A Rely/Guarantee Reasoning Framework using Computational Tree Logic

Joey W. Coleman
A Rely/Guarantee Reasoning Framework using Computational Tree Logic

J. W. Coleman

Abstract

This paper presents a formulation of a rely/guarantee reasoning framework using an extended version of computational tree logic called Relational CTL*. As a result of using Relational CTL* we gain a rely/guarantee reasoning framework which has the ability to directly reason about fairness constraints; and consider properties about the ordering and frequency of events at the level of development rules.
Bibliographical details

COLEMAN, J.W.

A Rely/Guarantee Reasoning Framework using Computational Tree Logic
[By] J. W. Coleman
(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-1229)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE

Abstract

This paper presents a formulation of a rely/guarantee reasoning framework using an extended version of computational tree logic called Relational CTL*. As a result of using Relational CTL* we gain a rely/guarantee reasoning framework which has the ability to directly reason about fairness constraints; and consider properties about the ordering and frequency of events at the level of development rules.

About the author

Joey earned a BSc (2001) in Applied Computer Science at Ryerson University in Toronto, Ontario. With that in hand he stayed on as a systems analyst in Ryerson's network services group. Following that he took a position at a post-dot.com startup as a software engineer and systems administrator. The technical problems encountered working on a large multi-threaded application sent him back to academia in search of better ways of dealing with concurrency. At Newcastle University he has since earned a MPhil (2005) and a PhD (2008) for work on semantics and formal methods. During his time at Newcastle he has been involved with several projects including the FP7 RODIN project, working on methodology, and was associated with the EPSRC DIRC project. He is currently involved with the EPSRC "Splitting (Software) Atoms Safely" project, working on atomicity in software development methods. His interests cover a broad range of topics in computer science, though the focus has been primarily on programming language semantics and the use of formal methods to model concurrent systems. Recent work has involved rely/guarantee reasoning and structural operational semantics.

Suggested keywords

RELY/GUARANTEE
TEMPORAL LOGIC
CTL*
A Rely/Guarantee Reasoning Framework using Computational Tree Logic

Joey W. Coleman
School of Computing Science
Newcastle University
NE1 7RU, UK
e-mail: j.w.coleman@ncl.ac.uk
December 16, 2010

Abstract

This paper presents a formulation of a rely/guarantee reasoning framework using an extended version of computational tree logic called Relational CTL*. As a result of using Relational CTL* we gain a rely/guarantee reasoning framework which has the ability to directly reason about fairness constraints; and consider properties about the ordering and frequency of events at the level of development rules.

1 Introduction

Interference is the defining characteristic of concurrent software systems. Many formalisms have been developed to reason about interference; this paper is concerned with the family of formalisms gathered under the banner of rely/guarantee reasoning, in particular as described in [CJ07] and the author’s thesis [Col08]. Rely/guarantee reasoning frameworks follow the format of Hoare’s rules for reasoning about program developments [Hoa69]; unlike Hoare’s rules rely/guarantee development rules deal with concurrent systems explicitly.

The key element of Coleman and Jones’ formulation of a rely/guarantee framework, for our purposes, is that it has been proven to be sound with respect to a language which has a structural operational semantic (SOS) definition. Definitions given in the form of a SOS give rise to a labelled transition system; this property implies that, ultimately, rely/guarantee reasoning can be considered to be about properties of these labelled transition systems.

Computational tree logic (CTL*) [CES86] is also used to reason about transition systems and is suitable for reasoning about events which happen an arbitrary but finite number of times. Reasoning about these events in a rely/guarantee

\[\text{References}\]

[1] See Plotkin’s work structural operational semantics [Plo81, Plo04].
framework is more difficult: rely/guarantee conditions focus only on two states—two points in time—without direct reference to the past. Conditions which need to reference the occurrence of past events requires either that the system records the event in some fashion, or the use of auxiliary variables about the system.

The transition systems which correspond to rely/guarantee specifications and those used by CTL bear (at least) a superficial resemblance. The first aim of this effort is to deepen that resemblance and attempt to make a usable framework for relating CTL and rely/guarantee. Secondary goals include providing a convenient way of extending rely/guarantee to cleanly express progress conditions.

In [BKP84] there is a set of Hoare-like development rules that are formulated in terms of temporal logic. This set includes a rule for parallel program composition, but suffers from the disadvantage that properties must be expressed as predicates over single program states. Typical rely/guarantee reasoning frameworks allow for relational properties over pairs of program states.

This paper is structured as follows: in Section 2 we describe the semantics of a family of programming languages. This provides properties which constrain the type of systems in which we are interested. Section 3 describes the usual, non-temporal formulation of rely/guarantee frameworks and relates them to the language semantics. Section 4 describes our relational version of CTL*, called Relational CTL*. Section 5 gives a formulation of a rely/guarantee reasoning system which uses Relational CTL* as its reasoning system. We then conclude the paper with Section 6, describing some future work. Sections 2 through 4 set out the technical background of our effort, and Section 5 gives the resulting framework.

There are two major contributions in this work: first, we have a rely/guarantee framework that can directly express fairness and progress conditions; and second, we have a rely/guarantee framework which can reason about the ordering and frequency of events with more precision than traditional Jones-style frameworks. However, the trade-off from these benefits results in development rules in the framework which are, arguably, more verbose and harder to understand. Notwithstanding the increased verbosity of the development rules, this framework does offer an alternate take on rely/guarantee reasoning in general.

2 Programming Language Semantics

The usual formulations of rely/guarantee reasoning frameworks are typically grounded in the semantics of some language to ensure that the resulting development rules are sound. This is not always explicit, though this is a requirement for actual development purposes. This section gives a semantic specification for a family of languages that the rest of the paper depends upon.

