Alignment Completeness for Relational Hoare Logics

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 c satisfies $P \rightsquigarrow Q$ if its terminating executions, from initial states that satisfy P, end in states that satisfy Q. To relate programs c and d, use binary relations $\mathcal{R}, \mathcal{S}$ over states: c and d satisfy $\mathcal{R} \approx\!> \mathcal{S}$ if pairs of their terminating executions, from $\mathcal{R}$-related initial states, end in $\mathcal{S}$-related states. For program equivalence, take $\mathcal{R}$ and $\mathcal{S}$ to be the identity on states.

Hoare logics (HL) provide sound rules to infer correctness judgments $c : P \rightsquigarrow Q$, conventionally written $\{P\}\, c\, \{Q\}$, for various programming languages. In this paper we confine attention to simple imperative commands and focus on Relational Hoare logics (RHL) which provide rules to infer judgments which we write as $c \mid d : \mathcal{R} \approx\!> \mathcal{S}$. Such properties go beyond program equivalence. Using primed variables to refer to the second of two related states, the judgment $c \mid d : x = x' \approx\!> y < y'$ 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, $c \mid c : x = x' \approx\!> y = y'$ 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 motived early work on RHL [1].

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 $x = x' \rightsquigarrow z = z'$. To prove the judgment $c0; c0' : x = x' \rightsquigarrow z = z'$ in HL we need the assertion $z = x! \wedge x = x'$ at the semicolon, to get $z = x! \wedge z' = x'! \wedge x = x'$ following $c0'$, from which $z = z'$ follows. The spec $x = x' \rightsquigarrow z = z'$ 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 (as famously observed in [2]).

There is a simple way to prove $c0 \mid c0' : x = x' \approx\!> z = z'$. Consider the execution pairs to be aligned step-by-step, and note that at each aligned pair of states we have $y = y' \wedge z = z'$, 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 $y := x \mid y' := x' : x = x' \approx\!> y = y'$ and $z := 1 \mid z' := 1 : y = y' \approx\!> (y = y' \wedge z = z')$ infer that $y := x; z := 1 \mid y' := x'; z' := 1 : (x = x') \approx\!> (y = y' \wedge z = z')$. 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 DSEQ 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. VIII.

The touchstone for Hoare logics is Cook's completeness theorem [3] which says any true correctness judgment $c : P \rightsquigarrow Q$ 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

978-1-6654-4895-6/21/$31.00

c0: $(*\ z := x!\ *)$ *y:= x; z:= 1;* **while** $y \neq 0$ **do** *z:= z∗y; y:= y−1* **od**
c1: $(*\ z := 2^x\ *)$ *y:= x; z:= 1;* **while** $y \neq 0$ **do** *z:= z∗2; y:= y−1* **od**
c2: $(*\ z := x!\ *)$ *y:= x; z:= 1; w:= 0;* **while** $y \neq 0$ **do if** $w\ \%\ 2 = 0$ **then** *z:= z∗y; y:= y−1* **fi**; *w:= w+1* **od**
c3: $(*\ z := 2^x\ *)$ *y:= x; z:= 1; w:= 0;* **while** $y \neq 0$ **do if** $w\ \%\ 3 = 0$ **then** *z:= z∗2; y:= y−1* **fi**; *w:= w+1* **od**

Fig. 1. Example programs (where % is modulo).

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 $c \mid c' : \mathcal{R} \approx\!> \mathcal{S}$ from the HL judgment $c; c' : \mathcal{R} \rightsquigarrow \mathcal{S}$, 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 representative collections 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 $P \rightsquigarrow Q$ 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 $P \rightsquigarrow Q$, there is a proof in HL of $c : P \rightsquigarrow Q$ using only judgments of the form $b : R \rightsquigarrow S$ with b ranging over subprograms of c and R, S the assertions annotating the entry and exit points of b.*[1]

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 $c0 \mid c0' : x = x' \approx\!> z = z'$, choose pairs at the lockstep positions, and the abovementioned conjunction $y = y' \wedge z = z'$. 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 $c0 \mid c2 : x = x' \approx\!> z = z'$ 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. II 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. III 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. 6.

automaton on pairs of states that serves to represent aligned steps of two computations.

Sec. IV gives the first alignment completeness theorem: Given a valid annotation of a product automaton for $c|c' : P \approx\!\!> Q$ 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. V gives the alignment completness theorem for lockstep alignment: if $c \mid c' : P \approx\!\!> Q$ 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 SEQPROD 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 SEQPROD. As a worked example we show that $c2$ majorizes $c3$ 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.

Additional details and proofs are in the appendix of an extended version of the paper (https://arxiv.org/abs/2101.11730).

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 $(Ctrl, Sto, init, \mathit{fin}, \mapsto)$ where Sto is a set (the data stores), $Ctrl$ is a finite set (the control points) that contains distinct elements $init$ and fin, and $\mapsto \subseteq (Ctrl \times Sto) \times (Ctrl \times Sto)$ is the transition relation. We require $(n, s) \mapsto (m, t)$ to imply $n \neq \mathit{fin}$ 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 ctrl, 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 τ is ***terminated*** provided τ is finite and $\mathsf{ctrl}(\tau_{-1}) = \mathit{fin}$, where τ_{-1} denotes the last state of τ. An ***initial trace*** is one such that $\mathsf{ctrl}(\tau_0) = init$. We allow traces of length one, in which case $\tau_{-1} = \tau_0$, but a terminated initial trace has plural length because $init \neq \mathit{fin}$.

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 τ to satisfy $P \rightsquigarrow Q$ means that $\tau_0 \models P$ and $\mathsf{ctrl}(\tau_{-1}) = \mathit{fin}$ imply $\tau_{-1} \models Q$, in which case we write $\tau \models P \rightsquigarrow Q$. Then A satisfies $P \rightsquigarrow Q$, written $A \models P \rightsquigarrow Q$, just if all its finite initial traces do. For ***non-stuck semantics*** there are two conditions: (i) $\tau_0 \models P$ and $\mathsf{ctrl}(\tau_{-1}) = \mathit{fin}$ imply $\tau_{-1} \models Q$, and (ii) $\tau_0 \models P$ and $\mathsf{ctrl}(\tau_{-1}) \neq \mathit{fin}$ imply $\tau_{-1} \mapsto -$, where $\tau_{-1} \mapsto -$ 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 $P \rightsquigarrow Q$ 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 $\mathsf{CFG}(A)$ to be the rooted directed graph with vertices $Ctrl$, root $init$, and an edge $n \to m$ iff $\exists s, t.\ (n, s) \mapsto (m, t)$. For our purposes CFGs are unlabelled; we write $n \to m$ 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 (ctrl) over a trace τ of A we get its ***control path***, $\mathsf{cpath}(\tau)$, i.e., the sequence of control points in τ.

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

As an example, Fig. 2 shows a labelled version of program $c0$. Fig. 3 shows the CFG of the automaton for $c0; \mathsf{skip}$[6]. 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 $vs \in \mathsf{segs}(A, K)$ define relation $\overset{vs}{\longmapsto}$ by $(n, s) \overset{vs}{\longmapsto} (m, t)$ iff there is a trace τ of A with $\mathsf{cpath}(\tau) = vs$ and $\tau_0 = (n, s)$ and $\tau_{-1} = (m, t)$. Notice that $\overset{vs}{\longmapsto}$ need not be total, even in the typical case that the underlying relation $\mapsto$ is never stuck on non-fin 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 $\overset{vs}{\longmapsto}$.

Given automaton A, cutpoint set K, and spec $P \rightsquigarrow Q$, an ***annotation*** is a function an from K to store predicates such that $an(init) = P$ and $an(\mathit{fin}) = Q$. The requirement $init \neq \mathit{fin}$ ensures that annotations exist for any spec.

$c0:\quad y :=^1 x; z :=^2 1; \mathsf{while}^3\ y \neq 0\ \mathsf{do}\ z :=^4 z * y; y :=^5 y - 1\ \mathsf{od}$

$c4:\quad y :=^1 x; z :=^2 24; w :=^3 0; \mathsf{while}^4\ y \neq 4\ \mathsf{do}\ \mathsf{if}^5\ w\,\%\,2 = 0\ \mathsf{then}\ z :=^6 z * y; y :=^7 y - 1\ \mathsf{else}\ \mathsf{skip}^8\ \mathsf{fi}; w :=^9 w + 1\ \mathsf{od}$

$c5:\quad y :=^1 x; z :=^2 16; w :=^3 0; \mathsf{while}^4\ y \neq 4\ \mathsf{do}\ \mathsf{if}^5\ w\,\%\,3 = 0\ \mathsf{then}\ z :=^6 z * 2; y :=^7 y - 1\ \mathsf{else}\ \mathsf{skip}^8\ \mathsf{fi}; w :=^9 w + 1\ \mathsf{od}$

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

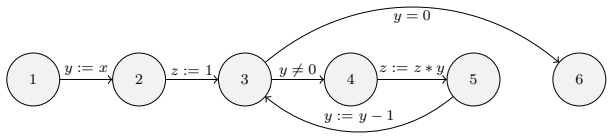

Fig. 3. CFG of the automaton $\mathsf{aut}(c0; \mathsf{skip}^6)$, with suggestive edge labels.

