Distributed System Proofs in Constructor Based Reachability Logic

Paul Pok Hym Ng[a]
[a]ECE, University of Illinois at Urbana-Champaign, ppng2@illinois.edu

Abstract

Proving properties of distributed systems is not an easy task and often uses traditional interactive theorem provers is a herculean task. In this article, we explore the use of constructor based reachability logic and rewrite theories for proving distributed systems. First discovered and designed by Stephen Skeirik [14] for his PhD thesis, reachability logic is a theory generic proof system meaning all tactics and strategies apply for all possible systems or programming languages. Since its inception, constructor based reachability logic has been further improved and augmented with a tool written in Maude. In this study, we demonstrate the specification and proof of distributed systems. In addition we show how the usage of CBRL and rewrite theories is much more intuitive and easier to use in comparison to traditional meth distributed systems.

Keywords: Rewrite Theory, Reachability Logic, Maude, Distributed Systems, Interactive Theorem Provers, Theorem Proving, Formal Methods

1 Introduction

It is complicated to prove distributed systems is complicated due to their non-determinism and non-terminating properties.

There are two approaches to prove distributed systems: bounded model checking and theorem proving. The former, uses languages such as TLA+ to check all possible instances of a distributed system up to a certain bound by using user bounds such as, number of iterations, and number of variables in a system. However, due to the enumeration of states it is often prohibitive to test all possible states that a distributed system can enter. The latter approach, solves the previous issue by performing a mathematical analysis on the distributed system. By mathematically reasoning about properties we can prove properties that hold for all possible states of the system.

Constructor based reachability logic (CBRL) is an instance of the theorem proving approach, which uses rewrite theories written in the language Maude. In the later sections, we will give a crash course on rewrite theories, CBRL, examples of systems and proofs with increasing complexity, and finally a comparison to other theorem provers on the market (Isabelle and Coq).

2 Goals

Originally, the goal was to prove the soundness and completeness of a concurrent garbage collection algorithm for actors [12]. The algorithm would be used to kill off actors that are no longer being used by a server. An example of this would be Discord calls. When a user disconnects from a call, a garbage collector detect that the actor (caller) is now garbage and will clean up data structures spawned when they joined the call.

The author had new ideas of how to improve the algorithm and give it a much needed performance boost. As such we would have need of another automated proof for soundness and completeness of the system. This would would lead to a new interesting result. In fact, the specification for this algorithm was already specified and completed in Maude (will be included with this paper). However we ran into many problems when trying to prove soundness and completeness of this algorithm within the CBRL tool.

During the early months before Stephen's tool

matured, proving such a complex algorithm proved prohibitive. This led me to attempt a rebuild of the system in Coq and Isabelle. However, due to my lack of understanding and skill in using these languages, I was still unable to prove "Choice" (discussed below).

From thereon, the project's goals shifted towards building and verifying systems of increasing complexity in order to debug, fix, and test `rltool`, the tool that implements CBRL. We have dedicated a section on the difficulties and pathological bugs we encountered along the way.

I would like to show my sincere appreciation to Stephen for resolving the bugs in a timely manner and allow me to complete the proofs shown in this paper in a timely manner.

3 Preliminaries

In this section we will give a quick rundown of rewriting theories and CBRL. With this knowledge we hope to convey the ideas of CBRL and how to utilize it. For a more complete survey of rewriting logic and CBRL please refer to [9] [14].

3.1 Rewrite Theories

Rewrite theories are one way of describing distributed systems. A rewrite theory is a 3-tuple (Σ, E, R). Here Σ represents the function symbols ($+, -$ in natural numbers) and E represents the equations (implementation of $+, -$ in natural numbers) used to describe the state of a distributed system. In addition, (Σ, E) provides us with all the terms in our initial algebra $T_{\Sigma/E}$. For example, in the theory of Peano numbers this would contain the values $0, s(0), s(s(0)), s(0) + s(s(s(0)))$.... These are terms built only with Σ and E. The transitions between states describing the evolution of the distributed system are modeled by R the rewrite rules in our system. Rules can be conditional or unconditional and are applied to *modulo* our equations E. This means that a rewrite can apply functions in or use predicates in (Σ, E) to evaluate and trigger the rewrite rule.

3.2 Reachability Logic

CBRL is *theory generic* which allows it to use the same inference rules for different systems with varies semantics. This is different from Hoare logic which requires a redesign of inference rules per language. Often, the number of rules are in the high tens and take a long time to prove the completeness/soundness of. In CBRL, as long as we have a suitable $\mathcal{R}$, rewrite theory, as an input we can use its inference rules. The rewrite theory must have the following properties.

1. The theory must be terminating: All terms must rewrite to constructor types
2. Sort decreasing: Every rewrite must either write to the same level of sort or decrease towards the constructor level
3. Coherent: Two equivalent terms must rewrite to the same ground term
4. Church Rosser: The order which rewrites are applied should not change the end result

A reachability logic formula has the following form

$$A \rightarrow^{\circledast} B \iff u \mid \phi \rightarrow^{\circledast} v \mid \psi$$

where u, v are terms representing the state of a system and ϕ and ψ are predicates. The arrow with a star represents a reachability logic formula rather than a rewrite rule.