The language family used in this paper is given in terms of the least relation satisfying a collection of inference rules, along with the type descriptions and well-formedness constraints on the relevant entities. This is the usual way of describing a language in the style of structural operational semantics [Plo81, Plo04]. The ac-
tual instances of the language family that we are interested in is not relevant beyond the properties described below; however, we have in mind languages such as those in the author’s joint paper with Jones [CJ07] and the author’s thesis [Col08].

The status of the system at any given instant is represented by a configuration and is simply a pair consisting of a program and a state. The set of all programs is denoted $\Pi$, with individuals in the set denoted as $\pi$ and decorated variations thereof. The set of all states is denoted by $\Sigma$, with individuals denoted as $\sigma$ and variations thereof. States are partial mappings from identifiers, $Id$, to values, $Value$, giving the type $\Sigma = Id \rightarrow Value$. The set of all configurations is denoted $\text{Config} = \Pi \times \Sigma$; individual configurations are typically denoted as $c$ and decorated variations thereof.

The primary transition relation is denoted $\lambda \rightarrow$ where $\lambda$ is either $p$ or $e$, respectively representing program transitions and environmental transitions. The actual definition of the program transition relation is dependent on the specific language. However, the following properties must hold on any proposed transition relation.

1. Program termination is indicated by a transition to a configuration of the form $(\text{nil}, \sigma)$ where $\text{nil}$ is a constant indicating the empty program and $\sigma$ is the final state at termination.

2. Configurations of the form $(\text{nil}, \sigma)$ are not in the domain of the program transition relation, $p \rightarrow$. This ensures that no further computation can occur due to the program.

3. All program transitions result in a change to the program element of the configuration. Thus, all transitions of the form $(\pi, \sigma) \xrightarrow{p} (\pi', \sigma')$ imply that $\pi \neq \pi'$.

4. Environmental transitions, $e \rightarrow$, never alter the program component; thus, all transitions of the form $(\pi, \sigma) \xrightarrow{e} (\pi', \sigma')$ imply that $\pi = \pi'$.

5. For any configuration $(\pi, \sigma)$, if $\pi$ is not $\text{nil}$ then there must be a successor configuration under the $p \rightarrow$ transition, assuming a suitable state, $\sigma$.

The definition of the environmental transition relation can be given in terms of a secondary relation, $\text{Rely}$.

$$\text{Rely}(\sigma, \sigma') \quad (\pi, \sigma) \xrightarrow{p} (\pi', \sigma')$$

The secondary relation, $\text{Rely}$, anticipates the rely condition of Section 3; it is sufficient, for the moment, to know that $\text{Rely}$ encodes the changes that the environment may make to the state.

The single critical property of the environmental transition is that it must preserve the program component through all transitions; this is the fourth point in the list above. Beyond that constraint, the environmental transition may, in general, be reflexive and transitive as is convenient.
Finally, $\lambda \rightarrow^*$ denotes the usual reflexive and transitive closure of $\lambda \rightarrow$, with $p \rightarrow^*$ and $e \rightarrow^*$ also denoting their respective closures.

### 3 Classical Rely/Guarantee

A program specification in a rely/guarantee reasoning framework is typically written

$$\{ \text{Pre}, \text{Rely} \} \pi \{ \text{Guar}, \text{Post} \}$$

where

- **Pre** is the pre condition, a predicate over states. It characterizes the set of initial states under which the program will perform correctly.

- **Rely** is the rely condition, a relation over states. This relation characterizes the possible changes that the program’s environment may make to the state component of the configuration. The program must tolerate these changes during its execution and behave correctly despite them.

- $\pi$ is the program itself.

- **Guar** is the guarantee condition, a relation over states. This relation is a constraint on the behaviour of the program; every step –i.e. every semantic transition– the program makes during execution must satisfy this relation with respect to change in the associated states.

- **Post** is the post condition, a relation over states. The post condition is a constraint upon the program: it relates the initial state component at the start of execution to the state component at termination of the program. The program’s execution must result in a state which satisfies the post condition relative to the starting state.

All four of the conditions must be total over (pairs of) states. We assume that it is always possible to determine whether or not a state satisfies a condition, despite the fact that the conditions themselves are usually expressed using the logic of partial functions [JM94]. To this end we will assume that if the formulation of a condition is not defined with respect to a given (pair of) states then the condition is not satisfied; i.e. if the condition does not denote, we consider it equivalent to false for the purposes of evaluating program satisfaction.

A rely/guarantee specification describes the desired behaviour of a program in four conditions: assuming a state satisfying **Pre** and environmental behaviour within **Rely** then program $\pi$ will behave within **Guar** and produce a state satisfying **Post** when paired with its initial state. One further constraint on a program that satisfies a rely/guarantee specification is that, when executed starting in any situation which satisfies **Pre** and **Rely**, the program must terminate.

---

$^2$Some formulations define the post condition to be a predicate. We prefer to use a relation.
To emphasise, the pre and rely conditions are both assumptions which are made about the environment in which the program will run; we assume the conditions hold and do not need to take any steps to check their validity. The programmers are only responsible for ensuring that the program acts in such a way as to satisfy the guarantee and post conditions.

A valid rely/guarantee specification must be *satisfiable*; this comes from the notion of the same name in used by VDM [Jon90]. Satisfiability means that, for all elements in the pre condition of a specification, there must exist some element which satisfies the post condition when paired with the first element. This is typically expressed as

\[ \forall \vec{\sigma} \cdot \left( \text{Pre}(\vec{\sigma}) \Rightarrow \exists \sigma \cdot \text{Post}(\vec{\sigma}, \sigma) \right) \]

in VDM literature, though here we will use the more concise point-free form

\[ \text{Pre} \subseteq \text{dom Post} \quad \text{Satisfiable} \]

This property must hold for any rely/guarantee specification but is not sufficient on its own. The interference from the environment must be taken into account and the domain of the guarantee condition must be a superset of the range of the rely condition, i.e.

\[ \text{rng Rely} \subseteq \text{dom Guar} \quad \text{Progress} \]

This ensures that any state resulting from interference can be handled by the program. The converse condition,

\[ \text{rng Guar} \subseteq \text{dom Rely} \quad \text{Composable} \]

requiring that any state arising as a result of program activity must be one which the environment can act upon, is not strictly required. It is not difficult to imagine the utility of a program (or construct therein) which suspends environmental activity; however, a near analogue of this property must be satisfied to be able to compose two rely/guarantee specifications.

