel of the proof method. To choose only representative successors for \( \varphi \), the set of operations \( T \) is partitioned into two complementary sets \( Q \) and \( Q(= T \setminus Q) \). Instead of showing that each \( \alpha \in T \) which is enabled when \( \varphi \) holds will produce a state satisfying \( \psi \), the aim is to show that:

1. By executing operations from \( Q \), enough successors for the states satisfying \( \varphi \) are generated.

2. When executing any operation from the set \( Q \) from a state satisfying \( \varphi \), a state satisfying \( \psi \) is reached.
A third predicate \( \delta \) is used to achieve the first goal. It is used to show that from a state satisfying \( \varphi \), as long as no operation from \( Q \) is executed, each operation from \( Q \) is either disabled or independent of all the operations of \( Q \). By the independence conditions, any operation of \( Q \) enabled at \( s \) cannot be disabled by an operation of \( Q \). Thus, there are two cases: One is that there is an infinite sequence of \( Q \) operations and hence at least one of the successors of \( s \) is produced by some element of \( Q \) (this justice-like property relies on the maximality of the set of traces in each run and will be elaborated in the proof below). The other case is that all the sequences of the class are finite and thus, some element \( a \) from \( Q \) which is enabled when \( \varphi \) holds is eventually executed. In both cases, by independence (using the collapse), \( a \) can be commuted with every previous adjacent operation from \( Q \) that occurred after \( s \). The last premise of the rule asserts that by executing any operation from \( Q \) which is enabled in a state satisfying \( \varphi \), a state satisfying \( \psi \) is reached.

\[
\text{STEP} \\
\begin{align*}
S1 & \quad \varphi \rightarrow (\delta \land \forall_{\alpha \in Q} cn_\alpha) \\
S2 & \quad \text{for each } \alpha \in Q, \delta \land cn_\alpha \rightarrow wp_\alpha(\delta) \\
S3 & \quad \delta \rightarrow \land_{\alpha \in Q} (\neg cn_\alpha \lor \land_{\beta \in Q} \Delta_{\alpha, \beta}) \\
S4 & \quad \text{for each } \alpha \in Q, \varphi \land cn_\alpha \rightarrow wp_\alpha(\psi) \\
\varphi & \rightarrow \exists \psi
\end{align*}
\]

**Theorem 3.4** The proof rules are sound.

**Proof.** The only nonobvious rule is STEP. Assume that all its premises hold. Let \( t \) be a trace satisfying \( \varphi \) which belongs to \( \text{traces}(II) \) for some run \( II \). By \( S1 \), \( t \) satisfies \( \delta \) and at least one operation of \( Q \) is enabled at \( \text{fin}_t \). Furthermore, by the premise \( S2 \), while executing after \( i \) (appending to it) only operations from \( Q \), \( \delta \) is kept invariant. By \( S3 \), when \( \delta \) holds, each operation of \( Q \) is either disabled or the conditions for its independence with all the operations from \( Q \) hold. Hence, from the conditions of collapses in Definition 2.2, the same operations from \( Q \) which are enabled in \( t \) remains enabled after executing any number of operations from \( Q \). Moreover, if an operation from \( Q \) is finally executed, it can be commuted with all the operations of \( Q \) executed since \( t \). By the premise \( S4 \), by executing any operation of \( Q \) from a trace satisfying \( \varphi \), a trace satisfying \( \psi \) is obtained.

If \( II \) is a finite run, it is obvious that the maximal trace \( t' \) of \( II \) subsuming \( t \) cannot avoid executing all the operations from \( Q \) after \( t \), because otherwise, at least one operation from \( Q \) remains enabled.

Assume now that \( II \) is an infinite run, and all the traces of \( II \) subsuming \( t \) avoid executing operations from \( Q \) after \( t \). Then, this set of traces is not maximal. To see this, take some operation \( \alpha \in Q \) which is enabled in \( t \). Then, \( \delta \) (using \( S2 \) and \( S3 \)) guarantees that \( \alpha \) is enabled in every trace of \( II \) subsuming \( t \). For each such trace, \( t' = [\sigma, \nu] \) form a new trace \( t'' = [\sigma, \nu, \alpha] \).

