Alignment Completeness for Relational Hoare Logics (with appendix)

Ramana Nagasamudram
Stevens Institute of Technology

David A. Naumann
Stevens Institute of Technology

Abstract—Relational Hoare logics (RHL) provide rules for reasoning about relations between programs. Several RHLs include a rule we call sequential product that infers a relational correctness judgment from judgments of ordinary Hoare logic (HL). Other rules embody sensible patterns of reasoning and have been found useful in practice, but sequential product is relatively complete on its own (with HL). As a more satisfactory way to evaluate RHLs, a notion of alignment completeness is introduced, in terms of the inductive assertion method and product automata. Alignment completeness results are given to account for several different sets of rules. The notion may serve to guide the design of RHLs and relational verifiers for richer programming languages and alignment patterns.

I. INTRODUCTION

A common task in programming is to ascertain whether a modified version of a program is equivalent to the original. For programs with deterministic results, equivalence can be formulated simply: From any initial state, if both programs terminate then their final states are the same. This termination-insensitive property is akin to partial correctness: A program terminates then their final states are the same. This termination and alignment patterns.

RHLs and relational verifiers for richer programming languages

evaluate RHLs, a notion of alignment completeness is introduced, in terms of the inductive assertion method and product automata. The touchstone for Hoare logics (HL) which we write as \( c | d : R \Rightarrow S \) infer that \( R \Rightarrow S \). Relations go beyond program equivalence. Using primed variables to refer to the second of two related states, the judgment \( c | d : x = x' \Rightarrow y < y' \) expresses that when run from states that agree on the initial value of \( x \), the final value of \( y \) produced by \( c \) is less than the final value of \( y \) from \( d \) (i.e., \( d \) majorizes \( c \)). As another example, \( c | c : x = x' \Rightarrow y = y' \) relates \( c \) to itself and says that the final value of \( y \) is determined by the initial value of \( x \). Such dependency properties arise in many contexts including compiler optimization and security analysis which motivated early work on RHL.

Suppose the variables of \( d \) are disjoint from those of \( c \). For example let \( d \) be a copy of \( c \) with all the variables renamed by adding primes. Then a relation on states amounts to a predicate on primed and unprimed variables as in the preceding informal notation. Moreover terminated executions of \( (c;d) \) are in bijection with pairs of terminated executions of \( c \) and \( d \). Put differently, \( c;d \) serves as a product program, much like products in automata theory. The product program lets us prove relational properties using HL, but in general the technique is unsatisfactory. As an example, consider the simple program \( c0 \) in Fig. [1] which computes in \( z \) the factorial of \( x \). Let \( c0' \) be a renamed copy, so determinacy of \( c0 \) can be expressed by \( x = x' \Rightarrow z = z' \). To prove the judgment \( c0; c0' : x = x' \Rightarrow z = z' \) in HL we need the assertion \( z = x! \land x = x' \) at the semicolon, to get \( z = x! \land z' = x'! \land x = x' \) following \( c0' \), from which \( z = z' \) follows. The spec \( x = x' \Rightarrow z = z' \) involves nothing more than equalities, yet the proof requires nonlinear arithmetic. This illustrates the general problem that the technique requires strong functional properties and thus nontrivial loop invariants.

There is a simple way to prove \( c0 | c0' : x = x' \Rightarrow z = z' \). Consider the execution pairs to be aligned step-by-step, and note that at each aligned pair of states we have \( y = y' \land z = z' \), given \( x = x' \) initially. RHLs feature rules for compositional reasoning about similarly-structured subprograms, which embody informal patterns of reasoning and enable the use of simple assertions with alignment of computations expressed in terms of syntax. For example, from judgments \( y := x | y' := x' : x = x' \Rightarrow y = y' \) and \( z := 1 | z' := 1 : y = y' \Rightarrow (y = y' \land z = z') \) infer that \( y := x; z := 1 | y' := x'; z' := 1 : (x = x') \Rightarrow (y = y' \land z = z') \). A number of RHLs have appeared in the literature, but even for the simple imperative language we see no convergence on a common core set of rules. Besides closely “synchronized” rules like the sequence rule for the preceding inference (see \( \text{Step in Fig. 10} \)), there are sound rules to relate a command to skip (Fig. 12) or to a differently-structured command. For an example of the latter see Sec. VII.

The touchstone for Hoare logics is Cook’s completeness theorem [3] which says any true correctness judgment \( c : P \Rightarrow Q \) can be derived using the rules. To be precise, HL relies on entailments between assertions and the rules are complete relative to completeness of assertion reasoning. Moreover, completeness requires expressiveness of the assertion language, meaning that weakest preconditions can be expressed, for whatever types of data are manipulated by the program [4]. These considerations are not important for
the ideas in this paper. We follow the common practice of treating assertions and their entailments semantically \(^5\) (i.e., by shallow embedding in our metalanguage).

RHLs often feature a rule we dub “sequential product”, which infers \(c \mid c' : \mathcal{R} \leadsto S\) from the HL judgment \(c; c' : \mathcal{R} 

\leadsto S\), formalizing the idea of product program. It is a useful rule, to complement the rules for relational judgments about similarly-structured programs. For example, in the regression verification \(^6\) scenario with which we began, i.e., equivalence between two versions of a program, same-structure rules can be used to relate the unchanged parts, while sequential product can be used for the revised parts if they differ in control structure. This is how programmers might reason informally about a small change in a big program.

But there is a problem. Consider the logic comprised of the sequential product rule together with a sound and complete HL. This is complete, in the sense of Cook, for relational judgments! The problem is well known to RHL experts. Completeness is the usual means to determine a sufficient and parsimonious set of inference rules, but completeness fails to discriminate between a RHL that supports compositional proofs using facts about aligned subprograms and an impoverished RHL with only sequential product.

The conceptual contribution of this paper is the notion of alignment completeness, which discriminates between rules in terms of different classes of alignments. The technical contributions are four alignment completeness theorems, for representational capabilities of RHL rules.

To explain the idea our first step is to revisit Floyd-Hoare logic. Floyd \(^7\) made precise the inductive assertion method (IAM) already evident in work by Turing \(^8\) (see \(^9\)). To prove \(P \leadsto Q\) by IAM, provide assertions at control points in the program, at least \(P\) at the initial point and \(Q\) at program exit. We call this an annotation; it is familiar to programmers in the form of assert statements, and it gives rise to verification conditions. A valid annotation is one where the verification conditions are true and every loop in control flow is “cut” by an annotation. A valid annotation constitutes a proof by induction. HL is complete in a sense that we call Floyd completeness: For any valid annotation of a program \(c\) for a spec \(P \leadsto Q\), there is a proof in HL of \(c : P \leadsto Q\) using only judgments of the form \(b \vdash R \leadsto S\) with \(b\) ranging over subprograms of \(c\) and \(R, S\) the assertions annotating the entry and exit points of \(b\).

Now consider relating two programs. An annotation should attach relations to designated pairs of points in the control flow of the two programs. For the example judgment \(c0 \mid c0' : x = x' \Rightarrow z = z'\), choose pairs at the lockstep positions, and the abovementioned conjunction \(y = y' \land z = z'\). Validity of an annotation is defined in terms of execution pairs aligned in accord with the designated pairs of control points: at aligned steps, the asserted relations hold. To make alignment precise we use product automata. A product represents a particular pattern of alignment; if it is adequate in the sense of covering all execution pairs then the IAM can be applied to prove relational properties of the two programs. A set of RHL rules is then alignment complete, for a given class of alignment automata, if for any valid annotated automaton there is a derivation using the rules and only the assertions and judgments associated with the annotated automaton.

This paper formalizes the idea and gives some representative results of this kind: for sequential product, for strict lockstep, and also for data-dependent alignment of loop iterations. To see the need for the latter, consider program \(c2\) in Fig. \(1\) in which some iterations have no effect on \(y\) or \(z\). We can prove \(c0 \mid c2 : x = x' \Rightarrow z = z'\) using only simple equalities and without reasoning about factorial, provided the effectful iterations are aligned in lockstep while the gratuitous iterations of \(c2\) (when \(w\) is odd) proceed with \(c0\) considered to be stationary.

The notion of Floyd completeness should be no surprise to readers familiar with Floyd-Hoare logic or related topics like software model checking. But the authors are unaware of any published result of this form. A related idea is proof outline logic \(^4\), \(^10\), which formalizes commands with correct embedded assertions. Rules for proof outlines have verification conditions which imply an annotation is valid. In addition to showing Cook-style soundness and completeness results for a proof outline logic, Apt et al. prove a “strong soundness” theorem \(^4\) Thm. 3.3] which says that if the program’s proof outline is provable then in any execution each assertion is true when control is at that point. The converse would be a way to formalize Floyd completeness. Strong soundness is phrased in terms of transition semantics (small steps). By contrast, the fact that reasoning in HL is compositional in terms of control structure is beautifully reflected in proofs of soundness and (Cook) completeness based on denotational semantics \(^4\).

In this paper we only consider rules that are sound, in the usual sense, and we have no need to formalize alignment soundness.

Outline: Sec. \(\[\[\) lays the groundwork by spelling out Floyd completeness for HL, in terms of automata with explicit control points, including automata based on small-step semantics of labelled commands. Sec. \(\[\[\[\) formalizes product automata: ways in which a pair of automata can be combined into an

---

\(^1\)The precise result depends on details of the HL rules and may require mildly adjusted judgments in addition to those directly given by the annotation; see Thm. \(\[\).

---

\(^2\)For any valid annotation of a program

\[c0 \mid c0' : x = x' \Rightarrow z = z'\]

choose pairs at the lockstep positions, and the abovementioned conjunction

\[y = y' \land z = z'\].

Validity of an annotation is defined in terms of execution pairs aligned in accord with the designated pairs of control points: at aligned steps, the asserted relations hold. To make alignment precise we use product automata. A product represents a particular pattern of alignment; if it is adequate in the sense of covering all execution pairs then the IAM can be applied to prove relational properties of the two programs. A set of RHL rules is then alignment complete, for a given class of alignment automata, if for any valid annotated automaton

\[\text{there is a derivation using the rules and only the assertions and judgments associated with the annotated automaton.}\]

This paper formalizes the idea and gives some representative results of this kind: for sequential product, for strict lockstep, and also for data-dependent alignment of loop iterations. To see the need for the latter, consider program \(c2\) in Fig. \(1\) in which some iterations have no effect on \(y\) or \(z\). We can prove

\[c0 \mid c2 : x = x' \Rightarrow z = z'\]

using only simple equalities and without reasoning about factorial, provided the effectful iterations are aligned in lockstep while the gratuitous iterations of \(c2\) (when \(w\) is odd) proceed with \(c0\) considered to be stationary.

The notion of Floyd completeness should be no surprise to readers familiar with Floyd-Hoare logic or related topics like software model checking. But the authors are unaware of any published result of this form. A related idea is proof outline logic \(^4\), \(^10\), which formalizes commands with correct embedded assertions. Rules for proof outlines have verification conditions which imply an annotation is valid. In addition to showing Cook-style soundness and completeness results for a proof outline logic, Apt et al. prove a “strong soundness” theorem \(^4\) Thm. 3.3] which says that if the program’s proof outline is provable then in any execution each assertion is true when control is at that point. The converse would be a way to formalize Floyd completeness. Strong soundness is phrased in terms of transition semantics (small steps). By contrast, the fact that reasoning in HL is compositional in terms of control structure is beautifully reflected in proofs of soundness and (Cook) completeness based on denotational semantics \(^4\).

In this paper we only consider rules that are sound, in the usual sense, and we have no need to formalize alignment soundness.

Outline: Sec. \(\[\[\) lays the groundwork by spelling out Floyd completeness for HL, in terms of automata with explicit control points, including automata based on small-step semantics of labelled commands. Sec. \(\[\[\[\) formalizes product automata: ways in which a pair of automata can be combined into an
automaton on pairs of states that serves to represent aligned steps of two computations.

Sec. [V] gives the first alignment completeness theorem: Given a valid annotation of a product automaton for \( c \mid c' : P \Rightarrow Q \) that executes \( c \) to termination and then executes \( c' \), there is a proof using just the sequential product rule and the rules of HL — and using only judgments for subprograms with pre- and postconditions given by the annotation.

Sec. [VI] gives the alignment completeness theorem for lockstep alignment: if \( c \mid c' : P \Rightarrow Q \) is witnessed by such an alignment with valid annotation, then it can be proved without HL, using just the RHL rules that relate same-structured programs (see Fig. 10). And again the judgments used are those associated with the annotation. These rules are not complete in the usual sense, but lockstep reasoning is sufficient in some practical situations.

The theorem of Sec. [VI] accounts for the combination of \( \text{SEQ}_{\text{PROD}} \) with the lockstep RHL rules. This and the preceding results are for alignments that can be described in terms of which control points are aligned. Sec. [VII] accounts for conditional alignment of loop iterations, using a rule due to Beringer [11]. Our fourth alignment completeness theorem is for a logic including that rule together with lockstep rules but not \( \text{SEQ}_{\text{PROD}} \). As a worked example we show that \( c \mid 2 \) majorizes \( c_3 \) for sufficiently large \( x \).

Sec. [VIII] discusses related work and open questions. Some recent works on relational verification use alignments that can be understood as more sophisticated product automata for which alignment complete rules remain to be designed.

This document extends the LICS’21 version with an appendix that contains additional details and proofs.

II. PRELIMINARIES AND FLOYD COMPLETENESS

In order to connect Floyd’s theory with Hoare’s we formulate the IAM in terms of transition systems with an explicit finite control flow graph (CFG). We consider ordinary program syntax, with a standard structural operational semantics and Hoare logic, but with labels used to define the transition system of a given “main program”.

