COMPUTING SCIENCE

Refining rely-guarantee thinking

Ian J. Hayes, Cliff B. Jones and Robert J. Colvin

TECHNICAL REPORT SERIES

No. CS-TR-1334 May 2012

TECHNICAL REPORT SERIES

No. CS-TR-1334 May, 2012

Refining rely-guarantee thinking

I.J. Hayes, C.B. Jones and R.J. Colvin

Abstract

Reasoning about concurrent programs can be very difficult due to the possibility of interference. The fundamental insight of Rely-Guarantee thinking is that developing concurrent designs can only be made compositional if the development method offers ways to record and reason about the interference that is inherent in concurrency. The original presentation of rely-guarantee rules used keywords to mark the various predicates and even the read/write frames of operations. Subsequent papers have moved to a more general message of "rely-guarantee thinking" but retained this VDM flavour and have typically presented a development style in terms of inference rules based on Hoare-like triples, extended to quintuples to accommodate rely and guarantee conditions. Morgan's refinement calculus presents concise rules that lend themselves to algebraic arguments. This paper reports on a complete reformulation of the key ideas of rely-guarantee reasoning in a refinement calculus style. As is shown, this indicates new useful and intuitive manipulations of rely/guarantee specifications. The approach makes use of two new commands: a guarantee command (guar g _ c) that behaves like the command c but also guarantees every atomic step satisfies the relation g, and a rely command (rely r _ c) that behaves like c provided any interference steps from the environment satisfy the relation r or stutter. Further notational developments result from the use of a more compact notation to indicate the read/write frame of a command. The new rules are justified with respect to an operational semantics presented in the Colvin style.

Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

HAYES, I.J., JONES, C.B., COLVIN, R.J.

Refining rely-guarantee thinking
[By] I.J. Hayes, C.B. Jones, R.J. Colvin
Newcastle upon Tyne: Newcastle University: Computing Science, 2012.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1334)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1334

Abstract

Reasoning about concurrent programs can be very difficult due to the possibility of interference. The fundamental insight of Rely-Guarantee thinking is that developing concurrent designs can only be made compositional if the development method offers ways to record and reason about the interference that is inherent in concurrency. The original presentation of rely-guarantee rules used keywords to mark the various predicates and even the read/write frames of operations. Subsequent papers have moved to a more general message of “rely-guarantee thinking” but retained this VDM flavour and have typically presented a development style in terms of inference rules based on Hoare-like triples, extended to quintuples to accommodate rely and guarantee conditions. Morgan’s refinement calculus presents concise rules that lend themselves to algebraic arguments. This paper reports on a complete reformulation of the key ideas of rely-guarantee reasoning in a refinement calculus style. As is shown, this indicates new useful and intuitive manipulations of rely/guarantee specifications. The approach makes use of two new commands: a guarantee command (guar g _ c) that behaves like the command c but also guarantees every atomic step satisfies the relation g, and a rely command (rely r _ c) that behaves like c provided any interference steps from the environment satisfy the relation r or stutter. Further notational developments result from the use of a more compact notation to indicate the read/write frame of a command. The new rules are justified with respect to an operational semantics presented in the Colvin style.

About the authors

Ian Hayes is a Professor in the School of Information Technology and Electrical Engineering at the University of Queensland. His research interests are in the field of formal methods for the specification and development of software, especially for real-time systems. His current research is on the use of time bands and teleo-reactive programming for the development of real-time systems. He is a fellow of the British Computer Society.

Cliff Jones is a Professor of Computing Science at Newcastle University. He is now applying research on formal methods to wider issues of dependability. Until 2007 his major research involvement was the five university IRC on "Dependability of Computer-Based Systems" of which he was overall Project Director - he is now PI of the follow-on Platform Grant "Trustworthy Ambient Systems" (TrAmS) (also EPSRC). He is also PI on an EPSRC-funded project "Splitting (Software) Atoms Safely" and coordinates the "Methodology" strand of the EU-funded RODIN project. As well as his academic career, Cliff has spent over twenty years in industry. His fifteen years in IBM saw among other things the creation -with colleagues in Vienna- of VDM which is one of the better known "formal methods". Under Tony Hoare, Cliff wrote his doctoral thesis in two years (and enjoyed the family atmosphere of Wolfson College). From Oxford, he moved directly to a chair at Manchester University where he built a world-class Formal Methods group which -among other projects- was the academic lead in the largest Software Engineering project funded by the Alvey programme (IPSE 2.5 created the "mural"(Formal Method) Support Systems theorem proving assistant). Cliff is a Fellow of the Royal Academy of Engineering (FREng), ACM, BCS, and IET. He has been a member of IFIP Working Group 2.3 (Programming Methodology) since 1973 (and was Chair from 1987-96).

Robert Colvin uses formal specification languages to abstract, generalise, and integrate theories of brain activity. In collaboration with Ian Hayes he has developed a style of formal semantics that allows straightforward integration of state and synchronous communication, and has used this as the base for describing hybrid and real-time behaviour. His current interest is in integrating theories about how the brain tells time with the behaviour of Pavlovian conditioning.

Suggested keywords

FORMAL METHODS
CONCURRENCY
RELY/GUARANTEE REASONING
SPECIFICATION STATEMENTS
INVARIANTS

Under consideration for publication in Formal Aspects of Computing

Refining rely-guarantee thinking

Ian J. Hayes[1] Cliff B. Jones[2] and Robert J. Colvin[3]

[1]School of Information Technology and Electrical Engineering, The University of Queensland, Australia
[2]School of Computing Science, Newcastle University, UK
[3]Queensland Brain Institute, The University of Queensland, Australia

Abstract. Reasoning about concurrent programs can be very difficult due to the possibility of interference. The fundamental insight of Rely-Guarantee thinking is that developing concurrent designs can only be made compositional if the development method offers ways to record and reason about the interference that is inherent in concurrency. The original presentation of rely-guarantee rules used keywords to mark the various predicates and even the read/write frames of operations. Subsequent papers have moved to a more general message of "rely-guarantee thinking" but retained this VDM flavour and have typically presented a development style in terms of inference rules based on Hoare-like triples, extended to quintuples to accommodate rely and guarantee conditions. Morgan's refinement calculus presents concise rules that lend themselves to algebraic arguments. This paper reports on a complete reformulation of the key ideas of rely-guarantee reasoning in a refinement calculus style. As is shown, this indicates new useful and intuitive manipulations of rely/guarantee specifications. The approach makes use of two new commands: a guarantee command $(\mathbf{guar}\ g \bullet c)$ that behaves like the command c but also guarantees every atomic step satisfies the relation g, and a rely command $(\mathbf{rely}\ r \bullet c)$ that behaves like c provided any interference steps from the environment satisfy the relation r or stutter. Further notational developments result from the use of a more compact notation to indicate the read/write frame of a command. The new rules are justified with respect to an operational semantics presented in the Colvin style.

1. Introduction

The rely-guarantee rules of [Jon81, Jon83] provide a compositional approach to reasoning about concurrent processes (the most accessible reference is [Jon96]; an exhaustive analysis of various compositional and non-compositional approaches can be found in [dR01]). Based on many other contributions such as [CJ00, CJ07] these rules have been absorbed into a more general rely-guarantee "thinking" as exemplified in [JP11]. The basic idea is simple: in order to develop a process π divorced from its surrounding components one needs to take into account interference from the processes which form the (parallel) environment of π. This is done by assuming that any interleaving step of the environment of π satisfies a rely condition r. The rely condition records assumptions the developer is invited to make about possible interference from the environment. Conversely, each process is associated with a guarantee condition, which must be proved to be the limit of interference that it can inflict on the other processes in its environment. Both rely and guarantee are expressed as binary relations over states.

When an (abstract) specification is refined into a parallel composition of specified components, each component specification, s, has an associated rely condition, r, and guarantee condition, g. A specification s is refined assuming the interference steps of the environment are bounded by r, but the refinement must also ensure every atomic step taken by the implementation is bounded by g.

The aim of this paper is to represent rely-guarantee thinking in a refinement calculus context. There is from the

Correspondence and offprint requests to: Ian.Hayes@itee.uq.edu.au

outset a good match in that both VDM and the refinement calculus use post conditions that are relations over pairs of states and the refinement calculus offers a compact notation for expressing the read/write frames of VDM. The refinement calculus [Bac81, Mor88, Mor94, Bv98, Mor87] is described in Section 2.2 below.

To simplify reasoning about types and invariants in the refinement calculus, Carroll Morgan and Trevor Vickersintroduced an "invariant command": $(\mathbf{inv}\, p \bullet c)$. A step of the execution of c is also allowed by $(\mathbf{inv}\, p \bullet c)$ only if the step maintains the invariant p (a predicate of a single state). If in a particular state the only steps available to c would all break p, then $(\mathbf{inv}\, p \bullet c)$ is infeasible (in refinement calculus terms), also known as unsatisfiable (in VDM terms).

The way in which an invariant constrains a program is similar to the way in which a guarantee constrains a program: an invariant constrains each state, while a guarantee constrains each atomic transition between states. This was already noted in [CJ00] but here the similarity is taken further in that a novel command of the form $(\mathbf{guar}\, g \bullet c)$ is introduced. The idea behind this new command was motivated by the analogy with the invariant command of Morgan and Vickers. The command $(\mathbf{guar}\, g \bullet c)$ behaves as c but it only allows atomic steps which either stutter or satisfy the relation g between their before-state and after-state. Any step that c alone could take that does not stutter or satisfy g is not a valid step for $(\mathbf{guar}\, g \bullet c)$. If in a particular state the only steps available to c would not stutter nor satisfy g, then $(\mathbf{guar}\, g \bullet c)$ is not feasible. The stuttering steps allow a process to perform internal steps that do not affect the shared state.

The refinement calculus makes use of a specification command of the form $[p,\ q]$, in which p is a predicate giving its precondition and q is a relation (expressed as a two-state predicate) giving its postcondition [Mor88]. When started in a state satisfying p any implementation of $[p,\ q]$ must terminate in a state satisfying q with respect to the starting state. For any initial state that satisfies p the command $(\mathbf{guar}\, g \bullet [p,\ q])$ not only satisfies the postcondition q but also only uses atomic steps which each satisfy the relation g. At this level there is no particular notion of granularity of atomicity; all that is required is that the atomic steps (whatever they turn out to be) of any implementation of $(\mathbf{guar}\, g \bullet [p,\ q])$ all individually satisfy g, while the complete sequence of steps satisfies q.

The main advantage in introducing the guarantee command is that it facilitates the separation the concern of refining a command from that of showing that the refined code adheres to a guarantee. Because the guarantee command is monotonic with respect to refinement of the command in its body, standard refinement laws can be used to refine its body. A separate set of refinement laws is used to distribute and eliminate the guarantee. In saying this, there is no suggestion that the advantages of "rely-guarantee thinking" in providing a top-down development method should be forgotten. There is one caveat though: a valid refinement of the body may introduce steps that become infeasible when constrained by the guarantee. This means that in doing the refinement one needs to be aware of the enclosing guarantee context in order to ensure that the refinement respects it. Section 3 explores guarantee commands in detail.

The second novel construct is a command of the form $(\mathbf{rely}\, r \bullet c)$, which reflects c being "implemented" in an environment that can impose interference that respects the rely condition r. The relation r records an assumption about every atomic step of the environment of the command: either the step satisfies r or it stutters. The stronger the rely condition the more constrained the environment and hence the easier it is to implement c. The empty relation is the strongest rely condition: it represents only stuttering interference from the environment. Common rely conditions are those that require certain variables are not modified, or those that restrict the way in which variables may change (e.g. only increase). Section 4 explores rely commands in detail as well as combining rely and guarantee commands.

Section 5 explores refinement laws that introduce parallelism using both rely and guarantee constructs. The laws for introducing parallelism are proved making use of the more fundamental laws for rely and guarantee constructs developed in the earlier sections. The semantics of the language is given in Section 7. Section 6 applies the laws to an example. First the programming language and refinement are introduced in Section 2.

2. Programming language and refinement

2.1. Syntax

Assume a set of variables *Var* and values *Val*. The syntax of expressions and commands is given in Figure 1. The command **nil** represents the terminated program, **abort** has any terminating or nonterminating behaviour, $\{p\}$ represents a precondition assumption, $[q]$ is the specification command with postcondition the relation q, the operator "$\sqcap$" is used for nondeterministic choice, "$\Cap$" is used for command conjunction (a specification rather than implementation construct), c^* is c iterated zero or more but finitely many times (nondeterministically), $\langle\langle r \rangle\rangle$ executes a single atomic step that satisfies the relation r. The command $[p,\ q]$ is an abbreviation for $(\{p\}\ ;\ [q])$. Sequential composition has a higher precedence than the other binary operators and can be omitted when no confusion arises, e.g. $(\{p\}\ ;\ [q])$ can be written as $\{p\}\,[q]$. The other commands are explained in the paper.

Let $x \in Var$, $v \in Val$, "$\oplus$" stand for a binary operator and "$\ominus$" stand for a unary operator.

$$e ::= v \mid x \mid e \oplus e \mid \ominus e$$

Let p be a predicate, e an expression, b a boolean expression, q, g and r relations, and x a variable.

$$c ::= \mathbf{nil} \mid \mathbf{abort} \mid \{p\} \mid [q] \mid [p, q] \mid x := e \mid c_1 \,;\, c_2 \mid c_1 \parallel c_2 \mid c_1 \sqcap c_2 \mid c_1 \Cap c_2 \mid c^* \mid \langle\langle r \rangle\rangle \mid (\mathbf{guar}\, g \bullet c) \mid (\mathbf{rely}\, r \bullet c) \mid \mathbf{if}\, b\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2 \mid \mathbf{while}\, b\, \mathbf{do}\, c$$

Fig. 1. Syntax of expressions and commands

The syntax of predicates and relations is not given in detail here. A *relation* is expressed as a predicate over a pair of states: the before state is represented by unprimed variables and the after state by primed variables. The notation $p_0 \Rrightarrow p_1$ means p_0 implies p_1 for all states, and $q_0 \Rrightarrow q_1$ means q_0 implies q_1 for pairs of before and after states. Central to the semantics is the notion of composing relations, i.e.

$$q_0 \fatsemi q_1 = (\exists\, Var'' \bullet q_0[Var''/Var'] \wedge q_1[Var''/Var])$$

in which the final state variables of the first relation (Var') and the initial state variables of the second relation (Var) are identified by replacing them both with fresh intermediate state variables Var''. The reflexive, transitive closure of a relation r is denoted by r^*. Note that $(r \vee \mathrm{id})^* = r^*$.

2.2. Program refinement

The refinement calculus [Bac81, Mor88, Mor94, MV94, Bv98, Mor87] provides a systematic approach to program development based on step-wise refinement from a specification to code. It treats a specification as a command $[p,\ q]$ in the language. In this paper as well as the specification command we introduce two new commands $(\mathbf{guar}\, g \bullet c)$ and $(\mathbf{rely}\, r \bullet c)$ which extend the expressive power of specifications to allow reasoning about interference between concurrently executing commands. The most general form of specification is thus $(\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p,\ q]))$. The statement that this specification is refined by a command c is written

$$(\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p,\ q])) \sqsubseteq c$$

and is equivalent to the Jones-style Hoare logic statement $\{p, r\}\ c\ \{g, q\}$. Dingel [Din00, Din02] has also produced a refinement calculus that supports rely-guarantee style reasoning. One difference between his approach and that presented here is that he used a four-tuple of pre, rely, guarantee and post conditions rather than the separate rely and guarantee constructs used here (see Sect. 8 for further comparison with Dingel's research).

In the standard sequential refinement calculus refinement of a command c by a command d, written $c \sqsubseteq d$, is defined in terms of weakest precondition predicate transformers. Because we wish to deal with concurrent execution of processes, this form of semantics is inadequate. Instead we use an operational semantics based on traces formed from atomic steps which is detailed in Section 7. The guarantee and rely commands are introduced via their defining properties, which are justified with respect to the semantics. As the new constructs are introduced we describe how their semantics differs from that in the standard sequential refinement calculus. For program constructs not involving relies, guarantees and concurrency their semantics reduces to be almost equivalent to their semantics in the sequential refinement calculus and hence the more familiar laws of that calculus can be used in those contexts. We make use of some basic laws from the refinement calculus which are still valid in our extended language.

Law 1 (consequence). For any predicates p_0 and p_1, and relations q_0 and q_1, provided $p_0 \Rrightarrow p_1$ and $p_0 \wedge q_1 \Rrightarrow q_0$,

$$[p_0,\ q_0] \sqsubseteq [p_1,\ q_1]$$

3. The guarantee command

Intuition for the semantics of a command of the form $(\mathbf{guar}\, g \bullet c)$ is provided by some examples of guarantee commands and their refinements.

3.1. Examples

The laws referred to here are given in Section 3.2.

1. Refine a specification enclosed in a guarantee by an assignment which respects the guarantee and implements the specification.

   $\mathbf{guar}\, x < x' \bullet x : [x' = x + 1]$
   $\sqsubseteq$ by Law 16 (guarantee-introduce-assignment) as $x' = x + 1 \Rrightarrow x < x'$
   $x := x + 1$

2. Use two atomic steps that both satisfy the guarantee.

   $\mathbf{guar}\, x < x' \bullet x : [x' = x + 2]$
   $\sqsubseteq$ by Law 3 (guarantee-monotonic) as $x : [x' = x + 2] \sqsubseteq (x : [x' = x + 1]\ ;\ x : [x' = x + 1])$
   $\mathbf{guar}\, x < x' \bullet (x : [x' = x + 1]\ ;\ x : [x' = x + 1])$
   $=$ by Law 8 (distribute-guarantee) part (1) sequential
   $(\mathbf{guar}\, x < x' \bullet x : [x' = x + 1])\ ;\ (\mathbf{guar}\, x < x' \bullet x : [x' = x + 1])$
   $\sqsubseteq$ by example 1 above (twice)
   $x := x + 1\ ;\ x := x + 1$

3. A guarantee may restrict a choice. As every atomic step must satisfy the relation $x < x'$ or stutter, the whole must satisfy the reflexive transitive closure of this relation: $(x < x')^*$.