Let \( \text{traces}(II) \) be the set of newly formed traces. We will show now that for each pair of traces \( t_1 \in \text{traces}(II), t_2 \in \text{traces}(II) \) there exists some trace \( t_3 \in \text{traces}(II) \) which subsumes both \( t_1 \) and \( t_2 \). Let \( t_4 \in \text{traces}(II) \) be the trace satisfying \( t_4[\text{fin}_{t_4}, \alpha]' = t_2 \) (\( t_2 \) was constructed from \( t_4 \) by adding the operation \( \alpha \)). Let \( t_5 \in \text{traces}(II) \) be the trace subsuming both \( t_1 \) and \( t_4 \). The trace \( t_3 = t_5[\text{fin}_{t_5}, \alpha] \) subsumes both \( t_1 \) and \( t_2 \) (\( t_5 \models cn_\alpha \) because \( t \models t_4 \subseteq t_5 \)). Similarly, consistency
among pairs of traces in \( \text{traces}(\Pi) \) can be shown. Hence, the traces in \( \text{traces}(\Pi) \cup \text{traces}(\Pi) \) are pairwise consistent, which contradicts the maximality of \( \text{traces}(\Pi) \).

**Theorem 3.5** The proof rules are complete for verifying total correctness.

**Proof.** It is always possible to choose \( Q = T, Q = \emptyset \). That is, we may choose not to exploit the independence. In that case, the proof method is reduced to other methods for proving total correctness, for example to [28].

Returning to the first example in Section 3.1, the total correctness proof is done using two well-founded inductions. The formula \( \varphi_1 \) describes the states obtained by executing only operations from \( L \). In order to show progress, we have to use a parametric formula where the value of the parameter decreases with each single operation from \( L \) which is executed. A closer look at the program reveals that \( y_1 \) is decreasing with every traversal of the loop \( l_1, l_2, l_3, l_4 \).

Thus, a well-founded ordering which decreases with every step of \( L \) can be formulated by taking the lexicographical order \( (\mathcal{N} \times \omega_1, \prec_L) \) where \( \mathcal{N} \) is the set of natural numbers with the usual "less than" order, and \( \omega_1 \equiv (L \cup \{\text{halt} \}), l_4 \Rightarrow l_1 \Rightarrow l_2 \Rightarrow l_3 \rightarrow \text{halt} \). Let \( \varphi_1(a, b) \) be the parametrized first order formula

\[
m = m_1 \land n \geq y_1 \geq n - k \land y_3 = n \times (n - 1) \times \ldots \times \left( y_1 + (1 \neq 0, 1, 0) \right) \land y_2 = 0 \land a = y_1 \land b = 1
\]

A second parametric formula is used to show progress from a state in which the left part (the operations in \( L \)) has terminated and only operations from \( M \) are enabled. Again, a lexicographic order \( (\mathcal{N} \times \omega_2, \prec_L) \) is used where \( \omega_2 \equiv (M \cup \{\text{halt} \}), m_3 \succ m_4 \succ m_5 \succ m_6 \succ m_7 \succ m_8 \succ m_9 \succ \text{halt} \). Let \( \varphi_2(a, b) \) be the parametrized first order formula

\[
l = \text{halt} \land y_1 = n - k \land y_3 = \frac{n \times (n - 1) \times \ldots \times (n - k + 1)}{1 \times 2 \times \ldots \times (y_2 - (m = m_3 \lor m = m_4, 1, 0))} \land y_2 \leq k \land a = k - y_2 \land b = m
\]

describing states obtained by executing operations from \( M \) after none of the operations of \( L \) is enabled (a multiplicity of zero elements is defined to be 1).

The proof proceeds as follows:

Using first order logic,

\[
\Theta \rightarrow \varphi_1(n, l_1)
\]  

(1)

Using first order logic and (1),

\[
\Theta \rightarrow \exists a \exists b \varphi_1(a, b)
\]  

(2)

Using \text{step} with \( \delta = \text{true}, Q = L, \) and \( Q = M, \) and the given collapse,

\[
(n - k, \text{halt}) \ll (a, b) \land \varphi_1(a, b) \rightarrow \text{EX} \exists a' \exists b'((a', b') \ll (a, b) \land \varphi_1(a', b'))
\]  

(3)

Using first order logic,

\[
\varphi_1(n - k, \text{halt}) \rightarrow \varphi_2(k, m_1)
\]  

(4)

15
Using (3), (4) and the rules SIMP,
\[ \varphi_1(a, b) \rightarrow EF(\varphi_2(k, m_1) \lor \exists a' \forall b'((a', b') \ll_1 (a, b) \land \varphi_1(a', b'))) \]  
(5)

Using (2), (5) and IND,
\[ \Theta \rightarrow EF\varphi_2(k, m_1) \]  
(6)