Floyd automata, specs and correctness: An automaton is a tuple \((\text{Ctrl}, \text{Sto}, \text{init}, \text{fin}, \rightarrow)\) where \( \text{Sto} \) is a set (the data stores), \( \text{Ctrl} \) is a finite set (the control points) that contains distinct elements \( \text{init} \) and \( \text{fin} \), and \( \rightarrow \subseteq (\text{Ctrl} \times \text{Sto}) \times (\text{Ctrl} \times \text{Sto}) \) is the transition relation. We require \((n, s) \rightarrow (m, t)\) to imply \( n \neq \text{fin} \) and \( n \neq m \) and call these the finality and non-stuttering conditions respectively. Absence of stuttering loses no generality and facilitates definitions involving product automata. Let \( s, t \) range over stores and \( n, m \) over control points.

A pair \((n, s)\) is called a state and we write \( \text{ctrl}, \text{stor} \) for the left and right projections on states. A trace of an automaton is a non-empty sequence of states, consecutive under the transition relation. A trace \( \tau \) is terminated provided \( \tau \) is finite and \( \text{ctrl}(\tau_{-1}) = \text{fin} \), where \( \tau_{-1} \) denotes the last state of \( \tau \). An initial trace is one such that \( \text{ctrl}(\tau_0) = \text{init} \). We allow traces of length one, in which case \( \tau_{-1} = \tau_0 \), but a terminated initial trace has plural length because \( \text{init} \neq \text{fin} \).

We treat predicates semantically, i.e., as sets of stores. Define \( s \models P \) iff \( s \in P \) and define \( (n, s) \models P \) iff \( s \models P \).

Let us spell out two semantics for specs in terms of an automaton \( A \). The basic semantics is as follows. For a finite initial trace \( \tau \) to satisfy \( P \Rightarrow Q \) means that \( \tau_0 \models P \) and \( \text{ctrl}(\tau_{-1}) = \text{fin} \) imply \( \tau_{-1} \models Q \), in which case we write \( \tau \models P \Rightarrow Q \). Then \( A \) satisfies \( P \Rightarrow Q \), written \( A \models P \Rightarrow Q \), just if all its finite initial traces do. For non-stuck semantics there are two conditions: (i) \( \tau_0 \models P \) and \( \text{ctrl}(\tau_{-1}) = \text{fin} \) imply \( \tau_{-1} \models Q \), and (ii) \( \tau_0 \models P \) and \( \text{ctrl}(\tau_{-1}) \neq \text{fin} \) imply \( \tau_{-1} \rightarrow \rightarrow \), where \( \tau_{-1} \rightarrow \rightarrow \) means there is at least one successor state. A state with no successor is called stuck. Non-final stuck states are often used to model runtime faults. Again, \( A \) satisfies \( P \Rightarrow Q \) just if all its finite initial traces do. Non-stuck is important in practice and we consider it in passing but for clarity our main development is for basic semantics.

For an automaton \( A \), define \( \text{CFG}(A) \) to be the rooted directed graph with vertices \( \text{Ctrl} \), root \( \text{init} \), and an edge \( n \rightarrow m \) iff \( \exists s, t. (n, s) \rightarrow (m, t) \). For our purposes CFGs are unlabelled; we write \( n \rightarrow m \) for \((n, m)\) to avoid confusion with various other uses of pairs. A path is a non-empty sequence of vertices that are consecutive under the edge relation. By mapping the first projection \( (\text{ctrl}) \) over a trace \( \tau \) of \( A \) we get its control path, \( \text{cpath}(\tau) \), i.e., the sequence of control points in \( \tau \).

A cutpoint set for \( A \) is a set \( K \) of control points with \( \text{init} \in K \) and \( \text{fin} \in K \), such that every cyclic path in \( \text{CFG}(A) \) contains at least one element of \( K \). Define \( \text{segs}(A, K) \), the segments for \( K \), to be the finite paths between cutpoints that have no intermediate cutpoint. Formally, \( vs \in \text{segs}(A, K) \) iff \( vs \) is finite, \( \text{len}(vs) > 1 \), \( vs_0 \in K \), \( vs_{-1} \in K \), and \( \forall i. 0 < i < \text{len}(vs) - 1 \Rightarrow vs_i \notin K \). A segment \( vs \) is meant to refer to execution starting at control point \( vs_0 \) and ending at \( vs_{-1} \), hence the requirement \( \text{len}(vs) > 1 \). Note that \( \text{segs}(A, K) \) is finite because \( \text{CFG}(A) \) is finite and \( K \) cuts every cycle.

As an example, Fig. 2 shows a labelled version of program \( c_0 \). Fig. 3 shows the CFG of the automaton for \( c_0 \); skip0. The trailing skip serves to provide an end label. The figure shows code as edge labels, for clarity, but we do not formally consider edge-labelled CFGs. One cutpoint set is \( \{1, 3, 6\} \); its segments are \( \{1, 2, 3\} \), \( \{3, 4, 5, 3\} \), and \( \{3, 6\} \).

It is convenient to have notation for the effect of transitions along a segment. Given \( vs \in \text{segs}(A, K) \) define relation \( \tau \rightarrow vs \) by \( (n, s) \rightarrow vs (m, t) \) iff there is a trace \( \tau \) of \( A \) with \( \text{cpath}(\tau) = vs \) and \( \tau_0 = (n, s) \) and \( \tau_{-1} = (m, t) \). Notice that \( \tau \rightarrow vs \) need not be total, even in the typical case that the underlying relation \( \rightarrow \) is never stuck on non-f\( \text{in} \) states. For example, if \( vs_0 \) represents a conditional branch and in state \( (vs_0, s) \) the condition does not drive the automaton to \( vs_1 \) then \( (vs_0, s) \) is not in the domain of \( \tau \rightarrow vs \).

Given automaton \( A \), cutpoint set \( K \), and spec \( P \Rightarrow Q \), an annotation is a function \( an \) from \( K \) to store predicates such that \( an(\text{init}) = P \) and \( an(\text{fin}) = Q \). The requirement \( \text{init} \neq \text{fin} \) ensures that annotations exist for any spec.
c0: \ y := 1; x := z := 2; \text{while}^3 y \neq 0 \text{ do } z := z + y; y := y - 1 \od

c4: \ y := 1; z := 2; w := 3; \text{while}^4 y \neq 4 \text{ do } if^2 w \% 2 = 0 \text{ then } z := z + y; y := y - 1 \text{ else skip}^5 fi; w := w + 1 \od

c5: \ y := 1; z := 2; w := 3; \text{while}^4 y \neq 4 \text{ do } if^2 w \% 3 = 0 \text{ then } z := z + y; y := y - 1 \text{ else skip}^5 fi; w := w + 1 \od

Fig. 2. Example labelled commands; c4 and c5 are variations on c2 and c3 of Fig. 1.

![CFG of the automaton aut(c0; skip6)](image)

We lift \(\text{ann}^6\) to a function \(\text{ann}\) that yields states: \(\text{ann}(n) = \{ (n, s) \mid s = \text{ann}(n) \}\). Put differently: \(\text{ann}(n) = \{ n \} \times \text{ann}(n)\). For each \(vs\) in \(\text{seg}(A, K)\) there is a verification condition (VC):

\[
\text{post}((\text{vs}^n)(\text{ann}(vs_0))) \subseteq \text{ann}(vs_{-1}) \tag{1}
\]

Here \(\text{post}((\text{vs}^n))\) is the direct image i.e., strongest postcondition. Using the universal pre-image is equivalent to \(\text{ann}(vs_0) \subseteq \text{pr}((\text{vs}^n)(\text{ann}(vs_{-1})))\). The VC says that for every trace \(\tau\) with epath(\(\tau\)) = \(vs\), if \(\tau_0 \models \text{ann}(vs_0)\) then \(\tau_{-1} \models \text{ann}(vs_{-1})\). Annotation \(\text{ann}\) is valid if all the VC\(s\) are true. A segment represents a finite execution that follows that control path, so pre-image is the weakest precondition.

**Proposition 1** (soundness of IAM). Consider a valid annotation, \(\text{ann}\), of automaton \(A\) with cutpoint set \(K\), for \(P \leadsto Q\). In any initial trace \(\tau\) of \(A\) such that \(\tau_0 \models P\), at any position \(i, 0 \leq i < \text{len}(\(\tau\))\), we have \(\text{ctrl}(\tau_i) \in K \Rightarrow \tau_i \models \text{ann}(\text{ctrl}(\tau_i))\). Moreover \(A \models P \leadsto Q\).

**Proposition 2** (completeness of IAM). Suppose \(A \models P \leadsto Q\) and let \(K\) be a cutpoint set. Then there is an annotation on \(K\) that is valid.

A full annotation of an automaton is one where the cutpoint set is all control points. Using strongest postconditions, including disjunction at control points, one can show:

**Lemma 3.** Any valid annotation can be extended to a full annotation that is valid.

In fact the extension can be constructed efficiently, for many assertion languages and programming languages.

**Labelled commands:** A few tedious but routine technical details need to be spelled out in order to precisely formulate the main results. Hoare logic is about programs; to make connections with automata we use syntax with labels \(n \in \mathbb{Z}\):

\[
c ::= \text{skip}^n \mid x := n \mid c; c \mid c \big| n \leftarrow c \mid \text{if}^n e \text{ then } c \text{ else } c \text{ fi } \mid \text{while}^n e \text{ do } c \text{ od}
\]

Here metavariable \(x\) ranges over a set \(\text{Var}\) of variable names and \(e\) ranges over integer expressions. The form \(c \big| n \leftarrow d\) is for nondeterministic choice. We also use metavariables \(b, d, c, \ldots\) for commands.

Let \(\text{lab}(c)\) be the label of command \(c\), defined recursively in the case of sequence: \(\text{lab}(c; d) = \text{lab}(c)\). Let \(\text{labs}(c)\) be the set of labels that occur in \(c\). The label of a command can be understood as its entry point. We focus on programs of the form \(c; \text{skip}^\text{fin}\) where \(\text{fin} \in \mathbb{N}\) serves as the exit label for \(c\). Such a program can take at least one step, even if \(c\) is just skip; this fits with our formulation of automata.

Negative labels are used in the transition semantics, but for most purposes we are concerned with “main programs” which are required to have unique, non-negative labels. This is formalized by the predicate \(\text{ok}(c)\) defined straightforwardly. Fig. 2 gives example labelled commands. The transition semantics is standard except for the manipulation of labels, which is done in a way that facilitates definitions to come later.

As in the discussion of automata, let \(s\) and \(t\) range over stores —but here we use \textit{variable stores}, i.e., total mappings from \textit{Var} to \(\mathbb{Z}\). We write \(s[x := i]\) for the store like \(s\) but mapping \(x\) to \(i\). A \textit{configuration} \(\langle c, s\rangle\) pairs a labelled command with a store, and we let \(\text{ctrl}(\langle c, s\rangle) = c\) and \(\text{store}(\langle c, s\rangle) = s\).

The transition relation \(\rightarrow\) is defined in Fig. 4. In a configuration reached from an \(\text{ok}\) command, the only negative labels are those introduced by the transitions for assignment and while, which introduce negative labels on skip commands. For while, one transition rule duplicates the loop body, creating non-unique labels. For every \(c, s\), either \(\langle c, s\rangle\) has a successor or \(c\) is \(\text{skip}^n\) for some \(n \in \mathbb{Z}\). Assume integer expressions are everywhere defined, so configurations are not stuck under \(\rightarrow\) unless the program is a single skip.
The automaton of a program: If \( \mathsf{ok}(c) \) and \( m \in \mathsf{labs}(c) \), let \( \mathsf{sub}(m, c) \) be the sub-command of \( c \) with label \( m \). For example, consider \( c_0 \) in Fig. 2 then \( \mathsf{sub}(2, c_0) \) is \( x := 1 \) and \( \mathsf{sub}(3, c_0) \) is \( y := 0 \). To manipulate the CFG of a program of the form \( c; \mathsf{skip}^n \) that is \( \mathsf{ok} \) (so, \( \mathsf{fin} \notin \mathsf{labs}(c) \)), we define functions \( \mathsf{fsuc} \) and \( \mathsf{elab} \). For motivation, the control flow successors of the loop, \( \mathsf{sub}(3, c_0) \), in \( c_0; \mathsf{skip}^6 \) are 3 and 6. Whereas 3 is inside \( c_0 \), 6 is not. We call 6 the following successor, given by \( \mathsf{fsuc}(3, c_0, 6) \). In general, \( \mathsf{fsuc}(m, c, f) \) is defined by recursion on \( c \); see Fig. 5. For example, \( \mathsf{fsuc}(\mathsf{sub}(3, c_0), 6) = 6 \) and \( \mathsf{fsuc}(\mathsf{sub}(5, c_0), 7) = 3 \). For another example, let \( c \) be \( x > 0 \) then \( x := 2 \) \( x - 1 \); \( y := x \) else \( \mathsf{skip}^4 \); then \( \mathsf{fsuc}(2, c, 5) = 3 \) and \( \mathsf{fsuc}(1, c, 5) = \mathsf{fsuc}(3, c, 5) = \mathsf{fsuc}(4, c, 5) = 5 \).

For subcommand \( b \) of \( c \) we define \( \mathsf{elab}(b, c, \mathsf{fin}) \) to be the exit label, i.e., the label to which control goes after every path through \( b \). In case \( b \) is a conditional, loop, choice, assignment or skip, let \( \mathsf{elab}(b, c, \mathsf{fin}) = \mathsf{lab}(\mathsf{elab}(b), c, \mathsf{fin}) \). In case \( b \) is a sequence \( b_0; b_1 \), let \( \mathsf{elab}(b, c, \mathsf{fin}) = \mathsf{elab}(b_1, c, \mathsf{fin}) \), i.e., the exit of a sequence is the exit of its last command.