We lift an to a function $\hat{an}$ that yields states: $\hat{an}(n) = \{(n, s) \mid s \models an(n)\}$. Put differently: $\hat{an}(n) = \{n\} \times an(n)$. For each vs in $\mathsf{segs}(A, K)$ there is a ***verification condition*** (***VC***):

$$\mathsf{post}(\overset{vs}{\longmapsto})(\hat{an}(vs_0)) \subseteq \hat{an}(vs_{-1}) \tag{1}$$

Here $\mathsf{post}(\overset{vs}{\longmapsto})$ is the direct image i.e., strongest postcondition. Using the universal pre-image,[2] (1) is equivalent to $\hat{an}(vs_0) \subseteq \mathsf{pre}(\overset{vs}{\longmapsto})(\hat{an}(vs_{-1}))$. The VC says that for every trace τ with $\mathsf{cpath}(\tau) = vs$, if $\tau_0 \models an(vs_0)$ then $\tau_{-1} \models an(vs_{-1})$. Annotation an is ***valid*** if all the VCs 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, an, of automaton A with cutpoint set K, for $P \rightsquigarrow Q$. In any initial trace τ of A such that $\tau_0 \models P$, at any position i, $0 \leq i < len(\tau)$, we have $\mathsf{ctrl}(\tau_i) \in K \Rightarrow \tau_i \models an(\mathsf{ctrl}(\tau_i))$. Moreover $A \models P \rightsquigarrow Q$.*

Proposition 2 (completeness of IAM)**.** *Suppose $A \models P \rightsquigarrow Q$ 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 joins, 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 $n \in \mathbb{Z}$:

$$\begin{aligned} c ::= \ & \mathsf{skip}^n \mid x :=^n e \mid c; c \mid c \sqcup^n c \\ & \mid \mathsf{if}^n\ e\ \mathsf{then}\ c\ \mathsf{else}\ c\ \mathsf{fi} \mid \mathsf{while}^n\ e\ \mathsf{do}\ c\ \mathsf{od} \end{aligned}$$

Here metavariable x ranges over a set Var of variable names and e ranges over integer expressions. The form $c \sqcup^n d$ is for nondeterministic choice. We also use metavariables $b, d, c', \ldots$ for commands.

[2]For relation R on states and set X of states, $\mathsf{pre}(R)(X) = \{\alpha \mid \forall \beta.\ \alpha R \beta \Rightarrow \beta \in X\}$ and $\mathsf{post}(R)(X) = \{\beta \mid \exists \alpha.\ \alpha \in X \wedge \alpha R \beta\}$.

$$\frac{s(e) \neq 0}{\langle \mathsf{if}^n\ e\ \mathsf{then}\ c\ \mathsf{else}\ d\ \mathsf{fi},\ s\rangle \rightarrowtriangle \langle c,\ s\rangle}$$

$$\frac{s(e) = 0}{\langle \mathsf{if}^n\ e\ \mathsf{then}\ c\ \mathsf{else}\ d\ \mathsf{fi},\ s\rangle \rightarrowtriangle \langle d,\ s\rangle}$$

$$\frac{s(e) \neq 0}{\langle \mathsf{while}^n\ e\ \mathsf{do}\ c\ \mathsf{od},\ s\rangle \rightarrowtriangle \langle c; \mathsf{while}^n\ e\ \mathsf{do}\ c\ \mathsf{od},\ s\rangle}$$

$$\frac{s(e) = 0}{\langle \mathsf{while}^n\ e\ \mathsf{do}\ c\ \mathsf{od},\ s\rangle \rightarrowtriangle \langle \mathsf{skip}^{-n},\ s\rangle} \qquad \frac{\langle c,\ s\rangle \rightarrowtriangle \langle d,\ t\rangle}{\langle c; b,\ s\rangle \rightarrowtriangle \langle d; b,\ t\rangle}$$

$$\langle x :=^n e,\ s\rangle \rightarrowtriangle \langle \mathsf{skip}^{-n},\ [s \mid x\colon s(e)]\rangle \qquad \langle \mathsf{skip}^n; c,\ s\rangle \rightarrowtriangle \langle c,\ s\rangle$$

$$\langle c \sqcup^n d,\ s\rangle \rightarrowtriangle \langle c,\ s\rangle \qquad \langle c \sqcup^n d,\ s\rangle \rightarrowtriangle \langle d,\ s\rangle$$

Fig. 4. Command semantics (with n ranging over $\mathbb{Z}$).

Let $\mathsf{lab}(c)$ be the label of command c, defined recursively in the case of sequence: $\mathsf{lab}(c; d) = \mathsf{lab}(c)$. Let $\mathsf{labs}(c)$ 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 $c; \mathsf{skip}^{fin}$ where $fin \in \mathbb{N}$ 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 $\mathsf{ok}(c)$ 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 ***variable stores***, i.e., total mappings from Var to $\mathbb{Z}$. We write $[s \mid x\colon i]$ for the store like s but mapping x to i. A ***configuration*** $\langle c,\ s\rangle$ pairs a labelled command with a store, and we let $\mathsf{ctrl}\langle c,\ s\rangle = c$ and $\mathsf{stor}\langle c,\ s\rangle = s$.

The transition relation $\rightarrowtriangle$ is defined in Fig. 4. In a configuration reached from an 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 skip^n for some $n \in \mathbb{Z}$. Assume integer expressions are everywhere defined, so configurations are not stuck under $\rightarrowtriangle$ unless the program is a single skip.

$$
\begin{array}{ll}
\mathsf{fsuc}(n, \mathsf{skip}^n, f) & = f \\
\mathsf{fsuc}(n, x :=^n e, f) & = f \\
\mathsf{fsuc}(n, c; d, f) & = \mathsf{fsuc}(n, c, \mathsf{lab}(d)) \text{ , if } n \in \mathsf{labs}(c) \\
 & = \mathsf{fsuc}(n, d, f) \text{ , otherwise} \\
\mathsf{fsuc}(m, \mathsf{while}^n\ e\ \mathsf{do}\ c\ \mathsf{od}, f) & = f \text{ , if } m = n \\
 & = \mathsf{fsuc}(m, c, n) \text{ , otherwise} \\
\mathsf{fsuc}(m, \mathsf{if}^n\ e\ \mathsf{then}\ c\ \mathsf{else}\ d\ \mathsf{fi}, f) & = \mathsf{fsuc}(m, c, f) \text{ , if } m \in \mathsf{labs}(c) \\
 & = \mathsf{fsuc}(m, d, f) \text{ , if } m \in \mathsf{labs}(d) \\
 & = f \text{ , otherwise (i.e., } m = n) \\
\mathsf{fsuc}(m, c \sqcup^n d, f) & = \mathsf{fsuc}(m, c, f) \text{ , if } m \in \mathsf{labs}(c) \\
 & = \mathsf{fsuc}(m, d, f) \text{ , if } m \in \mathsf{labs}(d) \\
 & = f \text{ , otherwise (i.e., } m = n)
\end{array}
$$

Fig. 5. Following successor $\mathsf{fsuc}(n, c, f)$, assuming $\mathsf{ok}(c)$, $n \in \mathsf{labs}(c)$, and $f \in \mathbb{N} \setminus \mathsf{labs}(c)$.

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 $c0$ in Fig. 2, then $\mathsf{sub}(2, c0)$ is $z :=^2 1$ and $\mathsf{sub}(3, c0)$ is $\mathsf{while}^3\ y \neq 0\ \mathsf{do} \ldots \mathsf{od}$. To manipulate the CFG of a program of the form $c; \mathsf{skip}^{fin}$ that is ok (so, $fin \notin \mathsf{labs}(c)$), we define functions fsuc and elab. For motivation, the control flow successors of the loop, $sub(3, c0)$, in $c0; \mathsf{skip}^6$ are 3 and 6. Whereas 3 is inside $c0$, 6 is not. We call 6 the ***following successor***, given by $\mathsf{fsuc}(3, c0, 6)$. In general, $\mathsf{fsuc}(n, c, f)$ is defined by recursion on c; see Fig. 5. For example, $\mathsf{fsuc}(\mathsf{sub}(3, c0), c0, 6) = 6$ and $\mathsf{fsuc}(\mathsf{sub}(5, c0), c0, 6) = 3$. For another example, let c be $\mathsf{if}^1\ x > 0\ \mathsf{then}\ x :=^2 x - 1; y :=^3 x\ \mathsf{else}\ \mathsf{skip}^4\ \mathsf{fi}$; 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 $\mathsf{elab}(b, c, fin)$ 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 $\mathsf{elab}(b, c, fin) = \mathsf{fsuc}(\mathsf{lab}(b), c, fin)$. In case b is a sequence $b_0; b_1$, let $\mathsf{elab}(b, c, fin) = \mathsf{elab}(b_1, c, fin)$, i.e., the exit of a sequence is the exit of its last command.[3]

Now we can define the CFG for an ok program $c; \mathsf{skip}^{fin}$, 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 $\{fin\} \cup \mathsf{labs}(c)$. There is no control flow successor of fin. For $n \in \mathsf{labs}(c)$ there are one or two successors, described by cases on $\mathsf{sub}(n, c)$:

$\mathsf{sub}(n, c)$	successor(s) of n in the CFG
$x :=^n e$	$n \to \mathsf{fsuc}(n, c, fin)$
skip^n	$n \to \mathsf{fsuc}(n, c, fin)$
$\mathsf{if}^n\ e\ \mathsf{then}\ d_0\ \mathsf{else}\ d_1\ \mathsf{fi}$	$n \to \mathsf{lab}(d_0)$ and $n \to \mathsf{lab}(d_1)$
$d_0 \sqcup^n d_1$	$n \to \mathsf{lab}(d_0)$ and $n \to \mathsf{lab}(d_1)$
$\mathsf{while}^n\ e\ \mathsf{do}\ d\ \mathsf{od}$	$n \to \mathsf{lab}(d)$ and $n \to \mathsf{fsuc}(n, c, fin)$

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

- $\exists d.\ \langle \mathsf{sub}(n, c),\ s\rangle \twoheadrightarrow \langle d,\ t\rangle \wedge \mathsf{lab}(d) \geq 0 \wedge m = \mathsf{lab}(d)$
- $\exists d.\ \langle \mathsf{sub}(n, c),\ s\rangle \twoheadrightarrow \langle d,\ t\rangle \wedge \mathsf{lab}(d) < 0 \wedge m = \mathsf{fsuc}(n, c, fin)$
- or $\mathsf{sub}(n, c) = \mathsf{skip}^n \wedge m = \mathsf{fsuc}(n, c, fin) \wedge t = s$

The first two cases use the semantics of Fig. 4 for a subcommand on its own. The second case uses 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 $\mathsf{aut}(c; \mathsf{skip}^{fin})$ are terminated ones.