Finally, we also require that all elements of the pre condition be in the domain of the guarantee; thus,

\[ \text{Pre} \subseteq \text{dom Guar} \quad \text{Startable} \]

ensures that the program may always act immediately, without having to wait for some interference to alter its initial state first.

We view program execution as a sequence of configurations with each successive configuration in the sequence satisfying the semantic transition relation \( \lambda \mapsto \), as described in Section 2. These sequences are members of the set \( \text{Config}^\omega \), and we will denote individual program components of a configuration within a sequence as \( \pi_i \), and individual state components of a configuration as \( \sigma_i \), where \( i \) is an index into the sequence. Indexing uses the natural numbers and start from 0.
The assumptions about the pre and rely conditions can be formalised in this sequence model as

\[ \text{Pre}(\sigma_0) \quad \text{Pre Assumption} \]

and

\[ \forall i \in \mathbb{N} \cdot \pi_i = \pi_{i+1} \Rightarrow \text{Rely}(\sigma_i, \sigma_{i+1}) \quad \text{Rely Assumption} \]

respectively. Any sequence that does not satisfy these two properties is not relevant: a designer of a program that satisfies a rely/guarantee specification does not need to consider such sequences.

The model of execution for rely/guarantee reasoning assumes that the program, if not finished, will always have a chance to execute. In terms of the execution sequences this means that there will never be an infinitely long subsequence of only environment transitions. Though a subsequence of only environment transitions may be arbitrarily long, it must be finite. This corresponds to the requirement that the language semantics always allow \( p \rightarrow \) transitions, but strengthens it to exclude infinitely long subsequences of \( e \rightarrow \) transitions.

All relevant sequences that correspond to a possible execution of a program satisfying a rely/guarantee specification must, in turn, satisfy the following properties. The program must eventually terminate,

\[ \exists i \in \mathbb{N} \cdot \pi_i = \text{nil} \quad \text{Termination} \]

it must satisfy the guarantee condition for every step it makes,

\[ \forall i \in \mathbb{N} \cdot \pi_i \neq \pi_{i+1} \Rightarrow \text{Guar}(\sigma_i, \sigma_{i+1}) \quad \text{Behaviour} \]

and it must satisfy the post condition at the point where it terminates.

\[ \exists i \in \mathbb{N} \cdot (\neg \exists j \in \mathbb{N} \cdot j < i \land \pi_j = \text{nil}) \land \pi_i = \text{nil} \land \text{Post}(\sigma_0, \sigma_i) \quad \text{Correctness} \]

Though it is possible –for small programs– to reason directly in terms of the semantic relation as is done in [HJ08, Hug11], it is more practical to use development rules. The cost of development rules is that each rule must be proven sound with respect to the semantics; this cost only needs to be paid once, however. Examples of such soundness proofs can be found in [CJ07, Col08], which use structural induction, and in [dR01], which uses trace-based reasoning.

Below is a typical decomposition development rule for a parallel composition construct in a possible language.

\[
\{\text{Pre, Rely}_l\} \pi_l \{\text{Guar}_l, \text{Post}_l\} \\
\{\text{Pre, Rely}_r\} \pi_r \{\text{Guar}_r, \text{Post}_r\} \\
\text{Rely} \Rightarrow \text{Rely}_l \lor \text{Rely}_r \\
\text{Guar}_l \lor \text{Guar}_r \Rightarrow \text{Guar} \\
\text{Guar}_l \Rightarrow \text{Rely}_r \\
\text{Guar}_r \Rightarrow \text{Rely}_l \\
\text{Pre} \land \text{Post}_l \land \text{Post}_r \land (\text{Guar}_l \lor \text{Guar}_r \lor \text{Rely}_l \lor \text{Rely}_r)^* \Rightarrow \text{Post} \\
\{\text{Pre, Rely}\} \pi_l \parallel \pi_r \{\text{Guar}, \text{Post}\}
\]
The first two hypotheses describe the form of the rely/guarantee specifications for the two programs that are to be composed. Note that the rely condition in each specification includes the guarantee condition of the other specification; this ensures that both programs can tolerate interference from the other. The third hypothesis ensures that the overall rely condition of the composed program does not violate the assumed rely conditions of the component programs. The fourth hypothesis is a constraint on the behaviour of both programs to ensure that they behave within the guarantee condition of the composed specification. The fourth and fifth hypotheses ensure that the behaviour of each component program does not violate the assumed rely condition of its counterpart. The last hypothesis is a check to ensure that the composition of the two programs satisfies the post condition of the composed specification; this hypothesis is structured to capture the effects of interaction between the two programs.

A simplification of this rule is used in [CJ07] and elsewhere and is only one of the possibilities for such a rule; this version is structured to match the example in Section 5.4.2.

4 Relational CTL*

We will assume that the reader is familiar with temporal logics in general and with CTL* [Eme90] in particular. Relational CTL* is based on CTL* as formulated by Emerson, though we have omitted the until (U) binary operator; the reasons for this are described in [Col10]. It should be possible to include the until operator in a version of Relational CTL*, however, its use is not necessary in this paper.

The primary motivation behind this variant of CTL* is the need, in Jones-style rely/guarantee reasoning, to directly refer to properties over pairs of states; that is, the need to directly handle relations over the system state at different points in time. The rely, guarantee, and post conditions in a Jones-style rely/guarantee system are all relational and the overall reasoning system benefits from this. Temporal logics generally are formulated on the use of a set of atomic propositions that are predicates over single states. This presents some difficulties when encoding a rely/guarantee system into a temporal logic, and it is our position that the difficulty is best avoided. To this end we have generated this variant of CTL* using a set of atomic propositions which contains relations.