Now we can define the CFG for an \( \mathsf{ok} \) program \( c; \mathsf{skip}^n \) and with this in mind define an automaton with the same CFG to represent the program. The nodes of the CFG are the control points \( \{\mathsf{fin}\} \cup \mathsf{labs}(c) \). There is no control flow successor of \( \mathsf{fin} \). For \( n \in \mathsf{labs}(c) \) there are one or two successors, described by cases on \( \mathsf{sub}(n, c) \):

\[
\begin{align*}
\mathsf{sub}(n, c) & = \vdash n. e \quad \mathsf{successor}(s) \text{ of } n \text{ in the CFG} \\
\mathsf{sub}(n, c) & = \vdash n. \mathsf{fsuc}(n, c, \mathsf{fin}) \\
\mathsf{sub}(n, c) & = \vdash n. \mathsf{skip}^n \\
\mathsf{sub}(n, c) & = \vdash n. \mathsf{lab}(d_0) \text{ and } n. \mathsf{lab}(d_1) \\
\mathsf{sub}(n, c) & = \vdash n. \mathsf{lab}(d_0) \text{ and } n. \mathsf{lab}(d_1) \\
\mathsf{sub}(n, c) & = \vdash n. \mathsf{fsuc}(n, c, \mathsf{fin}) \\
\end{align*}
\]

The automaton of an \( \mathsf{ok} \) program \( c; \mathsf{skip}^n \), written \( \mathsf{aut}(c; \mathsf{skip}^n) \), is \( (\mathsf{labs}(c) \cup \{\mathsf{fin}\}, (\mathsf{Var} \rightarrow \mathbb{Z}), \mathsf{lab}(c), \mathsf{fin}, \mathsf{⇒}) \) where \( (n, s) \Rightarrow (m, t) \) iff either

- \( \mathsf{elab}(m, c, s) \Rightarrow (d, t) \land \mathsf{lab}(d) \geq 0 \land m = \mathsf{lab}(d) \)
- \( \mathsf{elab}(m, c, s) \Rightarrow (d, t) \land \mathsf{lab}(d) < 0 \land m = \mathsf{lab}(d) \)
- or \( \mathsf{sub}(m, c) = \mathsf{skip}^n \land m = \mathsf{fsuc}(n, c, \mathsf{fin}) \land t = s \)

The first two cases use the semantics of Fig. 4 for a subcommand on its own. The second case uses \( \mathsf{fsuc} \) for a subcommand that has terminated. The third case handles skip which on its own would be stuck, but which should take a step when it occurs as part of a sequence. The only stuck states of \( \mathsf{aut}(c; \mathsf{skip}^n) \) are terminated ones.

For any traces \( \tau \rightarrow v \) \( \mathsf{aut}(\tau) = \mathsf{lab}(v) \) and \( \mathsf{stor}(\tau) = \mathsf{lab}(v) \) \( \mathsf{lab}(\mathsf{ctrl}(\tau)) = \mathsf{lab}(v) \), for all \( i, 0 < i < \mathsf{len}(\tau) \).

**Lemma 4.** Suppose \( \mathsf{ok}(c; \mathsf{skip}^n) \). Let \( A \) be \( \mathsf{aut}(c; \mathsf{skip}^n) \) and let \( x \) be a store.

1. For any trace \( \tau \rightarrow (c; \mathsf{skip}^n, s) \rightarrow \rightarrow \), there is a trace \( v \rightarrow \rightarrow A \rightarrow \rightarrow (c; \mathsf{skip}^n, s) \rightarrow \rightarrow \), such that \( \tau \rightarrow v \).
2. For any trace \( \tau \rightarrow A \rightarrow \rightarrow (c; \mathsf{skip}^n, s) \rightarrow \rightarrow \), there is a trace \( \tau \rightarrow (c; \mathsf{skip}^n, s) \rightarrow \rightarrow \), such that \( \tau \rightarrow v \).

For a full annotation, the segments are exactly the paths of length two, i.e., the edges of the CFG. This enables a straightforward description of the VCs for the automaton of a program (not unlike the VCs given by Floyd [7] for flowchart programs).

**Lemma 5 (VCs for programs).** Consider an \( \mathsf{ok} \) program \( c; \mathsf{skip}^n \) and a full annotation, \( an \), of its automaton. For each control edge \( n \rightarrow m \), the VC [7] can be expressed as in Fig. 6.

The conditions are derived straightforwardly from the semantic definitions. Although we are treating assertions as sets of stores, we use formula notations like \( \wedge \) and \( \Rightarrow \), rather than \( \cap \) and \( \subseteq \), for clarity. We write \( an(n) \land c \) to abbreviate \( an(n) \cap \{s \mid s(e) = 0\} \). For a set \( P \) of program stores, we use substitution notation \( P^c \) with standard meaning: \( s \in P^c \) iff \( \{s \mid s(e) \in P\} \). The Floyd completeness of Hoare Logic: Fig. 7 gives the rules of HL. We write \( \vdash \) to indicate derivability using the rules. As usual, the semantics is that for all \( s, t \), if \( s \models P \) and \( \langle c, s \rangle \rightarrow^* \langle \mathsf{skip}^n, t \rangle \) then \( t \models Q \). We write this as \( \models c : P \rightarrow Q \). As is well known, the rules are sound: \( \vdash c : P \rightarrow Q \) implies \( \models c : P \rightarrow Q \).

A corollary of Lemma 4 is that if \( \mathsf{ok}(c; \mathsf{skip}^n) \) then \( \mathsf{aut}(c; \mathsf{skip}^n) \Rightarrow P \rightarrow Q \) iff \( \mathsf{aut}(c; \mathsf{skip}^n) \Rightarrow P \rightarrow Q \). The trailing skip loses no generality. By semantics, \( \models c : P \rightarrow Q \) iff \( \vdash c : P \rightarrow Q \). In terms of proofs, the two are equi-derivable.

Given a valid annotation for \( \mathsf{aut}(c; \mathsf{skip}^n) \) and \( P \rightarrow Q \), by Prop. 1 we have \( \mathsf{aut}(c; \mathsf{skip}^n) \Rightarrow P \rightarrow Q \), so by the corollary of Lemma 4 we have \( \models c : \mathsf{skip}^n : P \rightarrow Q \) and thus \( \models c : P \rightarrow Q \). So by the standard (Cook) completeness result for HL [3], [4] there is a proof of \( c : P \rightarrow Q \). The idea of Floyd completeness is that from a valid annotation one may obtain a proof that essentially uses only judgments given directly by the annotation. For a full annotation, \( an \), of \( \mathsf{aut}(c; \mathsf{skip}^n) \), define the associated judgments to be:

- for subprograms \( b \) of \( c \), the judgments

\[
\begin{align*}
b : \mathsf{lab}(b) & \Rightarrow \mathsf{lab}(b) \\
b & \Rightarrow \mathsf{elab}(b, c, \mathsf{fin})
\end{align*}
\]

- for a loop, or then-branch of a conditional, with test \( e \); \( b \Rightarrow \mathsf{lab}(b) \land e \Rightarrow \mathsf{elab}(b, c, \mathsf{fin}) \)

- for the else-branch of a conditional with test \( e \);
Fig. 6. VCs for the automaton of a program $c$; skip$^{fin}$ and full annotation $an$.

![Diagram](image-url)
which we call agreement. For example, \( x \equiv x \Rightarrow z \equiv z \) expresses that the final value of \( z \) is determined by the initial value of \( x \). Owing to our use of non-zero integers to represent truth (in semantics Fig. 9), we also need a different form, \( e \equiv e' \), to express agreement on truth value; it denotes \( \{(s', s) \mid s(e) \neq 0 \iff s(e') \neq 0\} \). We also write \( e \neq e' \) for the set \( \{(s, t) \mid s(e) \neq 0 \} \) and \( e \neq e' \) for \( \{(s, t) \mid t(e) \neq 0\} \).

In the presence of nondeterminism, the \( \forall \exists \)-properties expressed by specs \( \mathcal{R} \models S \) are not the only properties of interest. For example, the program \( x := x + 1 \land x := x + 2 \) does not satisfy \( x \equiv x \Rightarrow x \equiv x \), but it does satisfy the “possibilistic noninterference” property that for any two stores \( s, s' \) that agree on \( x \), and any run from \( s \), there exists a run from \( s' \) with the same final value for \( x \). Such \( \forall \exists \)-properties are beyond the scope of this paper. The nondeterministic problem \( (y := 0 \lor y := 1); x := x + 1 \) does satisfy \( x \equiv x \Rightarrow x \equiv x \).

A product of automata \( A \) and \( A' \) is meant to represent a chosen alignment of steps of \( A \) with steps of \( A' \). In many cases, the control points of a product are pairs \((n, m)\) from underlying automata, but we continue to use identifiers \( n, m \) for control points regardless of their type.

A product of \( A \) and \( A' \) is an automaton \( \Pi_{A,A'} \) of the form \((C, (\text{Sto} \times \text{Sto'}), i, f, \Rightarrow)\), together with functions \( \text{lit} : C \to \text{Ctrl} \) and \( \text{rt} : C \to \text{Ctrl'} \) such that the following hold: First, \( \text{lit}(i) = \text{init}, \text{rt}(i) = \text{init}', \text{lit}(f) = \text{fin}, \) and \( \text{rt}(f) = \text{fin}' \). Second, \( (n, (s, s')) \Rightarrow (m, (t, t')) \) implies either

\[
\begin{align*}
\text{left-only.} & \quad ((n, n'), (s, s')) \Rightarrow_{\text{lok}} ((m, m'), (t, t')) \iff \text{iff} (n, s) \Rightarrow (m, t) \text{ and } (n', s') \Rightarrow (m', t') \\
\text{right-only.} & \quad ((n, n'), (s, s')) \Rightarrow_{\text{req}} ((m, m'), (t, t')) \iff \text{iff} (n, s) = (m, t) \text{ and } (n, s') \Rightarrow (m', t'), \\
\text{ interleaved.} & \quad ((n, n'), (s, s')) \Rightarrow_{\text{rsk}} ((m, m'), (t, t')) \iff \text{iff} (n, s) = (m, t) \text{ and } (n', s') \Rightarrow (m', t').
\end{align*}
\]

As an example of the \( \text{ctrl-conditioned} \) form, the lockstep-control product restricts only-lockstep to additionally require the two executions to follow the same control path. (So it only reaches a linear number out of the quadratically many control points.) This can be described by taking \( L = R = \emptyset \) and defining the condition for joint steps by \((n, n') \in J \) iff \( n = n' \). The reader can check that the \( \text{ctrl-conditioned} \) form subsumes the others listed above.

A product can also be conditioned on data, as explored in Sec. VII Fig. 8 depicts a product of \( \emptyset \) with itself, in which an iteration of the loop body on just the left (resp. right) side may happen only when the store relation \( L \) (resp. \( R \)) holds.

The only-lockstep form is not adequate, in general, because a terminated state can be reached on one side before the other side terminates. But even lockstep-control can be \( \mathcal{R} \)-adequate in some cases, for example let \( \mathcal{R} \) be equality of stores and \( A = A' \), then lockstep-control is \( \mathcal{R} \)-adequate if \( A \) is deterministic.

The interleaved product is strongly adequate—but not very helpful, since so many pairs of control points are reachable.

The sequential form is adequate but not strongly adequate: if \( \tau \) is a finite prefix of a divergent trace, and \( \tau' \) has length > 1, the sequential product never finishes on the left and so does not cover \( \tau' \).

Eager-lockstep is adequate, and strongly adequate if the underlying automata have no stuck states. It shows the need for prefix in the definition of strong adequacy: if \( \tau \) is shorter than \( \tau' \), the product automaton needs to extend \( \tau \) by further steps in order to cover \( \tau' \).

Auxiliary control state can be used to ensure strong adequacy. The dovetail product has \( C = \text{Ctrl} \times \text{Ctrl'} \times \{0, 1\} \) with \( \text{lit}(n, n', i) = n \) and \( \text{rt}(n, n', i) = n' \). Let \((n, n', i, (s, s')) \Rightarrow_{\text{dov}} ((m, m', j), (t, t')) \) iff either

\[
\begin{align*}
&i = 0, j = 1, ((n, n')(s, s')) \Rightarrow_{\text{lo}} ((m, m'), (t, t')) \text{ or} \\
&i = 1, j = 0, ((n, n')(s, s')) \Rightarrow_{\text{req}} ((m, m'), (t, t')) \text{ or} \\
&\text{one of the last two eager-lockstep cases apply (one side terminated)}.
\end{align*}
\]

The most general notion of product allows auxiliary store in addition to auxiliary control. We return to this in Sec. VIII.

Owing to the definition of stores of a product, a relation \( \mathcal{R} \subseteq \text{Sto} \times \text{Sto'} \) from stores of \( A \) to stores of \( A' \) is the same thing as a predicate on stores of a product \( \Pi_{A,A'} \). So a
Fig. 8. Conditionally aligned loop product automaton for $c_0 | c_0$, with informal edge labels including alignment guards $L, R$.

relational spec $R \approx S$ for $A, A'$ can be seen as a unary spec $R \leadsto S$ for $A, A'$.

**Theorem 7.** For the basic semantics of specs, if $\Pi_{A,A'}$ is an $R$-adequate product of $A, A'$ then $\Pi_{A,A'} \models R \leadsto S$ iff $A|A' \models R \leadsto S$. For the non-stuck semantics, if $\Pi_{A,A'}$ is a strongly $R$-adequate product then $\Pi_{A,A'} \models R \leadsto S$ implies $A|A' \models R \leadsto S$.

This lifts IAM to a method for proving a relational judgment for programs: construct their automata, define a product $\Pi$ and an annotation $\alpha$; prove $R$-adequacy of $\Pi$ and validity of $\alpha$.

The number of cutpoints needed for a product may be on the order of the product of the number for the underlying automata. But some products have fewer. Unreachable cutpoints can be annotated as false so the corresponding VCs are vacuous.

**IV. A ONE-RULE COMPLETE RHL**

Recall the sequential product rule sketched in Sec. I. Sequential product is adequate so by Theorem 7 it can be used to prove correctness for any relational spec. This leads to the well known rule that we now study in detail.