[3]Formally the definition of elab is by recursion on its first argument. We can define elab as a function of b because unique labels rules out multiple occurrences of a subprogram. And it needs to be a function of command b, not its label, to handle sequences.

For any traces τ via $\twoheadrightarrow$ and υ via $\mapsto$, define $\tau \asymp \upsilon$ iff $len(\tau) = len(\upsilon)$ and $\mathsf{stor}(\tau_i) = \mathsf{stor}(\upsilon_i)$ and $\mathsf{lab}(\mathsf{ctrl}(\tau_i)) = \mathsf{ctrl}(\upsilon_i)$, for all i, $0 \leq i < len(\tau)$.

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

(i) For any trace τ *from* $\langle c; \mathsf{skip}^n,\ s\rangle$ *via* $\twoheadrightarrow$, *there is a trace* υ *of* A *from* $(\mathsf{lab}(c), s)$ *via* $\mapsto$, *such that* $\tau \asymp \upsilon$.

(ii) For any trace υ *of* A *from* $(\mathsf{lab}(c), s)$ *via* $\mapsto$, *there is a trace* τ *from* $\langle c; \mathsf{skip}^n,\ s\rangle$ *via* $\twoheadrightarrow$, *such that* $\tau \asymp \upsilon$.

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* ok *program* $c; \mathsf{skip}^{fin}$ *and a full annotation,* an, *of its automaton. For each control edge* $n \to m$, *the VC (1) 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 $an(n) \wedge e$ to abbreviate $an(n) \cap \{s \mid s(e) \neq 0\}$. For a set P of program stores, we use substitution notation P^x_e with standard meaning: $s \in P^x_e$ iff $[s \mid x\colon s(e)] \in P$.

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 $\langle c, s\rangle \twoheadrightarrow^* \langle \mathsf{skip}^n,\ t\rangle$ then $t \models Q$. We write this as $\models c : P \rightsquigarrow Q$. As is well known, the rules are sound: $\vdash c : P \rightsquigarrow Q$ implies $\models c : P \rightsquigarrow Q$.

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

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

- for subprograms b of c, the judgments
$$b : an(\mathsf{lab}(b)) \rightsquigarrow an(\mathsf{elab}(b, c, fin)) \qquad (2)$$
- $b : an(\mathsf{lab}(b)) \wedge e \rightsquigarrow an(\mathsf{elab}(b, c, fin))$ where b is the body of a loop, or then-branch of a conditional, with test e;
- $b : an(\mathsf{lab}(b)) \wedge \neg e \rightsquigarrow an(\mathsf{elab}(b, c, fin))$ where b is the else-branch of a conditional with test e;

if $b = \mathsf{sub}(n, c)$ is…	and $n \to m$ in CFG is…	then the VC is equivalent to…
skip^n	$m = \mathsf{elab}(b, c, \mathit{fin})$	$an(n) \Rightarrow an(m)$
$x :=^n e$	$m = \mathsf{elab}(b, c, \mathit{fin})$	$an(n) \Rightarrow an(m)^x_e$
$\mathsf{if}^n\ e\ \mathsf{then}\ d_0\ \mathsf{else}\ d_1\ \mathsf{fi}$	$m = \mathsf{lab}(d_0)$	$an(n) \wedge e \Rightarrow an(m)$
$\mathsf{if}^n\ e\ \mathsf{then}\ d_0\ \mathsf{else}\ d_1\ \mathsf{fi}$	$m = \mathsf{lab}(d_1)$	$an(n) \wedge \neg e \Rightarrow an(m)$
$\mathsf{while}^n\ e\ \mathsf{do}\ d\ \mathsf{od}$	$m = \mathsf{lab}(d)$	$an(n) \wedge e \Rightarrow an(m)$
$\mathsf{while}^n\ e\ \mathsf{do}\ d\ \mathsf{od}$	$m = \mathsf{elab}(b, c, \mathit{fin})$	$an(n) \wedge \neg e \Rightarrow an(m)$
$d_0 \sqcup^n d_1$	m is $\mathsf{lab}(d_0)$ or $\mathsf{lab}(d_1)$	$an(n) \Rightarrow an(m)$

Fig. 6. VCs for the automaton of a program $c; \mathsf{skip}^{\mathit{fin}}$ and full annotation an.

$$\textsc{Skip}\ \ \mathsf{skip} : P \rightsquigarrow P \qquad \textsc{Ass}\ \ x := e : P^x_e \rightsquigarrow P \qquad \textsc{Seq}\ \ \frac{c : P \rightsquigarrow R \quad d : R \rightsquigarrow Q}{c; d : P \rightsquigarrow Q}$$

$$\textsc{If}\ \ \frac{c : P \wedge e \rightsquigarrow Q \quad d : P \wedge \neg e \rightsquigarrow Q}{\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ d\ \mathsf{fi} : P \rightsquigarrow Q}$$

$$\textsc{Wh}\ \ \frac{c : P \wedge e \rightsquigarrow P}{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od} : P \rightsquigarrow P \wedge \neg e} \qquad \textsc{Choice}\ \ \frac{c : P \rightsquigarrow Q \quad d : P \rightsquigarrow Q}{c \sqcup d : P \rightsquigarrow Q}$$

$$\textsc{Conseq}\ \ \frac{P \Rightarrow R \quad c : R \rightsquigarrow S \quad S \Rightarrow Q}{c : P \rightsquigarrow Q}$$

Fig. 7. Rules of HL (command labels elided).

- $b : an(\mathsf{lab}(b)) \rightsquigarrow an(\mathsf{lab}(b)) \wedge \neg e$ where b is a loop with test e; and
- $x :=^n e : an(m)^x_e \rightsquigarrow an(m)$ where $m = \mathsf{elab}(x :=^n e, c, \mathit{fin})$.

Theorem 6 (Floyd completeness)**.** *Consider an* ok *program* $c; \mathsf{skip}^{\mathit{fin}}$*, and a valid annotation,* an*, of* $\mathsf{aut}(c; \mathsf{skip}^{\mathit{fin}})$ *for* $P \rightsquigarrow Q$*. Then there is proof in HL of* $c : P \rightsquigarrow Q$ *using only the associated judgments of* an*.*

A corollary is Cook completeness, i.e., $\models c : P \rightsquigarrow Q$ implies $\vdash c : P \rightsquigarrow Q$, using Prop. 2.

To prove the theorem, we first prove that

$$\vdash b : an(\mathsf{lab}(b)) \rightsquigarrow an(\mathsf{elab}(b, c, \mathit{fin})) \tag{3}$$

for every subprogram b of c. The claim (3) is proved by structural induction on c. In each case, we use one instance of the syntax-directed rule for b, and in some cases also Conseq. The base cases are the assignments and skip commands in c. For such a command b, let $m = \mathsf{elab}(b, c, \mathit{fin})$.

- If b is skip^n, we have $\vdash \mathsf{skip}^n : an(m) \rightsquigarrow an(m)$ by rule Skip, and Lemma 5 gives $an(n) \Rightarrow an(m)$ (using validity of an), so by Conseq we get $\vdash \mathsf{skip}^n : an(n) \rightsquigarrow an(m)$
- if b is $x :=^n e$, we have $\vdash x :=^n e : an(m)^x_e \rightsquigarrow an(m)$ by rule Ass, and Lemma 5 gives $an(n) \Rightarrow an(m)^x_e$, so by Conseq we get $\vdash x :=^n e : an(n) \rightsquigarrow an(m)$

The induction step is as follows. (Other cases in appendix.)

- If b is the sequence $b_0; b_1$, by induction we have $\vdash b_0 : an(\mathsf{lab}(b_0)) \rightsquigarrow an(\mathsf{elab}(b_0, c, \mathit{fin}))$ and $\vdash b_1 : an(\mathsf{lab}(b_1)) \rightsquigarrow an(\mathsf{elab}(b_1, c, \mathit{fin}))$. We have $\mathsf{elab}(b_0, c, \mathit{fin}) = \mathsf{lab}(b_1)$ and $\mathsf{elab}(b_0; b_1, c, \mathit{fin}) = \mathsf{elab}(b_1, c, \mathit{fin})$, so by Seq we get $\vdash b : an(\mathsf{lab}(b)) \rightsquigarrow an(\mathsf{elab}(b, c, \mathit{fin}))$.
- If b is $\mathsf{if}^n\ e\ \mathsf{then}\ d_0\ \mathsf{else}\ d_1\ \mathsf{fi}$, by induction we have $\vdash d_0 : an(\mathsf{lab}(d_0)) \rightsquigarrow an(\mathsf{elab}(d_0, c, \mathit{fin}))$ and $\vdash d_1 : an(\mathsf{lab}(d_1)) \rightsquigarrow an(\mathsf{elab}(d_1, c, \mathit{fin}))$. Lemma 5 gives $an(n) \wedge e \Rightarrow an(\mathsf{lab}(d_0))$ and $an(n) \wedge \neg e \Rightarrow an(\mathsf{lab}(d_1))$ so by Conseq we get $\vdash d_0 : an(n) \wedge e \rightsquigarrow an(\mathsf{elab}(d_0, c, \mathit{fin}))$ and $\vdash d_1 : an(n) \wedge \neg e \rightsquigarrow an(\mathsf{elab}(d_1, c, \mathit{fin}))$. By definitions we have $\mathsf{elab}(b, c, \mathit{fin}) = \mathsf{elab}(d_0, c, \mathit{fin}) = \mathsf{elab}(d_1, c, \mathit{fin})$. So rule If yields $\vdash b : an(n) \rightsquigarrow an(\mathsf{elab}(b, c, \mathit{fin}))$.