In the non-parametric case (easier), both A and B have no shared variables. Then given a $\mathcal{R}$ we interpret the terms in A and B (u and v) in the initial algebra $\mathcal{T}_{\mathcal{R}}$ where a term u can be rewritten to v. Equivalently we can say a state transition from $\mathcal{R} \vdash u \rightarrow v \iff [u] \rightarrow [v]$ (where [ ] is the equivalence class operator) gives us the result that computation is equivalent to deduction.

Moreover, our terms u and v in a reachability logic formula must be constructors. Where a constructor Ω is a subset of Σ. A simple example is is shown below. In the theory of Peano natural numbers the two constructors are s and 0 the successor. These are **not** functions on natural numbers but ground terms used to construct a natural number. A term composed completely of constructors is on the left and the right is a term built from operators in Σ ($+$ in this example).

$$s(s(s(0))) = s(0) + s(s(0))$$

ϕ and ψ in a reachability logic formula represent a predicate which only use conjunction ($\wedge$) and disjunction ($\vee$). These are known as constrained pattern predicates in CBRL and will be explained in later sections.

3.3 Order Sorted Algebras

In Maude, all functional modules (ones without rewrite rules) and system modules (ones with rewrite rules) are described with an order sorted algebra. An order sorted algebra contains the following Σ.

$\Sigma = (S, \leq, \Sigma)$ where S is a set of sorts (classes in OOP), $\leq$ a binary relation telling us about the "largeness" of sorts (class inheritance in OOP) and Σ the set of function operators.

Furthermore, we also have the following properties for an algebra (S, Σ) algebra.

1. If $s \leq s'$ then algebra $A' \leq$ algebra A
2. If $f : w \to s$, $f : w' \to s' \in f_{[s]}^{[s_1],...,[s_n]}$ (a function f goes from w (w has input sorts $s_1, ..., s_n$, and output s has sort s AND a function goes from w' to s where s' is also a sub-sort of s), and $\bar{a} \in A^w \cap A^{w\prime}$ ($\bar{a}$ is a term that exists in both algebras) then we have the result $A_{f:w\to s}(\bar{a}) = A_{f:w'\to s'}(\bar{a})$

An example of property 1 is sort Nat and a sort Int. A natural number is a subsort of integers as the set of all natural numbers is contained within the set of integers. In the case of property 2 we can look at the following example.

$$(5).Nat + (6).Nat = (5).Int + (6).Int$$

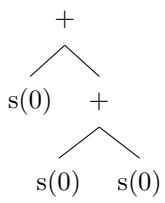
Both of these obviously evaluate to the same value, use the same operator, and are different algebras.

This concludes our introduction to order sorted algebras.

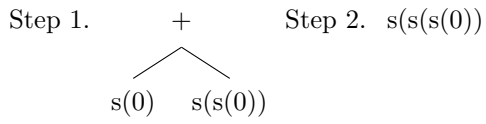
3.4 Rewriting Logic

A rewrite theory is used to model a distributed system in our case. $\mathcal{R}$ is a rewrite theory composed of $(\Sigma, E \cup B, R)$ where Σ are function symbols, E are equations, B are axioms, and R are rewrite rules.

All terms in a rewrite theory can be represented as a tree. The tree for $s(0) + (s(0) + s(0))$ is shown below.



All equations or rules that operate on the term will have to work from the leaves of the tree. The equations simpify each subterm until we have only a root term represented by constructors only. Below we see the steps used by the + operator to simplify the tree above.



Recall that other than R, the rest of a rewrite theory is an order sorted algebra. An unconditional rewrite rule is of the form

$$R_i = u \to v$$

Where a term u can be mutated into term v. An example of this is shown below.

$$N; L \to L$$

The rule above states the following: Given a list, with concatenation operator ";", containing a number N and its tail L (which may be empty), remove the number N from the list. Note that this rule can not be applied if the list is empty because it is impossible to match an empty list to a non-empty list. In addition to unconditional rules, we also have conditional rules which can be applied only if a condition is true.

$$N; L \to N; N; L \mid if\ N > 0$$

This conditional rule can only duplicate an element in our list if the number N is greater than 0. Furthermore, a rewrite theory has the following properties.

1. Unconditional equations $u = v$ must have the same shared variables (no free variables) and no repeated variables.
2. Conditional equations can be re-oriented into conditional rewrite rules
3. The rules are ground coherent with E modulo B

Ground coherence [5] is the property where all ground terms (constructor terms) can be written modulo associativity which is in general undecidable $([t]_{Assoc} \to [t']_{Assoc})$.

These three conditions give us the initial reachability model $\mathcal{T}_{\mathcal{R}}$ and the initial algebra $T_{\Sigma/E\cup B}$'s isomorphism to the canonical term algebra $C_{\Sigma/E\cup B}$. A canonical term algebra is the simplest form of representing a term and is composed completely by constructors of the algebra.

3.5 Constructor Decomposition

Constructor decomposition is the the subset theory obtained from $(\Sigma, E \cup B)$ which operates completely on the constructors.

More concretely we turn $(\Sigma, E \cup B)$ into $(\Sigma, B, \overrightarrow{E})$ where $\overrightarrow{E}$ represents the equations reoriented into rewrite rules (conditional if we have a conditional equation). Then $(\Omega, B_\Omega, \overrightarrow{E_\Omega})$ is a constructor decomposition only if the following properties are true.