Although the definition of automaton allows an arbitrary set for stores, in HL we require stores to be mappings on variables, specifically on the set $\text{Var}$. To express a product as a single program we need to encode a pair of such stores as a single one. Let $\text{Var}^*$ be the set of fresh variable names $x^*$ such that $x \in \text{Var}$. Let $\dot{\text{dot}} : \text{Var} \rightarrow \text{Var}^*$ be the obvious bijection. Given $s : \text{Var} \rightarrow \mathbb{Z}$ and $t : \text{Var}^* \rightarrow \mathbb{Z}$, the union $s \cup t$ is a function $\text{Var} \cup \text{Var}^* \rightarrow \mathbb{Z}$ (treating functions as sets of ordered pairs). For $s$ and $t$ of type $\text{Var} \rightarrow \mathbb{Z}$ define $s + t = s \cup (t \text{dot}^{-1})$, so $s + t : \text{Var} \cup \text{Var}^* \rightarrow \mathbb{Z}$ faithfully represents $(s, t)$. For relation $R$ on variable stores, let $R^\oplus$ be the predicate on $\text{Var} \cup \text{Var}^* \rightarrow \mathbb{Z}$ given by $R^\oplus = \{s + t \mid (s, t) \in R\}$. We overload the name dot for the function renaming variables of an expression, and for command $c$ on $\text{Var}$ let $\dot{\text{dot}}(c)$ be the command on $\text{Var}^*$ obtained by renaming; this leaves labels unchanged.

Define semantic substitution $R^\oplus_{\text{dot}e'}$ by $(s, t) \in R^\oplus_{\text{dot}e'}$ iff $[\{s \mid x : s(x), [t \{x \rightarrow t(e')])\}] \in R$. (Here $x$ need not be distinct from $x'$.) Substitution is preserved by the encoding:

$$R^\oplus_{\text{dot}e'} \models \left( R^\oplus_{\text{dot}e'} \right)_{e,e'} \quad (4)$$

(Using simultaneous substitution on the right; it can as well be written $((R^\oplus_{\text{dot}e'})_{e,e'})^\oplus$. We write $R^\oplus_{\text{dot}e'}$ for substitution that leaves the right side unchanged. We refrain explicit notation to distinguish between correctness judgments for $\text{Var}$-programs and those for $\text{Var} \cup \text{Var}^*$-programs.

**Lemma 8.** Let $c, d$ be programs on $\text{Var}$ and $R, S$ be relations on $\text{Var}$-stores. Let $d' = \text{dot}(d)$, so $c; d'$ is a $(\text{Var} \cup \text{Var}^*)$-program. Then $c; d : R^\oplus \leadsto S^\oplus$ iff $c ; d : R \leadsto S$.

The lemma implies soundness of the following sequential product rule (sometimes called self composition):

$$\text{SEQPROD} \quad d' = \text{dot}(d) \quad c; d' : R^\oplus \leadsto S^\oplus \quad c \mid d : R \leadsto S$$

**Proposition 9** (one-rule complete RHL). The logic comprised of SEQPROD and the rules of HL (Fig. 7) is complete.

To prove this, suppose $c ; d : R \leadsto S$. By Lemma 8 we have $c ; d' : R^\oplus \leadsto S^\oplus$ where $d' = \text{dot}(d)$. By completeness of HL we have $c ; d' : R^\oplus \leadsto S^\oplus$ so SEQPROD yields $c ; d : R \leadsto S$, q.e.d.

Analogous to Floyd completeness (Thm. 6), we can directly prove a stronger result. It refers to the associated judgments of an annotation of a sequential product. These are similar to those defined preceding Thm. 6, but now they have the form $b : an(n, 1)^\oplus \leadsto an(m, 1)^\oplus$ (with $b$ a subprogram of $c$ with $n = \text{lab}(b)$ and $m$ the end label), the form $b : a(n, fin, m)^\oplus \leadsto an(fin, m)^\oplus$, and the variations like preceding Thm. 6. The exact definition of associated judgment becomes evident in the proof to follow.

**Theorem 10** (alignment completeness for sequential product). If $a(n)$ is a valid full annotation of the sequential product of $\text{aut}(c; \text{skip}^\text{fin})$ and $\text{aut}(d; \text{skip}^\text{fin})$, for spec $R \leadsto S$, then $c \mid d : R \leadsto S$ in the logic comprised of HL and SEQPROD. Moreover this judgment can be proved using only the associated judgments.

As a corollary, any valid annotation of a sequential product gives a provable judgment, using loop invariants given by the annotation, because the annotation can be extended to a full one (Lemma 3). We choose to state all the theorems for full annotations, just to avoid a more complicated definition for the associated judgments.

The proof relies on an analysis like Lemma 5 but for VCs of sequential product. Assume w.l.o.g. that $\text{lab}(c) = 1 = \text{lab}(d)$. As a first step, consider the CFG of the sequential product $\Pi$ of $\text{aut}(c; \text{skip}^\text{fin})$ and $\text{aut}(d; \text{skip}^\text{fin})$. Initial states of $\Pi$ have the form $((1, 1), (s, t))$, final states have the form $((\text{fin}, \text{fin}), (s, t))$, and edges of the CFG are of two forms: $(n, 1) \rightarrow (m, 1)$ for $n \rightarrow m$ in the CFG of $c; \text{skip}^\text{fin}$ and $((\text{fin}, n)) \rightarrow (\text{fin}, m)$ for $n \rightarrow m$ in the CFG of $d; \text{skip}^\text{fin}$.

Now assume $a(n)$ is a full annotation of $1$. Because steps of $\Pi$ represent execution of a program on one side or the other, the VCs are similar to those in Fig. 6 except that they pertain to control points of the forms $(n, 1)$ and $(\text{fin}, n)$. To
save space we just give some illustrative cases in Fig. 9 using
some notations from Sec. [11].

Let \( Q = an(fin, 1) \). We will use \( Q^\oplus \) as the intermediate
assertion for rule SEQ in a proof of \( c; dot(d) : R^\oplus \sim S^\oplus \)
which can then be used in SEQPROD to obtain \( c|d : R \approx \approx S \).
To obtain proofs of \( c : R^\oplus \sim Q^\oplus \) and \( dot(d) : Q^\oplus \sim S^\oplus \), we
use the VCs of Fig. 9 in an argument similar to the proof of Thm. [6]. By induction on \( c \) we can show, for all subcommands \( b \) of \( c \):

\[
\vdash b : an(lab(b), 1)^\oplus \sim an(m, 1)^\oplus
\]  

(5)

where \( m = \text{lab}(b, c, \text{fin}) \). By induction on \( d \) we can show, for all subcommands \( b \) of \( d \):

\[
\vdash \text{dot}(b) : \text{an(fin, lab(b))}^\oplus \sim \text{an(fin, m)}^\oplus
\]  

(6)

where \( m = \text{lab}(b, d, \text{fin}) \). In proving (6) using Fig. 9 we use
that \( (an(fin, n) \land [e])^\oplus = an(fin, n)^\oplus \land d \) and \( \text{an(fin, n)}^\oplus = (an(fin, n)^\oplus)^\oplus \). Instantiating (5) and (6)
we get \( \vdash c : an(1, 1)^\oplus \sim an(1, 1)^\oplus \)

\[
\vdash \text{dot}(d) : \text{an(fin, 1)}^\oplus \sim \text{an(fin, m)}^\oplus
\]  

Thus \( \vdash c : R^\oplus \sim Q^\oplus \) and \( \vdash \text{dot}(d) : Q^\oplus \sim S^\oplus \), because
\( an(1, 1) = R, an(fin, 1) = Q, \) and \( \text{an(fin, m)} = S \).

V. A LOGIC OF LOCKSTEP ALIGNMENT

So far we have that SEQPROD is complete in the sense of
Cook, and alignment complete with respect to alignments
represented by sequential product. It is not complete with
respect to other classes of alignments. For example, consider
lockstep-control alignments. As mentioned in Sec. [1] such an
alignment enables to prove \( c|c : x = x \Rightarrow x = z \) using an
annotation with intermediate relations only \( y = y \land z = z \).
A proof using SEQPROD with \( c|c : \text{dot}(c) \) requires to assert
\( z = x! \land x = x^* \) at the semicolon, and to use factorial in
invariants. This is far beyond the assertions and judgments
associated with the annotation of the lockstep product.

Fig. 10 gives rules for relational judgments sometimes
called “diagonal” [12] because they relate same-structured
programs. They are typical of RHLs [11, 13] and we call
them lockstep because they embody lockstep-control alignment,
with side conditions for agreement of tests. The relational
version of CONSEQ is included in Fig. 10 because it is needed in order
for this collection of rules to be complete for lockstep-control alignment, which we make precise in
Theorem 11. These rules are not complete in the sense of
Cook, for relational judgments in general, because they do not apply to differently-structured commands and do not support reasoning about differing control paths. For example, the
monotonicity property \( x \leq x' \Rightarrow y \leq y' \) is satisfied by
if \( x > 0 \) then \( y := x + 1 \) else \( y := x \). But \( x \leq x' \) does not
imply agreement on the value of \( x > 0 \).

In a lockstep-control product, the CFG edges have the form
\( (n, n) \rightarrow (m, m) \). For this to be sufficient for \( R \)-adequacy, the
code paths reached from initial state-pairs satisfying \( R \) need to
be the same. Thus lockstep control is not adequate for
programs with choice except in trivial cases like \( x := 0 \cup x := 0 \).
Choice is ruled out in theorem.

Say \( c \) and \( c' \) have same control, written sameCtl \((c, c') \), if
\( \text{labs}(c) = \text{labs}(c') \) and for each \( n \in \text{labs}(c) \) the programs
\( \text{sub}(n, c) \) and \( \text{sub}(n, c') \) are the same kind: both are assignments,
both are skip, both are if, and so on; moreover \( n \) has the
same control flow successors in \( c \) and \( c' \). Put differently:
\( c' \) can be obtained from \( c \) by renaming variables and replacing
expressions in assignments and branch conditions, but no other
changes. For example, \( x := y + z \) has same control as
\( x := y^3, x \) and \( x \) and \( c_4 \) and \( c_5 \) in Fig. 13. Same control implies
identical CFGs.

For \( c, c' \) with same control, and their lockstep-control product
\( \Pi \), VCs for a full annotation are given in Fig. 11. As usual, the VCs for conditional have the test or its negation as
antecedent, because the VC embodies the program semantics
while assuming control is along a particular path; see (1).

Regardless of whether the annotation is full, if the cutpoints
include all branch conditions, and for each point \( n \) with branch
conditions \( e, e' \), respectively, we have \( an(n, n) \Rightarrow e \equiv e' \),
then \( \Pi \) is \( R \)-adequate (if \( an \) is valid). This is because by
program semantics and definition of lockstep-control, the only
stuck non-terminated states are those where the program is
a conditional branch and the conditions disagree (so the
successor control points differ).

As with Prop. 2 and Theorem 10, an annotation determines a set of what we call associated judgments. For
lockstep automata, the associated judgments are much like
those defined preceding Prop. 2 only doubled, like \( b|b' : an(lab(b), lab(b)) \Rightarrow an(m, m) \) where \( m = \text{lab}(b, c, \text{fin}) \),
together with those obtained by adding if-tests and so forth.
For lack of space we refrain from spelling them out.

**Theorem 11** (alignment completeness for lockstep product).
*Suppose \( c \) and \( c' \) are choice-free and satisfy sameCtl \((c, c') \).* Let \( \Pi \) be the lockstep-control product of \( a(u(c); \text{skip}) \) and
\( a(u(c'); \text{skip}) \) and let \( an \) be a valid full annotation of \( \Pi \) for
\( \Pi \Rightarrow S \). Assume that for any branch point \( n \) with tests \( e, e' \)
we have \( an(n, n) \Rightarrow e \equiv e' \). Then \( \vdash c|c' : \Pi \Rightarrow S \) in
the logic comprising just the rules of Fig. 10. Moreover this can
be proved using only the associated judgments.

Informally, for a relation between two programs with the
same control structure, one may be able to argue that under
precondition \( R \) every pair of executions follows the same
control path. To formalize such an argument, the annotation at
each branch should include that their tests agree. The theorem
says this pattern of reasoning is covered by the rules of Fig. 10.

Such reasoning does not apply in the presence of pure
nondeterministic choice. Choice is sometimes used to model
externally determined inputs. An alternative is to model inputs
using additional variables, on which the precondition can
assert agreement.

To prove the theorem, we use that \( c \) and \( c' \) satisfy
sameCtl \((c, c') \). We show, by induction on structure of \( c \), that
if \( b = \text{sub}(n, c) \) is... and \((n, 1) \rightarrow (m, 1)\) in CFG is... then the VC is equivalent to...

\[
\begin{align*}
\textbf{@} \quad & x := e \quad m = \text{elab}(b, c, \text{fin}) \quad an(n, 1) \Rightarrow an(m, 1) \\
\text{if}^n \ e & \quad b_0 \sqsupset \quad b_1 \\
\text{if}^n e & \quad b_0 \sqsupset b_1 \\
\text{if}^n \ e & \quad b_0 \sqsupset b_1
\end{align*}
\]

Fig. 9. Selected VCs for sequential product of \( \text{aut}(c; \text{skip}^{lm}) \) and \( \text{aut}(d; \text{skip}^{lm}) \) and full annotation \( an \).

The arguments for sequence and while are similar, using rules \( \text{DSEQ} \) and \( \text{DWH} \). That completes the proof of (7), as \( c \) and \( c' \) are choice-free. Instantiating (7) with \( c, c' \) completes the proof.

Theorem 11 pertains to a restricted class of program pairs. We could relax the sameCtl condition slightly, to allow an assignment to match skip, still using lockstep control for the product. Then the VCs of Fig. 11 would include a case for \( x := e \) on the left and skip\(^n\) on the right, with VC \( an(n, n) \Rightarrow an(m, m) \) where \( m = \text{elab}(b, c, \text{fin}) = \text{elab}(b', c', \text{fin}) \). To get an assignment complete logic one would add the axiom \( x := e \mid \text{skip} : R\langle e \rangle \Rightarrow R \) (and a similar VC and rule for assignment on the right).

VI. COMBINING LOCKSTEP WITH SEQUENTIAL

A common practical problem is regression verification: equivalence of two programs that differ in that some subprogram has been replaced by another. We can describe this as equivalence of \( \hat{c}[b] \) and \( \hat{c}'[b'] \), using the usual notation \( \hat{c}[b] \) for a program context \( \hat{c}[] \) with a designated subprogram \( b \).