To prove the theorem we instantiate (3) with c itself, to get $\vdash c : an(\mathsf{lab}(c)) \rightsquigarrow an(\mathsf{elab}(c, c, \mathit{fin}))$. Now $\mathit{init} = \mathsf{lab}(c)$ by definition of $\mathsf{aut}(c; \mathsf{skip}^{\mathit{fin}})$, and an is an annotation for $P \rightsquigarrow Q$, so $an(\mathsf{lab}(c)) = an(\mathit{init}) = P$ and $an(\mathsf{elab}(c, c, \mathit{fin})) = an(\mathit{fin}) = Q$. Thus we have obtained $\vdash c : P \rightsquigarrow Q$, highlighting the associated judgments, q.e.d.

In the rest of the paper, we assume without mention that all considered programs satisfy ok.

III. Relational judgments and product automata

Let $A = (\mathit{Ctrl}, \mathit{Sto}, \mathit{init}, \mathit{fin}, \mapsto)$ and $A' = (\mathit{Ctrl}', \mathit{Sto}', \mathit{init}', \mathit{fin}', \mapsto')$ be automata. A relational spec $\mathcal{R} \approx\!\!> \mathcal{S}$ is comprised of relations $\mathcal{R}$ and $\mathcal{S}$ from Sto to Sto'. We write $s, s' \models \mathcal{R}$ iff $(s, s') \in \mathcal{R}$ and $(n, s), (n', s') \models \mathcal{R}$ iff $s, s' \models \mathcal{R}$. Finite traces τ of A and τ' of A' satisfy $\mathcal{R} \approx\!\!> \mathcal{S}$, written $\tau, \tau' \models \mathcal{R} \approx\!\!> \mathcal{S}$, just if $\tau_0, \tau'_0 \models \mathcal{R}$, $\mathit{ctrl}(\tau_{-1}) = \mathit{fin}$, and $\mathit{ctrl}(\tau'_{-1}) = \mathit{fin}'$ imply $\tau_{-1}, \tau'_{-1} \models \mathcal{S}$. The pair A, A' satisfies $\mathcal{R} \approx\!\!> \mathcal{S}$, written $A | A' \models \mathcal{R} \approx\!\!> \mathcal{S}$, just if all pairs of finite initial traces do.

In passing we will consider the ***non-stuck semantics of relational specs***: in addition to the above conditions, it requires, for all finite initial τ, τ' such that $\tau_0, \tau'_0 \models \mathcal{R}$, that $\mathit{ctrl}(\tau_{-1}) \neq \mathit{fin}$ implies $\tau_{-1} \mapsto -$ and $\mathit{ctrl}(\tau'_{-1}) \neq \mathit{fin}'$ implies $\tau'_{-1} \mapsto' -$.

In casual examples, we use primed identifiers in specs to refer to the second execution. The sequential product rule involves renaming identifiers in order to encode two computations as one, but our product constructions and RHL rules do not require programs to act on distinct variables. We continue to use primes on metavariables to aid the reader, but introduce notations like $x \doteq x$ which expresses equality of the values of x in two states with the same variables. For program expressions, $e \doteq e'$ denotes the relation $\{(s, s') \mid s(e) = s'(e')\}$

which we call ***agreement***. For example, $x \doteq x \approx\!> z \doteq z$ 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. 4), we also need a different form, $e \mathrel{\mathring{=}} e'$, to express agreement on truth value; it denotes $\{(s,s') \mid s(e) \neq 0 \text{ iff } s'(e') \neq 0\}$. We also write $\{\!|e|\!\}$ for the set $\{(s,t) \mid s(e) \neq 0\}$ and $|\!|e|\!|$ for $\{(s,t) \mid t(e) \neq 0\}$.

In the presence of nondeterminacy, the $\forall\forall$-properties expressed by specs $\mathcal{R} \approx\!> \mathcal{S}$ are not the only properties of interest. For example, the program $x := x+1 \sqcup x := x+2$ does not satisfy $x \doteq x \approx\!> x \doteq x$, 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 program $(y := 0 \sqcup y := 1); x := x+1$ does satisfy $x \doteq x \approx\!> x \doteq x$.

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 $(C, (Sto \times Sto'), i, f, \Mapsto)$, together with functions $\mathsf{lt} : C \to Ctrl$ and $\mathsf{rt} : C \to Ctrl'$ such that the following hold: First, $\mathsf{lt}(i) = init$, $\mathsf{rt}(i) = init'$, $\mathsf{lt}(f) = fin$, and $\mathsf{rt}(f) = fin'$. Second, $(n, (s,s')) \Mapsto (m, (t,t'))$ implies either

- $(\mathsf{lt}(n), s) \mapsto (\mathsf{lt}(m), t)$ and $(\mathsf{rt}(n), s') = (\mathsf{rt}(m), t')$, or
- $(\mathsf{lt}(n), s) = (\mathsf{lt}(m), t)$ and $(\mathsf{rt}(n), s') \mapsto' (\mathsf{rt}(m), t')$, or
- $(\mathsf{lt}(n), s) \mapsto (\mathsf{lt}(m), t)$ and $(\mathsf{rt}(n), s') \mapsto' (\mathsf{rt}(m), t')$.

The second condition says each step of the product represents a step of A, a step of A', or both. For states of $\Pi_{A,A'}$, define $\mathsf{left}(n, (s,s')) = (\mathsf{lt}(n), s)$ and $\mathsf{right}(n, (s,s')) = (\mathsf{rt}(n), s')$. We extend left and right to traces of $\Pi_{A,A'}$ by $\mathsf{left}(T) = \mathsf{destutter}(\mathsf{map}(\mathsf{left}, T))$ and $\mathsf{right}(T) = \mathsf{destutter}(\mathsf{map}(\mathsf{right}, T))$. Observe that $\mathsf{left}(T)$ is a trace of A and $\mathsf{right}(T)$ a trace of A'. (Any repeated states in $\mathsf{map}(\mathsf{left}, T)$ are not transitions of A, owing to the non-stuttering condition for automata.)

A product is ***$\mathcal{R}$-adequate*** if it covers all terminated initial traces from $\mathcal{R}$-states: For all terminated initial traces τ of A and τ' of A' such that $\tau_0, \tau_0' \models \mathcal{R}$, there is a trace T of $\Pi_{A,A'}$ with $\tau = \mathsf{left}(T)$ and $\tau' = \mathsf{right}(T)$. A product is ***adequate*** if it is adequate for all initial pairs of states (i.e., true-adequate). A product is ***strongly $\mathcal{R}$-adequate*** if it covers prefixes of diverging traces, in addition to terminated ones, that is: For all finite initial traces τ, τ' of A, A' such that $\tau_0, \tau_0' \models \mathcal{R}$, there is a trace T of $\Pi_{A,A'}$ such that $\tau \preceq \mathsf{left}(T)$ and $\tau' \preceq \mathsf{right}(T)$, where $\preceq$ means prefix. Moreover, it does not have ***one-sided divergence***: there is no infinite trace T, with $T_0 \models \mathcal{R}$, with i such that $\mathsf{left}(T_j) = \mathsf{left}(T_i)$ for all $j > i$, or $\mathsf{right}(T_j) = \mathsf{right}(T_i)$ for all $j > i$. Note that strong $\mathcal{R}$-adequacy implies $\mathcal{R}$-adequacy.

Here are some products defined for arbitrary A, A', taking C to be $Ctrl \times Ctrl'$ and lt, rt to be fst, snd.

only-lockstep. $((n,n'),(s,s')) \Mapsto_{olck} ((m,m'),(t,t'))$ iff
$(n,s) \mapsto (m,t)$ and $(n',s') \mapsto' (m',t')$.

left-only. $((n,n'),(s,s')) \Mapsto_{lo} ((m,m'),(t,t'))$ iff
$(n,s) \mapsto (m,t)$ and $(n',s') = (m',t')$.

right-only. $((n,n'),(s,s')) \Mapsto_{ro} ((m,m'),(t,t'))$ iff
$(n,s) = (m,t)$ and $(n',s') \mapsto' (m',t')$.

interleaved. The union $\Mapsto_{lo} \cup \Mapsto_{ro}$.

eager-lockstep. $((n,n'),(s,s')) \Mapsto_{elck} ((m,m'),(t,t'))$ iff
$((n,n'),(s,s')) \Mapsto_{olck} ((m,m'),(t,t'))$, or
$n = fin$ and $((n,n'),(s,s')) \Mapsto_{ro} ((m,m'),(t,t'))$, or
$n' = fin'$ and $((n,n'),(s,s')) \Mapsto_{lo} ((m,m'),(t,t'))$

sequential. $((n,n'),(s,s')) \Mapsto_{seq} ((m,m'),(t,t'))$ iff
$n' = init'$ and $((n,n'),(s,s')) \Mapsto_{lo} ((m,m'),(t,t'))$, or
$n = fin$ and $((n,n'),(s,s')) \Mapsto_{ro} ((m,m'),(t,t'))$.

ctrl-conditioned. Given subsets L, R, J of $Ctrl \times Ctrl'$, define
$((n,n'),(s,s')) \Mapsto_{cnd} ((m,m'),(t,t'))$ iff
$(n,n') \in L$ and $((n,n'),(s,s')) \Mapsto_{lo} ((m,m'),(t,t'))$, or
$(n,n') \in R$ and $((n,n'),(s,s')) \Mapsto_{ro} ((m,m'),(t,t'))$, or
$(n,n') \in J$ and $((n,n'),(s,s')) \Mapsto_{olck} ((m,m'),(t,t'))$.

As an example of the 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 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 $c0$ with itself, in which an iteration of the loop body on just the left (resp. right) side may happen only when the store relation $\mathcal{L}$ (resp. $\mathcal{R}$) 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 τ is a finite prefix of a divergent trace, and τ' has length > 1, the sequential product never finishes on the left and so does not cover τ'.

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 τ is shorter than τ', the product automaton needs to extend τ by further steps in order to cover τ'.