1. $t =_\Omega t' \iff t =_B t'$
2. $t = t!_{\overrightarrow{E_\Omega}, B_\Omega} \iff t = t!_{\overrightarrow{E}, B}$
3. $C_{\Sigma_\Omega/E_\Omega, B_\Omega} = C_{\Sigma/E,B}$

The three statements above essentially state that there must be 1. equality on terms on both algebras, 2. the fully rewritten terms which are also equivalent in the decomposition and original, and 3. the canonical algebras which are equivalent.

3.6 Why Care About Decompositions?

Decomposition allows us to easily reason on the constructors. By doing so, it allows CBRL to significantly reduce the computation required to simplify goals and calculate something known as the congruence closure (discussed later). In the figure below we can see the relationship. The major focus here is the diagonal dot-

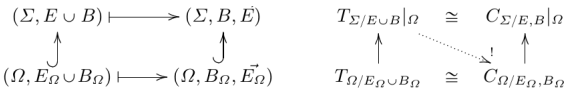


Figure 1: Theory Inclusions (Left) and Initial Algebra Isomorphisms (Right) [14]

ted arrow on the right of Figure 1. Recall that our specification of a rewrite theory gives an initial algebra $T_{\Sigma/E\cup B}$. This algebra is easy for us as humans to read and use. For example,

$$s(s(s(s(s(s(s(0)))))))$$

the canonical reprsentation of eight is much harder for a human to read than,

$$8$$

However, the canonical term algebra post decomposition, is much easier for the computer to reason with because less rewrites need to be applied. In shorts, this allows us to write our rewrite theories in a human readable format while doing all the proofs at the constructor level. In addition to this fact, usually the constructor decomposition is so simple that there are no equations E_Ω. This further reduces the amount of computation required for the process. The problem is associativity. Associativity by itself is not decidable [10]. Other combinations of associativity, commutativity, and identity axioms are decidable [10].

4 Constructor Based Reachability Logic

Now we shall cover the proof system, its inference rules, and how to describe invariants.

4.1 Constrained Pattern Predicates

A constrained pattern predicate is in the form of $u \mid \phi$ where u is a term and ϕ is a quantifier free formula. u must be part of the canonical term algebra and thus described using constructor terms.

$$N \mid N > 0$$

In this example, we are showing a constrained pattern predication on a natural number N which can only apply if N is greater than 0.

4.2 Describing Invariants

To describe invariants with a pattern predicate, we should recall our reachability formula.

$$A \to^{\circledast} B$$

A and B are pattern predicates. The left side A represents the precondition and the right side B represents a midcondition. A midcondition is not the same as postcondition as it only needs to hold along the path **at some point**.

However, due to the non-terminating nature of a distributed system, we are not able to actually prove an invariant. This is because all reachability formulas are vacuously true. In order to solve this problem, we need to add a **stop** operator which pauses the distributed system at anytime. After this, we will be able to check on our invariant. In all of our subsequent case studies, the stop operator we use is `[...]` which wraps the whole system state.

This allows us to compute both "system invariants" and "inductive invariants". A system invariant is one which holds true eventually in the system (midcondition). An inductive invariant is one that must be true in the precondition and after taking one step (midcondition).

4.3 Comparison to Hoare Logic and Linear Temporal Logic

We will now draw a parallel between Hoare Logic and Linear Temporal Logic.

In the case of Hoare Logic, we have the following.

$$\{A\}\mathcal{R}\{B\}$$

Translated into CBRL we have the following where $\mathcal{R}_\mathcal{L}$ represents the semantics of the programming language as a rewrite theory.

$$\{p : init \mid \phi\}\mathcal{R}_\mathcal{L}\{< skip : S > \mid \psi\}$$

Where the precondition ϕ represents the initial state of the program and p representing the input program. *skip* represents the empty program state after we have processed all lines of code and ψ is our program termination condition. Please refer to [14] and [1] for more information on how this can be done.

In the case of LTL, we have the following equivalences.

$$A \to^{\circledast} B \iff \forall Y \mid A \to \diamond B \vee \Box enabled$$

$$A \to^{\circledast} [B] \iff \forall Y \mid A \to \circ B$$

$$B \to^{\circledast} [B] \iff \circ B$$

4.4 T-Consistency

All reachability formulas must be something known as T-Consistent. This means that there must exists a unifier (substitution) such that the precondition is an instance of the midcondition with parameters Y under the substitution.

$$N + M \mid true \to^{\circledast} O \mid true$$

In the example above N, M, O are all integers. And because we can match the sum of $N + M$ as an arbitrary integer O, this reachability formula is T-consistent.

The following example is not T-Consistent, because the midcondition's term is a boolean value rather than a simple integer.

$$N + M \mid true \to^{\circledast} true \mid true$$

4.5 Inference System

The essential goal of a proof and its sequents (steps, next goals) are in the form of

$$[\mathcal{A}, \mathcal{C}] \vdash_T u\ \phi \to^{\circledast} \bigvee_i v_i \mid \psi_i$$

$\mathcal{A}$ represents the axioms you as a user input. $\mathcal{C}$ are circularies which contain formulas that we wish to prove simultaneously. All formulas must be T-Consistent at every application of tactics. Otherwise, the proof will failure.