Informal reasoning might go by lockstep except for \( b' \).

In this section we consider a more general situation, relating \( \hat{c}[b] \) to \( \hat{c}'[b'] \) where the contexts \( \hat{c}[] \) and \( \hat{c}'[] \) have the same structure, as in Sec. V. We consider the logic comprised of \( \text{SEQPROD} \), \( \text{HL} \), and the rules of Fig. 10. The corresponding form of automata has both lockstep and one-sided sequential steps, where lockstep execution is used for similar control structure...
and one-sided only for the designated subprograms (which may have arbitrarily different structure).

To be precise, define sameExcept$(c, c’, b, b’, beg, end, fin)$ iff there are contexts $c[]$ and $c’[]$ such that

- $c = c[b]$ and $c’ = c’[b’]$
- $beg = \text{lab}(b) = \text{lab}(b’)$ and $\text{labs}(b) \cap \text{labs}(b’) = \{\text{beg}\}$
- $end = \text{elab}(b, c, fin) = \text{elab}(b’, c’, fin)$.
- $\text{sameCtrl}(c[\text{skip}^R], c’[\text{skip}^R])$
- $c[]$ and $c’[]$ are choice free (but $b, b’$ may have choice).

The setup encompasses any pair of programs, because it includes the extreme case where $c[]$ is nothing more than the hole to be filled, i.e., $c[b] = b$ and $c’[b’] = b’$; put differently, $c = b$ and $c’ = b’$. Here is an example that is only a little beyond what is encompassed by Theorem 11:

- $c$ is if $x > y$ then $y := y; x := x$ else skip fi and
- $c’$ is if $y < x$ then $x := 2$ else skip fi
- $b$ is $y := x; x := 0$ and $b’$ is a single assignment $x := 2$

Here $beg = 2$ and $\text{elab}(b, c, fin) = fin$. The example does not satisfy sameCtrl (because $b$ and $b’$ do not match).

To describe the product we use extra control state, for which we assume a set of three tags $\{\text{lock}, \text{lo}, \text{ro}\}$ to designate lockstep, left-only, and right-only steps. Let $\text{Ctrl}$ and $\text{Ctrl’}$ be the control sets for the automata of $c; \text{skip}^\text{fin}$ and $c’; \text{skip}^\text{fin}$ respectively, assuming as before that lab(c) = 1 = lab(c’). Recall $\text{Ctrl} = \text{labs}(c; \text{skip}^\text{fin})$ and $\text{Ctrl’} = \text{labs}(c’; \text{skip}^\text{fin})$. The control set of the product is $\text{Ctrl} \times \text{Ctrl’} \times \{\text{lock, lo, ro}\}$ and it and $\text{rt}$ are the first two projections. The initial and final control points are $(1, \text{lock})$ and $(\text{fin}, \text{lock}, \text{lock})$. Define $\Rightarrow$ as follows, where $\Rightarrow$ is from aut(c; skip fin), $\Rightarrow’$ is from aut(c’; skip fin), and $\Rightarrow_{\text{lock}}$ is the lockstep-control product based on those. For all $n, m, s, t, s’, t’$:

(i) $((n, n, \text{lock}), (s, s')) \Rightarrow ((m, m, \text{lock}), (t, t'))$

(ii) $((n, m, \text{lock}), (s, s')) \Rightarrow ((m, m, \text{lock}), (t, t'))$

(iii) $((n, m, \text{lock}), (s, s')) \Rightarrow ((m, m, \text{lock}), (t, t'))$

Rule (ii) enters left-only mode, (iii) continues, (iv) switches to right-only, (v) continues, and (vi) resumes lockstep. Like the lockstep-control product, this gets stuck at branch points outside the designated subprograms $b, b’$, if tests don’t agree.

Theorem 12. Suppose sameExcept$(c, c’, b, b’, beg, end, fin)$ and $\Pi$ is the product defined above. Suppose $an$ is a valid full annotation of $\Pi$ for $\Rightarrow S$. Suppose for all branch points $n \in \text{labs}(c) \setminus \{\text{beg}\}$ with branch conditions $e, e’$, we have $an(n, n, \text{lock}) \Rightarrow e \doteq e’$. Then $\forall c’ c : \Rightarrow S$ in the logic comprised of HL, the rules of Fig. 10 and $\text{SEQProd}$, using only the associated judgments.

As an exercise the reader may like to modify the product in this section to handle the special case where $b$ is $b0 \sqcup b1$ and $b’$ is $b0’ \sqcup b1’$ by nondeterministically choosing between four sequential executions ($b0’$ after $b0$, or $b1’$ after $b0$, etc.). Then recover alignment completeness by adding a relational proof rule that relates a choice to a choice, with four premises.

VII. Conditionally aligned loops

We want to prove $c2$ majorizes $c3$, that is, the judgment $c2 \triangleright c3 : x = x’ \land x > 3 \Rightarrow z > z’$, by aligning the iterations in which $y$ gets updated, maintaining invariant $y = y’ \land z > z’$ and allowing the no-op iterations to happen independently. To do so we use this rule 11:

\begin{align*}
\text{caWhile} & \quad c \triangleright c’ : Q \land [e \lor [e’]] \land \neg L \land \neg R \Rightarrow Q \\
& \quad \text{skip} : Q \land L \land [e] \Rightarrow Q \\
& \quad \text{skip} : Q \land R \land [e’] \Rightarrow Q \\
& \quad Q \Rightarrow e \doteq e’ \lor (L \land [e]) \lor (R \land [e’])
\end{align*}

while $e$ do $c$ od | while $e’$ do $c’$ od : $Q \Rightarrow Q \land \neg (e \lor e’)$

There are three premises, which strengthen the invariant $Q$ in three different ways. The first premise relates both loop bodies, like the lockstep loop rule in Fig. 10. The second and third premises each relate a loop body to skip, under preconditions strengthened by relations $L$ or $R$. The side condition ensures these are adequate to cover all cases. Later we consider the example of $c2, c3$ using $\{w \neq 2 \neq 0\}$ for $L$ and $\{w’ \neq 3 \neq 0\}$ for $R$. For caWhile to be useful, the proof system we consider has the lockstep rules of Fig. 10 together with one-sided rules of Fig. 12.

In order to focus on this situation, we consider relating same-control programs $c, c’$ with a distinguished label $\text{beg}$ such that sub(beg, c) is a loop—and so is sub(beg, c’), since we assume sameCtrl(c, c’). Let $\text{Ctrl}, \text{Ctrl’}$ and $\Rightarrow, \Rightarrow’$ be the control sets and transition relations for their automata (noting that $\text{Ctrl}’ = \text{Ctrl}$). As in Sec. VI we define a product with control $\Pi \times \text{Ctrl’} \times \text{Tag}$ where $\text{Tag} = \{\text{lock, lo, ro}\}$. The transition relation $\Rightarrow$ is defined with respect to given relations $\Pi$, $L$ and $R$. Unlike a ctrl-conditioned automaton, some transitions are conditioned on the stores. If $n$ is the label of the distinguished loop’s body, there are three transitions from the top of the loop into its body: $(\text{beg, beg, lock}) \hat{\rightarrow} (n, n, \text{lock})$ if neither $L$ nor $R$ holds, $(\text{beg, beg, lock}) \hat{\rightarrow} (n, \text{beg, lo})$ if $L$ holds, and $(\text{beg, beg, lock}) \hat{\rightarrow} (n, \text{beg, ro})$ if $R$ holds. Fig. 8 depicts an example. (See appendix for details.)

Theorem 13. Consider $c, c’, \text{beg, L, R such that sameCtrl}(c, c’), c$ and $c’$ are choice-free, sub(beg, c) is a loop, and $L, R$ are store relations. Let $\Pi$ be the product described above for $c; \text{skip}^\text{fin}, c’; \text{skip}^\text{fin}$. Suppose $an$ is a valid full annotation of $\Pi$ for $\Rightarrow Q$. Assume

- (a) for all branch points $n \in \text{labs}(c) \setminus \{\text{beg}\}$ with branch conditions $e, e’$, we have $an(n, n, \text{lock}) \Rightarrow e \doteq e’$;
- (b) $an(\text{beg, beg, lock}) \Rightarrow e \doteq e’ \lor (L \land [e]) \lor (R \land [e’])$

where $e, e’$ are the tests of the loops at beg.

Then $\forall c \triangleright c’ : \Rightarrow Q \Rightarrow Q$ in the logic comprised of the rules of Fig. 10, Fig. 12 and caWhile, using only the associated judgments.
AssSkip
\[ x := e \mid \text{skip} : R^x_\| \Rightarrow R \]  
SkipSkip
\[ \text{skip} \mid \text{skip} : R \Rightarrow R \]  
SeqSkip
\[ e \mid \text{skip} : R \Rightarrow Q \quad d \mid \text{skip} : Q \Rightarrow S \]  
IfSkip
\[ \begin{aligned} &c \mid \text{skip} : R \land \{ e \} \Rightarrow S \\ &d \mid \text{skip} : R \land \{ \neg e \} \Rightarrow S \end{aligned} \]
\[ \text{if } e \text{ then } c \text{ else } d \text{ fi } \mid \text{skip} : R \Rightarrow S \]
WhSkip
\[ c \mid \text{skip} : Q \land \{ e \} \Rightarrow Q \]  
\[ \text{while } e \text{ do } c \text{ od } \mid \text{skip} : Q \Rightarrow Q \land \{ \neg e \} \]

Fig. 12. One-side rules (symmetric right-side rules omitted).

Assumption (a) is about annotations of aligned branch points, even inside the distinguished loop at beg, but only in the part of the product CFG that executes in lockstep. For \( n \) a branch point inside that loop, the control points \((n, n, lo)\) and \((n, n, ro)\) are not reachable, by construction of \( \Pi \). Assumption (b) is like the side condition of rule cawhile and ensures adequacy.

Instead of showing how \( c_2 \) majorizes \( c_3 \), we consider variations \( c_4, c_5 \) in Fig. 2. This lets us avoid complications in the invariant needed to handle the first few iterations of \( c_2, c_3 \) (where \( z > z' \) does not hold under precondition \( x = x' \)). Whereas the originals maintain the invariants \( x! = z \times y! \) (for \( c_2 \)) and \( 2^x = z \times 2^y \) (for \( c_3 \)) and \( y \geq 0 \) (for both), the variations maintain \( x! = 4! = z \times y! \) and \( 2^x \times 2^4 = z \times 2^y \) and \( y \geq 4 \), owing to initializations of \( z = 4! = 24 \) and \( z = 2^4 = 16 \). We shall prove \( x = x' \land x > 3 \Rightarrow z > z' \) for \( c_4; \text{skip}^0 \) and \( c_5; \text{skip}^0 \). We use the product construction of this section, for alignment conditions mentioned earlier: \( \mathcal{L} \) is \( \{ w \% 2 \neq 0 \} \) and \( \mathcal{R} \) is \( \{ w' \% 3 \neq 0 \} \). So the transition on edge \((4, 4, lck)\)→\((5, 4, lo)\) is guarded by \( \{ w \% 2 \neq 0 \} \), the transition \((4, 4, lck)\)→\((4, 5, ro)\) is guarded by \( \{ w' \% 2 \neq 0 \} \), and \((4, 4, lck)\)→\((5, 5, lck)\) is guarded by \( \neg \mathcal{L} \land \neg \mathcal{R} \). As loop invariant we choose
\[ S : \quad y = y' \land y > 3 \land z > z' > 0 \]
Define the annotation \( an \) as follows.

\[
\begin{array}{ll}
(1, 1, lck) & an(n,m,tag) \quad x = x' \land x > 3 \quad \text{precondition} \\
(2, 2, lck) & \quad y = y' \land y > 3 \\
(3, 3, lck) & \quad y = y' \land y > 3 \land z > z' > 0 \\
(4, 4, lck) & \quad S \\
(5, 5, lck) & \quad S \land y > 4 \land \neg \mathcal{L} \land \neg \mathcal{R} \\
(6, 6, lck) & \quad S \land y > 4 \land w \% 2 = 0 \iff w' \% 3 \\
(7, 7, lck) & \quad S \land y > 4 \land w \% 2 = 0 \iff w' \% 3 \\
(8, 8, lck) & \quad S \land y > 4 \land w \% 2 \neq 0 \iff w' \% 3 \\
(9, 9, lck) & \quad S \\
(10, 5, lck) & \quad S \land \mathcal{L} \\
(10, 4, lck) & \quad S \\
(11, 5, lck) & \quad S \\
(12, 4, lck) & \quad S \\
& \quad S \\
(13, 5, ro) & \quad S \\
(14, 4, ro) & \quad S \\
& \quad S \\
(15, 9, ro) & \quad S \\
(0, 0, lck) & \quad z > z' \quad \text{postcondition}
\end{array}
\]

Let \( an(m,n,tag) = \text{false} \) for all others. The automaton never reaches \((6, 8, lck)\) or \((8, 6, lck)\), owing to the lockstep conditions. It never reaches \((6, 4, lo)\) or \((7, 4, lo)\) because \( \mathcal{L} \) contradicts the if-test; likewise \((4, 6, ro)\), \((4, 7, ro)\), and \( \mathcal{R} \). The annotation is valid.

VIII. Discussion

We introduced a notion of completeness relative to a designated class of alignments, and showed alignment completeness for four illustrative sets of RHL rules. In passing, we defined and proved Floyd completeness for HL. For simplicity we used the basic semantics of specs. Non-stuck semantics is preferable for richer languages and the requisite adjustments to HL and RHL rules are straightforward and well known (e.g., the precondition for assignment includes a definedness condition). We highlighted what adequacy means for non-stuck semantics, but refrained from spelling out alignment completeness results for it.

In this section we point out related work and directions for further development. For more extensive reviews of related work on RHLs, some starting points are [14]–[16] and [17]. Naumann [17] proposes the idea of alignment completeness but only in vague terms.