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

- $i = 0$, $j = 1$, $((n,n')(s,s')) \Mapsto_{lo} ((m,m'),(t,t'))$, or
- $i = 1$, $j = 0$, $((n,n')(s,s')) \Mapsto_{ro} ((m,m'),(t,t'))$, or
- one of the last two eager-lockstep cases apply (one side terminated).

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 $\mathcal{R} \subseteq Sto \times Sto'$ 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

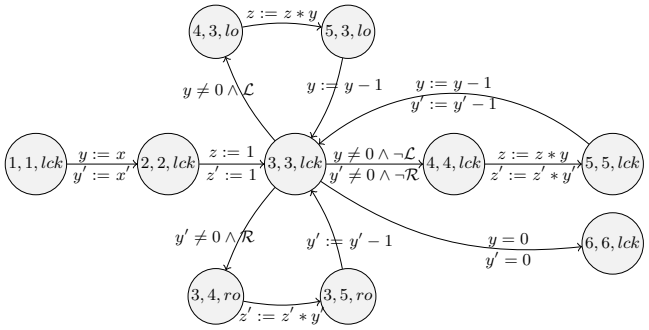

Fig. 8. Conditionally aligned loop product automaton for $c0|c0$, with informal edge labels including alignment guards $\mathcal{L}, \mathcal{R}$.

relational spec $\mathcal{R} \approx\!\!> \mathcal{S}$ for A, A' can be seen as a unary spec $\mathcal{R} \rightsquigarrow \mathcal{S}$ for $\Pi_{A,A'}$.

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

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

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

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

$$(\mathcal{R}^{x|x'}_{e|e'})^\oplus = (\mathcal{R}^\oplus)^{x,x'^\bullet}_{e,e'^\bullet} \tag{4}$$

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

Lemma 8. *Let c, d be programs on Var and $\mathcal{R}, \mathcal{S}$ be relations on Var-stores. Let $d' = \mathsf{dot}(d)$, so $c; d'$ is a $(\mathsf{Var} \cup \mathsf{Var}^\bullet)$-program. Then $\models c; d' : \mathcal{R}^\oplus \rightsquigarrow \mathcal{S}^\oplus$ iff $\models c|d : \mathcal{R} \approx\!\!> \mathcal{S}$.*

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

$$\textsc{SeqProd} \quad \frac{d' = \mathsf{dot}(d) \qquad c; d' : \mathcal{R}^\oplus \rightsquigarrow \mathcal{S}^\oplus}{c \mid d : \mathcal{R} \approx\!\!> \mathcal{S}}$$

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

To prove this, suppose $\models c|d : \mathcal{R} \approx\!\!> \mathcal{S}$. By Lemma 8 we have $\models c; d' : \mathcal{R}^\oplus \rightsquigarrow \mathcal{S}^\oplus$ where $d' = \mathsf{dot}(d)$. By completeness of HL we have $\vdash c; d' : \mathcal{R}^\oplus \rightsquigarrow \mathcal{S}^\oplus$ so SeqProd yields $\vdash c|d : \mathcal{R} \approx\!\!> \mathcal{S}$, 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 $b : an(n, 1)^\oplus \rightsquigarrow an(m, 1)^\oplus$ (with b a subprogram of c with $n = \mathsf{lab}(b)$ and m the end label), the form $b : an(\mathit{fin}, n)^\oplus \rightsquigarrow an(\mathit{fin}, m)^\oplus$, 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 an is a valid full annotation of the sequential product of $\mathsf{aut}(c; \mathsf{skip}^{\mathit{fin}})$ and $\mathsf{aut}(d; \mathsf{skip}^{\mathit{fin}})$, for spec $\mathcal{R} \rightsquigarrow \mathcal{S}$, then $\vdash c \mid d : \mathcal{R} \approx\!\!> \mathcal{S}$ 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 $\mathsf{lab}(c) = 1 = \mathsf{lab}(d)$. As a first step, consider the CFG of the sequential product Π of $\mathsf{aut}(c; \mathsf{skip}^{\mathit{fin}})$ and $\mathsf{aut}(d; \mathsf{skip}^{\mathit{fin}})$. Initial states of Π have the form $((1, 1), (s, t))$, final states have the form $((\mathit{fin}, \mathit{fin}), (s, t))$, and edges of the CFG are of two forms: $(n, 1) \to (m, 1)$ for $n \to m$ in the CFG of $c; \mathsf{skip}^{\mathit{fin}}$ and $(\mathit{fin}, n) \to (\mathit{fin}, m)$ for $n \to m$ in the CFG of $d; \mathsf{skip}^{\mathit{fin}}$.

Now assume an is a full annotation of Π. Because steps of Π 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 (fin, n). To

save space we just give some illustrative cases in Fig. 9, using some notations from Sec. III.

Let $\mathcal{Q} = an(\mathit{fin}, 1)$. We will use $\mathcal{Q}^{\oplus}$ as the intermediate assertion for rule SEQ in a proof of $c; \mathsf{dot}(d) : \mathcal{R}^{\oplus} \rightsquigarrow \mathcal{S}^{\oplus}$ which can then be used in SEQPROD to obtain $c|d : \mathcal{R} \approx\!\!> \mathcal{S}$. To obtain proofs of $c : \mathcal{R}^{\oplus} \rightsquigarrow \mathcal{Q}^{\oplus}$ and $\mathsf{dot}(d) : \mathcal{Q}^{\oplus} \rightsquigarrow \mathcal{S}^{\oplus}$, 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:

$$\vdash b : an(\mathsf{lab}(b), 1)^{\oplus} \rightsquigarrow an(m, 1)^{\oplus} \tag{5}$$

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

$$\vdash \mathsf{dot}(b) : an(\mathit{fin}, \mathsf{lab}(b))^{\oplus} \rightsquigarrow an(\mathit{fin}, m)^{\oplus} \tag{6}$$

where $m = \mathsf{elab}(b, d, \mathit{fin})$. In proving (6) using Fig. 9, we use that $(an(\mathit{fin}, n) \wedge \lfloor e \rfloor)^{\oplus} = an(\mathit{fin}, n)^{\oplus} \wedge \mathsf{dot}(e)$ and $(an(\mathit{fin}, n)|_e^{|x})^{\oplus} = (an(\mathit{fin}, n)^{\oplus})^{x^{\bullet}}_{\mathsf{dot}(e)}$. Instantiating (5) and (6) we get $\vdash c : an(1, 1)^{\oplus} \rightsquigarrow an(\mathit{fin}, 1)^{\oplus}$
$\vdash \mathsf{dot}(d) : an(\mathit{fin}, 1)^{\oplus} \rightsquigarrow an(\mathit{fin}, \mathit{fin})^{\oplus}$
Thus $\vdash c : \mathcal{R}^{\oplus} \rightsquigarrow \mathcal{Q}^{\oplus}$ and $\vdash \mathsf{dot}(d) : \mathcal{Q}^{\oplus} \rightsquigarrow \mathcal{S}^{\oplus}$, because $an(1, 1) = \mathcal{R}$, $an(\mathit{fin}, 1) = \mathcal{Q}$, and $an(\mathit{fin}, \mathit{fin}) = \mathcal{S}$.

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. I, such an alignment enables to prove $c0|c0 : x \doteq x \approx\!\!> z \doteq z$ using an annotation with intermediate relations only $y \doteq y \wedge z \doteq z$. A proof using SEQPROD with $c0; \mathsf{dot}(c0)$ requires to assert $z = x! \wedge x = x^{\bullet}$ 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 [1], [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 $x \leq x' \approx\!\!> y \leq y'$ is satisfied by if $x > 0$ then $y := x + 1$ else $y := x$ fi, 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 $\mathcal{R}$-adequacy, the code paths reached from initial state-pairs satisfying $\mathcal{R}$ need to be the same. Thus lockstep control is not adequate for programs with choice except in trivial cases like $x := 0 \sqcup x := 0$. Choice is ruled out in theorem.

Say c and c' have ***same control***, written $\mathsf{sameCtl}(c, c')$, if $\mathsf{labs}(c) = \mathsf{labs}(c')$ and for each $n \in \mathsf{labs}(c)$ the programs $\mathsf{sub}(n, c)$ and $\mathsf{sub}(n, c')$ 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 in 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, $x :=^5 y + z$ has same control as $w :=^5 x - 1$; so too $c4$ and $c5$ in Fig. 2. Same control implies identical CFGs.

For c, c' with same control, and their lockstep-control product Π, 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 $an(n, n) \Rightarrow e \stackrel{\circ}{=} e'$, then Π is $\mathcal{R}$-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 $b|b' : an(\mathsf{lab}(b), \mathsf{lab}(b)) \approx\!\!> an(m, m)$ where $m = \mathsf{elab}(b, c, \mathit{fin})$, 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 $\mathsf{sameCtl}(c, c')$. Let Π be the lockstep-control product of $\mathsf{aut}(c; \mathsf{skip}^{\mathit{fin}})$ and $\mathsf{aut}(c'; \mathsf{skip}^{\mathit{fin}})$ and let an be a valid full annotation of Π for $\mathcal{R} \rightsquigarrow \mathcal{S}$. Assume that for any branch point n with tests e, e' we have $an(n, n) \Rightarrow e \stackrel{\circ}{=} e'$. Then $\vdash c|c' : \mathcal{R} \approx\!\!> \mathcal{S}$ 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 $\mathcal{R}$ 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 $\mathsf{sameCtl}(c, c')$. We show, by induction on structure of c, that

if $b = \mathsf{sub}(n, c)$ is…	and $(n,1)\to(m,1)$ in CFG is…	then the VC is equivalent to…
$x :=^n e$	$m = \mathsf{elab}(b, c, \mathit{fin})$	$an(n,1) \Rightarrow an(m,1)^{x\mid}_{e\mid}$
$\mathsf{if}^n\ e\ \mathsf{then}\ b_0\ \mathsf{else}\ b_1\ \mathsf{fi}$	$m = \mathsf{lab}(b_0)$	$an(n,1) \wedge \{\!\mid e \mid\!\} \Rightarrow an(m,1)$
$b_0 \sqcup^n b_1$	m is $\mathsf{lab}(b_0)$ or $\mathsf{lab}(b_1)$	$an(n,1) \Rightarrow an(m,1)$

if $b = \mathsf{sub}(n, d)$ is…	and $(\mathit{fin}, n)\to(\mathit{fin}, m)$ in CFG is…	then the VC is equivalent to…
$x :=^n e$	$m = \mathsf{elab}(b, d, \mathit{fin})$	$an(\mathit{fin}, n) \Rightarrow an(\mathit{fin}, m)^{x}_{\mid e}$
$\mathsf{if}^n\ e\ \mathsf{then}\ b_0\ \mathsf{else}\ b_1\ \mathsf{fi}$	$m = \mathsf{lab}(b_0)$	$an(\mathit{fin}, n) \wedge \{\!\mid e \mid\!\} \Rightarrow an(\mathit{fin}, m)$
$b_0 \sqcup^n b_1$	m is $\mathsf{lab}(b_0)$ or $\mathsf{lab}(b_1)$	$an(\mathit{fin}, n) \Rightarrow an(\mathit{fin}, m)$