In the following syntax and semantics, $P$ and $Q$ and their decorated variants, refer to specific relational atomic propositions; $a$ and $b$ and their decorated variants refer to arbitrary Relational CTL* formulae; $M$ refers to a model; $s$ and its decorated variants refer to specific states in a model; and $x$ and its decorated variants refers to a temporal path. Note that superscripted variations on paths, i.e. $x^i$, refers to the suffix of path $x$ starting at the $i + 1$st state; thus, $x^0 = x$ and $x^1$ is the suffix of $x$ starting from the second state.

Two function-like notations are used in the semantic definitions which follow. First, $x \in \text{paths}(s)$ indicates that a specific sequence of states, $x$, is a possible path
Each atomic proposition $P$ is a state formula.

If $a$, $b$ are state formulae then so are $a \land b, \neg a$

If $a$ is a path formula then $Ea, Aa$ are state formulae

If $a$ is a state formula then so is $Sa$

Every state formula is a path formula

If $a$, $b$ are path formulae then so are $a \land b, \neg a$

If $a$ is a path formula then so are $Xa, Fa, Ga$

If $a$ is a path formula then so is $Sa$

Figure 1: Syntax of Relational CTL*

| S1 | $M, s_h, s_0 \models P$ iff $P \in L(s_h, s_0)$ |
| S2a | $M, s_h, s_0 \models a \land b$ iff $M, s_h, s_0 \models a$ and $M, s_h, s_0 \models b$ |
| S2b | $M, s_h, s_0 \models \neg a$ iff $\neg(M, s_h, s_0 \models a)$ |
| S3a | $M, s_h, s_0 \models Ea$ iff $\exists x \in \text{paths}(s_0) \cdot M, s_0, x \models a$ |
| S3b | $M, s_h, s_0 \models Aa$ iff $\forall x \in \text{paths}(s_0) \cdot M, s_0, x \models a$ |
| S4  | $M, s_h, s_0 \models Sa$ iff $M, s_0, s_0 \models a$ |
| P1  | $M, s_h, x \models a$ iff $M, s_h, \text{first}(x) \models a$ |
| P2a | $M, s_h, x \models a \land b$ iff $M, s_h, x \models a$ and $M, s_h, x \models b$ |
| P2b | $M, s_h, x \models \neg a$ iff $\neg(M, s, x \models a)$ |
| P3a | $M, s_h, x \models Xa$ iff $M, s_h, x^1 \models a$ |
| P3b | $M, s_h, x \models Fa$ iff $\exists i \cdot M, s, x^i \models a$ |
| P3c | $M, s_h, x \models Ga$ iff $\forall i \cdot M, s, x^i \models a$ |
| P4  | $M, s_h, x \models Sa$ iff $M, \text{first}(x), x \models a$ |

Figure 2: Semantics of Relational CTL*

starting at $s$ in the model; the model is taken from the context of the use. Second, $\text{first}(x)$ is a reference to the initial state in the given sequence, $x$. Thus, it is true that $\forall s \cdot \forall x \in \text{paths}(s) \cdot \text{first}(x) = s$.

The syntax of Relational CTL* is given in figure 1. Formulae are comprised of atomic propositions, the usual logical connectives $\land$ and $\neg$; the CTL* path quantifiers $E$ and $A$; the CTL* path operators $X$, $F$, and $G$; and our $\text{shift}$ operator $S$. All of these elements—with the exception of atomic propositions and the $\text{shift}$ operator— are defined in the usual way.

A model, $M$, in Relational CTL* is defined in a similar manner as in CTL*. A model is a tuple, $(S, R, L)$: $S$ is the set of states; $R$ is the accessibility relation between states; and $L$ is the interpretation, mapping a pair of states to the set of atomic propositions which hold for that pair.

An assertion using a state formula in Relational CTL* is written

$M, s_h, s_0 \models a$
where $M$ is the ground model, both $s_h$ and $s_0$ are states, and $a$ is the formula that we are interested in. The second state, $s_0$, performs the same function as the single state in a regular CTL* formula; it acts as the reference from which $p$ is interpreted and from which paths are rooted by the A and E quantifiers. The first state, $s_h$, is a “held-aside” state and provides one of the pair of states over which atomic propositions –relations– are checked.

Semantic rule S1 in figure 2 gives the basic case for atomic propositions. An atomic proposition, $P$, holds in a model $M$ given states $s_h$ and $s_0$ if and only if $P$ is in the set of atomic propositions which $L$ designates to be true for the pair $(s_h, s_0)$.

An assertion using a path formula in Relational CTL* is written

$$M, s_h, x \models a$$

where the difference relative to the previous state-based assertion is a the path $x$. Path-based assertions are essentially the same as in CTL*.

The shift operator has the effect of replacing the held-aside state, as can been seen in semantic rules S4 and P4. For state assertions, the shift operator replaces the held-aside state with the reference state $s_0$; for path assertions the held-aside state is replaced with the initial state of the path $x$.

We will sometimes use the abbreviation $M, s_0 \models p$ in place of $M, s_0, s_0 \models p$ and similarly for path formulae. Thus,

$$M, s_0 \models p \triangleleft M, s_0, s_0 \models p$$

This shorthand is convenient for the rules in Section 5 where the initial held-aside state is always the same as the initial state.