Product automata appear in many places (e.g., [12], [18], [19]) and are similar to control flow automata [20], [21]. Francez [12] observes that sequential product is complete relative to HL, but does not work that out in detail. Its completeness is featured in Barthe et al. [22], for the special case of relating a program to itself; it is clear that it holds generally as noted in [23]. Beringer [11] proves semantic completeness of sequential product and leverages it to derive rules including two conditionally aligned loop rules. The variant cawhile is featured in Banerjee et al. [24], [25]; the latter uses a form of dovetail product for non-stuck semantics. A RHL for deterministic programs is proved complete based on sequential product by Barthe et al. [26]. Sousa and Dillig [27] give a rule like seqprod for \( k \)-products; their Theorem 2 is completeness relative to completeness of an underlying HL. Wang et al. [28] prove a similar result specialized to program equivalence. The basis of these results is that the product under consideration is adequate, to use our term. Francez gives an adequacy result of this sort, for eager-lockstep, and Eilers et al. [29] do the same for a more general form of \( k \)-product that uses eager-lockstep for loops. Aguirre et al. [30] prove completeness of a RHL for higher order functional programs, via embedding in a unary logic.

By contrast with these Cook-style completeness results, alignment completeness is with respect to a designated class of alignments. Our results are for classes of alignments defined in terms of a limited class of product automata without auxiliary store, although ghost variables can be used in the relations \( \mathcal{L} \) and \( \mathcal{R} \) of rule cawhile. The most general notion of alignment would be a function from pairs of traces to alignments thereof [31]. Even restricted to computable functions, this is
far beyond the scope of known RHL rules, even using mixed-
structure rules like this one adapted from \cite{12}.

\[
\begin{align*}
\text{while } e \land e_0 \text{ do } b \text{ od } & | c : P \Rightarrow Q \\
\text{while } e \text{ do } b \text{ od } & | d : Q \Rightarrow R \\
\text{while } e \text{ do } b \text{ od } & | c; d : P \Rightarrow R
\end{align*}
\]

Several practical works use more sophisticated product con-
structions \cite{19,32}, including some that use auxiliary store \cite{33,34}. Clochard et al. \cite{34} highlight the efficacy of (unary) deductive verification applied to product programs, as well as going beyond \( \forall \forall \) (as do \cite{18}). Bringing more sophisticated alignments within the purview of RHL could have the familiar benefits of HL, like bringing the principles of conjunctive and disjunctive decomposition together with the IAM. However, merely collecting a large number of mixed-structure and data-conditioned rules would be inelegant and likely fall short of covering all useful alignments. An alternative is to leverage HL in the way \( \text{SEQ} \text{PROD} \) does, but for a wider range of product encodings. This might be achieved using a subsidiary judgment that connects two commands with a third that is an adequate product, as explored in \cite{15,25}, and taking into account the encoding of two stores by one for different data models \cite{22,35,36}.

Another subsidiary judgment that broadens the applicability of basic RHL rules is correctness-preserving rewriting, which is common in verification tools but is seldom made explicit in HLs. The RHL of \cite{24} includes an unconditional rewriting relation \( \cong \) and rule to infer \( c' : R \Rightarrow S \) from \( d'd' : R \Rightarrow S \) if \( c \cong d \) and \( c' \cong d' \). Using simple equivalences like if \( E \) then \( C \) else \( S \); while \( E \) do \( C \) od \( \cong \) while \( E \) do \( C \) od, together with a version of \text{cAWHILE}, they prove a loop tiling transformation which had been used to argue for working directly with automata \cite{18}. Unrolling examples \( c \text{ and } c \) a few times would address the problem we dodged by using \( c \text{ and } c \). Equivalences valid in Kleene algebra with tests \cite{57} are a natural candidate to connect with some class of product automata.

Alignment completeness characterizes sets of rules in terms of classes of alignments. If different classes could be defined using combinators for product automata, one might obtain a more uniform and comprehensive theory leading to more systematic development of relational verification tools.

Acknowledgments

Thanks to Anindya Banerjee and anonymous reviewers for helpful suggestions. The authors were partially supported by NSF award CNS 1718713 and the second author was partially supported by ONR N00014-17-1-2787.

REFERENCES

\begin{enumerate}
\item[1] N. Benton, “Simple relational correctness proofs for static analyses and program transformations,” in \textit{POPL}, 2004.
\item[2] T. Terauchi and A. Aiken, “Secure information flow as a safety problem,” in \textit{Static Analysis Symposium (SAS)}, 2005.
\item[3] S. A. Cook, “Soundness and completeness of an axiom system for program verification,” \textit{SIAM J. Comput.}, vol. 7, no. 1, 1974.
\item[4] K. R. Apt, F. S. de Boer, and E.-R. Olderog, \textit{Verification of Sequential and Concurrent Programs}, 3rd ed. Springer, 2009.
\item[5] T. Nipkow, “Hoare logics for recursive procedures and unbounded nondeterminism,” in \textit{Computer Science Logic}, 2002.
\item[6] D. Gorla and O. Strichman, “Regression verification,” in \textit{46th ACM Design Automation Conference}, 2009.
\item[7] R. Floyd, “Assigning meaning to programs,” in \textit{Symposium on Applied Mathematics 19, Mathematical Aspects of Computer Science}, 1967.
\item[8] A. Turing, “On checking a large routine,” in \textit{Report of a Conference on High Speed Automatic Calculating Machines}, 1949.
\item[9] K. R. Apt and E. Olderog, “Fifty years of Hoare’s logic,” \textit{Formal Asp. Comput.}, vol. 31, no. 6, 2019.
\item[10] S. Owicki and D. Gries, “An axiomatic proof technique for parallel programs,” \textit{Acta Inf.}, vol. 6, 1976.
\item[11] L. Beringer, “Relational decomposition,” in \textit{Interactive Theorem Proving (ITP)}, 2011.
\item[12] N. Francez, “Product properties and their direct verification,” \textit{Acta Informatica}, vol. 20, 1983.
\item[13] H. Yang, “Relational separation logic,” \textit{Theo. Comp. Sci.}, vol. 375, 2007.
\item[14] B. Beckert and M. Ulbrich, “Trends in relational program verification,” in \textit{Principled Software Development}, 2018.
\item[15] G. Barthe, J. M. Crespo, and C. Kunz, “Product programs and relational program logics,” J. Logical and Algebraic Methods in Programming, vol. 85, no. 5, 2016.
\item[16] K. Maillard, C. Hritcu, E. Rivas, and A. V. Muylder, “The next 700 relational program logics,” \textit{Proc. ACM Program. Lang.}, vol. 4, no. POPL, 2020.
\item[17] D. A. Naumann, “Thirty-seven years of relational Hoare logic: remarks on its principles and history,” in \textit{ISOLA}, 2020, extended version at https://arxiv.org/abs/2007.06421.
\item[18] G. Barthe, J. M. Crespo, and C. Kunz, “Beyond 2-safety: Asymmetric product programs for relational program verification,” in \textit{LICS}, 2013.
\item[19] B. R. Churchill, O. Padon, R. Sharma, and A. Aiken, “Semantic program alignment for equivalence checking,” in \textit{PLDI}, 2019.
\item[20] M. Heizmann, J. Hoenicke, and A. Podelski, “Software model checking for people who love automata,” in \textit{CAV}, 2013.
\item[21] T. Lange, M. R. Neuhäuser, and T. Noll, “IC3 software model checking on control flow automata,” in \textit{FMCAD}, 2015.
\item[22] G. Barthe, P. R. D’Argenio, and T. Rezk, “Secure information flow by self-composition,” in \textit{IEEE CSFW}, 2016, see extended version \cite{58}.
\item[23] G. Barthe, J. M. Crespo, and C. Kunz, “Relational verification using product programs,” in \textit{Formal Methods}, 2011.
\item[24] A. Banerjee, D. A. Naumann, and M. Nikouei, “Relational logic with framing and hypotheses,” in \textit{FSTTCS}, 2016, technical report at https://arxiv.org/abs/1611.08892.
\item[25] A. Banerjee, R. Nagasamudram, M. Nikouei, and D. A. Naumann, “A relational program logic program with data abstraction and dynamic framing,” 2019, available at https://arxiv.org/abs/1910.14560.
\item[26] G. Barthe, B. Grégoire, J. Hsu, and P. Strub, “Coupling proofs are probabilistic program products,” in \textit{POPL}, 2017.
\item[27] M. Sousa and I. Dillig, “Cartesian Hoare Logic for verifying k-safety properties,” in \textit{PLDI}, 2016.
\item[28] Y. Wang, I. Dillig, S. K. Lahiri, and W. R. Cook, “Verifying equivalence of database-driven applications,” \textit{Proc. ACM Program. Lang.}, vol. 2, no. POPL, 2018.
\item[29] M. Eilers, P. Müller, and S. Hitz, “Modular product programs,” \textit{ACM Trans. Program. Lang. Syst.}, vol. 42, no. 1, 2020.
\item[30] A. Aguirre, G. Barthe, M. Gaborud, D. Garg, and P. Strub, “A relational logic for higher-order programs,” \textit{J. Funct. Program.}, vol. 29, 2019.
\item[31] M. Kovács, H. Seidl, and B. Finkbeiner, “Relational abstract interpretation for the verification of 2-hypersafety properties,” in \textit{CCS}, 2013.
\item[32] R. Shenmer, A. Gurfinkel, S. Shoham, and Y. Vizel, “Property directed self composition,” in \textit{CAV}, 2019.
\item[33] T. Girka, D. Mentré, and Y. Régis-Gianas, “Verifiable semantic difference languages,” in \textit{PPDP}, 2017.
\item[34] M. Clochard, C. Marché, and A. Paskevich, “Deductive verification with ghost monitors,” \textit{Proc. ACM Program. Lang.}, vol. 4, no. POPL, 2020.
\item[35] D. A. Naumann, “From coupling relations to mates invariants for secure information flow,” in \textit{ESORICS}, 2006.
\item[36] L. Beringer and M. Hofmann, “Secure information flow and program logics,” in \textit{IEEE CSF}, 2007.
\item[37] D. Kozen, “On Hoare logic and Kleene algebra with tests,” \textit{ACM Trans. Comput. Log.}, vol. 1, no. 1, 2000.
\item[38] G. Barthe, P. R. D’Argenio, and T. Rezk, “Secure information flow by self-composition,” \textit{Math. Struct. Comput. Sci.}, vol. 21, no. 6, 2011.
\end{enumerate}
Definitions

Definition of \(\text{destutter}(\tau)\) for trace \(\tau\):

\[
\text{destutter}(\tau) = \tau \text{ if } lcn(\tau) = 1
\]

\[
\text{destutter}(\gamma_0 :: \gamma_1 :: \tau) = \text{destutter}(\gamma_1 :: \tau) \text{ if } \gamma_0 = \gamma_1
\]

\[
\text{destutter}(\gamma_0 :: \gamma_1 :: \tau) = \gamma_0 :: \text{destutter}(\gamma_1 :: \tau) \text{ otherwise}
\]

Definition of \(\text{lab}(c)\):

\[
\text{lab}(\text{skip}^n) = n
\]

\[
\text{lab}(x := e) = n
\]

\[
\text{lab}(c; d) = \text{lab}(c) \text{ if } m \in \text{labs}(c)
\]

\[
\text{lab}(\text{if}^n \ e \text{ then } c \text{ else } d \ fi) = n
\]

\[
\text{lab}(\text{while}^n \ e \text{ do } c \ od) = n
\]

\[
\text{lab}(c \cup^m d) = n
\]

Definition of \(\text{labs}(c)\):

\[
\text{labs}(\text{skip}^n) = \{n\}
\]

\[
\text{labs}(x := e) = \{n\}
\]

\[
\text{labs}(c; d) = \text{labs}(c) \cup \text{labs}(d)
\]

\[
\text{labs}(\text{if}^n \ e \text{ then } c \text{ else } d \ fi) = \{n\} \cup \text{labs}(c) \cup \text{labs}(d)
\]

\[
\text{labs}(\text{while}^n \ e \text{ do } c \ od) = \{n\} \cup \text{labs}(c)
\]

\[
\text{labs}(c \cup^m d) = \{n\} \cup \text{labs}(c) \cup \text{labs}(d)
\]

Appendix

Fig. 13 defines the \(\text{ok}\) condition.

Fig. 14 defines \(\text{sub}(m, c, \text{eod})\).

As an example how labels are manipulated in the transition semantics, here is an \(\text{ok}\) command and its transitions. We omit stores but assume \(x = 1\) initially.

\[
\begin{align*}
\text{while}^1 x > 0 \text{ do } x := x - 1 \text{ od}; & \text{skip}^3 \\
\rightarrow x := x - 1 \text{ ; while}^1 x > 0 \text{ do } x := x - 1 \text{ od}; & \text{skip}^3 \\
\rightarrow \text{skip}^2 \text{ ; while}^1 x > 0 \text{ do } x := x - 1 \text{ od}; & \text{skip}^3 \\
\rightarrow \text{while}^1 x > 0 \text{ do } x := x - 1 \text{ od}; & \text{skip}^3 \\
\rightarrow \text{skip}^1 \text{ ; skip}^3 & \rightarrow \text{skip}^1
\end{align*}
\]

Apropos the Floyd completeness Thm. 8 one can dispense with all forms of associated judgment except 8 by reformulating some of the HL rules to more directly match the judgments of a valid annotation so rule 

\[
\text{Cons} \Rightarrow \text{Ass}
\]

so by induction we have \(\vdash d : \text{ann}(\text{lab}(d)) \Rightarrow \text{ann}(an)\).