4.6 Inference Rules

In this section, we will list the inference rules being used in CBRL. Further details and conditions on these inference rules can be found in [14].

1. Subsumption: Discharges trivial formulas. There are 2 cases. One, the precondition is an instance of the midcondition with parameters Y or it is vacuously impossible.
2. Step: Use rewrite rules to evolve the system state by step. This is used internally by the auto rule.
3. Axiom: Use a trusted axiom to take "multiple rewrite steps". This is a very powerful inference rule since if being used incorrectly may discharge a proof incorrectly. This is used internally by the auto rule.
4. Split: Split a constrained pattern predicate into an equivalent one
5. Case: Create new goals with a complete covering of constructor patterns
6. Substitution: Allows you to solve a conjunction of equalities in the precondition
7. Auto: Takes a step, applies all available axioms, and checks T-Consistency.

Let us give a few examples of when each inference rule is being used. All variables used below are natural numbers.

Subsumption:

$$0 \mid true \to^{\circledast} s(0) \mid true$$

can be subsumed by a rather generalized pattern

$$N \mid true \to^{\circledast} s(M) \mid true$$

Split:

$$N \mid true \to^{\circledast} s(M) \mid true$$

can be split into the following conjunction of preconditions. While it might not be useful in this case, splitting is often being used to give the tool more information to proceed further with the proof. Below, we split on the fact that either N is 0 or $s(M)$ (nonzero). This can be done because of the equivalence $\phi \iff ((\phi \wedge \psi) \vee (\phi \wedge \neg\psi))$. In the example below $(true \wedge N = 0) \vee (true \wedge N \neq 0)$. This gives us two new goals.

$$N \mid true \wedge N = 0 \to^{\circledast} s(M) \mid true$$

$$N \mid true \wedge N \neq 0 \rightarrow^{\circledast} s(M) \mid true$$

Case:

$$N \mid true \rightarrow^{\circledast} s(M) \mid true$$

can be cased upon the cover set of natural numbers. In other words all the possible constructor patterns. This gives us two new goals.

$$0 \mid true \rightarrow^{\circledast} s(M) \mid true$$

$$s(I) \mid true \rightarrow^{\circledast} s(M) \mid true$$

In general, the number of new rules one needs to generate when using case is the number of constructors that a sort has.

5 Describing Distributed Systems

To describe a distributed system for CBRL as input, we use a language known as Maude. As mentioned above, Maude uses order sorted equational theories with rewrite rules to give us rewrite theories.

First, let's see a simple example of a rewrite theory and its proof in `rltool` the implementation of CBRL in Maude.

As a side note, Maude itself is not a very well documented language. Please refer to the manual for more information on Maude [4]. We have explained essential keywords below in order for the reader to understand both modules and proofs.

1. `fmod ... endfm`: Describes a functional module which can only contain operators and equations. These describe our equational theories.
2. `mod ... endm`: Describes a module which can contain rules.
3. `sorts` and `subsorts`: Defines our types and $\leq$ relation on them.
4. `op`: Defines are operations
5. `[...]`: Defines additional properties of an operator

   `ctor`: Constructor

   `assoc/comm`: Associativity and commutativity

   `metadata "N"`: Provides the CBRL tool with a termination order. This is specifically required for contextual rewriting [15]. This is not used in Choice which uses variant satisfiability [10] (not discussed here).

Maude allows one to define their own syntax for an operator. Underscores represent input variables, and other symbols represent the operator being used. For example,

```
op _=_ : Nat Nat -> Bool .
```

defines a binary mixfix operator that takes two natural numbers and returns a boolean. The semantics of the operator are then defined with equations. An example of the equality definition is shown below. `N` and `M` represent natural numbers.

```
eq (0 = 0) = true .
eq (N = N) = true .
eq (0 = N) = false .
eq (N = 0) = false .
eq (N = M) = false .
eq (M = N) = false .
```

5.1 Some Important Notes

Because we are using a Maude module for proofs, we need to do take into account several things. No builtin modules can be used. This is because the underlying implementation for things such as real numbers or integers is just the C data type. Which have no formal definitions attached and thus unprovable. This of course includes booleans and natural numbers which are used throughout this case study. As such we need to define our own functional modules with the semantics of booleans and natural numbers.

5.2 Choice: A Simple Example

Hereby, we are giving one of the canonical examples being used throughout the Maude manual. "Choice" is a module that that turns a multiset into a singleton. It non-deterministically pulls out any singleton in the multiset. One will notice that `fmod` (function module) only contains contructors and equations. `mod` contains rewrite rules.

```
set include BOOL off .
fmod CHOICE-DATA is
  *** Sorts
  sorts Nat MSet State Pred .
  subsorts Nat < MSet .
  *** Constructors
  op zero : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op __ : MSet MSet -> MSet [ctor assoc comm] .
  op {_} : MSet -> State [ctor] .
  op tt : -> Pred [ctor] .
  *** MSet containment
  op _=C_ : MSet MSet -> Pred [ctor] .
  vars U V : MSet . var N : Nat .
```

```
  eq U =C U = tt .
  eq U =C U V = tt .
endfm
mod CHOICE is pr CHOICE-DATA .
  vars U V : MSet .
  *** Choose a singleton
  rl [choice] : {U V} => {U} .
endm
```