   $\mathbf{guar}\, x < x' \bullet [x' = x + 1 \lor x' = x - 1]$
   $=$ by Law 10 (trading-post-guar))
   $\mathbf{guar}\, x < x' \bullet [(x' = x + 1 \lor x' = x - 1) \land (x < x')^*]$
   $=$ as $(x < x')^* = (x \leq x')$
   $\mathbf{guar}\, x < x' \bullet [(x' = x + 1 \lor x' = x - 1) \land x \leq x']$
   $= \mathbf{guar}\, x < x' \bullet [x' = x + 1]$

4. A specification constrained by a guarantee g cannot be implemented if there is no sequence of atomic steps satisfying g that satisfy the postcondition of the specification overall.

   $\mathbf{guar}\, x < x' \bullet [x' = x - 1]$
   $=$ by Law 10 (trading-post-guar)
   $\mathbf{guar}\, x < x' \bullet [x' = x - 1 \land (x < x')^*]$
   $= \mathbf{guar}\, x < x' \bullet [x' = x - 1 \land (x \leq x')]$
   $= \mathbf{guar}\, x < x' \bullet [\mathsf{false}]$

3.2. Laws for refining guarantee commands

The following laws are straightforward to prove from the semantics of guarantee.

Law 2 (guarantee-true). For any command c, $\quad (\mathbf{guar}\, \mathsf{true} \bullet c) = c$.

Law 3 (guarantee-monotonic). For any commands c and d, and relation g,

$c \sqsubseteq d \quad \Rightarrow \quad (\mathbf{guar}\, g \bullet c) \sqsubseteq (\mathbf{guar}\, g \bullet d)$.

Law 4 (strengthen-guarantee). For any command c and relations g_0 and g_1,

$(g_0 \lor \mathrm{id} \Rrightarrow g_1) \quad \Rightarrow \quad (\mathbf{guar}\, g_1 \bullet c) \sqsubseteq (\mathbf{guar}\, g_0 \bullet c)$.

Law 5 (guarantee-stutter). For any relation g and command c, $\quad (\mathbf{guar}\, g \bullet c) = (\mathbf{guar}\, g \lor \mathrm{id} \bullet c)$.

Law 6 (introduce-guarantee). For any command c and relation g, $\quad c \sqsubseteq (\mathbf{guar}\, g \bullet c)$.

Law 7 (nested-guarantees). For a command c and relations g_0 and g_1,

$$(\mathbf{guar}\, g_0 \bullet (\mathbf{guar}\, g_1 \bullet c)) = (\mathbf{guar}\, g_0 \wedge g_1 \bullet c)\ .$$

A guarantee on a composite command may be distributed to its component commands.

Law 8 (distribute-guarantee). For any relation g and commands c and d the following hold.

$$\mathbf{guar}\, g \bullet (c\ ;\ d) = (\mathbf{guar}\, g \bullet c)\ ;\ (\mathbf{guar}\, g \bullet d) \tag{1}$$
$$\mathbf{guar}\, g \bullet (c \parallel d) = (\mathbf{guar}\, g \bullet c) \parallel (\mathbf{guar}\, g \bullet d) \tag{2}$$
$$\mathbf{guar}\, g \bullet (c \sqcap d) = (\mathbf{guar}\, g \bullet c) \sqcap (\mathbf{guar}\, g \bullet d) \tag{3}$$
$$\mathbf{guar}\, g \bullet (c \Cap d) = (\mathbf{guar}\, g \bullet c) \Cap (\mathbf{guar}\, g \bullet d) \tag{4}$$
$$\mathbf{guar}\, g \bullet (c^*) = (\mathbf{guar}\, g \bullet c)^* \tag{5}$$

A precondition within a guarantee is a precondition of the whole.

Law 9 (guarantee-precondition). For any (precondition) predicate p, relation g and command c,

$$(\mathbf{guar}\, g \bullet \{p\}c) = \{p\}(\mathbf{guar}\, g \bullet c)\ .$$

The execution of any command enclosed in a guarantee consists of zero or more atomic execution steps each of which must satisfy g and hence any such execution sequence satisfies the reflexive, transitive closure of g. The following law allows a postcondition ensuring g^* to be traded for a guarantee of g on every atomic step.

Law 10 (trading-post-guar). For any relations g and q,

$$(\mathbf{guar}\, g \bullet [q \wedge g^*]) = (\mathbf{guar}\, g \bullet [q])\ .$$

A guarantee g on an atomic step that satisfies q must satisfy both q and the guarantee.

Law 11 (guarantee-atomic). For relations g and q,

$$(\mathbf{guar}\, g \bullet \langle\!\langle q \rangle\!\rangle) = \langle\!\langle q \wedge (g \vee \mathrm{id}) \rangle\!\rangle\ .$$

Definition 12 (feasible-guarantee-specification). A specification command within a guarantee $(\mathbf{guar}\, g \bullet [p,\ q])$ is feasible if for all states that satisfy the precondition, there exists some final state that satisfies the postcondition and the reflexive transitive closure of the guarantee, i.e. $p \Rrightarrow (\exists\, Var' \bullet q \wedge g^*)$.

The refinement calculus as described by Carroll Morgan [Mor94] includes a version of a specification command $x\!:[p,\ q]$ with an explicit frame x representing the set of variables that may be changed by it. Using the notation $\bar{x}$ to stand for all variables other than x, and $\mathrm{id}(\bar{x})$ to stand for the identity relation on all variables other than x, in the sequential refinement calculus $x\!:[p,\ q]$ is defined as $[p,\ q \wedge \mathrm{id}(\bar{x})]$ where this specification implicitly has all variables in its frame. In the context of concurrent programs this definition is not strong enough as it allows the following refinement,

$$x\!:[x' = y + 1] \sqsubseteq y := y + 1;\ x := y;\ y := y - 1$$

which, although it modifies y, leaves its final value the same as its initial value and hence satisfies the specification on the left. If a concurrent process accesses y during the execution this may lead to unexpected results. Hence we suggest the following more suitable definition in the context of concurrency, which does not allow the above refinement.

Definition 13 (specification-with-frame). For any set of variables x, predicate p and relation q,

$$x\!:[p,\ q] \mathrel{\widehat{=}} (\mathbf{guar}\, \mathrm{id}(\bar{x}) \bullet [p,\ q])\ .$$

Law 14 (assignment). For any variable x and expression e,

$$x\!:[def(e),\ x' = e] \sqsubseteq x := e$$

where $def(e)$ is a predicate that is true in just those states in which e is well defined, e.g. the expression x/y is well defined in states in which $y \neq 0$.

In the standard sequential refinement calculus the refinement in Law 14 is an equivalence but here it is a strict refinement. The standard refinement calculus gives the following equivalences.

$$x := x + 2 = x\!:[x' = x + 2] = (x\!:[x' = x + 1]\ ;\ x\!:[x' = x + 1]) = (x := x + 1\ ;\ x := x + 1)$$

However, in the context of guarantees, the above cannot be considered equivalent because $x := x + 2$ satisfies a guarantee like $even(x) \Rightarrow even(x')$, whereas $x := x + 1 \; ; \; x := x + 1$ does not (and hence assuming x is initially even, a process running in parallel with $x := x + 1 \; ; \; x := x + 1$ may observe an odd value of x, whereas a process running in parallel with $x := x + 2$ will not).

The (operational) semantics of an assignment statement $x := e$ evaluates its expression to a value v via a number of stuttering steps and then assigns x that value (see Section 7). No variables other than x may be modified and the only modification allowed to x is to set it to the new value; this rules out an assignment that sets x to some intermediate value before assigning x its final value v. This interpretation is required in the context of concurrency because if x can be set to an intermediate value, a parallel process may observe the intermediate value and alter its behaviour.

Law 15 (guarantee-assignment). For any predicate p, relation g, variable x and expression e, such that $p \wedge x' = e \wedge \mathrm{id}(\bar{x}) \Rrightarrow g \vee \mathrm{id}$,

$$(\mathbf{guar}\, g \bullet \{p\}x := e) \sqsubseteq x := e \; .$$

Law 16 (guarantee-introduce-assignment). For a predicate p, relations g and q, variable x and expression e, such that $p \wedge x' = e \wedge \mathrm{id}(\bar{x}) \Rrightarrow def(e) \wedge q \wedge (g \vee \mathrm{id})$,

$$(\mathbf{guar}\, g \bullet x{:}[p,\ q]) \sqsubseteq x := e \; .$$

Proof. A number of steps in the proof rely on Law 3 (guarantee-monotonic).

$$\begin{array}{ll} & (\mathbf{guar}\, g \bullet x{:}[p,\ q]) \\ = & \quad \text{by Law 1 (consequence) as } p \wedge x' = e \wedge \mathrm{id}(\bar{x}) \Rrightarrow def(e) \wedge q \\ & (\mathbf{guar}\, g \bullet \{p\}x{:}[def(e),\ x' = e]) \\ \sqsubseteq & \quad \text{by Law 14 (assignment)} \\ & (\mathbf{guar}\, g \bullet \{p\}x := e) \\ \sqsubseteq & \quad \text{by Law 15 (guarantee-assignment) as } p \wedge x' = e \wedge \mathrm{id}(\bar{x}) \Rrightarrow g \vee \mathrm{id} \\ & x := e \end{array}$$

□

Law 17 (conditional). For any relation g, boolean expression b, and commands c and d,

$$(\mathbf{guar}\, g \bullet \mathbf{if}\, b\, \mathbf{then}\, c\, \mathbf{else}\, d) = \mathbf{if}\, b\, \mathbf{then}\, (\mathbf{guar}\, g \bullet c)\, \mathbf{else}\, (\mathbf{guar}\, g \bullet d) \; .$$

Law 18 (loop). For any relation g, boolean expression b, and command c,

$$(\mathbf{guar}\, g \bullet \mathbf{while}\, b\, \mathbf{do}\, c) = \mathbf{while}\, b\, \mathbf{do}(\mathbf{guar}\, g \bullet c) \; .$$

3.3. Guarantee invariants

A special case of a guarantee relation is that a single-state predicate is an invariant of every atomic step. The following notation is used as an abbreviation in this case.

Definition 19 (guarantee-invariant). For a single-state predicate p and command c,

$$(\mathbf{guar\text{-}inv}\, p \bullet c) \mathrel{\widehat{=}} (\mathbf{guar}(p \Rightarrow p') \bullet c)$$

where by convention p' stands for the predicate p with all program variables (Var) replaced by their primed counterparts (Var'), i.e. p in the after state.

An interesting aspect of an invariant predicate is that the relation $(p \Rightarrow p')$ is both reflexive and transitive and hence

$$(p \Rightarrow p')^* \equiv (p \Rightarrow p') \tag{6}$$

that is, for any number of steps (zero or more) if each step maintains the invariant, then the whole does [Hay10]. This fact can be combined with Law 10 (trading-post-guar) to give the following law.

Law 20 (trading-post-guarantee-invariant). For any single state predicates p_0 and p_1, and relation q,

$$[p_0,\ (p_1 \Rightarrow p_1') \wedge q] \sqsubseteq (\mathbf{guar\text{-}inv}\, p_1 \bullet [p_0,\ q])$$

The effect of this law is to take a property that is required to be invariant overall and require it to be invariant for every atomic step of the computation – a stronger requirement.

Proof.

$$
\begin{array}{ll}
& [p_0,\ (p_1 \Rightarrow p_1') \wedge q] \\
\sqsubseteq & \text{by Law 1 (consequence) as post condition is strengthened using (6)} \\
& [p_0,\ (p_1 \Rightarrow p_1')^* \wedge q] \\
\sqsubseteq & \text{by Law 10 (trading-post-guar)} \\
& \mathbf{guar}(p_1 \Rightarrow p_1') \bullet [p_0,\ q] \\
= & \text{by Definition 19 (guarantee-invariant)} \\
& \mathbf{guar\text{-}inv}\, p_1 \bullet [p_0,\ q]
\end{array}
$$

□

3.4. Extended example (sequential version)

By way of example, the following uses guarantee invariants to give a somewhat unconventional derivation of a program that, given an array v, finds the least index t for which a predicate p holds,[1] and if p does not hold for any element of v, sets t to $len(v)+1$. As shown in Sect. 6, the unconventional approach is useful in the context of deriving a concurrent implementation from the same specification. The specification of the *findp* program is

$$findp \mathrel{\widehat{=}} t : [(t' = len(v) + 1 \vee satp(v, t')) \wedge notp(v, \mathrm{dom}(v), t')] \qquad \lhd$$

where

$$
\begin{array}{l}
satp(v,t) \mathrel{\widehat{=}} t \in \mathrm{dom}(v) \wedge p(v(t)) \\
notp(v,s,t) \mathrel{\widehat{=}} (\forall\, i \in s \bullet i < t \Rightarrow \neg\, p(v(t))) \wedge 1 \leq t \leq len(v) + 1
\end{array}
$$

The first refinement step introduces an invariant $(t = len(v) + 1 \vee satp(v,t))$ and an initialisation of t that establishes the invariant.[2] We follow the convention that a refinement applies to the most recent command marked with "$\lhd$".

$$
\begin{array}{l}
\sqsubseteq t : [t' = len(v) + 1]\,; \\
\quad t : [t = len(v) + 1,\ ((t = len(v) + 1 \vee satp(v,t)) \Rightarrow (t' = len(v) + 1 \vee satp(v,t'))) \wedge notp(v, \mathrm{dom}(v), t')] \qquad \lhd
\end{array}
$$

The second specification can be refined using Law 20 (trading-post-guarantee-invariant).

$$
\begin{array}{l}
\sqsubseteq \mathbf{guar\text{-}inv}\, t = len(v) + 1 \vee satp(v,t) \bullet \\
\qquad t : [t = len(v) + 1,\ notp(v, \mathrm{dom}(v), t')] \qquad \lhd
\end{array}
$$

The body of this involves a quantification within *notp* which can be refined using a loop with fresh control variable c and loop invariant $notp(v, \mathrm{dom}(v), c)$. The invariant is trivially established if c is set to one.

$$
\begin{array}{l}
\sqsubseteq c := 1; \\
\quad c, t : [c = 1 \wedge t = len(v) + 1,\ (notp(v, \mathrm{dom}(v), c) \Rightarrow notp(v, \mathrm{dom}(v), c')) \wedge t' = c'] \qquad \lhd
\end{array}
$$

This can again be refined using Law 20 (trading-post-guarantee-invariant).

$$
\begin{array}{l}
\sqsubseteq \mathbf{guar\text{-}inv}\, notp(v, \mathrm{dom}(v), c) \bullet \\
\qquad c, t : [c = 1 \wedge t = len(v) + 1,\ t' = c'] \qquad \lhd
\end{array}
$$

This would appear to say that all that is required is that the final values of t and c are equal, however, this is in the context of two guarantee invariants $t = len(v)+1 \vee satp(v,t)$ and $notp(v, \mathrm{dom}(v), c)$, both of which must be preserved by every atomic step. The body can be refined to a loop with an invariant describing the bounds on c where

$$bnd(c,t,v) \mathrel{\widehat{=}} 1 \leq c \leq t \leq len(v) + 1$$

[1] For brevity, it is assumed here that $p(x)$ is always defined (undefinedness is considered in [CJ07] but it has little bearing on the actual design).

[2] It is pointed out in [Jon10] that it is not, in general, good practice to copy the post condition of initialisation into the pre condition of what follows; what can be classed as "insipid development" is tolerated here since it is long-winded to define the appropriate subsets of the indices of v.

and a well founded relation reduces the difference $t - c$.

$\sqsubseteq$ **while** $c < t$ **do**
$\quad c, t : [bnd(c,t,v),\ bnd(c,t,v) \land t' - c' < t - c]$ $\lhd$

Note that as this is in the context of the two guarantee invariants both guarantee invariants are effectively invariants of the loop. The variant can be decreased by either increasing c or decreasing t. If c is increased the invariant $notp(v, \mathrm{dom}(v), c)$ must be maintained. To increase c by one this requires $\neg\, p(v(c))$. If t is decreased the invariant $t' = len(v) + 1 \lor satp(v, t')$ must be maintained. To decrease t to c this requires $p(v(c))$. Hence the body of the loop is refined by

$\sqsubseteq$ **if** $p(v(c))$ **then** $t : [t' = c]$ **else** $c : [c' = c + 1]$

If the guarantee invariants are distributed into the program this becomes.

if $p(v(c))$ **then**
$\quad$ **guar-inv**$(t = len(v) + 1 \lor satp(v,t)) \land notp(v, \mathrm{dom}(v), c) \bullet t : [t' = c]$
else
$\quad$ **guar-inv**$(t = len(v) + 1 \lor satp(v,t)) \land notp(v, \mathrm{dom}(v), c) \bullet c : [c' = c + 1]$

The branches of the conditional can be refined using Law 16 (guarantee-introduce-assignment) to the the following final program.

$t := len(v) + 1\ ;\ c := 1;$
while $t \neq c$ **do**
$\quad$ **if** $p(v(c))$ **then** $t := c$ **else** $c := c + 1$

4. The rely command

Given a parallel composition $(c \parallel d)$, it is not possible to use the sequential refinement laws to refine one branch, say c, because d may interfere with execution of c by modifying variables shared between c and d. In the context of Hoare logic, Jones [Jon81, Jon83] addressed this issue by introducing the notion of a *rely* condition, which is a relation r that bounds the possible interference caused by d. Every atomic step of d is required to satisfy the relation r.