The full set of semantic definitions for Relational CTL* are in figure 2. Let us consider a few examples in Relational CTL* to illustrate how the shift operator works. First, consider the basic assertion

$$M, s_h, s_0 \models P$$

where $P$ is some relation in the set of atomic propositions. This case is simple: $P$ holds if it is in the set designated by $L$ for the pair of states $(s_h, s_0)$; this is as noted earlier, but also given a graphical depiction. A similar assertion using the shift operator,

$$M, s_h, s_0 \models SP$$

holds if $P$ is in the set designated by $L$ for $(s_0, s_0)$. Semantic rule S4 means that this assertion is equivalent to $M, s_0, s_0 \models P$.

For the next and eventually operators we will consider a linear example,\(^3\) with $x$ being the path starting at $s_0$ and continuing with $s_i$ for $i \in \mathbb{N}$. An assertion such as

$$M, s_h, x \models XP$$

\(^3\)without loss of generality...
holds where $P$ is in the set designated by $L(s_h, s)$, as would be expected. There are two possibilities for adding a single shift operator to the contained formula: the first is

$$M, s_h, x \vDash SXP$$

which works out to checking $P$ against the pair $(s_0, s_1)$ (i.e. it is equivalent to $M, s_0, x \vDash XP$); the second is

$$M, s_h, x \vDash XSP$$

which works out to checking $P$ against the pair $(s_1, s_1)$. The difference between these last two assertions is when the held-aside state is replaced: that is, either before or after entering the context of the next operator. This makes the shift operator non-commutative with respect to the path operators.

The difference is similar for the eventually operator: $M, s_h, x \vDash SPF$ and $M, s_h, x \vDash FSP$ correspond to checking $P$ against $L(s_h, s_i)$ and $L(s_i, s_i)$.

Note that the subscript $i$ is bound per semantic rule $P3b$ for $F$.

The last linear example we will consider is representative of a pattern which occurs with some frequency in rely/guarantee reasoning. Consider the assertion

$$M, s_h, x \vDash FSXP$$

This assertion holds if $P$ is in the set designated by $L(s_i, s_{i+1})$; that is, the assertion holds when $P$ is eventually true of some transition between states along the path.

The path quantifiers, $A$ and $E$, commute with the shift operator; thus, $AS \equiv SA$ and $ES \equiv SE$. A proof of this follows trivially from the rules in figure 2.

Predicates can be encoded as relations which ignore one of the pair of states. However, if the set of atomic propositions contains only right-hand predicates then the shift operator becomes an identity operation in Relational CTL* and the logic becomes equivalent to CTL*.

5 Rely/Guarantee and Relational CTL*

We will now use Relational CTL* to express the properties that a must hold on a system if it is to satisfy a rely/guarantee specification. First, we must make a slight modification to the structure of the model used in Relational CTL*: a model, $M$, remains a tuple, but is now $(\text{Config}, \lambda \rightarrow, L)$. The set Config is the set of possible configurations from section 2; $\lambda \rightarrow$ is the semantic transition relation from the same place; and $L$ is still the interpretation per Section 4. We take the set of atomic propositions to include $\lambda \rightarrow$ (and, importantly, $p \rightarrow$ and $e \rightarrow$), as well as Pre, Rely, Guar and Post which correspond to the particular conditions of

---

Footnotes:

4 $M, s_h, x \vDash FP$ works out to checking $P$ against $L(s_h, s_i)$.

5 minus until
the rely/guarantee specification under consideration.\(^6\) We will also assume the inclusion of a proposition \(\textit{Done}\) that is a right-hand predicate which holds when the program in the configuration has terminated. We will also write \(c\) (and the usual variations) to indicate specific configurations in \(\text{Config}\).

With the logical framework set, let us reconsider the assumptions and constraints on rely/guarantee systems set out in section 3. The assumptions correspond to constraints on the set of interesting models — any model which does not conform to these constraints is not one which we would try to use rely/guarantee reasoning on.

To save space and aid the reader, we define a purely syntactic definition:

\[
\text{atDone}(P) \triangleq (\neg \textit{Done} \land X(\textit{Done} \land P))
\]

where \(P\) is some atomic proposition. This definition allows us to assert that \(P\) holds precisely when the program has terminated. Furthermore, the structure of the definition is such that it works well with the shift operator.

The first assumption is just that the pre condition holds on the initial configuration and is trivial.

\[
M, c_h, c_0 \models \textit{Pre}
\]

That the rely condition holds over transitions due to the environment is not difficult to express

\[
M, c_h, c_0 \models \Box \text{GSX} \left( (\xrightarrow{e}) \Rightarrow \textit{Rely} \right)
\]

but it does take a little bit of effort to decipher. In the formula above, starting from the \(\text{SX}\), we indicate that every \(\xrightarrow{e}\) transition conforms to the rely condition. The preceding \(\Box\) means that this applies to every transition in every path.

With the set of interesting models thus constrained, we can consider some constraints on rely/guarantee specifications. First, specifications must be satisfiable: that all elements in the pre condition must exist in the domain of the post condition is expressed as

\[
M, c_h, c_0 \models \textit{Pre} \Rightarrow \text{ESFPost}
\]

expressed in the weaker form that there does exist a path which leads to a point where the post condition is satisfied. The second half of satisfiability – that the pre condition be wholly within the domain of the guarantee – is written

\[
M, c_h, c_0 \models \textit{Pre} \Rightarrow \text{ESXGuar}
\]

and is very similar to the previous, but is only concerned with the first transition from the pre condition satisfying configuration.

\(^6\) \(\textit{Pre}\) is a right-hand predicate.
The guarantee holds over all program steps

\[ M, c_h, c_0 \models AGSX \left( (\neg^p \Rightarrow Guar) \right) \]  

**Behaviour**

The post condition holds at termination (if the pre condition holds initially)

\[ M, c_h, c_0 \models Pre \Rightarrow ASG (atDone(\text{Post})) \]  

**Correctness**

Eventual termination in all paths

\[ M, c_h, c_0 \models AGF Done \]  

**Termination**

Ensuring that a specification requires that a program always be able to deal with a state resulting from interference is expressed as \( \text{rng} R \subseteq \text{dom} \ G \) in section 3. Naïvely, this would be written

\[ M, c_h, c_0 \models AGSX (\text{Rely} \Rightarrow ESX Guar) \]

and means that after any transition conforming to the rely condition there must exist a transition which conforms to the guarantee condition. However, this formula has the problem that it is insensitive to whether or not the program has terminated; taking that into account is simple, however, and the formula becomes

\[ M, c_h, c_0 \models AGSX ((\text{Rely} \land \neg Done) \Rightarrow ESX Guar) \]  

**Progress**

which ensures that the program has also not terminated before asserting that there must be a possible transition conforming to the guarantee condition.