5.3 Proving Choice

In this case, we would like to prove the following invariant.

$$M \mid true \rightarrow N \mid N \in M$$

In English, we can reword this. Given any multiset M our system will be able to reach a state where M has been transformed to N a natural number which is a member of multiset M. Below is the proof of which and its output.

```
--- Load our module and the tool
load choice.maude
load ~/soft/maude/rltool/rltool.maude

--- Select the module and methods
--- for proving, varsat stands
--- for variant satisfiability
--- not discussed here
(select CHOICE .)
(use tool varsat for unsatisfiability on
    CHOICE-DATA .)
(use tool varsat for validity on CHOICE-DATA .)
--- Declare the variables
(declare-vars (M:MSet) U (N:Nat) .)
--- Declare the terminating states/
(def-term-set ({N}) | true .)
--- Declare our invariant
(add-goal end-with-singleton :
    ({M}) | true => ({N}) | (N =C M) = (tt) .)
--- Begin the proof
(start-proof .)
--- Create new goals with a cover set
(case ({M:MSet}) on M:MSet by
    (M1:MSet M2:MSet) U (N':Nat) .)
--- Step and apply axioms
(auto .)
(auto .)
 quit .

--- OUTPUT
...
Added goal(s):
      [end-with-singleton : {M:MSet} |||
true => {N:Nat} ||| tt = N:Nat =C M:MSet]
Command: add-goal end-with-singleton :
({M})| true =>({N})|(N =C M)=(tt).

Started proof:
      [1 | {M&5:MSet} ||| true
=> {N&6:Nat} ||| tt = N&6:Nat =C M&5:MSet]
Command: start-proof .

Cases rule generated:
      [8 | {M&8:MSet M&9:MSet} |||
true => {N&6:Nat} |||
tt = N&6:Nat =C(M&8:MSet M&9:MSet)]
Action consumed 1 of 1 active
goals and generated 1 goals
Command: case({M:MSet})on M:MSet
by(M1:MSet M2:MSet)U(N':Nat).

Auto Results:
[40 | {&5:MSet} ||| true => {N&7:Nat} |||
tt = N&7:Nat =C(&6:MSet &5:MSet)]
...
[46 | {&6:MSet &8:MSet} ||| true
=> {N&9:Nat} ||| tt = N&9:Nat =C(&6:MSet
      &8:MSet &5:MSet &7:MSet)]
Action consumed 1 goals and generated 7 goals
Command: auto .

Proof Completed.
Action consumed 7 goals and generated 0 goals
Command: auto .

Bye.
```

From here on only sections of specifications and proofs will be shown due to their length. Only the highlights are given in this article.

6 Simple Consensus Algorithm

In this section, we describe a simple consensus algorithm [8] between two parties over a channel. Each

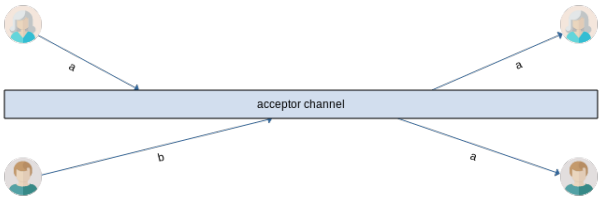


Figure 2: Consensus Protocol

party (user) sends a message onto an acceptor channel. The acceptor channel will then accept the first message it receives and reply to every subsequent party with the first received message. This can be seen in Figure 2 where the woman sends message a first be-

fore the man sending b. Therefore when the acceptor channel replies to both it sends a back.

6.1 Specifying the Consensus Algorithm

We use four modules in this consensus algorithm.

1. `IBOOL`: This is our self defined boolean operations and constructors
2. `MSGS`: This defines the messages that can be sent. Because we use two parties only we only have two constructors a and b one for each channel.
3. `STATE`: The state defines our system as a whole, as such it contains three parts, two parties and an acceptor channel.
4. `STATE-RULES`: This module contains the rewrite rules specifying the behavior of our distributed system.

Our state operator contains three parts separated by a pipe. The acceptor channel is in the middle and the parties are on each side. Each party contains two parts, the message it will send and a boolean which represents whether it has received a message from the acceptor channel.

```
--- partyA | acceptorCh | partyB
op _,_|_|_,_ : Msg Bool Msg Msg Bool ->
    State [ctor metadata "8"] .
```

The rewrite rules that we have in our consensus algorithm are as follows.

1. 2 rules representing each party choosing a message
2. 2 rules representing each party proposing a message
3. 2 rules representing each party accepting a consensus message
4. A stop rule allowing us to pause the system and prove invariants

The accept rule for party a is shown below.

```
--- ~M represents message equality
crl [accept-a] :
    A,false | C | B,T => C,true | C | B,T
    if C ~M empty = false .
```

This conditional rule can be read as follows: If party A has yet to receive a consensus message from the channel (false) and the consensus channel is not empty, change the message of party a to C and mark it as accepted by changing $false$ to $true$.

Note that, due to the nature of way a message was designed, `A,B,C` can represent any message (including an empty message) thus we need the condition on the value of `C` in order for the rule to trigger.