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

$$\textsc{dSkip}\ \ \mathsf{skip} \mid \mathsf{skip} : \mathcal{R} \approx\!\!> \mathcal{R} \qquad \textsc{dAss}\ \ x := e \mid x' := e' : \mathcal{R}^{x\mid x'}_{e\mid e'} \approx\!\!> \mathcal{R} \qquad \textsc{dSeq}\ \ \frac{c \mid c' : \mathcal{R} \approx\!\!> \mathcal{Q} \qquad d \mid d' : \mathcal{Q} \approx\!\!> \mathcal{S}}{c; d \mid c'; d' : \mathcal{R} \approx\!\!> \mathcal{S}}$$

$$\textsc{dIf}\ \ \frac{\mathcal{R} \Rightarrow e \mathrel{\mathring{=}} e' \qquad c \mid c' : \mathcal{R} \wedge \{\!\mid e \mid\!\} \wedge \{\!\mid e' \mid\!\} \approx\!\!> \mathcal{S} \qquad d \mid d' : \mathcal{R} \wedge \neg\{\!\mid e \mid\!\} \wedge \neg\{\!\mid e' \mid\!\} \approx\!\!> \mathcal{S}}{\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ d\ \mathsf{fi} \mid \mathsf{if}\ e'\ \mathsf{then}\ c'\ \mathsf{else}\ d'\ \mathsf{fi} : \mathcal{R} \approx\!\!> \mathcal{S}}$$

$$\textsc{dWh}\ \ \frac{\mathcal{Q} \Rightarrow e \mathrel{\mathring{=}} e' \qquad c \mid c' : \mathcal{Q} \wedge \{\!\mid e \mid\!\} \wedge \{\!\mid e' \mid\!\} \approx\!\!> \mathcal{Q}}{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od} \mid \mathsf{while}\ e'\ \mathsf{do}\ c'\ \mathsf{od} : \mathcal{Q} \approx\!\!> \mathcal{Q} \wedge \neg\{\!\mid e \mid\!\} \wedge \neg\{\!\mid e' \mid\!\}} \qquad \textsc{rConseq}\ \ \frac{\mathcal{P} \Rightarrow \mathcal{R} \qquad c \mid d : \mathcal{R} \approx\!\!> \mathcal{S} \qquad \mathcal{S} \Rightarrow \mathcal{Q}}{c \mid d : \mathcal{P} \approx\!\!> \mathcal{Q}}$$

Fig. 10. Lockstep (diagonal) syntax-directed rules.

if $b = \mathsf{sub}(n, c)$ and $b' = \mathsf{sub}(n, c')$ are…	and $(n,n)\to(m,m)$ in CFG is…	then the VC is equivalent to…
$x :=^n e$ and $x' :=^n e'$	$m = \mathsf{elab}(b, c, \mathit{fin}) = \mathsf{elab}(b', c', \mathit{fin})$	$an(n,n) \Rightarrow an(m,m)^{x\mid x'}_{e\mid e'}$
skip^n and skip^n	$m = \mathsf{elab}(b, c, \mathit{fin}) = \mathsf{elab}(b', c', \mathit{fin})$	$an(n,n) \Rightarrow an(m,m)$
$\mathsf{if}^n\ e\ \mathsf{then}\ b_0\ \mathsf{else}\ b_1\ \mathsf{fi}$ and $\mathsf{if}^n\ e'\ \mathsf{then}\ b'_0\ \mathsf{else}\ b'_1\ \mathsf{fi}$	$m = \mathsf{lab}(b_0) = \mathsf{lab}(b'_0)$	$an(n,n) \wedge \{\!\mid e \mid\!\} \wedge \{\!\mid e' \mid\!\} \Rightarrow an(m,m)$
$\mathsf{if}^n\ e\ \mathsf{then}\ b_0\ \mathsf{else}\ b_1\ \mathsf{fi}$ and $\mathsf{if}^n\ e'\ \mathsf{then}\ b'_0\ \mathsf{else}\ b'_1\ \mathsf{fi}$	$m = \mathsf{lab}(b_1) = \mathsf{lab}(b'_1)$	$an(n,n) \wedge \neg\{\!\mid e \mid\!\} \wedge \neg\{\!\mid e' \mid\!\} \Rightarrow an(m,m)$

Fig. 11. Selected VCs for lockstep-control product of $\mathsf{aut}(c; \mathsf{skip}^{\mathit{fin}})$ and $\mathsf{aut}(c'; \mathsf{skip}^{\mathit{fin}})$ with $\mathsf{sameCtl}(c, c')$, and full annotation an.

for every subprogram b of c, with corresponding subprogram b' in c':

$$\vdash b \mid b' : an(\mathsf{lab}(b), \mathsf{lab}(b)) \approx\!\!> an(m, m) \tag{7}$$

where $m = \mathsf{elab}(b, c, \mathit{fin})$. Note that $\mathsf{lab}(b) = \mathsf{lab}(b')$ and $m = \mathsf{elab}(b', c', \mathit{fin})$ by $\mathsf{sameCtl}(c, c')$. In the base case, b and b' are both assignments or both skip. We get that $an(\mathsf{lab}(b), \mathsf{lab}(b))$ implies the weakest precondition for the command to establish $an(m, m)$ by the first two rows in Fig. 11. So we can use dSkip or dAss, together with rConseq, to get (7). For the induction step, consider the case where b is $\mathsf{if}^n\ e\ \mathsf{then}\ b_0\ \mathsf{else}\ b_1\ \mathsf{fi}$ and b' is $\mathsf{if}^n\ e'\ \mathsf{then}\ b'_0\ \mathsf{else}\ b'_1\ \mathsf{fi}$. Let $m_0 = \mathsf{lab}(b_0) = \mathsf{lab}(b'_0)$ (using $\mathsf{sameCtl}$) and $m_1 = \mathsf{lab}(b_1) = \mathsf{lab}(b'_1)$. Let $p = \mathsf{elab}(b, c, \mathit{fin}) = \mathsf{elab}(b', c', \mathit{fin})$, noting that p is also the end label for the then and else parts. By induction we have $\vdash b_0 | b'_0 : an(m_0, m_0) \approx\!\!> an(p, p)$ and $\vdash b_1 | b'_1 : an(m_1, m_1) \approx\!\!> an(p, p)$. By the VCs we have $an(n,n) \wedge \{\!\mid e \mid\!\} \wedge \{\!\mid e' \mid\!\} \Rightarrow an(m_0, m_0)$ and $an(n,n) \wedge \neg\{\!\mid e \mid\!\} \wedge \neg\{\!\mid e' \mid\!\} \Rightarrow an(m_1, m_1)$ so using rConseq we get

$$\vdash b_0 | b'_0 : an(n,n) \wedge \{\!\mid e \mid\!\} \wedge \{\!\mid e' \mid\!\} \approx\!\!> an(p, p)$$
$$\vdash b_1 | b'_1 : an(n,n) \wedge \neg\{\!\mid e \mid\!\} \wedge \neg\{\!\mid e' \mid\!\} \approx\!\!> an(p, p)$$

By assumption of the theorem we have $an(n,n) \Rightarrow e \mathrel{\mathring{=}} e'$, which is the side condition of rule dIf, which yields:

$$\mathsf{if}^n\ e\ \mathsf{then}\ b_0\ \mathsf{else}\ b_1\ \mathsf{fi} \mid \mathsf{if}^n\ e'\ \mathsf{then}\ b'_0\ \mathsf{else}\ b'_1\ \mathsf{fi} : an(n,n) \approx\!\!> an(p, p)$$

The arguments for sequence and while are similar, using rules dSeq and 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 $\mathsf{sameCtl}$ 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 $x :=^n e$ on the left and skip^n on the right, with VC $an(n,n) \Rightarrow an(m,m)^{x\mid}_{e\mid}$ where $m = \mathsf{elab}(b, c, \mathit{fin}) = \mathsf{elab}(b', c', \mathit{fin})$. To get an alignment complete logic one would add the axiom $x := e \mid \mathsf{skip} : \mathcal{R}^{x\mid}_{e\mid} \approx\!\!> \mathcal{R}$ (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 and 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 SeqProd, 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 $\mathsf{sameExcept}(c, c', b, b', beg, end, fin)$ iff there are contexts $\hat{c}[\,]$ and $\hat{c'}[\,]$ such that

- $c = \hat{c}[b]$ and $c' = \hat{c'}[b']$
- $beg = \mathsf{lab}(b) = \mathsf{lab}(b')$ and $\mathsf{labs}(b) \cap \mathsf{labs}(b') = \{beg\}$
- $end = \mathsf{elab}(b, c, fin) = \mathsf{elab}(b', c', fin)$.
- $\mathsf{sameCtl}(\hat{c}[\mathsf{skip}^{beg}], \hat{c'}[\mathsf{skip}^{beg}])$
- $\hat{c}[\,]$ and $\hat{c'}[\,]$ are choice free (but b, b' may have choice)