Using \texttt{STEP} with \( \delta = \text{true} \), \( \Theta = M \) and \( \Theta = L \),
\[ ((0, \text{halt}_m) \ll_2 (a, b) \land \varphi_2(a, b)) \rightarrow EX \exists a' \forall b'((a', b') \ll_2 (a, b) \land \varphi_2(a', b')) \]  
(7)

Using first order logic,
\[ \varphi_2(0, \text{halt}_m) \rightarrow \psi \land \bigwedge_{a \in T} \neg \text{en}_a \]  
(8)

Using (7), (8) and the rules SIMP,
\[ \varphi_2(a, b) \rightarrow EF((\psi \land \bigwedge_{a \in T} \neg \text{en}_a) \lor \exists a' \forall b'((a', b') \ll_2 (a, b) \land \varphi_2(a', b'))) \]  
(9)

Using first order logic,
\[ \varphi_2(k, m_1) \rightarrow \exists a \forall b \varphi_2(a, b) \]  
(10)

Using (9), (10) and IND,
\[ \varphi_2(k, m_1) \rightarrow EF(\psi \land \bigwedge_{a \in T} \neg \text{en}_a) \]  
(11)

Using (6), (11) and applying SIMP4,
\[ \Theta \rightarrow EF(\psi \land \bigwedge_{a \in T} \neg \text{en}_a) \]  
(12)

4 Discussion

The semantics of conditional traces is a direct extension to trace semantics. As we have shown, a collapse is a formalism that extends independence [20] among operations. Instead of using a fixed relation on pairs of operations (predefined, or easily obtainable from the set of operations using some formation rules), a set of predicates satisfying the independence conditions are used. The motivation for using a collapse is practical: it designates when the order among the occurrence of two operations does not matter and can be exploited according to the convenience of a proof or to improve implementation considerations. In this section we discuss the benefits of using this semantics and compare it to other works.

The concept of \textit{locality} discussed earlier is reflected in the semantics: a specification language, such as the temporal logic ISTL [15] can easily express properties about the existence of execution sequences (in each run). A typical such formula assures the existence of sequences where processes which are executing some local task, independent of the other processes, progress in isolation to the progress of the other processes. The concept of \textit{sequentiality} is also facilitated. Considering communication closed layers, it is possible to assert about the existence of interleaving sequences in which the order of execution progresses layerwise. This was shown in [17].
to hold for fixed independence relations and obviously holds for conditional independence, pro-
vided that the generated runs obtained by a collapse contain the runs of the fixed independence
relation (this is achieved by weakening the fixed independence relation).

Assertions which are written as $\varphi \rightarrow EF\psi$ were shown in [15, 16, 29] to express properties
such as concurrency, immediate response, and serializability. However, one should be careful
when using conditional independence. Semantic properties such as concurrency rely upon in-
terpreting the independence as "can be executed in parallel". Thus, in this case, the collapse
should be chosen to agree with this interpretation. A complete proof system for fixed dependence
relations appears in [25] and can also be extended to handle conditional dependence.

The conditional trace notation provides a uniform framework for investigating various strate-
gies for defining collapses, such as those in [19, 8], as aids in verification, implementation, and
program optimization. The results of Back and Sere on refining atomicity [4, 5] can also be
used to enhance the independence which can be used in program verification. Sometimes it
is not possible to find appropriate independence predicates for the atomic operations, but the
program behaves as if the atomicity exists at a coarser grain, where independence does hold.
Using the techniques of [4, 5], it is sometimes possible to transform a given program into one
with coarser atomic operations (although, the aim of those papers is to obtain the opposite,
namely, refine the atomicity to achieve more concurrency). A possible extension to conditional
traces semantics can allow achieving the effect of coarsening without actually performing the
transformation. This is done by allowing independence conditions to be defined on sequences of
operations (such as $\Delta_{p,\psi,\mu}$). Then one sequence can be exchanged with another - even though
finer interleavings cannot be done.

Another possibility is to use collapses to prove that a program behaves as if it is constructed
from operations of coarser granularity. This can be done within the framework of ISTL, proving
properties which are related to serializability [7] of database operations. Namely, segments of
the program are shown to behave as serializable database transactions. Then, use coarser grained
operations and a collapse for these transactions to prove termination.

References

[1] R. Allen, K. Kennedy, Automatic translation of FORTRAN programs to vector forms,
Transactions on Programming Languages and Systems, 1987, 491-543.
[2] K. Apt, N. Francez, S. Katz, Appraising fairness in languages for distributed programming,
Science 10(1975), 119-135.
tronic Computers 15 (1966), 757-762.