Below we show a snippet of the `MSGS` module to demonstrate our point.

```
...
sort Msg .
op empty : -> Msg [ctor metadata "5"] .
op a : -> Msg [ctor metadata "6"] .
op b : -> Msg [ctor metadata "7"] .
...
```

6.2 Proving the Consensus Algorithm

The system invariant we want to prove is as follows.

$$A_1, B_1|C_1|B_1, TT_1 \mid init(A_1, B_1|C_1|B_1, TT_1) \rightarrow^{\circledast} \mathrm{A}_2, B_2|C_2|B_2, TT_2 \mid accepted?(A_2, B_2|C_2|B_2, TT_2)$$

The predicate `init` checks if the system is in an initial state `empty,false | empty | empty,false` where empty represents the empty message. `accepted?` checks if the system state has both actors accepting a consensus message and all the messages (both parties and acceptor channel) are the same. Below the `accepted?` operator is shown. For the true version we list the two possibilities of an accepted state and the false case should match any other system state. `S` represents the most general state variable.

```
eq accepted?(a,true | a | a,true) = true .
eq accepted?(b,true | b | b,true) = true .
eq accepted?(S) = false .
```

The proof is very simple and the outline is essentially the same as the Choice proof. In fact the proof is so simple that we do not need to apply any inference rules. We only need to apply `auto`.

7 Self-Stabilizing Construction of Spanning Trees

The next algorithm to be proven was proposed by Chen et al in 1991 [3]. The algorithm states as the following. Given a connected graph, the algorithm will maintain a spanning tree with each node knowing its level in a tree. Each node contains the following fields: its ID, level (depth in graph), a parent, and a list of neighboring nodes. The graph must contain a root node with no parent (we default a root to

parentID 0, nodeID 0). With the information of leveling and one parent we are able to then build a spanning tree.

The graph may lead to an unexpected state due to a perturbation in the system. A perturbation is represented by an incorrect level in a node. There are several cases. Firstly, a node that does not have a level of $parent + 1$ is in an error state. Secondly, a node that has level of $|V|$ (where $|V|$ is the number of vertices in our graph) is in an error state. From an error state, the algorithm being proposed uses 3 rules to return a valid state.

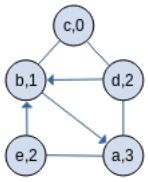


Figure 3: Invalid Tree state [3]

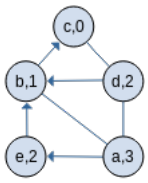


Figure 4: Valid Tree state [3]

Figures 3 and 4 depict two example tree states, where a node has a $(name, level)$. Arrows depict the spanning tree which is constructed by the algorithm.

7.1 Specifying the Self Stabilizing Tree Algorithm

There are two parts for this point. The system description and the rules describing the recovery algorithm.

7.1.1 Specification

A graph is represented as a list of nodes. We use ; as the concatenation operator.

```
op mtGraph : -> Graph
    [ctor metadata "28"] .
op _;_ : Graph Graph -> Graph
[ctor assoc comm metadata "29" id: mtGraph] .
op _;_ : Graph NeGraph -> NeGraph
[ctor assoc comm metadata "30" id: mtGraph] .
op _;_ : NeGraph Graph -> NeGraph
[ctor assoc comm metadata "31" id: mtGraph] .
```

A node is defined as follows

```
op _||_|->_::_ : iNat iNat iNat iNatList
    -> Node [ctor metadata "22"] .
```

the fields are as follows.

1. NodeID
2. Level
3. Parent
4. List of NodeIDs that are the neighbors of this node

7.1.2 Rules

There are 3 rules to comply with the algorithm.

1. Rule 0: $L(i) \neq n \wedge L(i) \neq L(p) + 1 \wedge L(p) \neq n \rightarrow L(i) := L(p) + 1$, if a node's level is not equal to the number of the nodes in the graph and it is not equal to the level of its $parent + 1$ then we assign the level of the node to $level(parent) + 1$.
2. Rule 1: $L(i) \neq n \wedge L(p) = n \rightarrow L(i) := n$, if a node's level is not equal to the number of the nodes in the graph and and its parent is equal to the number of nodes in the graph, assign the level of the node to be the number of nodes in the graph.
3. Rule 2: Let k be some neighbor of i, $L(i) = n \wedge L(k) < n - 1 \rightarrow L(i) := L(k) + 1; P(i) = k$, this is the recovery rule and it claims that if a node's level is equal to the number of nodes in the graph and it has a neighbor which does not, change the parent of the node to that neighbor and its level to $level(k) + 1$.

The reasoning behind why rule two is able to recover is because we always have a root node which **always** has level 0 and is never modified. Therefore the predicate $L(k) < n - 1$ can always be triggered even if all other nodes have level n.