Lemma 8 gives \(\text{ann}(a) \wedge e \Rightarrow \text{ann}(\text{lab}(d))\) so by 

\[
\text{Cons} \Rightarrow \text{Ass}
\]

we get \(\vdash d : \text{ann}(a) \wedge e \Rightarrow \text{ann}(an)\). Rule 

\[
\text{WH}
\]

yields \(\vdash b : \text{ann}(a) \Rightarrow \text{ann}(\text{lab}(b, c, \text{fin}))\) so by Cons we get \(\vdash b : \text{ann}(a) \Rightarrow \text{ann}(\text{lab}(b, c, \text{fin}))\).
• If $b$ is $d_0 \sqcup^n d_1$, by induction we have
  \[ \vdash d_0 : \text{an}((\text{lab}(d_0) \sim \text{an}(\text{lab}(d_0, c, \text{fin}))) \quad \text{and} \]
  \[ \vdash d_1 : \text{an}((\text{lab}(d_1) \sim \text{an}(\text{lab}(d_1, c, \text{fin}))) \].
  Lemma gives $\text{an}(n) \Rightarrow \text{an}(\text{lab}(d_0))$ and $\text{and}(n) \Rightarrow \text{an}(\text{lab}(d_1))$ so by Coseq we get
  \[ \vdash d : \text{an}(n) \sim \text{an}(\text{lab}(d_0, c, \text{fin})) \]
  \[ \vdash d_1 : \text{an}(n) \sim \text{an}(\text{lab}(d_1, c, \text{fin})). \]
  By definitions we have \( \text{lab}(b, c, \text{fin}) = \text{lab}(d_0, c, \text{fin}) = \text{lab}(d_1, c, \text{fin}). \) So by rule Choice we get
  \[ \vdash b : \text{an}(n) \sim \text{an}(\text{lab}(b, c, \text{fin})). \]

Proof of Theorem

For basic semantics this is left to the reader. We show that if $\Pi_{A,A'}$ is a strongly $R$-adequate product then, in non-stuck semantics, $\Pi_{A,A'} \models R \sim S$ implies $A|A' \models R \approx S$, and (ii) all finite and initial traces of $A$ and $A'$ are not stuck. Let $\tau, \tau'$ be finite initial traces of $A, A'$ such that $\tau_0, \tau'_0 \models R$.

(i) Assume $\tau$ and $\tau'$ are terminated, therefore $\text{ctrl}(\tau_{-1}) = \text{fin}$ and $\text{ctrl}(\tau'_{-1}) = \text{fin}'. $ Since $\Pi_{A,A'}$ is $R$-adequate (implied by strong $R$-adequacy), we can appeal to the result for basic semantics to conclude $\tau_{-1}, \tau'_{-1} \models S$. Hence, $\tau, \tau' \models R \approx S$ and we are done.

(ii) Assume $\tau$ is not terminated, i.e., $\text{ctrl}(\tau_{-1}) \neq \text{fin}$. Since $\Pi_{A,A'}$ is strongly $R$-adequate we can obtain a trace $T$ of $\Pi_{A,A'}$ such that $\tau \leq \text{left}(T)$. Now, there are two cases to consider. If $\tau < \text{left}(T)$, then the witness for $\tau_{-1} \mapsto -$ is the successor of $\tau_{-1}$ in $\text{left}(T)$. Otherwise, $\tau = \text{left}(T)$. Since $\Pi_{A,A'}$ satisfies $R \sim S$ in the non-stuck semantics of unary specs, and is not terminated (because $\text{ctrl}(\text{left}(T_{-1})) = \text{ctrl}(\tau_{-1}))$ we know $T \mapsto -$. This argument can be iterated, extending $T$, and after finitely many steps it must cover a left-step from $\tau_{-1}$ because otherwise it would contradict the one-side divisibility condition of strong $R$-adequacy of $\Pi_{A,A'}$. The argument for $\tau'$ is similar, so we are done.

Although for basic semantics the theorem is an equivalence, for non-stuck semantics the converse does not hold. For a counterexample, consider the interleaved product (which is strongly $R$-adequate) of $A, A'$ and augment its control points to include a tag from $\{0, 1\}$ (like the product in Sec. [VI]), with tag value 1 for the initial and final points. The transition relation is the same as of the interleaved product, except it is not enabled for control points with tag 0. Further, any step of the product may change the value of the tag nondeterministically. Let $\Pi$ be such a product. It is strongly $R$-adequate. But suppose $A|A' \models R \approx S$, so there are no stuck non-final traces of $A$ or $A'$. We do not have $\Pi \models R \sim S$. If $T$ is a finite initial trace of $\Pi$ such that $T_0 \models R$ and $T_{-1}$ is not final, we must show $T \mapsto -$. Clearly, this cannot be true if the tag of $\text{ctrl}(T_{-1})$ is 0; in this case, $T$ is stuck and $T_{-1}$ has no successor.

Proof of Lemma

By mutual implication. For traces $\tau, \nu$ with $\nu_0 = \tau_{-1}$ we write $\tau \circ \nu$ for the coalesced catenation, i.e., $\tau$ followed by all but the first element of $\nu$.

Assume $\models c; d : R \approx S$. To show $\models c|d : R \approx S$, consider terminated traces $\tau$ of $c$ and $\nu$ of $d$ (over Var). Suppose $\tau_0, \nu_0 \models R$, to show $\tau_{-1}, \nu_{-1} \models S$. By renaming we get a trace $\nu'$ of $d'$. Let $s = \text{stor}(\nu_0) \circ \text{dot}. $ By adding $s$ to the stores of $\nu$ we get a trace $\tau'$ of $d'$ that leaves the $\text{Var}^*$ part uncharged and ends ready to execute $d'$. Adding $\text{stor}(\tau_{-1})$ to the stores of $\nu'$ we get a trace $\tau''$ of $d'$ that leaves the $\text{Var}^*$ part uncharged.

So $\tau \circ \nu''$ is a terminated trace of $c; d'$ that ends with store $\text{stor}(\tau_{-1}) + \text{stor}(\nu_{-1}).$ We have $\tau_0, \nu_0 \models R$ iff $\tau_0 \cup s \models R$, whence $(\tau \circ \nu'')_0 \models R$. So from $\models c; d' : R \approx S$ we get $(\tau \circ \nu'')_{-1} \models S$, that is, $\text{stor}(\tau_{-1}) + \text{stor}(\nu_{-1}) \models S$. The latter is equivalent to $\text{stor}(\tau_{-1}), \text{stor}(\nu_{-1}) \models S$ which completes the proof that $\tau, \nu \models R \approx S$.

For the converse, assume $\models c|d : R \approx S.$ To show $\models c; d' : R \approx S$, consider any terminated trace of $c; d'$, which by semantics can be written as coalesced catenation $\tau \circ \nu$ of nonempty traces of $c$ and $d'$, with the $\text{Var}^*$-part of each $\tau_{-1} \equiv \text{to that of } \tau_0, \nu_0 \text{ and the } \text{Var}^*-\text{part of each } \nu_1 \equiv \text{to that of } \nu_{-1}$. By discarding the untouched parts, and renaming, we get terminated Var-traces $\tilde{\tau}$ of $c$ and $\tilde{\nu}$ of $d$. Moreover if $\tau_0 \models R$ then $\tilde{\tau}, \tilde{\nu}_0 \models R$ so using $\models c|d : R \approx S$ we get $\tilde{\tau}, \tilde{\nu}_{-1} \models S$. The latter is equivalent to $\nu_{-1} \models S$ whence $(\tau \circ \nu) \models R \approx S$.

Proof of Theorem

As with the previous theorems, the associated judgments are easily determined by inspecting the proof of the theorem so we refrain from spelling them out.

The proof relies on an analysis of the VCs. These include VCs for the lockstep part of the automaton, like those in Fig. [11] as well as VCs for the one-sided part of the automaton like those in Figs. [9] and we omit them. We use HL to obtain proofs for $b$ and $\text{dot}(b')$, then lift those using $\text{Seq}, \text{Prod}$, and complete the derivation of $\vdash c|c' : R \approx S$ using rules in Fig. [10] Here are more details.

By induction on $b$, using an argument similar to that proving [5], using left-side VCs, we have for all subcommands $d$ of $b$:

\[ \vdash d : \text{an}(\text{lab}(d), \text{beg}, \text{lo}) \sim \text{an}(\text{lab}(d, c), \text{beg}, \text{tag}) \]  \tag{8}

where $\text{tag} = \text{ro}$ if $m = \text{end}$, else $\text{tag} = \text{lo}$. (The different cases for $\text{tag}$ arise from the form of the CFG, in particular transitions of type (iii) versus (iv).) Then in particular the instantiation for $b$ itself is $\vdash b : \text{an}(\text{beg}, \text{les}, \text{lo}) \sim \text{an}(\text{end}, \text{beg}, \text{ro})$ with ending tag $\text{ro}$ because the paths through $b$ end with a transition of type (iv). Next, by induction on $b'$, a similar argument using right-side VCs shows for all subcommands $d'$ of $b'$:

\[ \vdash \text{dot}(d') : \text{an}(\text{end}, \text{lab}(d'), \text{ro}) \sim \text{an}(\text{end}, m, \text{tag}) \]  \tag{9}
where $m = \text{etab}(d', c)$ and $\text{tag} = \text{lck}$ if $m = \text{end}$ else $\text{tag} = \text{ro}$. Instantiating (8) and (9) we get

\[ \vdash b : \text{an}(\text{beg}, \text{beg}, \text{lo})^\ast \leadsto \text{an}(\text{end}, \text{beg}, \text{ro})^\ast \]

\[ \vdash \text{dot}(b') : \text{an}(\text{end}, \text{beg}, \text{ro})^\ast \leadsto \text{an}(\text{end}, \text{end}, \text{lck})^\ast \]

hence \[ \vdash b; \text{dot}(b') : \text{an}(\text{beg}, \text{beg}, \text{lo})^\ast \leadsto \text{an}(\text{end}, \text{end}, \text{lck})^\ast \]

by $\text{SEQ}$. So rule $\text{SEQPROD}$ yields

\[ \vdash b|b' : \text{an}(\text{beg}, \text{beg}, \text{lo}) \Rightarrow \text{an}(\text{end}, \text{end}, \text{lck}) \]  

(10)

Now show, for all subcommands $d$ of $c$ and corresponding subcommands $d'$ of $c'$, with $d$ not a proper subcommand of $b$ (so $\text{lab}(d') = \text{lab}(d)$ and $d'$ is not a proper subcommand of $b'$):

\[ \vdash d \mid d' : \text{an}(\text{lab}(d), \text{lab}(d), \text{tag}) \Rightarrow \text{an}(m, m, \text{tag}o) \]  

(11)

where $m = \text{etab}(d, c, \text{fin}) = \text{etab}(d', c, \text{fin})$ and $\text{tag} = \text{lo}$ if $\text{lab}(d) = \text{beg}$ (i.e., $d$ is $b$), and $\text{tag} = \text{lck}$ otherwise; and $\text{tag}o = \text{lo}$ if $m = \text{beg}$, else $\text{tag}o = \text{lck}$. (Here the tag conditions are due to transitions (i) versus (ii); noting that $\text{lab}(d) = \text{beg}$ implies $m \neq \text{beg}$.) The proof is by induction on structure of $c$ (which is structurally the same as $c'$). The base cases $b|b'$, pairs of assignments outside $b, b'$, and pairs of skip. For $b|b'$, (11) is just (10). For assignments and skip, we use the VCs just as in the proof of Theorem 11. The induction step to prove (11) is same-structure command relations. By analysis of the lockstep transitions of the product we get VCs like in Fig. 11 and using the assumption that the assertion's assertions imply test agreement on branches (outside $b$ and $b'$), we can show the rules in Fig. 10 suffice to prove what's needed.

Finally, instantiate $d, d'$ as $c, c'$ in (11), noting that $\text{lab}(d) = 1 \neq \text{beg}$ so $\text{tag} = \text{lck}$. So Theorem 12 is proved, since $\text{an}(1, 1, \text{lck}) = \mathbb{R}$ and $\text{an}(\text{fin}, \text{fin}, \text{lck}) = \mathbb{S}$.

To handle the general case where there are multiple designated subprograms, disjoint from each other, one can define the product in terms of a set $\text{BEG}$ of begin labels and $\text{END}$ of end labels (disjoint from $\text{BEG}$), replacing condition $m \neq \text{beg}$ with $n \notin \text{BEG}$ and adjusting the other transition rules accordingly.

To avoid the case distinctions on tags in the induction hypotheses (9) and (9), one could relax the definition of product automata to allow a product to take transitions with no effect on the underlying computation. In the present case we would use one from $(\text{beg}, \text{beg}, \text{lck})$ to $(\text{beg}, \text{beg}, \text{lo})$ and so forth. For adequacy, such steps must be bounded. This is akin to the familiar requirement that ghost code must be terminating [2].

**Exercise following Theorem 12**

Consider the situation assumed in Thm. 12 i.e., \text{sameExcept}(c, c', b, b', \text{beg}, \text{end}, \text{fin}), and suppose in addition that the distinguished subprograms $b$ and $b'$ are both choices, say \[ b_0 \sqcup b_1 \text{ and } b_0' \sqcup b_1' \]. The product used in Thm. 12 essentially executes $(b_0 \sqcup b_1); (b_0' \sqcup b_1')$ (ignoring the dot encoding, for clarity). A proof constructed in accord with the theorem will use $\text{SEQPROD}$ with a single relation $\mathbb{Q}$ at the semicolon. Then using the HL rule for choice the judgments involved will be $b_0 : \mathbb{R} \leadsto \mathbb{Q}, b_1 : \mathbb{R} \leadsto \mathbb{Q}, b_0' : \mathbb{Q} \leadsto \mathbb{S}, \text{ and } b_1' : \mathbb{Q} \leadsto \mathbb{S}$. The exercise is to modify the product to handle this situation by using a four-way nondeterministic choice between sequentially executing $b_0; b_0'; b_1; b_1'$, or $b_1; b_1'$. This corresponds to the relational four-precise rule for relating choices:

\[
\begin{array}{c}
\vdash \text{c} \mid \text{c'} : \mathbb{R} \leadsto \mathbb{S} \\
\text{d} \mid \text{d'} : \mathbb{R} \leadsto \mathbb{S} \\
\text{d} \mid \text{d'} : \mathbb{R} \leadsto \mathbb{S} \\
\end{array}
\]

(11)

(which is similar to the four-rule way relating if-else commands in some RHLs). Then $\text{SEQPROD}$ will be used for the judgments $b_0; b_0' : \mathbb{R} \leadsto \mathbb{S}, b_0; b_1' : \mathbb{R} \leadsto \mathbb{S}, b_1; b_0' : \mathbb{R} \leadsto \mathbb{S}, \text{ and } b_1; b_1' : \mathbb{R} \leadsto \mathbb{S}$. For these, we may choose four different intermediate assertions, by contrast with a single $\mathbb{Q}$ that works for all four cases.