### 5.1 Fairness

The **Progress** condition in Relational CTL*, above, is a sort of fairness condition, in the traditional temporal logic sense. The traditional structure of rely/guarantee reasoning frameworks are unable to (easily) specify such a condition with any more precision than as given in Section 3. The direct translation of the **Progress** condition into its Relational CTL* form suffers from the disadvantage that it uses the **Rely** and **Guar** propositions.

We propose the following constraint as a replacement

\[ M, c_h, c_0 \models AG \left( \neg Done \Rightarrow ESX (\frac{p}{p}) \right) \]  

**Fairness**

This assertion is formulated in terms of the transition relation, and is stronger than the **Progress** condition: it is possible that the intersection of the **Rely** and **Guar** propositions is non-empty, thereby allowing for environmental transitions which satisfy **Guar**. The use of the transition relation directly ensures that, in this case, it is always possible for a program step to occur from any given state (so long as the
program has not terminated). Strictly speaking, this assertion only guarantees non-blocking behaviour; full fairness —requiring that it always be possible for both the program and the environment to make a transition from all configurations— would be

\[ M, c_0, c_0 \models AG(\neg \text{Done} \Rightarrow (\text{ESX}(\rightarrow p) \land \text{ESX}(\rightarrow e))) \]

and is trivially satisfied so long as the environment can make null transitions.

A weaker candidate for Fairness could be

\[ M, c_0, c_0 \models AG(\neg \text{Done} \Rightarrow \text{AFSX}(\rightarrow p)) \]

which only requires that there always be a program transition in the future of every path (so long as the program has not terminated). This would allow for environmental transitions which block program execution for an arbitrary, though finite, number of transitions.

At the other extreme we could require strict alternation between program and environment transitions by asserting

\[ M, c_0, c_0 \models AG(\text{Done} \lor (\text{SX}(\rightarrow p) \Leftrightarrow \neg \text{XSX}(\rightarrow p))) \]

We would not recommend this assertion as a fairness condition, however, as it would likely complicate proofs regarding parallel composition.

### 5.2 Rely/Guarantee Specifications

A rely/guarantee specification in this system is still written

\[ \{ \text{Pre}, \text{Rely} \} \pi \{ \text{Guar}, \text{Post} \} \]

but that is taken as shorthand with the meaning

\[ \forall c_0 \in \text{Config} \cdot \text{fst} c_0 = \pi \Rightarrow (M, c_0 \models \text{Fairness} \land (\text{all-assumptions} \Rightarrow \text{all-constraints})) \]

where \( M \) is an appropriate model; all-assumptions is the conjunction of the Pre Assumption and the Rely Assumption; and all-constraints is the conjunction of Satisfiable, Startable, Behaviour, Correctness, and Termination.

### 5.3 An Additional Constraint on the Semantics

At this point we introduce a few additional constraints on the semantic relation. These constraints come from the work described in [CJ07, Col08] and, though they may not be strictly necessary, they set the frame so that soundness proofs in the style of our earlier work could be performed.

First, we assume that all language constructs supported by the semantics transition relation have only local effects. Specifically, we require that analogues of the
isolation/monotonicity lemmas hold for all of the language constructs. It should be possible to derive these rules from the semantics of the particular language in use.

As we will be considering development rules for sequential and parallel composition in the following part of this section, we give the isolation and enclosure lemmas for these constructs here.

\[
\text{Seq-L-Iso} \quad (\pi_1; \pi_2, \sigma) \xrightarrow{\lambda} (\pi_1', \pi_2, \sigma') \\
\text{Seq-L-Encl} \quad (\pi_1, \sigma) \xrightarrow{\lambda} (\pi_1', \sigma')
\]

The intuition behind the pair of lemmas for sequential composition is that a step of computation reduces a (sub-)program the same way, with the same effect on state, regardless of whether or not it is contained within a sequential composition.

We are also making the assumption, with these lemmas, that once the first half of a sequential composition has finished, the composition construct is just removed. This means that there are transitions in the semantic relation of the form \((\text{nil}; \pi, \sigma) \xrightarrow{\lambda} (\pi, \sigma)\). The alternative –having the composition construct remain until the second half completed– would necessitate transitions in the semantic relation of the form \((\text{nil}; \pi, \sigma) \xrightarrow{\lambda} (\text{nil}; \pi', \sigma')\) and \((\text{nil}; \text{nil}, \sigma) \xrightarrow{\lambda} (\text{nil}, \sigma)\), as well as two more rules, above.

\[
\text{Par-L-Iso} \quad (\pi_1 \parallel \pi_2, \sigma) \xrightarrow{\lambda} (\pi_1', \pi_2, \sigma) \\
\text{Par-R-Iso} \quad (\pi_1 \parallel \pi_2, \sigma) \xrightarrow{\lambda} (\pi_1', \pi_2, \sigma) \\
\text{Par-L-Encl} \quad \exists \pi_2 \cdot (\pi_1 \parallel \pi_2, \sigma) \xrightarrow{\lambda} (\pi_1', \pi_2, \sigma') \\
\text{Par-R-Encl} \quad \exists \pi_1 \cdot (\pi_1 \parallel \pi_2, \sigma) \xrightarrow{\lambda} (\pi_1', \pi_2, \sigma')
\]

The intuition behind these four lemmas for the parallel construct is similar to that for the sequential construct: any computation step made by a (sub-)program may also be made when enclosed within a parallel construct.