7.2 Proving the Self Stabilizing Tree Algorithm

The system invariant we want to prove is as follows.

$$G_1 \mid connected?(G_1)$$
$$\rightarrow^{\circledast}$$
$$\mathrm{G}_2 \mid connected?(G_2) \wedge connected?(buildMST(G_2))$$

It can be summarized in English below. Given a connected graph, do we reach a state after multiple rewrites do we reach a state where the new graph remains connected and after modifying the parents of each node are we still able to build a spanning tree? A spanning tree in our context is built from the parents of each node. Therefore in Figure 4, we can build the resulting spanning tree,

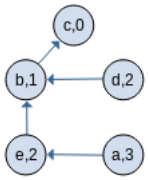


Figure 5: Spanning Tree built from Figure 4 [3]

It is important to point out that in our reachability formula G_1 represents **any** possible connected graph. This means that it could represent a graph that is already a spanning tree state or one that is in an invalid state. This simple representation format allows us to very simply represent all possible input graphs that satisfy our precondition.

The predicate `connected?(G)` returns true when a graph is connected. The predicate `buildMST(G)` returns a graph where we attempt to build a spanning tree. This is then further fed into the connected predicate to check if the graph is actually connected.

The connected predicate is defined as follows.

```
ceq connected?(G) = true
    if (graphToNeighbors(G, G, mtGraph) ~G G)
       = true .
ceq connected?(G) = false
    if (graphToNeighbors(G, G, mtGraph) ~G G)
       = false .
```

Where `graphToNeighbors` recursively pulls nodes out of a graph and builds a list of all the neighbors each node has. Note this operation (`graphToNeighbors`) is a set like operation where no nodeID can be duplicated in the resulting returned graph. Finally `G` represents the graph equality operation which recursively checks that both input graphs have the same nodeIDs.

Secondly, `buildMST`, takes an input graph and for each node in the graph we replace the neighbors list with the node's parent. Below we have shown the operator for `buildMST`.

```
eq buildMST(mtGraph) = mtGraph .
eq buildMST((A || B |-> C :: L) ; G) =
    (A || B |-> C :: C) ; buildMST(G) .
```

Note that unlike the consensus protocol, we do not explicitly constrain the set of initial states. Our only precondition is that it must be connected. Here we want to prove that we are able to reach a spanning tree from **any** connected graph.

Once again, the proof is simple and of the same format as the Choice proof. We only require `auto` to complete the proof.

8 Alternating Bit Protocol

Here we present the most complex, and hardest proof so far. This is also the distributed system that was able to discover the most amount of bugs in the CBRL tool (`rltool`).

The alternating bit protocol is a data level protocol in the OSI model. It models a lossy channel where a sender sends data from a sender to a receiver. The guarantees it provides are that the data will always be sent in order and will always be received (reliable data transfer).

The alternating bit protocol used here is built off of Rocha's PhD thesis on reachability analysis on rewrite theories [11].

The alternating bit protocol is defined as follows. The left side represents the sender. The right

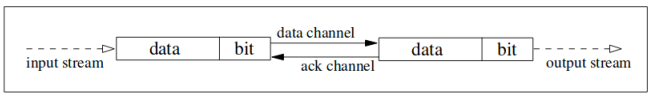


Figure 6: ABP [11]

side represents the receiver. There are two channels in the protocol: 1. one is used to send data from sender to receiver, 2. and the other is to send an acknowledgement from the receiver to the sender.

In addition to this, both sender and receiver hold a bit value. This bit alternates between sends and receives to ensures that the receiver must receive the previously sent packet before the sender sends the next one.

Here we describe life-cycle of one message. The

flip operator takes a bit and performs the following operation: $flip(on) \rightarrow off$ and $flip(off) \rightarrow on$

1. Sender sends: $(data_1, bit)$ on data channel
2. Receiver receives $(data_1, bit)$ and sends: (bit) as an acknowledgement
3. Sender receives acknowledgement flips its bit and sends: $(data2, flip(bit))$
4. Receiver receives $(data2, flip(bit))$ and sends: $(flip(bit))$
5. Repeat

8.1 Specification

The main state operator is shown below.

```
op _:_>_|_<_:_ :
iNat Bit BitPacketQueue BitQueue Bit iNatList
    -> Sys
[ctor metadata "20"] .
```

We will discuss what each entry in the `Sys` sort represents (left to right).

1. `iNat`: A natural number representing the message id we are sending
2. `Bit`: The sender bit
3. `BitPackQueue`: A queue of packets. A packet contains `(iNat, Bit)`. This packet contains the message id and the bit of the sender at the time the packet was sent.
4. `BitQueue`: A queue of bits. This queue of bits contain the acknowledgements that the receiver sends.
5. `Bit`: The receiver bit.
6. `iNatList`: A list of the received messages from the sender

Each of these function modules (iNat, Bit, and others) are quite similar by essentially only contain a concatenation operator. However, equality enrichment [7] was applied to help with the prover. If we had a simpler implementation of equivalence, contextual rewriting [15] would struggle more when computing the equivalence over an implication during the simplification of our reachability formula. The equality enrichment can be seen below.

```
--- list of naturals
fmod INAT-LIST is
pr INAT .
sort iNatList .
sort NeiNatList .
subsort iNat < NeiNatList .
subsort NeiNatList < iNatList .
op nil : -> iNatList [ctor metadata "10"] .
--- list concat
op __ : iNatList iNatList -> iNatList
    [ctor assoc prec 61 metadata "11"] .
op __ : NeiNatList NeiNatList -> NeiNatList
    [ctor ditto metadata "11"] .
--- list equivalence
op _~iNL_ : iNatList iNatList -> Bool
    [comm metadata "12"] .
var L : iNatList .
vars P Q R S : NeiNatList .
var N M : iNat .
--- Equality enrichment
eq P ~iNL P = true .
... omitted ...
eq (s(N) P) ~iNL (s(M) Q) =
   s(N) ~iNL s(M) and P ~iNL Q .
... omitted ...
eq (P 0) ~iNL (Q s(N)) = false .
--- Identity
eq L nil = L .
eq nil L = L .
endfm
```

All in all we have 16 different equality equations which covers all the possible equali ties, and checks on all possible combinations of lists (of natural numbers) and natural numbers.