A new command $(\textbf{rely}\, r \bullet c)$ is introduced; when it is put in an environment in which every atomic interference step satisfies the relation r or stutters, the composite behaviour implements c. The interference can be represented by the process $\langle\!\langle r \lor \mathrm{id} \rangle\!\rangle^*$, that is, the process that can do any finite number, zero or more, of atomic steps, each of which satisfies the relation r or stutters. The defining property of the rely command is

$$c \sqsubseteq (\textbf{rely}\, r \bullet c) \parallel \langle\!\langle r \lor \mathrm{id} \rangle\!\rangle^*$$

that is, the command $(\textbf{rely}\, r \bullet c)$ guarantees to implement c in the presence of interference steps bounded by r. To understand better the rely command and whether it is feasible, a few examples are examined.

1. $(\textbf{rely}\, x < x' \bullet [x + 1 \leq x'])$ guarantees that, when it is put in an environment that may increase x, the value of x is increased by at least one. A possible implementation of this rely command is to increment x by one. The environment may further increase x but together they guarantee that it is increased by at least one.
2. $(\textbf{rely}\, x < x' \bullet [x' = x])$ guarantees that, when it is put in an environment that may increase x, the value of x is unchanged. There is no possible implementation of this. Even the "obvious" implementation, **skip**, which does nothing is not a valid implementation because when put in an environment that may increase x, the overall effect may be to increase x.
3. $(\textbf{rely}\, x = x' \bullet [x' = x + 1])$ guarantees that, when put in an environment in which each interference step does not modify x (although it may modify variables other than x), x is incremented by exactly one. It may be implemented by the assignment $x := x + 1$. This assignment does not have to be performed atomically, i.e. it may be interleaved with interference that satisfies $x = x'$, which guarantees not to modify x but may arbitrarily modify any variables other than x. Because none of the interference steps modify x, the evaluation of $x + 1$ and its assignment to x are not affected.
4. $(\textbf{rely}\, x = x' \land y = y' \bullet [x' = x + 1])$ guarantees that, when put in an environment in which each interference step does not modify either x or y (although it may modify variables other than x and y), x is incremented by one. It puts no constraints on the final value of y. It may be implemented by the (non-atomic) assignments $(y := x+1;\ x := y)$,

but note that this pair of assignments does not implement example 3 above because the environment in example 3 may arbitrarily modify y between the two assignments.

5. (**rely** $\mathrm{id} \bullet [x' = x + 1]$) guarantees that, when put in an environment which does not introduce any interference, (although it may interleave stuttering steps), x is incremented by one. It may also be implemented by the (non-atomic) assignment $x := x + 1$.

4.1. Modelling interference

The rely condition represents a bound on the interference that the environment can inflict between atomic steps of a process. The command $\langle\!\langle r \rangle\!\rangle$ can be used to model a single interference step; it can perform a single atomic step that satisfies the relation r. The command $\langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*$ can then be used to model any finite number of interference steps, zero or more, each of which either satisfies r or stutters. A terminating command that guarantees every atomic step satisfies g or stutters is bounded by (refines) the interference $\langle\!\langle g \vee \mathrm{id} \rangle\!\rangle^*$.

Law 21 (refine-guarantee). For any predicate p, relation g, and command c that terminates from states satisfying p,

$$\{p\}\langle\!\langle g \vee \mathrm{id} \rangle\!\rangle^* \sqsubseteq (\textbf{guar}\ g \bullet \{p\}c) \tag{7}$$

Iteration can be implemented by doing no steps, or by doing one step and repeating the iteration.

Law 22 (refine-iteration). For any relation r,

$$\langle\!\langle r \rangle\!\rangle^* \sqsubseteq \textbf{nil}$$
$$\langle\!\langle r \rangle\!\rangle^* \sqsubseteq \langle\!\langle r \rangle\!\rangle\ ;\ \langle\!\langle r \rangle\!\rangle^*$$

Interference on a sequential composition can be distributed to its components; two sets of interference in parallel corresponds to the disjunction of the interferences; and interference on an atomic command can only precede or follow it.

Law 23 (interference-sequential). For any commands c_0 and c_1 and relation r,

$$(c_0\ ;\ c_1) \parallel \langle\!\langle r \rangle\!\rangle^* \quad = \quad (c_0 \parallel \langle\!\langle r \rangle\!\rangle^*)\ ;\ (c_1 \parallel \langle\!\langle r \rangle\!\rangle^*)\ .$$

Law 24 (parallel-interference). For any relations r_0 and r_1,

$$\langle\!\langle r_0 \rangle\!\rangle^* \parallel \langle\!\langle r_1 \rangle\!\rangle^* = \langle\!\langle r_0 \vee r_1 \rangle\!\rangle^*\ .$$

Law 25 (interference-atomic). For any relations q and r,

$$\langle\!\langle q \rangle\!\rangle \parallel \langle\!\langle r \rangle\!\rangle^* \quad = \quad \langle\!\langle r \rangle\!\rangle^*\ ;\ \langle\!\langle q \rangle\!\rangle\ ;\ \langle\!\langle r \rangle\!\rangle^*\ .$$

Repeated interference is equivalent to interference; interference before an parallel composition with the same interference can be absorbed; and provided a precondition is preserved by the interference it can be moved into a parallel composition.

Law 26 (repeated-interference). For any relation r,

$$\langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*\ ;\ \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^* = \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*\ .$$

Law 27 (absorb-interference). For any relation r and command c,

$$\langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*\ ;\ (c \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*) = (c \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*)\ .$$

Law 28 (interference-precondition). For any predicate p, relation r and command c, such that $r \Rrightarrow (p \Rightarrow p')$,

$$\{p\}(c \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*) = \{p\}(\{p\}c \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*)\ .$$

Proof.

$\{p\}(\{p\}c \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*)$
$=$ by Law 23 (interference-sequential)
$\{p\}((\{p\} \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*)\ ;\ (c \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*))$
$=$ by Law 25 (interference-atomic)

$\{p\}\ ;\ \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*\ ;\ \{p\}\ ;\ \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*\ ;\ (c \parallel \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*)$

$=$ as $r \Rrightarrow (p \Rightarrow p')$

$\{p\}\ ;\ \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*\ ;\ \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*\ ;\ (c \parallel \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*)$

$=$ by Law 26 (repeated-interference) and Law 27 (absorb-interference)

$\{p\}(c \parallel \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*)$

□

Law 29 (guarantee-interference-precondition). For any predicate p, relation g and commands c and d, such that $g \Rrightarrow (p \Rightarrow p')$,

$$\{p\}((\mathbf{guar}\, g \bullet c) \parallel d) = \{p\}((\mathbf{guar}\, g \bullet c) \parallel \{p\}d)\ .$$

Every atomic step of $(\mathbf{guar}\, g \bullet c)$ satisfies g or stutters and because p is preserved by g, p can be assumed as a precondition of d.

4.2. Defining properties of rely

In Law 30 below the interference is represented by the process $\langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*$ which can only perform a finite number of interference steps. If infinite interference were to be allowed, the interference could preempt execution of the command forever (unless one included fairness constraints). Because this paper only considers rules for terminating constructs, finite interference is sufficient. The command $(\mathbf{rely}\, r \bullet c)$ when run in parallel with interference that is bounded by r implements c.

Law 30 (rely). Given a command c that does not include any guarantee commands and a relation r, $(\mathbf{rely}\, r \bullet c)$ is the least command (under the refinement ordering) satisfying,

$$c \sqsubseteq (\mathbf{rely}\, r \bullet c) \parallel \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*\ . \tag{8}$$

That is,

$$(\mathbf{rely}\, r \bullet c) = \bigsqcap\{d \mid c \sqsubseteq (d \parallel \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*)\} \tag{9}$$

Because it is the least command satisfying this condition, for any d

$$c \sqsubseteq (d \parallel \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*) \Rightarrow (\mathbf{rely}\, r \bullet c) \sqsubseteq d\ . \tag{10}$$

For particular r and c the rely command may not be feasible because the set of d such that $c \sqsubseteq (d \parallel \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*)$ may be empty as in Example 2 above.

There may be any finite number, zero or more, of interference steps (each of which satisfies r or stutters) between any two program steps. Hence the interference between any two program steps satisfies r^*. For this reason some formulations [CJ07] of the rely-guarantee approach require r to be reflexive and transitive, so that $r^* = r$. Here we do not require r to be either reflexive or transitive but use its reflexive transitive closure where necessary.

Law 31 (rely-refinement). For a relation r, and a command c with no guarantees and a command d,

$$((\mathbf{rely}\, r \bullet c) \sqsubseteq d) \Leftrightarrow (c \sqsubseteq (d \parallel \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*))\ .$$

Proof. For the forward implication, assume $(\mathbf{rely}\, r \bullet c) \sqsubseteq d$, then from (8)

$$c \sqsubseteq ((\mathbf{rely}\, r \bullet c) \parallel \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*) \sqsubseteq (d \parallel \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*)$$

The reverse implication follows directly from (10). □

Law 30 does not allow a rely with a body that contains any guarantee commands, such as,

$$\mathbf{rely}\, r \bullet (\mathbf{guar}\, g \bullet c)\ .$$

The problem with allowing a guarantee command within a rely in Law 30 is that to show $(\mathbf{rely}\, r \bullet (\mathbf{guar}\, g \bullet c)) \sqsubseteq d$, Law 30 would require one to show $(\mathbf{guar}\, g \bullet c) \sqsubseteq (d \parallel \langle\!\langle r \vee \mathrm{id}\rangle\!\rangle^*)$ which inadvertently requires the interference r to guarantee g (which is undesired), as well as d guaranteeing g (which is desired). However, for a rely nested within a guarantee this problem does not occur.

Law 32 (guar+rely). For any relations g and r and commands c and d where c does not contain any guarantees,

$$(\mathbf{guar}\ g \bullet (\mathbf{rely}\ r \bullet c)) \sqsubseteq d,$$

provided both the following hold

$$c \sqsubseteq (d \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*) \tag{11}$$

$$(\mathbf{guar}\ g \bullet d) \sqsubseteq d\ . \tag{12}$$

Proof. From (11) and (10) we have $(\mathbf{rely}\ r \bullet c) \sqsubseteq d$ and hence by Law 3 (guarantee-monotonic) and (12)

$$\mathbf{guar}\ g \bullet (\mathbf{rely}\ r \bullet c) \sqsubseteq (\mathbf{guar}\ g \bullet d) \sqsubseteq d$$

□

Although the above laws are only valid for bodies that do not contain guarantees, the following equivalence allows one to move a guarantee out of a rely and use the above laws on the rely part.

Law 33 (swap-rely-guarantee). For any relations r and g, and command c,

$$\mathbf{rely}\ r \bullet (\mathbf{guar}\ g \bullet c) = \mathbf{guar}\ g \bullet (\mathbf{rely}\ r \bullet c)$$

4.3. Laws for refining rely commands

Law 34 (introduce-rely). For any command c with no guarantees, $c \sqsubseteq (\mathbf{rely}\ r \bullet c)$.

Proof. From (8) and Law 22 (refine-iteration), $c \sqsubseteq ((\mathbf{rely}\ r \bullet c) \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*) \sqsubseteq ((\mathbf{rely}\ r \bullet c) \parallel \mathbf{nil}) \sqsubseteq (\mathbf{rely}\ r \bullet c)$. □

A rely condition of id corresponds to interference of $\langle [\mathrm{id}] \rangle^*$, i.e. stuttering steps.

Law 35 (rely-id). For any command c, $(\mathbf{rely}\ \mathrm{id} \bullet c) = c$.

Proof. From Law 34 (introduce-rely), $c \sqsubseteq (\mathbf{rely}\ \mathrm{id} \bullet c)$. From Law 31 (rely-refinement), $(\mathbf{rely}\ \mathrm{id} \bullet c) \sqsubseteq c$ if $c \sqsubseteq (c \parallel \langle\!\langle \mathrm{id} \vee \mathrm{id} \rangle\!\rangle^*)$ but $(c \parallel \langle\!\langle \mathrm{id} \rangle\!\rangle^*) = c$ because finite stuttering can be ignored. □

Law 36 (weaken-rely). For any command c with no guarantees, and relations r_0 and r_1,

$$(r_0 \vee \mathrm{id} \Rrightarrow r_1) \Rightarrow (\mathbf{rely}\ r_0 \bullet c) \sqsubseteq (\mathbf{rely}\ r_1 \bullet c)\ .$$

Proof. By Law 31 (rely-refinement) $(\mathbf{rely}\ r_0 \bullet c) \sqsubseteq (\mathbf{rely}\ r_1 \bullet c)$ provided

$$\begin{array}{ll} & c \sqsubseteq (\mathbf{rely}\ r_1 \bullet c) \parallel \langle\!\langle r_0 \vee \mathrm{id} \rangle\!\rangle^* \\ \Leftarrow & \text{Law 1 (consequence) assuming } r_0 \vee \mathrm{id} \Rrightarrow r_1 \\ & c \sqsubseteq (\mathbf{rely}\ r_1 \bullet c) \parallel \langle\!\langle r_1 \vee \mathrm{id} \rangle\!\rangle^* \\ \equiv & \text{by Law 31 (rely-refinement)} \\ & (\mathbf{rely}\ r_1 \bullet c) \sqsubseteq (\mathbf{rely}\ r_1 \bullet c) \end{array}$$

□

Law 37 (stuttering-rely). For any relation r and command c with no guarantees, $(\mathbf{rely}\ r \vee \mathrm{id} \bullet c) = (\mathbf{rely}\ r \bullet c)$.

Proof. The refinement holds using Law 36 (weaken-rely) in both directions as $(r \vee \mathrm{id} \vee \mathrm{id}) \equiv (r \vee \mathrm{id})$. □

Law 38 (rely-monotonic). For any relation r and commands c and d with no guarantees,

$$c \sqsubseteq d \Rightarrow (\mathbf{rely}\ r \bullet c) \sqsubseteq (\mathbf{rely}\ r \bullet d)$$

Proof.

$$\begin{array}{ll} & (\mathbf{rely}\ r \bullet c) \sqsubseteq (\mathbf{rely}\ r \bullet d) \\ \equiv & \text{by Law 31 (rely-refinement)} \\ & c \sqsubseteq ((\mathbf{rely}\ r \bullet d) \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*) \\ \Leftarrow & \text{as Law 30 (rely) gives } d \sqsubseteq ((\mathbf{rely}\ r \bullet d) \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*) \\ & c \sqsubseteq d \end{array}$$

□

Refining the body of a rely command does not necessarily preserve feasibility. For example, $[x = 0,\ x < x'] \sqsubseteq [x' = 1]$ but while $(\mathbf{rely}\, x \leq x' \bullet [x = 0,\ x < x'])$ is feasible, $(\mathbf{rely}\, x \leq x' \bullet [x' = 1])$ is not feasible because the interference may increase x beyond one – see Definition 42 (tolerate-interference).

Law 39 (nested-rely). For a command c with no guarantees and relations r_0 and r_1,

$$(\mathbf{rely}\, r_0 \bullet (\mathbf{rely}\, r_1 \bullet c)) = (\mathbf{rely}\, r_0 \vee r_1 \bullet c)$$

Proof. Refinement from left to right requires successive applications of Law 31 (rely-refinement) and then Law 24 (parallel-interference).

$$\begin{array}{ll} & (\mathbf{rely}\, r_0 \bullet (\mathbf{rely}\, r_1 \bullet c)) \sqsubseteq (\mathbf{rely}\, r_0 \vee r_1 \bullet c) \\ \equiv & (\mathbf{rely}\, r_1 \bullet c) \sqsubseteq (\mathbf{rely}\, r_0 \vee r_1 \bullet c) \parallel \langle\!\langle r_0 \vee \mathrm{id} \rangle\!\rangle^* \\ \equiv & c \sqsubseteq (\mathbf{rely}\, r_0 \vee r_1 \bullet c) \parallel \langle\!\langle r_0 \vee \mathrm{id} \rangle\!\rangle^* \parallel \langle\!\langle r_1 \vee \mathrm{id} \rangle\!\rangle^* \\ \equiv & c \sqsubseteq (\mathbf{rely}\, r_0 \vee r_1 \bullet c) \parallel \langle\!\langle r_0 \vee r_1 \vee \mathrm{id} \rangle\!\rangle^* \\ \equiv & (\mathbf{rely}\, r_0 \vee r_1 \bullet c) \sqsubseteq (\mathbf{rely}\, r_0 \vee r_1 \bullet c) \end{array}$$

The reverse refinement uses Law 31 (rely-refinement) and Law 24 (parallel-interference).

$$\begin{array}{ll} & (\mathbf{rely}\, r_0 \vee r_1 \bullet c) \sqsubseteq (\mathbf{rely}\, r_0 \bullet (\mathbf{rely}\, r_1 \bullet c)) \\ \equiv & c \sqsubseteq (\mathbf{rely}\, r_0 \bullet (\mathbf{rely}\, r_1 \bullet c)) \parallel \langle\!\langle r_0 \vee r_1 \vee \mathrm{id} \rangle\!\rangle^* \\ \equiv & c \sqsubseteq ((\mathbf{rely}\, r_0 \bullet (\mathbf{rely}\, r_1 \bullet c)) \parallel \langle\!\langle r_0 \vee \mathrm{id} \rangle\!\rangle^*) \parallel \langle\!\langle r_1 \vee \mathrm{id} \rangle\!\rangle^* \\ \Leftarrow & \text{by Law 30 (rely) we have } (\mathbf{rely}\, r_1 \bullet c) \sqsubseteq (\mathbf{rely}\, r_0 \bullet (\mathbf{rely}\, r_1 \bullet c)) \parallel \langle\!\langle r_0 \vee \mathrm{id} \rangle\!\rangle^* \\ & c \sqsubseteq (\mathbf{rely}\, r_1 \bullet c) \parallel \langle\!\langle r_1 \vee \mathrm{id} \rangle\!\rangle^* \\ \equiv & (\mathbf{rely}\, r_1 \bullet c) \sqsubseteq (\mathbf{rely}\, r_1 \bullet c) \end{array}$$

□

A rely on a composite command may be distributed to its component commands.