### 5.4 Rely/Guarantee Development Rules

We will consider in detail the development rules for sequential and parallel composition, and we will comment on rules for assignment, conditional execution, and iteration.
It should be noted that we will assume the same model structure, $M$, as being part of each hypothesis and conclusion in each inference rule. I.e., each line should be taken to have “$\forall s_0 \cdot M, s_0 \models$” implicitly prepended.

Before describing the development rules, it is worth noting that this construction of the rely/guarantee framework has forced us to be much more explicit in the development rules about the details of what is happening at the semantic level. This leads to rules which are far more verbose than the usual Jones-style formulation.

5.4.1 Sequential Composition

$$\{Pre_1, Rely\} \pi_1 \{Guar, Post_1 \land Pre_2\}$$
$$\{Pre_2, Rely\} \pi_2 \{Guar, Post_2\}$$
$$\text{SA}[Pre_2 \land XRely \Rightarrow XPre_2]$$

$$\text{Seq-I} \quad \text{SA}[Pre_1 \land F[Post_1 \land Pre_2 \land SF[\text{atDone}(Post_2)]] \Rightarrow F[\text{atDone}(Post)]]$$

This development rule for sequential composition of two programs is a near-direct translation of the Jones-style rule for the same. The first two hypotheses set up the specifications which apply to the two component programs and their presentation here utilises the trick of incorporating the pre condition of the second program into the post condition of the first. The rely and guarantee conditions are common to both programs for simplicity; one would usually use a weakening rule during development to gain common rely and guarantee conditions from two specific program specifications. Both $Pre_1$ and $Pre_2$ are predicates – relations dependent only on the right-hand argument, in this system – which allows us to safely conjoin $Pre_2$ to $Post_1$ and have $Pre_2$ apply to the configuration after $\pi_1$ has terminated.

The third hypothesis serves to ensure that the second pre condition is preserved under interference satisfying the rely condition. This allows us to assert that the second pre condition will still be true when the second component program starts execution.

In Jones-style rely/guarantee, the fourth hypothesis is written

$$(Post_1 \land Pre_2) \circ Post_2 \Rightarrow Post$$

being that the relational composition of the two post conditions implies the overall post condition from the conclusion. This is captured by the fourth hypothesis through a subtle use of the $\text{atDone}$ definition.

Building the fourth hypothesis from right to left, we want to conclude that when the whole program has finished then the overall post condition holds; this gives us

$$F[\text{atDone}(Post)]$$

Two things are sufficient to conclude this. The first is that the pre condition of the first component program is true initially. the second part is the temporal encoding
of the notion that, at some instant after the initial state the post condition of the first component is satisfied, and at some instant after that the post condition of the second component program is satisfied. We also need to know that the post condition of the second component program is satisfied specifically at the instant when the overall program finishes. Together this gives

$$\text{Pre}_1 \land F[\text{Post}_1 \land \text{Pre}_2 \land \text{SF}[\text{atDone}(\text{Post}_2)]]$$

The shift operator in this sub-formula ensures that the second post condition is checked relative the correct initial state, i.e. the state between execution of the first and second component programs.

The two sub-formulae are linked via implication, and as it is a path formula, we then use the $A$ operator to ensure that it applies to all possible paths.

In terms of the model (as thought of as a tree), sequential composition can be thought of as a combination of first lifting the set of temporal paths of the first program so that the program text is wrapped in a sequence construct; then, second, splicing the paths after the first program finishes with corresponding paths from the set of paths belonging to the second program. In short, sequential composition is tree (path) splicing.

We will now step though the constraints described at the beginning of this section to see how this development rule preserves them.

**Satisfiable** That there exists a future state which satisfies the post condition given an initial state satisfying the pre condition is derivable from the fourth hypothesis.

**Startable** That the pre condition is wholly in the domain of the guarantee condition is trivially satisfied the by first hypothesis.

**Behaviour** That the guarantee condition holds over all program steps is trivially satisfied by observing that every program step “belongs” to one of the specifications in the first two hypotheses.

**Correctness** That the post condition holds at termination for all paths is satisfied by the fourth hypothesis, again.

**Termination** That all paths terminate can be proven inductively from the knowledge that the component programs terminate and that sequential composition terminates if and only if its components do.

**Progress/Fairness** That the composed program is always able to make a step on configurations resulting from interference is can also be shown inductively, similar to termination.
5.4.2 Parallel Composition

\{Pre, Rely_l\} π_I \{Guar_l, Post_l\}
\{Pre, Rely_r\} π_r \{Guar_r, Post_r\}
AGSX [Rely ⇒ Rely_l ∧ Rely_r]
AGSX [Guar_l ∨ Guar_r ⇒ Guar]
AGSX [Guar_l ⇒ Rely_r]
AGSX [Guar_r ⇒ Rely_l]

The rule for parallel composition is also a close translation as the usual Jones-style rule. The first two hypothesis establish the specifications of the component programs; the next two give weakened versions of the rely and guarantee conditions for the composed specification; and the fifth and sixth establish that the behaviour of one component program can be tolerated by the other.

The second through fifth hypotheses are assertions about the relationships between the rely and guarantee conditions of the specifications: in traditional Jones-style frameworks, these assertions are universal and must hold over all possible states. The use of the AGSX operator combination, here, allows us to make these assertions about the conditions such that they only need hold in the context of the valid models. This gives us a slightly stronger hypothesis in that, for instance, Guar_l need not imply Rely_r over any transition which the system could never make.

The last hypothesis, despite its apparent complexity, is a translation of a corresponding hypothesis in the Jones-style rule. In that style, the hypothesis would be written

\[\overset{\text{Pre} \land \text{Post}_l \land \text{Post}_r \land (\text{Rely}_l \lor \text{Rely}_r)^*}{\text{Post}}\]

The translation of this into our notation is comparatively verbose.