**Theorem 13** proof and automaton

Here is the automaton for the conditionally aligned loop product, under the conditions given in Sec. VII. Note that $\text{labs}(\text{sub}(\text{beg}, c)) = \text{labs}(\text{sub}(\text{beg}, c'))$ under those conditions.

(i) $(n, n, \text{lk}, (s, s')) \Rightarrow ((m, m, \text{lk}), (t, t'))$ and moreover if $n = \text{beg}$ then $s, s' \notin \mathbb{L}$ and $s, s' \notin \mathbb{R}$.

(ii) $(\text{beg}, \text{beg}, \text{lk}, (s, s')) \Rightarrow ((m, \text{beg}, \text{lo}), (s, s'))$.

(iii) $(\text{beg}, \text{beg}, \text{lk}, (s, s')) \Rightarrow ((m, \text{beg}, \text{lo}), (s, s'))$.

(iv) $(n, \text{beg}, \text{lo}, (s, s')) \Rightarrow ((m, \text{beg}, \text{lo}), (t, t'))$.

(v) $(\text{beg}, \text{beg}, \text{lk}, (s, s')) \Rightarrow ((m, \text{beg}, \text{lo}), (s, s'))$.

(vi) $(n, \text{beg}, \text{lo}, (s, s')) \Rightarrow ((m, \text{beg}, \text{lo}), (t, t'))$.

(vii) $(\text{beg}, \text{beg}, \text{lk}, (s, s')) \Rightarrow ((m, \text{beg}, \text{lo}), (s, s'))$.

(viii) $(n, \text{beg}, \text{lo}, (s, s')) \Rightarrow ((m, \text{beg}, \text{lo}), (t, t'))$.

Theorem 13 is proved as follows, with reference to the VCs in Fig. 10. The VCs reflect that only same-value-test paths are taken, except in $\text{lo}$ or $\text{ro}$ iterations under conditions $\mathbb{L}$ and $\mathbb{R}$ respectively. The omitted VCs include the symmetric right-only transitions and unreachable ones. As an aside, Fig. 17 instantiates some VCs for the $\text{e1}$, $\text{e5}$ example of Sec. VII.

Let $b$ be the body of the loop $\text{sub}(\text{beg}, c)$ and $b'$ the body of the loop $\text{sub}(\text{beg}, c')$ (it must be a loop, by $\text{sameCt}(c, c')$). Labels have no significance on the proof rules and we will use skip without a label in the following. One could as well choose fresh labels so as to work entirely with labelled commands.

By induction on $b$ we show its relation with skip: for all subcommands $d$ of $b$:

\[ \vdash \text{d} \mid \text{skip} : \text{an}(\text{etab}(d), \text{beg}, \text{lo}) \Rightarrow \text{an}(m, \text{beg}, \text{tag}) \]  

(12)

where $m = \text{etab}(d, c, \text{fin})$ and $\text{tag} = \text{lck}$ if $m = \text{beg}$, else $\text{tag} = \text{lo}$. For this purpose we use the left-side rules of Fig. 12 together with $\text{SEQ}$, in an argument similar to proofs of the previous theorems. For example, in case $d$ is an assignment $x := \text{n} \text{e}$ we use $\text{AssSkip}$ to get

\[ x := \text{n} \text{e} \mid \text{skip} : \text{an}(m, \text{beg}, \text{tag}) \]  

(13)
and then use $\text{rConseq}$ and the VC to strengthen $\text{an}(m, \text{beg}, \text{tag})$ to $\text{an}(n, \text{beg}, \text{tag})$.

The relevant VCs arise from type (ii), (iv), and (vi) transitions (see Fig. 16). This is similar to the proof of Theorem 10 in the sense that the VCs are for one-side execution, but there we work with unary judgments and here we are using relational judgments (and relations are sets of store pairs, not sets of stores).

By induction on $b'$ we show its relation with skip: for all subcommands $d'$ of $b'$:

$\vdash \text{skip} \mid d' : \text{an}(\text{beg}, \text{lab}(d'), \text{ro}) \approx \text{an}(\text{beg}, m, \text{tag})$ (13)

where $m = \text{lab}(d', c, \text{fin})$ and $\text{tag} = \text{lck}$ if $m = \text{beg}$, else $\text{tag} = \text{ro}$. For this purpose we use the right-side rules which mirror those shown in Fig. 12. The relevant VCs arise from type (iii), (v), and (vii) transitions.

By induction on $b$ we show its relation with $b'$: for all subcommands $d$ of $b$ and corresponding subcommands $d'$ of $b'$:

$\vdash d \mid d' : \text{an}(\text{lab}(d), \text{lab}(d), \text{lck}) \approx \text{an}(m, m, \text{lck})$ (14)

where $m = \text{lab}(d, c, \text{fin})$. For this purpose we use the lockstep rules of Fig. 10 similar to the proof of Theorem 11 and relying on assumption (a) of Theorem 13.

Instantiating (12), (13), and (14), noting that $\text{lab}(b, c, \text{fin}) = \text{beg}$, we get

$\vdash b \mid \text{skip} : \text{an}(\text{lab}(b), \text{beg}, \text{lo}) \approx \text{an}(\text{beg}, \text{beg}, \text{lck})$

$\vdash \text{skip} \mid b' : \text{an}(\text{lab}(b), \text{ro}) \approx \text{an}(\text{beg}, \text{beg}, \text{lck})$

$\vdash b \mid b' : \text{an}(\text{lab}(b), \text{lab}(b), \text{lck}) \approx \text{an}(\text{beg}, \text{beg}, \text{lck})$

Now using $\text{rConseq}$ and the VCs we get

$\vdash b \mid \text{skip} : \text{an}(\text{beg}, \text{beg}, \text{lck}) \land \{e\} \land \text{L} \Rightarrow \text{an}(\text{beg}, \text{beg}, \text{lck})$

$\vdash \text{skip} \mid b' : \text{an}(\text{beg}, \text{beg}, \text{lck}) \land \{e'\} \land \text{R} \Rightarrow \text{an}(\text{beg}, \text{beg}, \text{lck})$

$\vdash b \mid b' : \text{an}(\text{beg}, \text{beg}, \text{lck}) \land \{e\} \land \{e'\} \land \text{L} \land \text{L} \Rightarrow \text{an}(\text{beg}, \text{beg}, \text{lck})$

With these, using assumption (b) of the theorem, we use rule $\text{cWhile}$ to get this judgment for the loop itself:

$\text{sub}(\text{beg}, c) \mid \text{sub}(\text{beg}, c') : \text{an}(\text{beg}, \text{beg}, \text{lck}) \approx \text{an}(\text{beg}, \text{beg}, \text{lck}) \land \neg\{e\} \land \neg\{e'\}$ (15)

where $m = \text{lab}(\text{sub}(\text{beg}, c), c, \text{fin})$.

Now by induction on $c$ we show for all subcommands $d$ of $c$ and corresponding subcommands $d'$ of $c'$ such that $d$ is not a proper subcommand of the distinguished loop sub($\text{beg}, c$) (nor $d'$ of sub($\text{beg}, c'$)):

$\vdash d \mid d' : \text{an}(\text{lab}(d), \text{lab}(d), \text{lck}) \Rightarrow \text{an}(m, m, \text{lck})$ (16)

where $m = \text{lab}(d, c, \text{fin})$. As base cases we have (15) as well as the atomic commands outside the distinguished loop $b$. The induction steps are like those proving lockstep product Theorem 11.

Instantiating (16) with $c, c'$ we get

$\vdash c \mid c' : \text{an}(1, 1, \text{lck}) \Rightarrow \text{an}(\text{fin}, \text{fin}, \text{lck})$

As $\text{an}$ is an annotation for $\text{P} \Rightarrow \text{Q}$, this is $\vdash c \mid c' : \text{P} \Rightarrow \text{Q}$, q.e.d.

Rule $\text{cWhile}$ essentially subsumes $\text{dWh}$ of Fig. 10 in the presence of the rules of Fig. 12. To see why, suppose we have
the premise \( c \mid c' : Q \land \{ e \land \{ e' \} \} \Rightarrow Q \) and side condition \( Q \Rightarrow e \vdash e' \) of \( d\text{WH} \). Let \( L = \text{false} = R \). Then the side condition of \( c\text{WHILE} \), \( Q \Rightarrow e \vdash e' \lor (L \land \{ e \} ) \lor (R \land \{ e' \} ) \), is equivalent to \( Q \Rightarrow e \vdash e' \). We can obtain the premises \( c \mid \text{skip} : Q \land L \land \{ e \} \Rightarrow Q \) and \( c \mid \text{skip} : Q \land R \land \{ e' \} \Rightarrow Q \) of \( c\text{WHILE} \), using \( r\text{CONSEQ} \), from
\[
c \mid \text{skip} : \text{false} \Rightarrow Q \quad \text{and} \quad c \mid \text{skip} : \text{false} \Rightarrow Q
\]
and then use \( c\text{WHILE} \) to get the conclusion of \( d\text{WH} \). Obviously \( c \mid c' : \text{false} \Rightarrow Q \) is valid for any \( c, c', Q \). Technically, to make \( d\text{WH} \) a derived rule, we would need to add \( c \mid c' : \text{false} \Rightarrow Q \) as an axiom.

**Soundness of \( c\text{WHILE} \)**

Soundness of a rule similar to \( c\text{WHILE} \) is proved by Beringer [11] but since the rule is not widely known we sketch a proof here.

One way to prove soundness is by way of the conditionally aligned loop product. A product gives rise to a simple proof structure, namely induction on traces. For the sake of variety we sketch a different argument.

Suppose the premises hold:
\[
\begin{align*}
(i) & \quad \models c \mid c' : Q \land \{ e \land \{ e' \} \} \land \neg L \land \neg R \Rightarrow Q \\
(ii) & \quad \models c \mid \text{skip} : Q \land L \land \{ e \} \Rightarrow Q \\
(iii) & \quad \models c \mid \text{skip} : Q \land R \land \{ e' \} \Rightarrow Q
\end{align*}
\]

Suppose the side condition \( Q \Rightarrow e \vdash e' \lor (L \land \{ e \} ) \lor (R \land \{ e' \} ) \) is valid. To show validity of the conclusion, \( \models \text{while} e \text{ do } c \text{ od } \mid \text{false} \Rightarrow Q \), and \( \models \text{while} e' \text{ do } c' \text{ od } : Q \Rightarrow Q \land \neg \{ e \land \neg e' \} \). Consider any terminated traces \( \tau, \tau' \) of the left and right programs respectively, such that \( \tau_0, \tau'_0 \models Q \). We have \( \tau_1, \tau'_1 \models \neg \{ e \land \neg e' \} \) by program semantics. It remains to show \( \tau_{-1}, \tau'_{-1} \models Q \).

We prove the following claim: for any terminated traces \( \tau, \tau' \), if \( \tau_0, \tau'_0 \models Q \) then \( \tau_{-1}, \tau'_{-1} \models Q \). The proof is by induction on \( \text{iter}(\tau) + \text{iter}(\tau') \), where \( \text{iter}(\tau) \) is the number of iterations on the left, and likewise for \( \text{iter}(\tau') \) on the right.

The base case is \( \text{iter}(\tau) + \text{iter}(\tau') = 0 \) in which case \( \tau_0, \tau'_0 \) and we are done.

Otherwise, suppose \( \text{iter}(\tau) + \text{iter}(\tau') > 0 \). Then \( \tau(\{ e \} ) = 0 \) or \( \tau'(\{ e' \} ) = 0 \) since there is at least one iteration. So using the side condition \( Q \Rightarrow e = e' \lor (L \land \{ e \} ) \lor (R \land \{ e' \} ) \) and \( \tau_0, \tau'_0 \) we have that \( \tau_0, \tau'_0 \) satisfies \( \langle \{ e \land \{ e' \} \} \lor (L \land \{ e \} ) \lor (R \land \{ e' \} ) \rangle \). This is equivalent to
\[
\langle \{ e \land \{ e' \} \} \land \neg L \land \neg R \rangle \lor \langle L \land \{ e \} \lor (R \land \{ e' \} ) \rangle
\]

So it suffices to consider these three cases:

- \( \tau_0, \tau'_0 \models \{ e \land \{ e' \} \} \land \neg L \land \neg R \)
  - Both \( \tau \) and \( \tau' \) begin with an iteration; let \( \nu \) be the suffix of \( \tau \) after its first iteration and \( \nu' \) the suffix of \( \tau' \) after its first. By premise \( (i) \) we have \( \nu_0, \nu'_0 \models Q \). Now \( \text{iter}(\nu) + \text{iter}(\nu') = \text{iter}(\tau) + \text{iter}(\tau') - 2 \) so we can apply the induction hypothesis to \( \nu, \nu' \), which yields the result because \( \nu_{-1} = \tau_{-1} \) and \( \nu'_{-1} = \tau'_{-1} \).
- \( \tau_0, \tau'_0 \models L \land \{ e \} \)
  - So at least \( \tau \) has an iteration. Let \( \nu \) be the suffix of \( \tau \) after its first iteration. By premise \( (ii) \) we have \( \nu_0, \nu'_0 \models Q \).
- Now \( \text{iter}(\nu) + \text{iter}(\nu') = \text{iter}(\tau) + \text{iter}(\tau') - 1 \) so we can apply the induction hypothesis to \( \nu, \nu' \), which yields the result because \( \nu_{-1} = \tau_{-1} \) and \( \nu'_{-1} = \tau'_{-1} \).