Law 40 (distribute-rely). For any relation r and commands c, c_0 and c_1 with no guarantees the following hold.

$$\mathbf{rely}\, r \bullet (c_0 \sqcap c_1) = (\mathbf{rely}\, r \bullet c_0) \sqcap (\mathbf{rely}\, r \bullet c_1) \tag{13}$$

$$\mathbf{rely}\, r \bullet (c_0 ;\ c_1) \sqsubseteq (\mathbf{rely}\, r \bullet c_0) ;\ (\mathbf{rely}\, r \bullet c_1) \tag{14}$$

$$\mathbf{rely}\, r \bullet (c_0 \parallel c_1) \sqsubseteq (\mathbf{rely}\, r \bullet c_0) \parallel (\mathbf{rely}\, r \bullet c_1) \tag{15}$$

$$\mathbf{rely}\, r \bullet (c_0 \Cap c_1) \sqsubseteq (\mathbf{rely}\, r \bullet c_0) \Cap (\mathbf{rely}\, r \bullet c_1) \tag{16}$$

$$\mathbf{rely}\, r \bullet c^+ \sqsubseteq (\mathbf{rely}\, r \bullet c)^+ \tag{17}$$

Proof. For nondeterministic choice (13) we have the following.

$$\begin{array}{ll} & \mathbf{rely}\, r \bullet (c_0 \sqcap c_1) \\ = & \text{by (9)} \\ & \bigsqcap\{d \mid (c_0 \sqcap c_1) \sqsubseteq (d \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*)\} \\ = & \bigsqcap\{d \mid (c_0 \sqsubseteq (d \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*)) \vee (c_1 \sqsubseteq (d \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*))\} \\ = & \bigsqcap\{d \mid (c_0 \sqsubseteq (d \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*))\} \sqcap \bigsqcap\{d \mid (c_1 \sqsubseteq (d \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*))\} \\ = & \text{by (9)} \\ & (\mathbf{rely}\, r \bullet c_0) \sqcap (\mathbf{rely}\, r \bullet c_1) \end{array}$$

For sequential composition we have the following.

$$\begin{array}{ll} & \mathbf{rely}\, r \bullet (c_0 ;\ c_1) \sqsubseteq (\mathbf{rely}\, r \bullet c_0) ;\ (\mathbf{rely}\, r \bullet c_1) \\ \equiv & \text{by Law 31 (rely-refinement)} \\ & c_0 ;\ c_1 \sqsubseteq ((\mathbf{rely}\, r \bullet c_0) ;\ (\mathbf{rely}\, r \bullet c_1)) \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^* \\ \equiv & \text{by Law 23 (interference-sequential)} \end{array}$$

$c_0 \,;\; c_1 \sqsubseteq ((\mathbf{rely}\, r \bullet c_0) \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*) \,;\; ((\mathbf{rely}\, r \bullet c_1) \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*)$

$\Leftarrow$ by monotonicity

$(c_0 \sqsubseteq (\mathbf{rely}\, r \bullet c_0) \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*) \wedge (c_1 \sqsubseteq (\mathbf{rely}\, r \bullet c_1) \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*)$

Both these refinements hold by (8). The proofs for parallel and command conjunction are similar. For iteration we use fixed-point induction. In general,

$$d \sqsubseteq c \sqcap (c \,;\; d) \Rightarrow d \sqsubseteq c^+ \tag{18}$$

Applying this to (17) requires one to show,

$(\mathbf{rely}\, r \bullet c^+) \sqsubseteq (\mathbf{rely}\, r \bullet c) \sqcap ((\mathbf{rely}\, r \bullet c) \,;\; (\mathbf{rely}\, r \bullet c^+))$

which we show as follows.

$(\mathbf{rely}\, r \bullet c^+)$

$=$ unfolding iteration

$(\mathbf{rely}\, r \bullet (c \sqcap (c \,;\; c^+)))$

$=$ by part (13)

$(\mathbf{rely}\, r \bullet c) \sqcap (\mathbf{rely}\, r \bullet (c \,;\; c^+))$

$\sqsubseteq$ by part (14)

$(\mathbf{rely}\, r \bullet c) \sqcap ((\mathbf{rely}\, r \bullet c) \,;\; (\mathbf{rely}\, r \bullet c^+))$

□

The following law corresponds to Jones-style sequential introduction rule.

Law 41 (rely-guarantee-sequential). For preconditions p_0 and p_1, relations g, r, q_0 and q_1,

$(\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p_0,\ q_0 \fatsemi q_1])) \sqsubseteq (\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p_0,\ q_0 \wedge p_1'])) \,;\; (\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p_1,\ q_1]))$

Proof.

$(\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p_0,\ q_0 \fatsemi q_1]))$

$\sqsubseteq$ refinement to sequential composition and Law 38 (rely-monotonic) and Law 3 (guarantee-monotonic)

$(\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p_0,\ q_0 \wedge p_1'] \,;\; [p_1,\ q_1]))$

$\sqsubseteq$ by Law 40 (distribute-rely) and Law 8 (distribute-guarantee)

$(\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p_0,\ q_0 \wedge p_1'])) \,;\; (\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p_1,\ q_1]))$

□

4.4. Specifications and rely commands

A specification placed in an environment that can generate interference steps that satisfy r or stutter must *at least* be able to tolerate any finite number of r steps (zero or more) both before and after. Hence we introduce the following definition. Note that for relations $(r \vee \mathrm{id})^* = r^*$.

Definition 42 (tolerate-interference). A specification $[p,\ q]$ *tolerates interference r* provided both the following hold.

$$r^* \Rrightarrow (p \Rightarrow p') \tag{19}$$

$$p \wedge (r^* \fatsemi q \fatsemi r^*) \Rrightarrow q \tag{20}$$

Law 43 (tolerate-interference). A specification $[p,\ q]$ tolerates interference r provided

$$r \Rrightarrow (p \Rightarrow p') \tag{21}$$

$$p \wedge (r \fatsemi q) \Rrightarrow q \tag{22}$$

$$p \wedge (q \fatsemi r) \Rrightarrow q \tag{23}$$

These conditions are a slight generalisation of conditions **PR-ident**, **RQ-ident**, and **QR-ident** in [CJ07] in which r is assumed to be reflexive and transitive. They are also closely related to the the concept of *stabilisation* of p and q in the sense of Wickerson et al. [WDP10], although the latter works with single state postconditions rather than relations.

Note that these conditions do not guarantee one can implement $[p,\ q]$ because the conditions do not address interference while $[p,\ q]$ is executing, only before and after. Interference during $[p,\ q]$ is handled by distributing the rely. We may have that $[p,\ q]$ tolerates interference r according to Definition 42 and

$$[p,\ q] \sqsubseteq [p,\ q_0]\ ;\ [p_1,\ q_1]$$

but when these are placed in a rely context and the rely is distributed one gets

$$(\mathbf{rely}\, r \bullet [p,\ q_0])\ ;\ (\mathbf{rely}\, r \bullet [p_1,\ q_1])$$

but there is no guarantee that either $[p,\ q_0]$ or $[p_1,\ q_1]$ tolerate interference r. Hence a feasible refinement in the standard sequential refinement calculus may no longer be feasible in the context of a rely.

A specification command within a rely may be refined to an atomic step satisfying the specification provided it can tolerate interference satisfying the rely before and after the atomic step. Note that the following law is not valid if the right side is enclosed in a rely context.

Law 44 (rely-specification). For any predicate p, relations r and q, such that $[p,\ q]$ tolerates interference r,

$$(\mathbf{rely}\, r \bullet [p,\ q]) \sqsubseteq \{p\}\langle\langle q\rangle\rangle\ .$$

Proof.

$$\begin{array}{ll}
& (\mathbf{rely}\, r \bullet [p,\ q]) \sqsubseteq \{p\}\langle\langle q\rangle\rangle \\
\equiv & \text{by Law 31 (rely-refinement)} \\
& [p,\ q] \sqsubseteq ((\{p\}\langle\langle q\rangle\rangle) \parallel \langle\langle r \vee \mathrm{id}\rangle\rangle^*) \\
\Leftarrow & \text{by Law 28 (interference-precondition) as } r \Rrightarrow (p \Rightarrow p') \\
& [p,\ q] \sqsubseteq \{p\}(\langle\langle q\rangle\rangle \parallel \langle\langle r \vee \mathrm{id}\rangle\rangle^*) \\
\Leftarrow & \text{by Law 25 (interference-atomic)} \\
& [p,\ q] \sqsubseteq \{p\}(\langle\langle r \vee \mathrm{id}\rangle\rangle^*\ ;\ \langle\langle q\rangle\rangle\ ;\ \langle\langle r \vee \mathrm{id}\rangle\rangle^*) \\
\Leftarrow & [p,\ q] \sqsubseteq \{p\}([r^*]\ ;\ [q]\ ;\ [r^*]) \\
\Leftarrow & [p,\ q] \sqsubseteq \{p\}\,[r^* \fatsemi q \fatsemi r^*] \\
\Leftarrow & \text{by Law 1 (consequence)} \\
& p \wedge (r^* \fatsemi q \fatsemi r^*) \Rrightarrow q
\end{array}$$

□

If the free variables of p and q are left unchanged by the rely condition, the rely can be eliminated provided the specification becomes atomic because $p \wedge (\mathrm{id}(\mathit{vars}(q))^* \fatsemi q \fatsemi \mathrm{id}(\mathit{vars}(q))^*) \Rrightarrow q$ and $\mathrm{id}(\mathit{vars}(p))^* \Rrightarrow (p \Rightarrow p')$, which follows as for any set of variables W, $\mathrm{id}(W)^* \Rrightarrow \mathrm{id}(W)$.

Law 45 (rely-id-specification-atomic). For any predicate p, relation q, and set of variables W, such that $\mathit{vars}(p) \cup \mathit{vars}(q) \subseteq W$,

$$(\mathbf{rely}\, \mathrm{id}(W) \bullet [p,\ q]) \sqsubseteq \{p\}\langle\langle q\rangle\rangle$$

Proof. This follows by Law 44 (rely-specification) because $\mathrm{id}(W)^* \Rrightarrow (p \Rightarrow p')$ and $p \wedge (\mathrm{id}(W)^* \fatsemi q \fatsemi \mathrm{id}(W)^*) \Rrightarrow q$. □

Given that the free variables of p and q are not modified by the environment, it is tempting to use a non-atomic specification on the right. However, recall from Law 35 (rely-id) that a specification without a rely condition is treated as if the rely is id, i.e. only stuttering interference is allowed. In this case a refinement of $[p,\ q]$ is free to make use of variables outside W and these variables are not guaranteed to be stable according to the rely condition on the left side of the law. However, if any refinement of $[p,\ q]$ is restricted to only use variables in W, it will be an implementation of the left side. To address this issue we introduce a new language construct $(\mathbf{uses}\, W \bullet c)$ that can only be refined by a refinement of c if that refinement only uses (reads or writes) variables in W. When c has been refined to code this requirement can be discharged syntactically. The **uses** construct distributes through all language constructs in a

straightforward manner, although a little care is required with local variable declarations. The rules are straightforward and hence they are not spelled out here.

Law 46 (rely-id-specification). For any predicate p, relation q, and set of variables W, such that $vars(p) \cup vars(q) \subseteq W$,

$$(\textbf{rely}\ \mathrm{id}(W) \bullet [p,\ q]) \sqsubseteq (\textbf{uses}\ W \bullet [p,\ q])$$

Law 47 (assignment-rely-guarantee). For any variable x, expression e, and relations g, r and q, such that $r \Rrightarrow \mathrm{id}(vars(e) \cup \{x\})$ and $p \wedge x' = e \wedge \mathrm{id}(\bar{x}) \Rrightarrow def(e) \wedge q \wedge (g \vee \mathrm{id})$,

$$(\textbf{guar}\ g \bullet (\textbf{rely}\ r \bullet x\colon [p,\ q])) \sqsubseteq x := e\ .$$

Proof. Many of the steps in the proof implicitly use Law 3 (guarantee-monotonic).

$\textbf{guar}\ g \bullet (\textbf{rely}\ r \bullet x\colon [p,\ q])$

$=$ by Definition 13 (specification-with-frame), Law 33 (swap-rely-guarantee) and Law 7 (nested-guarantees)

$\textbf{guar}\ g \wedge \mathrm{id}(\bar{x}) \bullet (\textbf{rely}\ r \bullet [p,\ q])$

$\sqsubseteq$ by Law 36 (weaken-rely) by assumption $r \Rrightarrow \mathrm{id}(vars(e) \cup \{x\})$

$\textbf{guar}\ g \wedge \mathrm{id}(\bar{x}) \bullet (\textbf{rely}\ \mathrm{id}(vars(e) \cup \{x\}) \bullet [p,\ q])$

$\sqsubseteq$ by Law 46 (rely-id-specification)

$\textbf{guar}\ g \wedge \mathrm{id}(\bar{x}) \bullet (\textbf{uses}\ vars(e) \cup \{x\} \bullet [p,\ q])$

$=$ by Definition 13 (specification-with-frame)

$\textbf{uses}\ vars(e) \cup \{x\} \bullet (\textbf{guar}\ g \bullet x\colon [p,\ q])$

$\sqsubseteq$ by Law 16 (guarantee-introduce-assignment) as $p \wedge x' = e \wedge \mathrm{id}(\bar{x}) \Rrightarrow def(e) \wedge q \wedge (g \vee \mathrm{id})$

$\textbf{uses}\ vars(e) \cup \{x\} \bullet x := e$

$=$ as the assignment $x := e$ only uses variables in $vars(e) \cup \{x\}$

$x := e$

□

The VDM rules for rely-guarantee handle this issue via read and write frames consisting of sets of variables. The union of the variables in the read and write frames corresponds to the set of "used" variables, and in addition the variables in the read frame are constrained by a guarantee that they are not changed.

Law 48 (trade-rely-guarantee).

$$\textbf{guar}\ g \bullet (\textbf{rely}\ r \bullet [p,\ q \wedge (r \vee g)^*]) = \textbf{guar}\ g \bullet (\textbf{rely}\ r \bullet [p,\ q])$$

For any execution of the right side, every atomic program step satisfies the guarantee $(g \vee \mathrm{id})$ and every atomic environment step satisfies the rely condition $(r \vee \mathrm{id})$, and hence every atomic execution step (whether program or environment) satisfies $(r \vee g \vee \mathrm{id})$ and hence any finite sequence of such steps satisfies $(r \vee g \vee \mathrm{id})^*$, which is equivalent to $(r \vee g)^*$. Refinement from right to left is just a strengthening of the postcondition.

Law 49 (local-variable). For a command c, relation r and variable y which does not occur free in c,

$$(\textbf{guar}\ y' = y \bullet (\textbf{rely}\ r \bullet c)) \sqsubseteq (\textbf{var}\ y \bullet (\textbf{rely}\ r \wedge y' = y \bullet c))$$

Note that the guarantee on the left applies to any global occurrence of y (which cannot be modified by c on the right because all its references to y are to the new local y[3]), while the rely on the right refers to the local variable y, which because it is local to the process cannot be subject to external interference.

4.5. Nondeterministic expression evaluation

If a concurrent process is modifying variables used within an expression during its evaluation, the evaluation becomes nondeterministic. Properties of nondeterminsitic expression evaluation have been investigated elsewhere [CJ07, Col08,

[3] This guarantee can be violated in a language that allows another variable to be an alias for the global variable y.

HBDJ12]; here we use the results of those investigations. If an expression contains at most a single reference to a variable v that may be modified by the environment and all variables other than v are stable (unchanged by the environment) during its evaluation, then the value of the expression is its value in the state in which v is sampled. If the environment respects a rely condition r, then the evaluation state is related to the initial state by r^*.

For a boolean expression b satisfying the single reference property, if b is invariant under an interference step satisfying r, i.e. $r \Rrightarrow (b \Rightarrow b')$, then b is invariant over any finite number of interference steps, i.e. $r^* \Rrightarrow (b \Rightarrow b')$, and hence if b holds in the initial state, it will always evaluate to true in the presence of interference bounded by r.

If b_0 and b_1 are boolean expressions satisfying the single reference to a single variable property, and $b_0 \wedge b_1$ is maintained by r, then if $b_0 \wedge b_1$ holds in the initial state, $b_0 \wedge b_1$ will evaluate to true in the presence of interference bounded by r. This property can be extended to any finite number of conjuncts.

Law 50 (rely-conditional). For any predicate p, relations r and q, such that $[p,\ q]$ tolerates interference r, and boolean expressions b, b_0 and b_1 such that b_0 and b_1 satisfy the single-reference property and $p \wedge b \Rrightarrow b_0$ and $p \wedge r \Rrightarrow (b_0 \Rightarrow b_0')$, and $p \wedge \neg b \Rrightarrow \neg b_1$ and $p \wedge r \Rrightarrow (\neg b_1 \Rightarrow \neg b_1')$,

$$(\mathbf{rely}\, r \bullet [p,\ q]) \sqsubseteq \mathbf{if}\, b\, \mathbf{then}\, (\mathbf{rely}\, r \bullet [p \wedge b_0,\ q]\ \mathbf{else}\ (\mathbf{rely}\, r \bullet [p \wedge \neg b_1,\ q])\ \mathbf{fi}$$

If the rely condition implies that all the variables used in the boolean expression b are stable, then b_0 and b_1 can both be chosen to be b.