The setup encompasses any pair of programs, because it includes the extreme case where $\hat{c}[\,]$ is nothing more than the hole to be filled, i.e., $\hat{c}[b] = b$ and $\hat{c'}[b'] = b'$; 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 $\mathsf{if}^1\ x > y\ \mathsf{then}\ y :=^2 y; x :=^3 0\ \mathsf{else}\ \mathsf{skip}^4\ \mathsf{fi}$ and
- c' is $\mathsf{if}^1\ y \leq x - 1\ \mathsf{then}\ x :=^2 0\ \mathsf{else}\ \mathsf{skip}^4\ \mathsf{fi}$
- b is $y :=^2 y; x :=^3 0$ and b' is a single assignment $x :=^2 0$

Here $beg = 2$ and $\mathsf{elab}(b, c, fin) = fin$. The example does not satisfy $\mathsf{sameCtl}$ (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 $\{lck, lo, ro\}$ to designate lockstep, left-only, and right-only steps. Let $Ctrl$ and $Ctrl'$ be the control sets for the automata of $c; \mathsf{skip}^{fin}$ and $c'; \mathsf{skip}^{fin}$ respectively, assuming as before that $\mathsf{lab}(c) = 1 = \mathsf{lab}(c')$. Recall $Ctrl = \mathsf{labs}(c; \mathsf{skip}^{fin})$ and $Ctrl' = \mathsf{labs}(c'; \mathsf{skip}^{fin})$. The control set of the product is $Ctrl \times Ctrl' \times \{lck, lo, ro\}$ and lt and rt are the first two projections. The initial and final control points are $(1, 1, lck)$ and (fin, fin, lck). Define $\Mapsto$ as follows, where $\mapsto$ is from $\mathsf{aut}(c; \mathsf{skip}^{fin})$, $\mapsto'$ is from $\mathsf{aut}(c'; \mathsf{skip}^{fin})$, and $\Mapsto_{lckc}$ is the lockstep-control product based on those. For all n, m, s, t, s', t':

(i) $((n, n, lck), (s, s')) \Mapsto ((m, m, lck), (t, t'))$
if $((n, n), (s, s')) \mapsto_{lckc} ((m, m), (t, t'))$ and $n \notin \mathsf{labs}(b) \cup \mathsf{labs}(b')$ and $m \neq beg$

(ii) $((n, n, lck), (s, s')) \Mapsto ((beg, beg, lo), (t, t'))$
if $((n, n), (s, s')) \mapsto_{lckc} ((beg, beg), (t, t'))$ and $n \notin \mathsf{labs}(b) \cup \mathsf{labs}(b')$

(iii) $((n, beg, lo), (s, s')) \Mapsto ((m, beg, lo), (t, s'))$
if $m \neq end$ and $(n, s) \mapsto (m, t)$ and $n \in \mathsf{labs}(b)$

(iv) $((n, beg, lo), (s, s')) \Mapsto ((end, beg, ro), (t, s'))$
if $(n, s) \mapsto (end, t)$ and $n \in \mathsf{labs}(b)$

(v) $((end, n', ro), (s, s')) \Mapsto ((end, m', ro), (s, t'))$
if $m' \neq end$ and $(n', s') \mapsto' (m', t')$ and $n' \in \mathsf{labs}(b')$

(vi) $((end, n', ro), (s, s')) \Mapsto ((end, end, lck), (s, t'))$
if $(n', s') \mapsto' (end, t')$ and $n' \in \mathsf{labs}(b)$

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* $\mathsf{sameExcept}(c, c', b, b', beg, end, fin)$ *and* Π *is the product defined above. Suppose* an *is a valid full annotation of* Π *for* $\mathcal{R} \rightsquigarrow \mathcal{S}$. *Suppose for all branch points* $n \in \mathsf{labs}(c) \setminus (\mathsf{labs}(b) \cup \mathsf{labs}(b'))$, *with branch conditions* e, e', *we have* $an(n, n, lck) \Rightarrow e \mathrel{\mathring{=}} e'$. *Then* $\vdash c|c' : \mathcal{R} \approx\!\!> \mathcal{S}$ *in the logic comprised of HL, the rules of Fig. 10, and* 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 $c2 \mid c3 : x = x' \wedge x > 3 \approx\!\!> z > z'$, by aligning the iterations in which y gets updated, maintaining invariant $y = y' \wedge z > z'$ and allowing the no-op iterations to happen independently. To do so we use this rule [11]:

caWhile

$$\frac{\begin{array}{c} c \mid c' : \mathcal{Q} \wedge \{\!|e|\!\} \wedge \}\!|e'|\!\{ \wedge \neg\mathcal{L} \wedge \neg\mathcal{R} \approx\!\!> \mathcal{Q} \\ c \mid \mathsf{skip} : \mathcal{Q} \wedge \mathcal{L} \wedge \{\!|e|\!\} \approx\!\!> \mathcal{Q} \\ \mathsf{skip} \mid c' : \mathcal{Q} \wedge \mathcal{R} \wedge \}\!|e'|\!\{ \approx\!\!> \mathcal{Q} \\ \mathcal{Q} \Rightarrow e \mathrel{\mathring{=}} e' \vee (\mathcal{L} \wedge \{\!|e|\!\}) \vee (\mathcal{R} \wedge \}\!|e'|\!\{) \end{array}}{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od} \mid \mathsf{while}\ e'\ \mathsf{do}\ c'\ \mathsf{od} : \mathcal{Q} \approx\!\!> \mathcal{Q} \wedge \neg\{\!|e|\!\} \wedge \neg\}\!|e'|\!\{}$$

There are three premises, which strengthen the invariant $\mathcal{Q}$ 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 $\mathcal{L}$ or $\mathcal{R}$. The side condition ensures these are adequate to cover all cases. Later we consider the example of $c2, c3$ using $\{\!|w \,\%\, 2 \neq 0|\!\}$ for $\mathcal{L}$ and $\}\!|w' \,\%\, 3 \neq 0|\!\{$ for $\mathcal{R}$. 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 beg such that $\mathsf{sub}(beg, c)$ is a loop—and so is $\mathsf{sub}(beg, c')$, since we assume $\mathsf{sameCtl}(c, c')$. Let $Ctrl, Ctrl'$ and $\mapsto, \mapsto'$ be the control sets and transition relations for their automata (noting that $Ctrl' = Ctrl$). As in Sec. VI we define a product with control $Ctrl \times Ctrl' \times Tag$ where $Tag = \{lck, lo, ro\}$. The transition relation $\Mapsto$ is defined with respect to given relations $\mathcal{L}$ and $\mathcal{R}$. 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: $(beg, beg, lck) \rightarrow (n, n, lck)$ if neither $\mathcal{L}$ nor $\mathcal{R}$ holds, $(beg, beg, lck) \rightarrow (n, beg, lo)$ if $\mathcal{L}$ holds, and $(beg, beg, lck) \rightarrow (beg, n, ro)$ if $\mathcal{R}$ holds. Fig. 8 depicts an example.

Theorem 13. *Consider* $c, c', beg, \mathcal{L}, \mathcal{R}$ *such that* $\mathsf{sameCtl}(c, c')$, c *and* c' *are choice-free,* $\mathsf{sub}(beg, c)$ *is a loop, and* $\mathcal{L}, \mathcal{R}$ *are store relations. Let* Π *be the product described above for* $c; \mathsf{skip}^{fin}$, $c'; \mathsf{skip}^{fin}$. *Suppose* an *is a valid full annotation of* Π *for* $\mathcal{P} \rightsquigarrow \mathcal{Q}$. *Assume*

(a) for all branch points $n \in \mathsf{labs}(c) \setminus \{beg\}$ *with branch conditions* e, e', *we have* $an(n, n, lck) \Rightarrow e \mathrel{\mathring{=}} e'$; *and*

(b) $an(beg, beg, lck) \Rightarrow e \mathrel{\mathring{=}} e' \vee (\mathcal{L} \wedge \{\!|e'|\!\}) \vee (\mathcal{R} \wedge \}\!|e|\!\{)$ *where* e, e' *are the tests of the loops at* beg.

Then $\vdash c \mid c' : \mathcal{P} \approx\!\!> \mathcal{Q}$ *in the logic comprised of the rules of Fig. 10, Fig. 12, and* caWhile, *using only the associated judgments.*

$$\textsc{AssSkip}\ \ x := e \mid \mathsf{skip} : \mathcal{R}^{x|}_{e|} \mathrel{\approx\!\!>} \mathcal{R} \qquad \textsc{SkipSkip}\ \ \mathsf{skip} \mid \mathsf{skip} : \mathcal{R} \mathrel{\approx\!\!>} \mathcal{R}$$

$$\textsc{SeqSkip}\ \frac{c \mid \mathsf{skip} : \mathcal{R} \mathrel{\approx\!\!>} \mathcal{Q} \qquad d \mid \mathsf{skip} : \mathcal{Q} \mathrel{\approx\!\!>} \mathcal{S}}{c; d \mid \mathsf{skip} : \mathcal{R} \mathrel{\approx\!\!>} \mathcal{S}}$$

$$\textsc{IfSkip}\ \frac{c \mid \mathsf{skip} : \mathcal{R} \wedge \{\!|e|\!\} \mathrel{\approx\!\!>} \mathcal{S} \qquad d \mid \mathsf{skip} : \mathcal{R} \wedge \{\!|\neg e|\!\} \mathrel{\approx\!\!>} \mathcal{S}}{\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ d\ \mathsf{fi} \mid \mathsf{skip} : \mathcal{R} \mathrel{\approx\!\!>} \mathcal{S}}$$

$$\textsc{WhSkip}\ \frac{c \mid \mathsf{skip} : \mathcal{Q} \wedge \{\!|e|\!\} \mathrel{\approx\!\!>} \mathcal{Q}}{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od} \mid \mathsf{skip} : \mathcal{Q} \mathrel{\approx\!\!>} \mathcal{Q} \wedge \{\!|\neg e|\!\}}$$

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 Π. Assumption (b) is like the side condition of rule caWhile and ensures adequacy.