Working structurally, from the outside-in, we start by shifting the root of the temporal tree into the held-aside states so that all relational tests use the initial state on the left unless otherwise modified. Then we assert that the formula in square brackets applies over all paths from the root of the tree. Inside the square brackets is an implication which corresponds to the implication of the Jones-style formula. The consequent of the implication is just that the overall post condition, Post, holds precisely when the parallel construct is finished.

The antecedent of the implication is a fairly direct translation of the four conjunctions in the Jones-style formula. First, that the pre condition holds in the initial state, is just the Pre relation on its own; as there are no path operators to “move” the focus away from the initial state we do not require an equivalent to the VDM
past-state hook notation. The second and third conjuncts require that whenever one of the parallel branches has finished its corresponding post condition is satisfied. We need to not use the shift operator in these conjuncts as we need to test the post conditions relative to the initial state that was fixed by the $S$ at the beginning of the hypothesis. Note, also, that these two conjuncts require that the left and right post conditions be stable relative to interference. The last conjunct simply requires that every transition satisfies either the left or right rely condition; though this is trivially true due to the first two hypotheses, the presence of this conjunct is occasionally required to discharge the consequent (see [CJ07] for an example which requires this).

Stepping through the constraints described at the beginning of the section, as we did for sequential composition:

**Satisfiable** is given directly by the last hypothesis.

**Startable** is given by the first two hypotheses allowing that $Pre$ is in the domain of $Guar_l$ and $Guar_r$, then the fourth hypothesis allows us to derive that $Pre$ is in the domain of $Guar$.

**Behaviour** is given by the fact that all program steps satisfy one of either the left or right guarantee (first two hypotheses), or do not affect the state component at all (assumption about how parallel composition works). Then the fourth hypothesis generalises the branch guarantees to the composed guarantee.

**Correctness** is given by the last hypothesis, depending on the establishment of the left and right post conditions, which we get from the first two hypotheses.

**Termination** requires an inductive proof on the structure of the parallel construct. So long as the branches terminate and the parallel construct is not pathologically strange, then the composition will terminate.

**Progress/Fairness** is also an inductive proof, depending on the fact that both branches will make progress given their respective rely conditions.

### 5.4.3 Other Constructs

There are three more types of construct typically used in rely/guarantee frameworks: assignment, conditional execution, and iteration.

Assignment is distinctly difficult to deal with if it is not assumed to be atomic. The stance taken in [CJ07] is that individual assignments must be proven correct directly in terms of the underlying language semantics; there is no general development rule for non-atomic assignment. It is, unfortunately, more common to assume that the assignment construct is atomic, thus removing the need to reason about inference during the expression evaluation; this leads to a development rule similar

---

7Even though it’s not explicitly specified!
to the standard Hoare Logic rule for assignment. This sort of development rule should pose no difficulties in the framework presented in this paper.

Conditional execution poses no problems not already raised by assignment and how expression evaluation is handled. Iterative execution is similar, in that it can be decomposed into a combination of conditional execution and sequential composition, and raises no issues which require special treatment.\(^8\)

### 6 Conclusions and Future Work

We have presented a brief introduction of a rely/guarantee framework in the style of Jones and Coleman; shown how the structure of the underlying system can be linked to CTL*; extended CTL* as Relational CTL* to deal with relations as atomic propositions; and shown how a rely/guarantee reasoning framework can be constructed using Relational CTL*. A particularly useful result of constructing the framework in this way is that we gain the ability to reason about fairness in a direct manner, unlike traditional rely/guarantee reasoning frameworks.

It may be possible to build a rely/guarantee framework which uses Relational CTL* directly to define the conditions; this would allow for behaviour conditions which are much more expressive than the traditional rely/guarantee versions (and more expressive than those in Section 5 as well). It is more likely, however, that we would need to use Relational CTL\(_\perp\) as described in [Col10] as it also handles undefinedness. Doing so would make it possible to directly express the idea that an event must happen an arbitrary but specific number of times, without resorting to the use of auxiliary variables. It is our suspicion that making this change would result in a framework which is difficult to apply.

One potentially fruitful use of Relational CTL* relates to the Wait conditions as described by Stølen in [Stø91b, Stø91a]. Wait conditions characterise the set of configurations in which the program may be blocked on the assumption that the environment will also eventually unblock the program. This would be an obvious use of the temporal aspects of the system presented here, allowing for very precise Wait conditions, and possibly for proofs that a Wait condition is always satisfied. Unlike the usual rely/guarantee conditions, it might be appropriate to express a Wait condition in terms of Relational CTL*.

One of the advantages of using a branching-time logic is that it directly exposes the effects that interference has on the possible sequences of configurations reachable by a system. Some opinions suggest that linear-time logics are easier to deal with, so it may be useful to recast Section 5 in terms of a relational linear-time logic. Few of the constraints required by rely/guarantee reasoning use the E modality, so most would pose no problems in translation; those that assert the existence of a possible transition at every configuration would cause some trouble, however.

\(^8\)Termination for iterative execution can be shown through the use of well-founded relations in the same manner as in [CJ07].
As far as we know, the use of relations as atomic propositions in a temporal logic is unique, and is worth further exploration.

The construction of the rely/guarantee reasoning framework on top of Relational CTL* may not present any obvious benefits in practise over the traditional form, but it does have the distinct benefit of being able to give a clear, precise and usable fairness condition.

Acknowledgements Thanks are due to Cliff Jones for discussions and comments on this work and to Edsko de Vries for pointing out that the $S$ operator is orthogonal to the $F$ and $X$ operators, thereby simplifying the structure of Relational CTL* immensely.

The author would also like to gratefully acknowledge support from the EPSRC “TrAmS” platform grant.

References