Law 51 (rely-conditional-id). For a boolean expression b, predicate p and relations q and r, provided $[p,\ q]$ tolerates interference r and $r \Rrightarrow \mathrm{id}(vars(b))$,

$$(\mathbf{rely}\, r \bullet [p,\ q]) \sqsubseteq \mathbf{if}\, b\, \mathbf{then}\, (\mathbf{rely}\, r \bullet [p \wedge b,\ q])\ \mathbf{else}\ (\mathbf{rely}\, r \bullet [p \wedge \neg b,\ q])\ \mathbf{fi}$$

The law for a "while" loop treats the guard in a similar manner to the guard in a conditional. To guarantee termination of a loop a well-founded relation w is required. The body of the loop must satisfy w and in addition any interference step satisfying r must either satisfy the transitive closure w^+ or not change any of the variables of w, i.e. $\mathrm{id}(vars(w))$. The reflexive transitive closure w^* would be too strong here because it would require that r implies no variables at all are changed if r did not satisfy w^+.

Law 52 (rely-iteration). For any predicate p, relations r and q and well-founded relation w, such that $r \Rrightarrow (p \Rightarrow p')$ and $p \wedge r \Rrightarrow (w^+ \vee \mathrm{id}(vars(w)))$, and boolean expressions b_0 and b_1 satisfying the single-reference property, such that $p \wedge b \Rrightarrow b_0$ and $p \wedge r \Rrightarrow (b_0 \Rightarrow b_0')$, and $p \wedge \neg b \Rrightarrow \neg b_1$ and $p \wedge r \Rrightarrow (\neg b_1 \Rightarrow \neg b_1')$,

$$(\mathbf{rely}\, r \bullet \left[p,\ p' \wedge \neg b_1' \wedge (w^+ \vee \mathrm{id}(vars(w)))\right]) \sqsubseteq \mathbf{while}\, b\, \mathbf{do}(\mathbf{rely}\, r \bullet [p \wedge b_0,\ p' \wedge w])\ \mathbf{od}$$

If the rely condition implies that all the variables used in the boolean expression b are stable, then b_0 and b_1 can both be chosen to be b.

Law 53 (rely-iteration-id). For any predicate p, relations r and q and well-founded relation w, such that $r \Rrightarrow (p \Rightarrow p')$ and $p \wedge r \Rrightarrow (w^+ \vee \mathrm{id}(vars(w)))$, and $r \Rrightarrow \mathrm{id}(vars(b))$,

$$(\mathbf{rely}\, r \bullet \left[p,\ p' \wedge \neg b' \wedge (w^+ \vee \mathrm{id}(vars(w)))\right]) \sqsubseteq \mathbf{while}\, b\, \mathbf{do}(\mathbf{rely}\, r \bullet [p \wedge b,\ p' \wedge w])\ \mathbf{od}$$

5. Parallel refinement

Law 61 (introduce-parallel-spec-with-rely) and Law 62 (introduce-parallel-spec-weakened-relies) below refine a specification with a conjunction of postconditions to a parallel composition of specifications each with one of the conjuncts. As a step towards parallelism we introduce a conjunction operator between commands. At the semantics level it corresponds to intersection of behaviours and hence it is not an operator of the implementation language, only the specification/development language.

Law 54 (conjoined-specifications). For any predicate p and relations q_0 and q_1,

$$[p,\ q_0 \wedge q_1] \sqsubseteq [p,\ q_0] \Cap [p,\ q_1]\ .$$

Proof. A behaviour of $[p,\ q_0] \Cap [p,\ q_1]$ must be a behaviour of both $[p,\ q_0]$ and a behaviour of $[p,\ q_1]$, and hence it satisfies q_0 provided p holds initially as well as satisfying q_1 provided p holds initially, and hence it satisfies $q_0 \wedge q_1$ provided p holds initially, and thus it is a behaviour of $[p,\ q_0 \wedge q_1]$. □

Law 55 (refine-conjunction). For any commands c_0, c_1 and d, if $c_0 \sqsubseteq d$ and $c_1 \sqsubseteq d$, then $c_0 \Cap c_1 \sqsubseteq d$.

Proof. Any behaviour of d must also be a behaviour of both c_0 and c_1, and hence it is a behaviour of $c_0 \Cap c_1$. □

Law 56 (guarantee-bounds-interference). For any predicate p, relations g_0 and g_1 and command c that has no guarantees and terminates from states satisfying p,

$$\{p\}\langle\!\langle g_0 \vee \mathrm{id}\rangle\!\rangle^* \sqsubseteq (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p\}c)) \,.$$

Proof.

$\{p\}\langle\!\langle g_0 \vee \mathrm{id}\rangle\!\rangle^*$
$\sqsubseteq$ by Law 21 (refine-guarantee) as c terminates on p
$(\mathbf{guar}\, g_0 \bullet \{p\}c)$
$\sqsubseteq$ by Law 34 (introduce-rely) as $\{p\}c$ has no guarantees; Law 3 (guarantee-monotonic)
$(\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p\}c))$

□

Law 57 (introduce-parallel-interference). For any predicates p, and p_1, relations g_0 and g_1 such that $p \Rrightarrow p_1$, $g_0 \Rrightarrow (p_1 \Rightarrow p_1')$ and and command c containing no guarantees,

$$\{p\}(\mathbf{guar}(g_0 \vee g_1) \bullet c) \sqsubseteq (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p\}c)) \parallel \{p_1\}\langle\!\langle g_1 \vee \mathrm{id}\rangle\!\rangle^*$$

Proof. We start from Law 30 (rely) which gives the following property.

$\{p\}c \sqsubseteq (\mathbf{rely}\, g_1 \bullet \{p\}c) \parallel \langle\!\langle g_1 \vee \mathrm{id}\rangle\!\rangle^*$
$\Rightarrow$ by Law 3 (guarantee-monotonic)
$(\mathbf{guar}\, g_0 \vee g_1 \bullet \{p\}c) \sqsubseteq (\mathbf{guar}\, g_0 \vee g_1 \bullet ((\mathbf{rely}\, g_1 \bullet \{p\}c) \parallel \langle\!\langle g_1 \vee \mathrm{id}\rangle\!\rangle^*))$
$\Rightarrow$ Law 9 (guarantee-precondition)
$\{p\}(\mathbf{guar}\, g_0 \vee g_1 \bullet c) \sqsubseteq \{p\}(\mathbf{guar}\, g_0 \vee g_1 \bullet ((\mathbf{rely}\, g_1 \bullet \{p\}c) \parallel \langle\!\langle g_1 \vee \mathrm{id}\rangle\!\rangle^*))$

The right side of the above can now be refined to complete the proof.

$\{p\}(\mathbf{guar}\, g_0 \vee g_1 \bullet ((\mathbf{rely}\, g_1 \bullet \{p\}c) \parallel \langle\!\langle g_1 \vee \mathrm{id}\rangle\!\rangle^*))$
$=$ by Law 8 (distribute-guarantee) part (2) parallel
$\{p\}((\mathbf{guar}\, g_0 \vee g_1 \bullet (\mathbf{rely}\, g_1 \bullet \{p\}c)) \parallel (\mathbf{guar}\, g_0 \vee g_1 \bullet \langle\!\langle g_1 \vee \mathrm{id}\rangle\!\rangle^*))$
$\sqsubseteq$ by Law 4 (strengthen-guarantee) twice
$\{p\}((\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p\}c)) \parallel (\mathbf{guar}\, g_1 \bullet \langle\!\langle g_1 \vee \mathrm{id}\rangle\!\rangle^*))$
$=$ by Law 8 (distribute-guarantee) part (5) iteration
$\{p\}((\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p\}c)) \parallel (\mathbf{guar}\, g_1 \bullet \langle\!\langle g_1 \vee \mathrm{id}\rangle\!\rangle)^*)$
$\sqsubseteq$ by Law 11 (guarantee-atomic)
$\{p\}((\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p\}c)) \parallel \langle\!\langle g_1 \vee \mathrm{id}\rangle\!\rangle^*)$
$=$ by Law 29 (guarantee-interference-precondition) as $p \Rrightarrow p_1$ and $g_0 \Rrightarrow (p_1 \Rightarrow p_1')$
$\{p\}((\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p\}c)) \parallel \{p_1\}\langle\!\langle g_1 \vee \mathrm{id}\rangle\!\rangle^*)$
$\sqsubseteq$ by Law 1 (consequence)
$(\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p\}c)) \parallel \{p_1\}\langle\!\langle g_1 \vee \mathrm{id}\rangle\!\rangle^*$

□

Law 58 (refine-by-parallel). For any predicates p and p_1, relations g_0 and g_1, such that $p \Rrightarrow p_1$, $g_0 \Rrightarrow (p_1 \Rightarrow p_1')$, a command c_0 containing no guarantees, and a command c_1 that contains no guarantees and terminates on p_1,

$$\{p\}(\mathbf{guar}(g_0 \vee g_1) \bullet c_0) \sqsubseteq (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p\}c_0)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet \{p_1\}c_1))$$

Proof. Using Law 56 (guarantee-bounds-interference) with symmetric numbering one can deduce

$$\{p_1\}\langle\!\langle g_1 \vee \mathrm{id}\rangle\!\rangle^* \sqsubseteq (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet \{p_1\}c_1)) \tag{24}$$

That is used in the following proof.

$$
\begin{array}{ll}
& \{p\}(\mathbf{guar}(g_0 \vee g_1) \bullet c_0) \\
\sqsubseteq & \text{by Law 57 (introduce-parallel-interference) as } p \Rrightarrow p_1,\ g_0 \Rrightarrow (p_1 \Rightarrow p_1') \\
& (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p\}c_0)) \parallel \{p_1\}\langle\!\langle g_1 \vee \mathrm{id}\rangle\!\rangle^* \\
\sqsubseteq & \text{using (24)} \\
& (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p\}c_0)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet \{p_1\}c_1))
\end{array}
$$

□

Law 59 (introduce-parallel). For any predicates p, p_0 and p_1, relations g_0 and g_1, such that $p \Rrightarrow p_0 \wedge p_1$, $g_0 \Rrightarrow (p_1 \Rightarrow p_1')$ and $g_1 \Rrightarrow (p_0 \Rightarrow p_0')$, and commands c_0 and c_1 both containing no guarantees where c_0 terminates on p_0 and c_1 terminates on p_1,

$$\{p\}(\mathbf{guar}\, g_0 \vee g_1 \bullet (c_0 \Cap c_1)) \quad \sqsubseteq \quad (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p_0\}c_0)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet \{p_1\}c_1))$$

Proof. One can use Law 8 (distribute-guarantee) and then distribute the precondition as follows.

$$\{p\}(\mathbf{guar}\, g_0 \vee g_1 \bullet (c_0 \Cap c_1)) = (\{p\}(\mathbf{guar}\, g_0 \vee g_1 \bullet c_0)) \Cap (\{p\}(\mathbf{guar}\, g_0 \vee g_1 \bullet c_1))$$

Using Law 58 (refine-by-parallel) as $p \Rrightarrow p_0 \wedge p_1$, $g_0 \Rrightarrow (p_1 \Rightarrow p_1')$ and $g_1 \Rrightarrow (p_0 \Rightarrow p_0')$ one can deduce both the following.

$$
\begin{array}{lll}
\{p\}(\mathbf{guar}\, g_0 \vee g_1 \bullet c_0) & \sqsubseteq & (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p_0\}c_0)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet \{p_1\}c_1)) \\
\{p\}(\mathbf{guar}\, g_0 \vee g_1 \bullet c_1) & \sqsubseteq & (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet \{p_1\}c_1)) \parallel (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p_0\}c_0))
\end{array}
$$

Because parallel is commutative the right sides of both these refinements are the same and hence by Law 55 (refine-conjunction) the right side refines the conjoined left sides.

$$
\begin{array}{ll}
& (\{p\}(\mathbf{guar}\, g_0 \vee g_1 \bullet c_0)) \Cap (\{p\}(\mathbf{guar}\, g_0 \vee g_1 \bullet c_1)) \\
\sqsubseteq & (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p_0\}c_0)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet \{p_1\}c_1))
\end{array}
$$

□

Law 60 (introduce-parallel-with-rely). For any predicate p, p_0 and p_1, relations r, g_0 and g_1 such that $p \Rrightarrow p_0 \wedge p_1$, $g_0 \Rrightarrow (p_1 \Rightarrow p_1')$, and $g_1 \Rrightarrow (p_0 \Rightarrow p_0')$, and commands c_0 and c_1 both containing no guarantees and where c_0 terminates on p_0 and c_1 terminates on p_1,

$$(\mathbf{guar}\, g_0 \vee g_1 \bullet (\mathbf{rely}\, r \bullet \{p\}(c_0 \Cap c_1))) \sqsubseteq (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, r \vee g_1 \bullet \{p_0\}c_0)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, r \vee g_0 \bullet \{p_1\}c_1))$$

Proof.

$$
\begin{array}{ll}
& (\mathbf{guar}\, g_0 \vee g_1 \bullet (\mathbf{rely}\, r \bullet \{p\}(c_0 \Cap c_1))) \\
\sqsubseteq & \text{by Law 33 (swap-rely-guarantee) and Law 9 (guarantee-precondition)} \\
& (\mathbf{rely}\, r \bullet \{p\}(\mathbf{guar}\, g_0 \vee g_1 \bullet (c_0 \Cap c_1))) \\
\sqsubseteq & \text{by Law 59 (introduce-parallel) and Law 38 (rely-monotonic)} \\
& (\mathbf{rely}\, r \bullet ((\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet \{p_0\}c_0)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet \{p_1\}c_1)))) \\
\sqsubseteq & \text{by Law 40 (distribute-rely) and Law 33 (swap-rely-guarantee)} \\
& (\mathbf{rely}\, r \bullet (\mathbf{rely}\, g_1 \bullet (\mathbf{guar}\, g_0 \bullet \{p_0\}c_0))) \parallel (\mathbf{rely}\, r \bullet (\mathbf{rely}\, g_0 \bullet (\mathbf{guar}\, g_1 \bullet \{p_1\}c_1))) \\
= & \text{by Law 39 (nested-rely) twice and Law 33 (swap-rely-guarantee)} \\
& (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, r \vee g_1 \bullet \{p_0\}c_0)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, r \vee g_0 \bullet \{p_1\}c_1))
\end{array}
$$

□

Law 61 (introduce-parallel-spec-with-rely). For predicates p, p_0 and p_1, and relations r, q_0, q_1, g_0 and g_1, such that $p \Rrightarrow p_0 \wedge p_1$, $g_0 \Rrightarrow (p_1 \Rightarrow p_1')$, and $g_1 \Rrightarrow (p_0 \Rightarrow p_0')$,

$$
\begin{array}{ll}
& (\mathbf{rely}\, r \bullet [p,\ q_0 \wedge q_1 \wedge (r \vee g_0 \vee g_1)^*]) \\
\sqsubseteq & (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, r \vee g_1 \bullet [p_0,\ q_0])) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, r \vee g_0 \bullet [p_1,\ q_1]))
\end{array}
$$

Proof.

$(\mathbf{rely}\, r \bullet [p,\ q_0 \wedge q_1 \wedge (r \vee g_0 \vee g_1)^*])$

$\sqsubseteq$ by Law 6 (introduce-guarantee) of $g_0 \vee g_1$ and Law 48 (trade-rely-guarantee)

$(\mathbf{guar}\, g_0 \vee g_1 \bullet (\mathbf{rely}\, r \bullet [p,\ q_0 \wedge q_1]))$

$\sqsubseteq$ by Law 54 (conjoined-specifications)

$(\mathbf{guar}\, g_0 \vee g_1 \bullet (\mathbf{rely}\, r \bullet \{p\}([q_0] \Cap [q_1]))$

$\sqsubseteq$ by Law 60 (introduce-parallel-with-rely) from assumptions

$(\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, r \vee g_1 \bullet [p_0,\ q_0])) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, r \vee g_0 \bullet [p_1,\ q_1]))$

□

Law 62 (introduce-parallel-spec-weakened-relies). For predicates p, p_0 and p_1 and relations $r, r_0, r_1, q_0, q_1, g_0$ and g_1, such that $p \Rrightarrow p_0 \wedge p_1$, $g_0 \Rrightarrow (p_1 \Rightarrow p_1')$, $g_1 \Rrightarrow (p_0 \Rightarrow p_0')$, and $g_0 \Rrightarrow r_1$ and $g_1 \Rrightarrow r_0$,

$(\mathbf{rely}\, r \bullet [p,\ q_0 \wedge q_1 \wedge (r \vee g_0 \vee g_1)^*])$

$\sqsubseteq (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, r \vee r_0 \bullet [p_0,\ q_0])) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, r \vee r_1 \bullet [p_1,\ q_1]))$

Proof. The law follows from Law 61 (introduce-parallel-spec-with-rely) and two applications of Law 36 (weaken-rely). □

6. Extended example (concurrent version)

The example here is the same as that used in Section 3.4, however, the focus is on refining to a concurrent algorithm. The choice facilitates comparison with other publications: the example is taken from Susan Owicki's thesis [Owi75] (and is also used in [Jon81] and [dR01] to contrast the compositional rely/guarantee approach with the "Owicki/Gries" method).

6.1. Specification

The specification requires that *findp* sets the final value of variable t to the lowest index of array v such that $p(v(t))$ for some predicate p; it assumes that neither v nor t is changed by the environment; the frame prefix guarantees no changes to variables other than t are made.

$$\mathit{findp} \mathrel{\widehat{=}} \mathbf{rely}\, \mathrm{id}(\{v,t\}) \bullet t\colon [(t' = \mathit{len}(v) + 1 \vee \mathit{satp}(v,t')) \wedge \mathit{notp}(v, \mathit{dom}(v), t')] \qquad \lhd$$

For a parallel implementation the main change from Sect. 3.4 is to be precise about the rely condition. Note the frame of t on the specification command by Definition 13 (specification-with-frame) guarantees $\mathrm{id}(\bar{t})$, i.e. all variables other than t are never modified.