Instead of showing how $c2$ majorizes $c3$, we consider variations $c4, c5$ in Fig. 2. This lets us avoid complications in the invariant needed to handle the first few iterations of $c2, c3$ (where $z > z'$ does not hold under precondition $x = x'$). Whereas the originals maintain the invariants $x! = z * y!$ (for $c2$) and $2^x = z{*}2^y$ (for $c3$) and $y \geq 0$ (for both), the variations maintain $x!{*}4! = z{*}y!$ and $2^x{*}2^4 = z{*}2^y$ and $y \geq 4$, owing to initializations of $z = 4! = 24$ and $z = 2^4 = 16$. We shall prove $x = x' \wedge x > 3 \mathrel{\approx\!\!>} z > z'$ for $c4; \mathsf{skip}^0$ and $c5; \mathsf{skip}^0$. 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 $(4, 4, lck){\rightarrow}(5, 4, lo)$ is guarded by $\{\!|w \% 2 \neq 0|\!\}$, the transition $(4, 4, lck){\rightarrow}(4, 5, ro)$ is guarded by $|\!\{w' \% 3 \neq 0\}\!|$, and $(4, 4, lck){\rightarrow}(5, 5, lck)$ is guarded by $\neg\mathcal{L} \wedge \neg\mathcal{R}$. As loop invariant we choose

$$\mathcal{S}: \qquad y = y' \wedge y > 3 \wedge z > z' > 0$$

Define the annotation an as follows.

(n, m, tag)	$an(n, m, tag)$	
$(1, 1, lck)$	$x = x' \wedge x > 3$	precondition
$(2, 2, lck)$	$y = y' \wedge y > 3$	
$(3, 3, lck)$	$y = y' \wedge y > 3 \wedge z > z' > 0$	
$(4, 4, lck)$	$\mathcal{S}$	
$(5, 5, lck)$	$\mathcal{S} \wedge y > 4 \wedge \neg\mathcal{L} \wedge \neg\mathcal{R}$	
$(6, 6, lck)$	$\mathcal{S} \wedge y > 4 \wedge w \,\%\, 2 = 0 = w' \,\%\, 3$	
$(7, 7, lck)$	$\mathcal{S} \wedge y > 4 \wedge w \,\%\, 2 = 0 = w' \,\%\, 3$	
$(8, 8, lck)$	$\mathcal{S} \wedge y > 4 \wedge w \,\%\, 2 \neq 0 \neq w' \,\%\, 3$	
$(9, 9, lck)$	$\mathcal{S}$	
$(5, 4, lo)$	$\mathcal{S} \wedge \mathcal{L}$	
$(8, 4, lo)$	$\mathcal{S}$	
$(9, 4, lo)$	$\mathcal{S}$	
$(4, 5, ro)$	$\mathcal{S} \wedge \mathcal{R}$	
$(4, 8, ro)$	$\mathcal{S}$	
$(4, 9, ro)$	$\mathcal{S}$	
$(0, 0, lck)$	$z > z'$	postcondition

Let $an(m, n, tag) = false$ 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 variation 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 [12].

$$\frac{\mathsf{while}\ e \wedge e_0\ \mathsf{do}\ b\ \mathsf{od} \mid c : \mathcal{P} \approx\!\!> \mathcal{Q} \qquad \mathsf{while}\ e\ \mathsf{do}\ b\ \mathsf{od} \mid d : \mathcal{Q} \approx\!\!> \mathcal{R} \qquad \mathcal{Q} \wedge \{\!|\neg e|\!\} \Rightarrow \mathcal{R}}{\mathsf{while}\ e\ \mathsf{do}\ b\ \mathsf{od} \mid c; d : \mathcal{P} \approx\!\!> \mathcal{R}}$$

Several practical works use more sophisticated product constructions [19], [32], including some that use auxiliary store [33], [34]. Clochard et al. [34] highlight the efficacy of (unary) deductive verification applied to product programs, as well as going beyond $\forall\forall$ (as do [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 SeqProd 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 [15], [25], and taking into account the encoding of two stores by one for different data models [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 [24] includes an unconditional rewriting relation $\cong$ and rule to infer $c|c' : \mathcal{R} \approx\!\!> \mathcal{S}$ from $d|d' : \mathcal{R} \approx\!\!> \mathcal{S}$ if $c \cong d$ and $c' \cong d'$. Using simple equivalences like $\mathsf{if}\ E\ \mathsf{then}\ C\ \mathsf{else\ skip\ fi}; \mathsf{while}\ E\ \mathsf{do}\ C\ \mathsf{od} \cong \mathsf{while}\ E\ \mathsf{do}\ C\ \mathsf{od}$, together with a version of caWhile, they prove a loop tiling transformation which had been used to argue for working directly with automata [18]. Unrolling examples $c2$ and $c3$ a few times would address the problem we dodged by using $c4$ and $c5$. Equivalences valid in Kleene algebra with tests [37] 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

[1] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," in *POPL*, 2004.
[2] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *Static Analysis Symposium (SAS)*, 2005.
[3] S. A. Cook, "Soundness and completeness of an axiom system for program verification," *SIAM J. Comput.*, vol. 7, no. 1, 1974.
[4] K. R. Apt, F. S. de Boer, and E.-R. Olderog, *Verification of Sequential and Concurrent Programs*, 3rd ed. Springer, 2009.
[5] T. Nipkow, "Hoare logics for recursive procedures and unbounded nondeterminism," in *Computer Science Logic*, 2002.
[6] B. Godlin and O. Strichman, "Regression verification," in *46th ACM Design Automation Conference*, 2009.
[7] R. Floyd, "Assigning meaning to programs," in *Symposium on Applied Mathematics 19, Mathematical Aspects of Computer Science*, 1967.
[8] A. Turing, "On checking a large routine," in *Report of a Conference on High Speed Automatic Calculating Machines*, 1949.
[9] K. R. Apt and E. Olderog, "Fifty years of Hoare's logic," *Formal Asp. Comput.*, vol. 31, no. 6, 2019.
[10] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs," *Acta Inf.*, vol. 6, 1976.
[11] L. Beringer, "Relational decomposition," in *Interactive Theorem Proving (ITP)*, 2011.
[12] N. Francez, "Product properties and their direct verification," *Acta Informatica*, vol. 20, 1983.
[13] H. Yang, "Relational separation logic," *Theo. Comp. Sci.*, vol. 375, 2007.
[14] B. Beckert and M. Ulbrich, "Trends in relational program verification," in *Principled Software Development*, 2018.
[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.
[16] K. Maillard, C. Hritçu, E. Rivas, and A. V. Muylder, "The next 700 relational program logics," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, 2020.
[17] D. A. Naumann, "Thirty-seven years of relational Hoare logic: remarks on its principles and history," in *ISOLA*, 2020, extended version at https://arxiv.org/abs/2007.06421.
[18] G. Barthe, J. M. Crespo, and C. Kunz, "Beyond 2-safety: Asymmetric product programs for relational program verification," in *LFCS*, 2013.
[19] B. R. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic program alignment for equivalence checking," in *PLDI*, 2019.
[20] M. Heizmann, J. Hoenicke, and A. Podelski, "Software model checking for people who love automata," in *CAV*, 2013.
[21] T. Lange, M. R. Neuhäußer, and T. Noll, "IC3 software model checking on control flow automata," in *FMCAD*, 2015.
[22] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *IEEE CSFW*, 2004, see extended version [38].
[23] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *Formal Methods*, 2011.
[24] A. Banerjee, D. A. Naumann, and M. Nikouei, "Relational logic with framing and hypotheses," in *FSTTCS*, 2016, technical report at https://arxiv.org/abs/1611.08992.
[25] A. Banerjee, R. Nagasamudram, M. Nikouei, and D. A. Naumann, "A relational program logic with data abstraction and dynamic framing," 2019, available at https://arxiv.org/abs/1910.14560.
[26] G. Barthe, B. Grégoire, J. Hsu, and P. Strub, "Coupling proofs are probabilistic product programs," in *POPL*, 2017.
[27] M. Sousa and I. Dillig, "Cartesian Hoare Logic for verifying k-safety properties," in *PLDI*, 2016.
[28] Y. Wang, I. Dillig, S. K. Lahiri, and W. R. Cook, "Verifying equivalence of database-driven applications," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 2018.
[29] M. Eilers, P. Müller, and S. Hitz, "Modular product programs," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 1, 2020.
[30] A. Aguirre, G. Barthe, M. Gaboardi, D. Garg, and P. Strub, "A relational logic for higher-order programs," *J. Funct. Program.*, vol. 29, 2019.
[31] M. Kovács, H. Seidl, and B. Finkbeiner, "Relational abstract interpretation for the verification of 2-hypersafety properties," in *CCS*, 2013.
[32] R. Shemer, A. Gurfinkel, S. Shoham, and Y. Vizel, "Property directed self composition," in *CAV*, 2019.
[33] T. Girka, D. Mentré, and Y. Régis-Gianas, "Verifiable semantic difference languages," in *PPDP*, 2017.
[34] M. Clochard, C. Marché, and A. Paskevich, "Deductive verification with ghost monitors," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, 2020.
[35] D. A. Naumann, "From coupling relations to mated invariants for secure information flow," in *ESORICS*, 2006.
[36] L. Beringer and M. Hofmann, "Secure information flow and program logics," in *IEEE CSF*, 2007.
[37] D. Kozen, "On Hoare logic and Kleene algebra with tests," *ACM Trans. Comput. Log.*, vol. 1, no. 1, 2000.
[38] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," *Math. Struct. Comput. Sci.*, vol. 21, no. 6, 2011.