6.2. Framing

The majority of the laws developed above for specifications do not handle a specification with a frame, however, each law can can be promoted to a law that handles frames by converting a specification with a frame x to its equivalent unframed specification enclosed in a guarantee of $\mathrm{id}(\bar{x})$, then applying the law to the unframed specification (within the context of the guarantee) and then distributing the guarantee into the result and finally rewriting any specifications within a guarantee of $\mathrm{id}(\bar{x})$ to the equivalent framed specification. This process can be applied systematically to each law and hence the laws are not explicitly elaborated here, however, the development assumes these laws are available. (One could develop a systematic transformation to generate these derived laws automatically, but that is a topic for another paper.)

It's clear we could design a sequential implementation in which one process uses an index t and loops from 1 upwards until either $p(v(t))$ is satisfied or the index goes beyond $\mathit{len}(v)$. Alternatively, one could split into two[4]

[4] Generalising to an arbitrary number of threads presents no conceptual difficulties; in common with the earlier papers on this example, a two way split of the index values into even and odd is considered because this keeps formulae short.

processes as below and have each process consider a subset of the domain of v. This could lead to the need to "lock" t during updates and there is a better approach followed in Section 6.3.

6.3. Representing t using two variables

One can avoid the difficulties of sharing t by having a separate index for each of the concurrent processes and representing t by $min(ot, et)$.[5] A poor solution would then use disjoint parallelism where the two processes ignore each other's progress. But our interest is in an algorithm where the processes "interfere" to achieve better performance.

Two local variables ot and et are introduced with the intention that the minimum of ot and et will be the least index satisfying p.

$\sqsubseteq$ by Law 49 (local-variable) for ot and et and Law 1 (consequence) to strengthen the postcondition
var $ot, et \bullet$ **rely** $\mathrm{id}(\{v, t, ot, et\}) \bullet$
$\quad ot, et, t : [(min(ot, et) = len(v) + 1 \lor satp(v, t')) \land notp(v, dom(v), min(ot, et)) \land t' = min(ot', et')]$ $\lhd$

$\sqsubseteq$ sequential
$ot, et : [(min(ot', et') = len(v) + 1 \lor satp(v, min(ot', et'))) \land notp(v, dom(v), min(ot', et'))]$; $\lhd$
$t : [t' = min(ot, et)]$

A guarantee invariant can be used in a similar manner to that used in Sect. 3.4.

$\sqsubseteq$ by Law 20 (trading-post-guarantee-invariant) and sequential composition
$ot, et : [ot' = et' = len(v) + 1]$;
guar-inv $min(ot, et) = len(v) + 1 \lor satp(v, min(ot, et)) \bullet$
$\quad ot, et : [ot = et = len(v) + 1,\ notp(v, \mathrm{dom}(v), min(ot', et'))]$ $\lhd$

The body of this can be refined to a parallel composition but first we distribute the rely conditions into the body so that as much context information as possible is available to the parallel refinement.

rely $id(\{v, r, ot, et\}) \bullet$
$\quad ot, et : [ot = et = len(v) + 1,\ notp(v, \mathrm{dom}(v), min(ot', et'))]$ $\lhd$

6.4. Concurrency

The motivation for the parallel algorithm comes from the observation that the set of indices to be searched, $\mathrm{dom}(v)$, can be partitioned into the odd and even indices of v, namely $evens(v)$ and $odds(v)$, respectively, which can be searched in parallel.

$$notp(v, odds(v), min(ot', et')) \land notp(v, evens(v), min(ot', et')) \Rightarrow notp(v, \mathrm{dom}(v), min(ot', et'))$$

The next step is the epitome of rely-guarantee refinement: splitting the specification command.

$\sqsubseteq$ by Law 62 (introduce-parallel-spec-weakened-relies)
guar $ot' \le ot \land et' = et \bullet$ **rely** $et' \le et \land \mathrm{id}(\{ot, v\}) \bullet$
$\quad ot, et : [ot = et = len(v) + 1,\ notp(v, odds(v), min(ot', et'))]$
$\parallel$
guar $et' \le et \land ot' = ot \bullet$ **rely** $ot' \le ot \land \mathrm{id}(\{et, v\}) \bullet$
$\quad ot, et : [ot = et = len(v) + 1,\ notp(v, evens(v), min(ot', et'))]$

Note that the above is all in the context of the guarantee invariant given above. As with Sect. 3.4, the guarantee invariant will eventually need to be discharged for each atomic step but it is simpler to do that when the final code is available. However, during the development one needs to be aware of this requirement to avoid introducing code that is inconsistent with the guarantee invariant.

[5] Following the observation in [Jon07] that achieving rely and/or guarantee conditions is often linked with data reification, it would be possible to view $min(ot, et)$ as a representation of the abstract variable t; this point is not pursued here.

6.5. Refining the branches to code

For the first branch of the parallel, the guarantee $et' = et$ is equivalent to removing et from the frame of the branch.

$\sqsubseteq$ **guar** $ot' \leq ot \bullet$ **rely** $et' \leq et \wedge \mathrm{id}(\{ot, v\}) \bullet$
$ot : [ot = et = len(v) + 1,\ notp(v, odds(v), min(ot', et'))]$ $\lhd$

The body of this can be refined to sequential code in a manner similar to that used in Sect. 3.4, however, because the specification refers to et' it is subject to interference from the parallel (evens) process which may update et. That interference is however bounded by the rely condition which assumes the parallel process only ever decreases et.

$\sqsubseteq$ by Law 49 (local-variable) for oc
var $oc \bullet$ **rely** $et' \leq et \wedge \mathrm{id}(\{oc, ot, v\}) \bullet$
$oc, ot : [ot = et = len(v) + 1,\ notp(v, odds(v), min(ot', et'))]$ $\lhd$

As in Sect. 3.4 a loop invariant is introduced as a (stronger) guarantee invariant which is established by setting oc to one. The guarantee invariant combined with the postcondition $oc' \geq min(ot', et')$ implies the postcondition of the above specification. The postcondition $oc' \geq min(ot', et')$ uses "$\geq$" rather than "$=$" because the parallel process may decrease et.

$\sqsubseteq$ by sequential and Law 20 (trading-post-guarantee-invariant)
$oc : [oc' = 1]$;
guar-inv $notp(v, odds(v), oc) \bullet$
$oc, ot : [oc = 1 \wedge ot = et = len(v) + 1,\ oc' \geq min(ot', et')]$ $\lhd$

Collecting the rely conditions gives the following.

rely $et' \leq et \wedge \mathrm{id}(\{oc, ot, v\}) \bullet$
$oc, ot : [oc = 1 \wedge ot = et = len(v) + 1,\ oc' \geq min(ot', et')]$ $\lhd$

Bounding conditions on oc are needed for the loop invariant.

$bnd(oc, ot, v) \mathrel{\widehat{=}} 1 \leq oc \leq ot \leq len(v) + 1$

The invariant $bnd(oc, ot, v)$ is established as $oc = 1 \wedge ot = len(v) + 1$.

$\sqsubseteq$ **rely** $et' \leq et \wedge \mathrm{id}(\{oc, ot, v\}) \bullet$
$oc, ot : [bnd(oc, ot, v),\ bnd(oc, ot, v) \wedge oc' \geq min(ot', et')]$ $\lhd$

A while loop is introduced with an invariant $bnd(oc, ot, v)$ and a well founded relation based on a variant of $ot - oc$.

$\sqsubseteq$ **while** $oc < ot \wedge oc < et$ **do**
rely $et' \leq et \wedge \mathrm{id}(\{oc, ot, v\}) \bullet$
$oc, ot : [oc < ot \wedge bnd(oc, ot, v),\ bnd(oc, ot, v) \wedge ot' - oc' < ot - oc]$ $\lhd$
od

Because oc, ot and v are stable in the rely condition, the predicate $bnd(oc, ot, v)$ is an invariant of any number of interference steps, as is the conjunct $oc < ot$ of the guard. The other conjunct $oc < et$ is not an invariant of the interference because et may be decreased, and hence this conjunct is dropped from the precondition of the loop body. The negation of the guard, i.e. $ot \leq oc \vee et \leq oc$ is invariant of the interference because the first disjunct is stable under interference and if $et \leq oc$ holds and et is decreased $et \leq oc$ still holds.

The specification of the loop body only involves variables which are stable under interference and hence we can use Law 46 (rely-id-specification) to eliminate the rely condition from the code.

$\sqsubseteq$ by Law 36 (weaken-rely)
rely $\mathrm{id}(\{oc, ot, v\}) \bullet$
$oc, ot : [oc < ot \wedge bnd(oc, ot, v),\ bnd(oc, ot, v) \wedge ot' - oc' < ot - oc]$
$\sqsubseteq$ by Law 46 (rely-id-specification)
uses $oc, ot, v \bullet$
$oc, ot : [oc < ot \wedge bnd(oc, ot, v),\ bnd(oc, ot, v) \wedge ot' - oc' < ot - oc]$ $\lhd$

The refinement is now the same as that used in Sect. 3.4, which ensures the two guarantee invariants are maintained.

$\sqsubseteq$ **if** $p(v(oc))$ **then** $ot := oc$ **else** $oc := oc + 2$ **fi**

6.6. Collected code

The final steps in the refinement are to use Law 47 (assignment-rely-guarantee) to introduce a number of simple assignments.

$$
\begin{array}{l}
\textbf{var } ot, et \bullet \\
ot := len(v) + 1 \ ; \\
et := len(v) + 1 \ ; \\
\left(\begin{array}{l} \textbf{var } oc \bullet \\ oc := 1 \ ; \\ \textbf{while } oc < ot \wedge oc < et \textbf{ do} \\ \quad \textbf{if } p(v(oc)) \textbf{ then } ot := oc \\ \qquad\qquad \textbf{else } oc := oc + 2 \textbf{ fi} \\ \textbf{od} \end{array} \right. \\
\qquad\qquad \left\| \right. \\
\left. \begin{array}{l} \textbf{var } ec \bullet \\ ec := 2 \ ; \\ \textbf{while } ec < ot \wedge ec < et \textbf{ do} \\ \quad \textbf{if } p(v(ec)) \textbf{ then } et := ec \\ \qquad\qquad \textbf{else } ec := ec + 2 \textbf{ fi} \\ \textbf{od} \end{array} \right) \ ; \\
t := min(ot, et)
\end{array}
$$

7. Language semantics

Coleman and Jones [CJ07] give a proof of soundness of rely-guarantee rules with respect to a (conventional) small-step structural operational semantics [Plo04a, Plo04b]. In order to provide the semantics of a (non-atomic) specification command in a concurrent language (i.e. in the presence of interference), we give a "multiple-small-step" operational semantics. We use a form of structural operational semantics based on that introduced by Colvin and Hayes [CH11, CH09], where transitions are labelled by a relation on states, and configurations consist of just a command (that is, they do not include state).

The transition relation and label type is given below.

$$
\begin{array}{l}
\longrightarrow : \mathit{Label} \to \mathbb{P}(\mathit{Cmd} \times \mathit{Cmd}) \\
\ell ::= \{p\} \mid q \mid \mathsf{env}(q)
\end{array}
$$

For each label ℓ, the relation $\xrightarrow{\ell}$ is the least relation given by the subsequent rules. The semantics follows the approach of Peter Aczel [dBHdR99, dR01] in distinguishing between *direct* (or program) steps and *interference* (or environment) steps. A label $\ell \in \mathit{Label}$ is either a relation q if abstracting the direct step of the program, or $\mathsf{env}(q)$ if an interference step of the environment. Relation q is a relation between states, represented as a two-state predicate. Within q the name x refers to the pre-state value of the variable x and x' refers to its post-state value. A step $c \xrightarrow{\mathsf{env}(r)} c'$ states that c will allow any concurrent program step, q, where $q \Rightarrow r$.

7.1. Small-step semantics

The syntax of our language was given in Section 2. In Fig. 2 we give the semantics of that language, excluding specification commands. The reason for the exclusion is discussed in Section 7.2.

Rule 2 states that the special command **abort** can take any terminating or non-terminating sequence of steps, and hence may arbitrarily change the values of variables. Rule 1 states that an assertion $\{p\}$ may transition provided p holds in which case no variables are changed, and otherwise aborts. In both cases the transition relation q must be satisfiable, that is, $(\exists \sigma, \sigma' \bullet q(\sigma, \sigma'))$.

Rule 6 is straightforward: if c_1 may take a step then so may the sequential composition $(c_1 \ ; \ c_2)$, and when the first command has terminated the second may execute. The step of eliminating the **nil** is labelled with the relation id, which does not depend on any variables and leaves all variables unchanged. It plays a similar role to internal (τ) transitions in process algebras such as CSP [Hoa85] and CCS [Mil89]. Rule 7 lets either command in a choice take a step, resolving the nondeterminism in that command's favour. If both commands are **nil** the choice may be reduced to **nil**. Rule 5 unfolds a possibly infinite iteration c^ω to choose between executing an instance of c and then repeating c^ω or terminating. The rule for the terminating iteration c^* is given in Section 7.2. Rule 14 states that an atomic specification command immediately terminates through any step satisfying its postcondition.

The novel rules are those for the new constructs and concurrency. Rule 8(a) states that if c can make a transition q, the transition the command $(\textbf{guar}\, g \bullet c)$ can make is q restricted to satisfy the guarantee $g \vee \mathrm{id}$, provided that

transition is itself satisfiable. Any transitions not satisfying $g \vee \mathrm{id}$ are prevented from happening. Rule 8(b) states that the guarantee has no effect on environment steps of c.

From the semantics of the guarantee command, $(\mathbf{guar}\, g \bullet \{p\})$ can only perform atomic steps that satisfy g, even when the precondition p does not hold. This is different to $(\{p\}\,;\ (\mathbf{guar}\, g \bullet \mathbf{skip}))$ which can perform any sequence of steps whatsoever. As a general rule $\{p\} \sqsubseteq (\mathbf{guar}\, g \bullet \{p\})$ but not the converse.[6] Note that $(\mathbf{guar}\, g \bullet \{p\})$ can perform any finite or infinite sequence of steps, each of which satisfies g, if p does not hold initially. The following example has both a guarantee that x decreases and a precondition that x is non-zero: $(\mathbf{guar}\, x > x' \bullet \{x \neq 0\}\,[x' = x - 1])$. If x is initially zero (i.e. the precondition does not hold) it allows any behaviour that decreases x by any amount or stutters.

Rule 10 for a parallel composition of commands $c_1 \parallel c_2$ is defined so that program steps of one match with (and eliminate) environment steps of the other. This occurs only if the program step is allowed by the environment step.

7.2. Multi-step semantics

We now give a "multi-step" semantics, based on the small-step semantics. The advantage of using this style of semantics is that we can define the (terminating) traces of a non-atomic specification command; in a traditional Plotkin-style semantics it is not possible to reconstruct individual steps from a sequence of steps, unless a similar mechanism to ours is adopted. It also allows us to define the semantics of the general atomic command $\langle c \rangle$. The semantics are in Fig. 3. The concatenation of sequences ls_1 and ls_2 is written $\ell s_1.\ell s_2$ and the sequence with head ℓ and tail ℓs is written $\ell.\ell s$.

The transition $c \xrightarrow{\ell s}\!\!\!\!\rightarrow\ c'$ means that c can transition from c to c' via the sequence of multiple steps within ℓs. The rules for most of the basic commands are straightforwardly retrieved from the small-step semantics by combining steps, i.e.

$$\frac{c \xrightarrow{\ell} c'}{c \xrightarrow{\langle \ell \rangle}\!\!\!\!\rightarrow\ c'} \qquad \frac{c \xrightarrow{\ell s_1}\!\!\!\!\rightarrow\ c' \quad c' \xrightarrow{\ell s_2}\!\!\!\!\rightarrow\ c''}{c \xrightarrow{\ell s_1.\ell s_2}\!\!\!\!\rightarrow\ c''}$$

We describe the rules for specification commands, concurrency, and the interaction of specification commands with rely and guarantee contexts below.

Rule 11(a) states that $[p, q]$ may transition with any (finite) trace ℓs, provided the combined effect of ℓs implies the postcondition q, assuming p holds initially. This allows any modifications to any variables, in any order. Guarantee and rely contexts will prune these behaviours. Rule 11(b) states that if p does not hold initially then any behaviour is possible.

The notation ${}^{\circ}_{9}/\ell s$ is the composition of the labels in ℓs, i.e. is a generalised version of binary composition of relations. It treats environment and program steps equally, i.e.

$$\begin{array}{rcl} {}^{\circ}_{9}/\langle\rangle & = & \mathrm{id} \\ {}^{\circ}_{9}/(q.\ell s) & = & q \,{}^{\circ}_{9}\, ({}^{\circ}_{9}/\ell s) \\ {}^{\circ}_{9}/(env(r).\ell s) & = & r \,{}^{\circ}_{9}\, ({}^{\circ}_{9}/\ell s) \\ {}^{\circ}_{9}/(\{p\}.\ell s) & = & p \wedge ({}^{\circ}_{9}/\ell s) \end{array}$$

Rule 13 states that if ℓs is any consistent trace of c, where ℓs does not contain environment steps, then $\langle c \rangle$ may transition via ℓs. Hence, the semantics of $\langle c \rangle$ is to exclude any interference during the execution of c.

The semantics uses the following definitions, which are used to extract all program or environment steps from a mixed trace, and to abbreviate that all envionment steps satsify some rely.

$$\begin{array}{rcl} \mathsf{psteps}(\langle\rangle) & = & \langle\rangle \\ \mathsf{psteps}(q.\ell s) & = & q.\,\mathsf{psteps}(\ell s) \\ \mathsf{psteps}(\{p\}.\ell s) & = & \{p\}.\,\mathsf{psteps}(\ell s) \\ \mathsf{psteps}(\mathsf{env}(e).\ell s) & = & \mathsf{psteps}(\ell s) \end{array} \qquad \begin{array}{rcl} \mathsf{esteps}(\langle\rangle) & = & \langle\rangle \\ \mathsf{esteps}(q.\ell s) & = & \mathsf{esteps}(\ell s) \\ \mathsf{esteps}(\{p\}.\ell s) & = & \mathsf{esteps}(\ell s) \\ \mathsf{esteps}(\mathsf{env}(e).\ell s) & = & e.\,\mathsf{esteps}(\ell s) \end{array}$$

ℓs **within** $R \mathrel{\widehat{=}} (\forall\, q \in \mathrm{ran}(\ell s) \bullet q \Rrightarrow R)$

[6] Aside: Another possible choice of semantics is to ensure that $(\mathbf{guar}\, g \bullet \{p\})$ is equivalent to $\{p\}$. However this alternative would complicate the semantics. A further justification for the choice made in this paper is that the refinement laws all preserve the guarantee relation on the subcomponents right down to the atomic updates which also guarantee g.

$\longrightarrow$: *Label* $\to$ *Relation* $\to \mathbb{P}(Cmd \times Cmd)$
$\ell ::= \{p\} \mid q \mid \mathsf{env}(q)$
$c ::= \mathbf{nil} \mid \mathbf{abort} \mid \{p\} \mid [p,q] \mid c_1 \,;\, c_2 \mid c_1 \parallel c_2 \mid c_1 \sqcap c_2 \mid c_1 \Cap c_2 \mid c^* \mid c^\omega \mid \langle c \rangle \mid$
$(\mathbf{guar}\ g \bullet c) \mid (\mathbf{rely}\ r \bullet c) \mid$

Definitions :

$$
\begin{array}{rcl}
[p,q] & \widehat{=} & \{p\} \,;\, [q] \\
\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 & \widehat{=} & \langle [b \wedge \mathrm{id}] \rangle \,;\, c_1 \sqcap \langle [\neg b \wedge \mathrm{id}] \rangle \,;\, c_2 \\
\mathbf{while}\ b\ \mathbf{do}\ c & \widehat{=} & (\langle [b \wedge \mathrm{id}] \rangle \,;\, c)^* \sqcap \langle [\neg b \wedge \mathrm{id}] \rangle \\
c \xrightarrow{\ell} c' & \widehat{=} & (\forall R \bullet R \vdash\ c \xrightarrow{\ell} c')
\end{array}
$$

Rule 1 (Assertion).

$$\frac{\mathbf{sat}(p)}{\{p\} \xrightarrow{\{p\}} \mathbf{nil}} \qquad \frac{\mathbf{sat}(\neg p)}{\{p\} \xrightarrow{\{\neg p\}} \mathbf{abort}}$$

Rule 2 (Abort).

$$\mathbf{abort} \xrightarrow{\ell} \mathbf{abort} \qquad \mathbf{abort} \xrightarrow{\ell} \mathbf{nil}$$

Rule 3 (Abort/guar.).

$$\frac{c \xrightarrow{\{false\}} c'}{c \xrightarrow{\ell} c'}$$

Rule 4 (Strengthen).

$$\frac{q_1 \Rrightarrow q_0 \quad R \vdash\ c \xrightarrow{q_0} c' \quad \mathbf{sat}(q_1)}{R \vdash\ c \xrightarrow{q_1} c'}$$

Rule 5 (Iteration).

$$c^\omega \xrightarrow{\mathrm{id}} (c \,;\, c^\omega) \sqcap \mathbf{nil}$$

Rule 6 (Seq. comp.).

$$\frac{R \vdash\ c_1 \xrightarrow{\ell} c_1'}{R \vdash\ (c_1 \,;\, c_2) \xrightarrow{\ell} (c_1' \,;\, c_2)}$$

$$(\mathbf{nil} \,;\, c_2) \xrightarrow{\mathrm{id}} c_2$$

Rule 7 (Choice).

$$\frac{R \vdash\ c_1 \xrightarrow{\ell} c_1'}{R \vdash\ (c_1 \sqcap c_2) \xrightarrow{\ell} c_1'} \qquad \frac{R \vdash\ c_2 \xrightarrow{\ell} c_2'}{R \vdash\ (c_1 \sqcap c_2) \xrightarrow{\ell} c_2'}$$

$$(\mathbf{nil} \sqcap \mathbf{nil}) \xrightarrow{\mathrm{id}} \mathbf{nil}$$

Rule 8 (Guarantee).

$$\frac{R \vdash\ c \xrightarrow{q} c' \quad q \Rrightarrow g}{R \vdash\ (\mathbf{guar}\ g \bullet c) \xrightarrow{q} (\mathbf{guar}\ g \bullet c')} \qquad \frac{R \vdash\ c \xrightarrow{\ell} c' \quad \ell \neq q}{R \vdash\ (\mathbf{guar}\ g \bullet c) \xrightarrow{\ell} (\mathbf{guar}\ g \bullet c')}$$

Rule 9 (Rely).

$$\frac{R \vee r \vdash\ c \xrightarrow{\ell} c'}{R \vdash\ (\mathbf{rely}\ r \bullet c) \xrightarrow{\ell} (\mathbf{rely}\ r \bullet c')}$$

Rule 10 (Concurrency).

$$\frac{c_1 \xrightarrow{q} c_1' \quad c_2 \xrightarrow{\mathsf{env}(q)} c_2'}{(c_1 \parallel c_2) \xrightarrow{q} (c_1' \parallel c_2')} \qquad \frac{c_2 \xrightarrow{q} c_2' \quad c_1 \xrightarrow{\mathsf{env}(q)} c_1'}{(c_1 \parallel c_2) \xrightarrow{q} (c_1' \parallel c_2')} \qquad \frac{c_1 \xrightarrow{\mathsf{env}(e)} c_1' \quad c_2 \xrightarrow{\mathsf{env}(e)} c_2'}{(c_1 \parallel c_2) \xrightarrow{\mathsf{env}(e)} (c_1' \parallel c_2')}$$

$$\frac{c_1 \xrightarrow{\{p\}} c_1'}{(c_1 \parallel c_2) \xrightarrow{\{p\}} (c_1' \parallel c_2)} \qquad \frac{c_2 \xrightarrow{\{p\}} c_2'}{(c_1 \parallel c_2) \xrightarrow{\{p\}} (c_1 \parallel c_2')} \qquad \mathbf{nil} \parallel \mathbf{nil} \xrightarrow{\mathrm{id}} \mathbf{nil}$$

Fig. 2. Small-step semantics for a subset of the language

Assume $\ell s, \ell s_i$ are finite sequences of labels.

Rule 11 (Specification command).

$$\frac{(\overset{\circ}{9}/\ell s) \Rrightarrow q \quad \mathbf{sat}(\ell s) \quad \mathsf{esteps}(\ell s)\ \mathbf{within}\ R}{R \vdash\ [q] \xrightarrow{\ell s} \mathbf{nil}}$$

Rule 12 (Atomic rel).

$$\frac{\mathsf{psteps}(\ell s) = \langle r \rangle \quad \mathsf{esteps}(\ell s)\ \mathbf{within}\ R}{\langle\langle r \rangle\rangle \xrightarrow{\ell s} \langle\langle r \rangle\rangle}$$

Rule 13 (Atomic cmd).

$$\frac{c \xrightarrow{\ell se_1.\ell s.\ell se_2} \mathbf{nil} \qquad \mathsf{psteps}(\ell se_1) = \mathsf{psteps}(\ell se_2) = \langle\rangle \quad \mathsf{esteps}(\ell s)\ \mathbf{within}\ \mathrm{id}}{\langle c \rangle \xrightarrow{\ell se_1.(\overset{\circ}{9}/\ell s).\ell se_2} \mathbf{nil}}$$

Rule 14 (Atomic spec. cmd).

$$\frac{[q] \xrightarrow{\ell s} \mathbf{nil} \quad \mathsf{psteps}(\ell s) = \langle q \rangle}{\langle [q] \rangle \xrightarrow{\ell s} \mathbf{nil}}$$

Rule 15 (Concurrency — multistep (derived)).

$$\frac{c_1 \xrightarrow{\ell s_1} \mathbf{nil} \quad c_2 \xrightarrow{\ell s_2} \mathbf{nil} \quad def(\ell s_1 \parallel \ell s_2)}{c_1 \parallel c_2 \xrightarrow{\ell s_1 \parallel \ell s_2} \mathbf{nil}}$$

Rule 16 (Finite iteration).

$$\frac{(c\ ;\ c^*) \sqcap \mathbf{nil} \xrightarrow{\ell s} \mathbf{nil}}{c^* \xrightarrow{\ell s} \mathbf{nil}}$$

where

Definition 63 (Trace interleaving).

$$\begin{array}{rcl}
\langle\rangle \parallel \langle\rangle & = & \langle\rangle \\
q.\ell s_1 \parallel \mathsf{env}(q).\ell s_2 & = & q.(\ell s_1 \parallel \ell s_2) \\
\mathsf{env}(q).\ell s_1 \parallel q.\ell s_2 & = & q.(\ell s_1 \parallel \ell s_2) \\
\mathsf{env}(q).\ell s_1 \parallel \mathsf{env}(q).\ell s_2 & = & \mathsf{env}(q).(\ell s_1 \parallel \ell s_2) \\
\{p\}.\ell s_1 \parallel \ell s_2 & = & \{p\}.(\ell s_1 \parallel \ell s_2) \\
\ell s_1 \parallel \{p\}.\ell s_2 & = & \{p\}.(\ell s_1 \parallel \ell s_2)
\end{array}$$

Fig. 3. Multi-step semantics of the language

7.3. Refinement

Intuitively, $c \sqsubseteq d$ if every behaviour (trace) of d is a possible behaviour of c, where both are executed in a default rely context (id). A more general definition of refinement with respect to some rely context R is also useful. Only terminating executions are considered.

Definition 64 (Refinement).

$$c \sqsubseteq d \quad \widehat{=} \quad d \xrightarrow{\ell s} \mathbf{nil} \wedge (\mathsf{esteps}(\ell s)\ \mathbf{within}\ \mathsf{rely}(c) \vee \mathrm{id}) \Rightarrow (\exists\, \ell s' \bullet c \xrightarrow{\ell s'} \mathbf{nil} \wedge \ell s' \leq \ell s)$$

We need to allow weakening of assertion labels, otherwise direct trace inclusion would work. Define

$$\begin{array}{rcll}
\langle\rangle & \leq & \langle\rangle & \\
q.\ell s & \leq & q.\ell s' & \text{if } \ell s \leq \ell s' \\
\mathsf{env}(r).\ell s & \leq & \mathsf{env}(r).\ell s' & \text{if } \ell s \leq \ell s' \\
\{p'\}.\ell s & \leq & \{p\}.\ell s' & \text{if } p \Rrightarrow p' \text{ and } \ell s \leq \ell s'
\end{array}$$

The important property is that identical traces satisfy $\leq$ (i.e., $\ell s \leq \ell s$), and hence, for all ℓs,

$$d \xrightarrow{\ell s} \mathbf{nil} \wedge (\mathsf{esteps}(\ell s)\ \mathbf{within}\ \mathsf{rely}(c) \vee \mathrm{id}) \Rightarrow c \xrightarrow{\ell s} \mathbf{nil} \quad \Rightarrow \quad c \sqsubseteq d \tag{25}$$

This stronger property is typically holds and is used in the following proofs.

$e ::= v \mid x \mid e_1 \wedge e_2 \mid e_1 + e_2 \mid \ldots$
$c ::= x := e \mid [[b]]$

Rule 17 (Evaluate variable).

$$x \xrightarrow{x=v \wedge \mathrm{id}} v$$

Rule 18 (Evaluate addition).

$$\frac{e_1 \xrightarrow{\ell} e_1'}{e_1 + e_2 \xrightarrow{\ell} e_1' + e_2} \qquad \frac{e_2 \xrightarrow{\ell} e_2'}{v + e_2 \xrightarrow{\ell} v + e_2'} \qquad \frac{v = v_1 + v_2}{v_1 + v_2 \xrightarrow{\mathrm{id}} v}$$

Rule 19 (Assignment).

$$\frac{e \xrightarrow{\ell} e'}{x := e \xrightarrow{\ell} x := e'} \qquad x := v \xrightarrow{x'=v \wedge \mathrm{id}(\bar{x})} \mathbf{nil}$$

Rule 20 (Code-guard).

$$\frac{b \xrightarrow{\ell} b'}{[[b]] \xrightarrow{\ell} [[b']]} \qquad [[\mathit{true}]] \xrightarrow{\mathrm{id}} \mathbf{nil}$$

Rule 21 (Code environment steps).

$$\frac{c \in \{e, x := e\}}{c \xrightarrow{\mathsf{env}(\mathit{true})} c}$$

Fig. 4. Semantics of CODE

7.4. Proof of laws involving rely commands

We now provide prooofs for the following refinement laws from the body of the paper.

$$c = (\mathbf{rely}\,\mathrm{id} \bullet c) \tag{26}$$
$$c \sqsubseteq (\mathbf{rely}\, r \bullet c) \tag{27}$$
$$(\mathbf{rely}\, r_0 \vee r_1 \bullet c) = (\mathbf{rely}\, r_0 \bullet \mathbf{rely}\, r_1 \bullet c) \tag{28}$$
$$(\mathbf{rely}\, r \bullet \mathbf{guar}\, g \bullet c) = (\mathbf{guar}\, g \bullet \mathbf{rely}\, r \bullet c) \tag{29}$$
$$c \sqsubseteq (\mathbf{rely}\, r \bullet c) \parallel \langle [r \vee \mathrm{id}] \rangle^* \tag{30}$$

Plus: refinement rules for assertions, if, and while.

The proof of $c = (\mathbf{rely}\,\mathrm{id} \bullet c)$ is straightforward from Definition 25; the rely context of id does not change the form of the traces nor restrict the environment of the traces in each case.

The following properties hold by induction on the language.

$$c \xrightarrow{\ell s} \mathbf{nil} \Leftrightarrow c \xrightarrow{\ell s} \mathbf{nil} \tag{31}$$

$$c \xrightarrow{\ell s} \mathbf{nil} \wedge \mathsf{esteps}(\ell s)\ \mathbf{within}\ r_1 \Rightarrow c \xrightarrow{\ell s} \mathbf{nil} \tag{32}$$

In all the following proofs, let $\mathsf{rely}(c) = R$, and assume

$$\mathsf{esteps}(\ell s)\ \mathbf{within}\ R \vee \mathrm{id} \wedge \mathbf{sat}(\ell s) \tag{33}$$

Prove $c \sqsubseteq (\mathbf{rely}\, r \bullet c)$.

$(\mathbf{rely}\, r \bullet c) \xrightarrow{\ell s} \mathbf{nil}$

$\equiv$ Rule 9

$c \xrightarrow{\ell s} \mathbf{nil}$

$\equiv$ (31)

$c \xrightarrow{\ell s}\!\!\!\!\rightarrow$

$\equiv$ (32), (33)

$c \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil}$

$\Rightarrow$ (31)

$c \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil}$

QED

Prove $(\mathbf{rely}\, r_0 \vee r_1 \bullet c) = (\mathbf{rely}\, r_0 \bullet \mathbf{rely}\, r_1 \bullet c)$.

$(\mathbf{rely}\, r_0 \bullet \mathbf{rely}\, r_1 \bullet c) \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil}$

$\equiv$ Rule 9

$(\mathbf{rely}\, r_1 \bullet c) \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil}$

$\equiv$ Rule 9

$c \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil}$

$\equiv$ Rule 9

$(\mathbf{rely}\, r_0 \vee r_1 \bullet c) \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil}$

QED

Prove $(\mathbf{rely}\, r \bullet \mathbf{guar}\, g \bullet c) = (\mathbf{guar}\, g \bullet \mathbf{rely}\, r \bullet c)$.

$(\mathbf{rely}\, r \bullet \mathbf{guar}\, g \bullet c) \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil}$

$\equiv$ Rule 9

$(\mathbf{guar}\, g \bullet c) \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil}$

$\equiv$ Rule 8

$c \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil} \wedge \mathsf{psteps}(\ell s)\ \mathbf{within}\ g$

$\equiv$ Rule 9

$(\mathbf{rely}\, r \bullet c) \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil} \wedge \mathsf{psteps}(\ell s)\ \mathbf{within}\ g$

$\equiv$ Rule 8

$(\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet c)) \xrightarrow{\ell s}\!\!\!\!\rightarrow \mathbf{nil}$

QED

Prove $(\mathbf{rely}\, r \bullet c) \sqsubseteq (\mathbf{rely}\, r \bullet c) \parallel \langle [r \vee \mathrm{id}] \rangle^*$.

Exercise for the reader..

7.5. Axiomatic semantics

To compare with related work it is worth considering how the classic Jones-style RG quintuples could be tackled in our language.

The meaning of the quintuple $\{P, R\}c\{G, Q\}$ in the [CJ07] format is something like the following, for all σ:

$$(\sigma \in P \land (c, \sigma) \underset{R}{\longrightarrow} (\mathbf{nil}, \sigma') \quad \Rightarrow \quad (\sigma, \sigma') \in Q) \qquad \land \qquad \{P, R\} \vdash c \textbf{ within } G$$

(This is concocted from Theorem 17 and some other intuitions, so may not be right.)

Inference rule semantics We posit the the following definition of $\{P, R\}c\{G, Q\}$ in our language and semantics. For all satisfiable traces ℓs:

$$(\langle\!\langle P \land \mathrm{id} \rangle\!\rangle c \xrightarrow{\ell s} \mathbf{nil}) \land (\mathsf{esteps}(\ell s) \textbf{ within } R) \quad \Rightarrow \quad (\fatsemi/\ell s \Rightarrow Q) \land (\mathsf{psteps}(\ell s) \textbf{ within } G)$$

This states that c satisfies the specification if, for all traces starting in a state that satisfies P generated when the environment satisfies R, then the combined effect of ℓs implies the postcondition, and additionally that every step of c satisfies the guarantee.

Inference rules for new command types We now present (without proof) the relevant inference rules for the new abstract command types.

$$\frac{\{P, R\}c\{G, Q\} \quad R \Rrightarrow r}{\{P, R\}(\mathbf{rely}\, r \bullet c)\{G, Q\}} \qquad \frac{\{P, R\}c\{G', Q\} \quad (G' \land g) \Rrightarrow G}{\{P, R\}(\mathbf{guar}\, g \bullet c)\{G, Q\}} \qquad \frac{(P \land q) \Rrightarrow Q}{\{P, true\}[q]\{true, Q\}}$$

I remember Ian doing versions of these at one time, but I can't find the email (it may have been in an attachment). I think these rules are a bit different

The rule for assertion (perhaps also novel in the R/G world to this paper?), and the rule for sequential comp.

$$\frac{(P \land p \land \mathrm{id}) \fatsemi R^* \Rrightarrow Q}{\{P, R\}\{p\}\{\mathrm{id}, Q\}} \qquad \frac{\{P, R\}c_1\{G, P'\} \quad \{P', R\}c_2\{G, Q\} \quad (P' \fatsemi R^*) \Leftrightarrow P'}{\{P, R\}c_1 \,;\, c_2\{G, Q\}}$$

We can now prove $\{P, G\}(\{P\} \,;\, \mathbf{guar}\, G \bullet \mathbf{rely}\, R \bullet [Q'])\{R, Q\}$ when $(P \fatsemi R^*) \fatsemi Q' \Rrightarrow Q$.

$$\frac{\{P, R\}\{P\}\{true, P \fatsemi R^*\} \quad \dfrac{\dfrac{\{P \fatsemi R^*, R\}[Q']\{true, Q\}}{\{P \fatsemi R^*, R\}(\mathbf{rely}\, R \bullet [Q'])\{true, Q\}}}{\{P \fatsemi R^*, R\}(\mathbf{guar}\, G \bullet \mathbf{rely}\, R \bullet [Q'])\{G, Q\}}}{\{P, R\}(\{P\} \,;\, \mathbf{guar}\, G \bullet \mathbf{rely}\, R \bullet [Q'])\{G, Q\}}$$

Relationship between refinement and inference One way of relating the two is the following theorem.

$$\{P, R\}c\{G, Q\} \quad \Leftrightarrow \quad \{P\}(\mathbf{guar}\, G \bullet \mathbf{rely}\, R \bullet [Q]) \sqsubseteq c$$

Taking this as a desirable property, let use it as a way of approximating what the definition of refinement must be.

$$c \sqsubseteq d \quad \widehat{=} \quad d \xrightarrow{\ell s} \mathbf{nil} \quad \Rightarrow \quad (\exists\, \ell s' \bullet c \xrightarrow{\ell s'} \mathbf{nil} \land \ell s \approx \ell s')$$

Obviosuly we must define $\approx$ on traces. As discussed via email, pure equivalence does not suffice (if it did, we could simplify the above definition to just trace inclusion). The more likely result is that stuttering should be removed, although I think the semantics is saying that collecting the effect ($\fatsemi/\ell s$) is the basis. However, as discussed via email, that is also problematic when it comes to code.

We now ask, given $\{P\}(\mathbf{guar}\, G \bullet \mathbf{rely}\, R \bullet [Q]) \xrightarrow{\ell s} \mathbf{nil}$ according to our semantics, what properties of ℓs can we derive?

First we note the following two general properties of assertion-prefixed traces.

$$\frac{c \xrightarrow{\ell s} \mathbf{nil} \quad \mathbf{sat}(\{P\}.\ell s)}{\{P\}c \xrightarrow{\{P\}.\ell s} \mathbf{nil}} \qquad \frac{\mathbf{sat}(\{\neg P\}.\ell s)}{\{P\}c \xrightarrow{\{\neg P\}.\ell s} \mathbf{nil}}$$

The second property states that if $\neg P$ is satisfiable initially then c can generate any trace.

The property for guarantee is

$$\frac{c \xrightarrow{\ell s} \mathbf{nil} \quad \mathsf{psteps}(\ell s)\ \mathbf{within}\ G}{(\mathbf{guar}\ G \bullet c) \xrightarrow{\ell s} \mathbf{nil}}$$

This must hold, unless ℓs is an abort trace, in which the guarantee need not be satisfied.

For a rely command we focus on a specification command. As with assertions, we have two possibilities, covering the abort cases.

$$\frac{{}^{\circ}/_{9}/\ell s \Rrightarrow Q \quad \mathsf{esteps}(\ell s)\ \mathbf{within}\ R}{(\mathbf{rely}\ R \bullet [Q]) \xrightarrow{\ell s} \mathbf{nil}} \qquad \frac{\neg\, \mathsf{esteps}(\ell s)\ \mathbf{within}\ R}{(\mathbf{rely}\ R \bullet [Q]) \xrightarrow{\ell s} \mathbf{nil}}$$

The second property just states that the command generates any trace (behaves as abort) if the rely condition is not satisfied by the environment. Note that if instead of $[Q]$ here we used the more general c, in particular, if we allowed the case where c is code, then we would have to account for a sensible prefix trace up to the point where the rely is violated.

From these properties we may deduce:

$$\{P\}(\mathbf{guar}\ G \bullet \mathbf{rely}\ R \bullet [Q]) \xrightarrow{\ell s} \mathbf{nil} \quad \Leftrightarrow \\ \mathbf{sat}(\{P\}.\ell s) \Rightarrow (\mathsf{psteps}(\ell s)\ \mathbf{within}\ G \wedge (\mathsf{esteps}(\ell s)\ \mathbf{within}\ R \Rightarrow {}^{\circ}/_{9}/\ell s \Rrightarrow Q))$$

Now with a bit of hand waving, we can state that the requirement $\mathsf{psteps}(\ell s)$ **within** G need hold only when ℓs is not an abort trace, which means it need hold only when P holds (already accounted for), and when the rely is satisfied. With a bit of manipulation this gives:

$$\mathbf{sat}(\{P\}.\ell s) \wedge \mathsf{esteps}(\ell s)\ \mathbf{within}\ R \quad \Rightarrow \quad (\mathsf{psteps}(\ell s)\ \mathbf{within}\ G \wedge {}^{\circ}/_{9}/\ell s \Rrightarrow Q)$$

There is an obvious similarity with the definition for satisfiability given above, which is a good basis for continuing the proof..

8. Conclusions

8.1. Summary and comparisons

The idea of adding state assertions to imperative programs is crucial to the ability to decompose proofs about such program texts. The most influential source of this idea is [Flo67] although it is interesting to note that pioneers such as Turing and von Neumann recognised that something of the sort would aid reasoning. The key contribution of [Hoa69] was to change the viewpoint away from program annotations to a system of judgements about "Hoare triples". One huge advantage of this viewpoint was that it made clear a notion of compositional development.

Other key developments include the idea of data refinement (or reification) and the importance of data type invariants. This is not the place to attempt a complete history but it is important to note that the "refinement calculus" as described by Carroll Morgan brings together the strands of development for sequential programs into an elegant calculus in which algebraic properties are clear.

There are several extra challenges when one moves from confronting sequential programs to their concurrent counterparts. Here again, this is not the place for a complete catalogue of the issues — and certainly not a history. The aspect of parallelism that goes under the heading of data races can be divided into avoiding such races and reasoning about those that are unavoidable. Conventional wisdom indicates that concurrent separation logic [OYR09] is useful for the former way of reasoning and that rely-guarantee thinking is useful for the latter.

The current paper shows how the sort of explicit reasoning about interference that underlies rely-guarantee thinking can be recast into the refinement calculus mould. It transpires that there is a very good fit of basic objectives. This includes the simple observation that Morgan also embraced relations (rather than single state predicates) as the cornerstone of specifications; more important is the shared acknowledgement that compositional reasoning is a necessity if a method is to scale up to large problems.

Rather than treat a specification of a program as a four-tuple of pre, rely, guarantee and postcondition, the approach taken here has been to consider initially guarantees and relies separately, and rather than just apply them to a pre-post specification, allow them to be applied more generally to commands. The guarantee command $(\mathbf{guar}\ g \bullet c)$ constrains the behaviour of c so that only atomic steps that respect g are permitted. It generalises nicely to being applied to

an arbitrary command. Refining a guarantee command can be decomposed into refining the body of the command, distributing the guarantee into the refinement and then checking that each atomic step maintains the guarantee. Alternatively, in order to ensure that the guarantee is not broken, one can interleave refinement steps with checking that the guarantee is preserved. The choice of strategies is up to the developer. Motivation for the guarantee construct was drawn from the invariant construct of Carroll Morgan and Trevor Vickers and its behaviour in restricting the possible atomic steps mirrors the restrictions introduced by the invariant command on possible states. The guarantee construct is also related to a form of enforced property used in action system refinement in which every action of a system is constrained to satisfy a relation [DH10]. The guarantee construct can be thought of as providing a context in which its body is refined. Nickson and Hayes [NH97] have investigated tool support for contexts such as preconditions and the Morgan-Vickers invariant, and it is clear that the guarantee context could be treated similarly within tool support.

Invariants play an important role in reasoning about "while" loops because if a single iteration of a loop maintains an invariant, any finite number of iterations of the loop will also maintain the invariant. In a similar vein, if every atomic step of a computation maintains an invariant, the whole computation will also maintain the invariant. This motivates the introduction of the guarantee invariant construct (Sect. 3.3). As illustrated in the developments of *findp* as both a sequential (Sect. 3.4) and concurrent (Sect. 6) programs, one can make use of guarantee invariants to ensure that every atomic step of a computation maintains the invariant, and hence the whole computation (including any loops within it) also maintains the invariant. The negative is that one must ensure every atomic step maintains the invariant, although in many cases this is trivial if no variables within the invariant are modified by the step. Guarantee invariants also play a role in the context of concurrency because if every atomic step of a process c maintains an invariant, then a concurrent process d can rely on the invariant being maintained by any interference generated by c.

The thesis by Jürgen Dingel [Din00] and the more accessible [Din02] offer an approach towards a "refinement calculus" view of rely/guarantee reasoning. There are however clear differences about what constitutes such a view in Dingel's writings and in the current paper. Dingel takes a historical five-tuple view of specifications; in fact, his writings do not follow Jones' original relational view because he, for example, has post conditions that are sets of predicates of single states — on this point he follows Colin Stirling [Sti86]. To the current authors, the key reason for a refinement calculus view is to get away from any fixed packaging of the assertions that comprise a specification and to view guar/rely commands in a way that opens up a relational view of the constructs. Furthermore, this presents a path to a more convenient, relational calculus, record of the frames of commands. Given a basic set of commands and their basic laws, it is possible to derive more specific laws that often show nice algebraic relations. To give just one specific example, Law 10 (trading-post-guar) presents an intuitive rule for what often appear to be arbitrary manipulations in earlier papers.

The rely command $(\mathbf{rely}\ r \bullet c)$ guarantees to implement c under interference bounded by the relation r. In this case the generalisation to a body containing and arbitrary command is less successful because in order for $(\mathbf{rely}\ r \bullet c)$ to be feasible c must allow inference bounded by r, and this usually requires c to be in the form of a pre-post specification rather than more basic commands like assignments because the latter tend to be too restrictive to be feasible when included in a non-trivial rely.

The advantage of treating the guarantee and rely constructs separately is that we have been able to develop sets of laws specific to each. This has the advantage of providing a better understanding of the role of guarantees and relies. In particular the defining property of the rely construct

$$c \sqsubseteq (\mathbf{rely}\ r \bullet c) \parallel \langle\!\langle r \vee \mathrm{id} \rangle\!\rangle^*$$

given in Section 4.2 brings out the essence of the role of the rely condition.

Of course, the main point of introducing rely and guarantee constructs is to allow them to be used to express the bounds on interference in parallel compositions. To this end, in Section 5, we have developed refinement laws for parallel composition that have been proved using the more fundamental laws for guarantee and rely constructs along with laws about conjoined specifications/commands.

8.2. Further work

In the sequential refinement calculus because one is only concerned with the end-to-end behaviour from the initial state to the final state, any program can be reduced to an equivalent specification statement.[7] However for concurrent programs the intermediate behaviour of a process can be as important as its overall effect. The rely-guarantee approach

[7] Assuming that angelic choice is not included in the language.

augments pre-post specifications with rely and guarantee relations which allow a pre-rely-guarantee-post specification to express both the assumption its makes of steps made by its environment and the guarantee about the steps its takes. As rely (guarantee) conditions abstract all interference (program) steps, pre-rely-guarantee-post specifications are not rich enough to be able precisely express the behaviour of all processes.[8] One challenge is to increase the expressive power of rely and guarantee conditions to allow a more precise specification of a greater range of processes, while retaining the elegance of the rely-guarantee approach.

As should be clear from the preceding material, the reformulation of rely/guarantee thinking in a refinement calculus mould has suggested new notation and laws. A nice example is the shift from the heavy VDM-like keyword framing in which read/write access is declared to the compact prefix listing of write variables. There is however much scope for further research and progress. In the same area as framing, it would be useful to have a direct notation that showed that a "variable" essentially became a constant throughout a portion of code; in the same vein, [JP11] uses **owns**. Furthermore, the hope for deep links to Separation Logic could be advanced by laws that directly indicate separate access (cf. *ot*, *et* in Sect. 6.4).

There is also a clear case for reconsidering data reification in the new framework. The concept of "possible values" was introduced in [JP11] and linked to non-deterministic states in [HBDJ12] — re-examining the idea in the new framework might also be enlightening. Finally, at least one of the current authors would like to reexamine the trade offs in using "hooked" initial values in relations.

Acknowledgements. This research was supported in part by Australian Research Council Linkage Grant LP0989643 and the EPSRC(UK) TrAmS-2 Platform Grant and EU-funded DEPLOY funding. We would like to thank the referees of an earlier version of this paper for valuable feedback on that version.

References

[Bac81] R.-J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, February 1981.

[Bv98] R.-J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[CH09] R. J. Colvin and I. J. Hayes. CSP with hierarchical state. In M. Leuschel and H. Wehrheim, editors, *Integrated Formal Methods (IFM 2009)*, volume 5423 of *LNCS*, pages 118–135. Springer, 2009.

[CH11] R. J. Colvin and I. J. Hayes. Structural operational semantics through context-dependent behaviour. *Journal of Logic and Algebraic Programming*, 80(7):392–426, 2011.

[CJ00] Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction*, chapter 10, pages 277–307. MIT Press, 2000.

[CJ07] J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4):807–841, 2007.

[Col08] Joey W. Coleman. Expression decomposition in a rely/guarantee context. In Natarajan Shankar and Jim Woodcock, editors, *VSTTE*, volume 5295 of *Lecture Notes in Computer Science*, pages 146–160. Springer, 2008.

[dBHdR99] F. de Boer, U. Hannemann, and W. de Roever. Formal justification of the rely-guarantee paradigm for shared-variable concurrency: a semantic approach. In Jeannette Wing, Jim Woodcock, and Jim Davies, editors, *FM99 Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 714–714. Springer Berlin / Heidelberg, 1999.

[DH10] Brijesh Dongol and Ian J. Hayes. Compositional action system derivation using enforced properties. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *Mathematics of Program Construction (MPC)*, volume 6120 of *LNCS*, pages 119–139. Springer Verlag, 2010.

[Din00] Jürgen Dingel. *Systematic Parallel Programming*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-99-172.

[Din02] J. Dingel. A refinement calculus for shared-variable parallel and distributed programming. *Formal Aspects of Computing*, 14:123–197, 2002.

[dR01] W. P. de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[Hay10] Ian J. Hayes. Invariants and well-foundedness in program algebra. In A. Cavalcanti, D. Déharbe, M.-C. Gaudel, and J. Woodcock, editors, *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, volume 6255 of *LNCS*, pages 1–14. Springer-Verlag, 2010. Invited keynote paper.

[HBDJ12] Ian J. Hayes, Alan Burns, Brijesh Dongol, and Cliff B. Jones. Comparing models of nondeterministic expression evaluation. *The Computer Journal*, (submitted), 2012.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[8] Technically this can be overcome by adding some form of program counter to each process and labels to each step of the program but such an approach destroys the elegance of the rely/guarantee abstractions.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and System*, 5(4):596–619, 1983.

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[Jon07] C. B. Jones. Splitting atoms safely. *Theoretical Computer Science*, 375(1–3):109–119, 2007.

[Jon10] C. B. Jones. The role of auxiliary variables in the formal development of concurrent programs. In Cliff B. Jones, A. W. Roscoe, and Kenneth Wood, editors, *Reflections on the work of C.A.R. Hoare*, chapter 8, pages 167–188. Springer, 2010.

[JP11] Cliff B. Jones and Ken G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. *Formal Aspects of Computing*, 23(3):289–306, 2011.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mor87] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[Mor88] C. C. Morgan. The specification statement. *ACM Trans. on Prog. Lang. and Sys.*, 10(3), July 1988.

[Mor94] C. C. Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

[MV90] C. C. Morgan and T. N. Vickers. Types and invariants in the refinement calculus. *Science of Computer Programming*, 14:281–304, 1990.

[MV94] C. C. Morgan and T. N. Vickers, editors. *On the Refinement Calculus*. Springer-Verlag, 1994.

[NH97] R. Nickson and I. J. Hayes. Supporting contexts in program refinement. *Science of Computer Programming*, 29(3):279–302, 1997.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Department of Computer Science, Cornell University, 1975.

[OYR09] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. *ACM TOPLAS*, 31(3), April 2009. Preliminary version appeared in 31st POPL, pp268-280, 2004.

[Plo04a] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, July–December 2004.

[Plo04b] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, July–December 2004.

[Sti86] C. Stirling. A compositional reformulation of Owicki-Gries' partial correctness logic for a concurrent while language. In *ICALP'86*. Springer-Verlag, 1986. LNCS 226.

[WDP10] John Wickerson, Mike Dodds, and Matthew Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. Technical Report 774, Computer Laboratory, University of Cambridge, March 2010.