8.2 Rules

There are a total of 11 rules that cover the different steps which a system may take. These rules are all unconditional and use matching module in Maude's back-end to determine which rule will be taken at any time. We will describe how these rules work down below.

- stop: This is the stop operator for the CBRL tool
- send-1, send-2: These rules put a new BitPacket from a sender or a Bit from a receiver onto their respective queues
- recv-1a: Sender receives acknowledgement from sender that has the same bit .
- recAck: This rule represents the sender receiving an acknowledgement from the receiver
- addOutput: This rule takes a packet out of the BitPacketQueue, adds the message to the receiver's output list and flips the receiver's bit

- recvIgnore: This rule ignores a receive of a message from the sender to the receiver
- dropMsg: This drops a BitPacket message due to a lossy channel
- dropAck: This drops a Bit acknowledgment due to a lossy channel
- dup-1, dup-2: These duplicate either the BitPacket in the sender message queue or the Bit in receiver acknowledgement queue

Below we put a rule as an example.

```
--- N, N': Natural Numbers
--- B, B': Bits
--- (B,N): BitPacket (sender msg)
--- BPQ : BitPacketQueue
--- BQ : BitQueue (recv acks)
--- NL : Natural List
rl [addOutput] :
N : B > ( B' , N' ) ; BPQ | BQ < B' : NL
=> N : B > BPQ | BQ < flip(B') : (N' NL) .
```

This rule takes a message in the sender's queue, adds the message onto the receiver's output and flips the receiver's bit, if the receiver's bit is equivalent to the message's bit.

8.3 Invariant and Proof of the Alternating Bit Protocol

The invariant we wish to prove is an inductive invariant. Originally, in Rocha's paper [11], the invariant was defined in linear temporal logic and can be described as follows.

$$good - queues \wedge inv \rightarrow \circ(good - queues \wedge inv)$$

Recall that in CBRL, this can be essentially rewritten as the reachability logic formula.

$$B \rightarrow^{\circledast} [B]$$

`inv` (which uses the `gen-list` operator) represents the following property. The predicate `inv` is true when either:

1. The list in the receiver's output is equivalent to the list all the numbers preceding the sender's message id (including the sender).
2. Otherwise the list in the receiver's output concatenated with the current sender message id is equivalent to the list of all the numbers preceding the sender's message id (including the sender) .

Here is an example. The first covers the first case. The second covers the second case. The natural list in the receiver is denoted by the list in the parentheses. Space is our concatenation operator.

```
gen-list(5) = (5 4 3 2 1 0)
gen-list(5) = 5 (4 3 2 1 0)
```

`good-queues` is a conjunction of 2 other predicates. The predicates used is actually dependent on the values of the sender and receiver bit. The English description will be given below.

1. If both receiver and sender bits are `on` or `off`, a) all the bits in the receiver's acknowledgement queue must be all the same, b) the sender's message queue must either all have increasing message ids with corresponding bit flips (or is empty), or all the messages must be the same (or we have an empty queue).
2. If receiver and sender bits are different, a) the receiver's acknowledgement queue must either alternate or contain all of the same bit (or is empty), b) the sender's message queue must have all the same message bit pair (or be empty).

Throughout the proof, it was impossible to only `auto` through the proof and complete it. The lemmas being used are dependent on the rule but they share the same concepts behind it. For example in the `dropMsg` rule we have,

> Lemma 1: If the sender message queue is empty and our invariant is true then the invariant still holds after a dropped message.
>
> Lemma 2: If the sender queue is non-empty and our invariant is true, then after we drop a message we either have an empty queue and the invariant still holds true or we have a non-empty sender message queue where the invariant holds true.

In Lemma 1, we reason that the invariant must still be true because we did not change any values in the system state.

In Lemma 2, we reason that the invariant must still be true because if it was true before then we must still be true afterwards. There are two cases. If the queue has greater than one item, then the queue in question must have already satisfied the `good-queues` predicate which inductively reasons about the queue. Therefore, it must still be true regardless of an element being dropped. Alternatively, if the queue has one element, the invariant is vacuously true for an empty queue.

Below the truncated `dropMsg` proof is shown (... represents omissions). We use the `inv` keyword when declaring our invariant instead of `add-goal` here because we are trying to prove an inductive property of the system.

```
--- prove one rule at a time
(select-rls dropMsg .)
--- keyword for an inductive inv
(inv good-queues-invariant to ''[_:_>_|_<_:_'] on (N1 : B1 > BPQ1 | BQ1 < B1' : NL1)
    | (good-queues(N1:B1>BPQ1|BQ1<B1':NL1))
    = (true)
    /\ (inv(N1 : B1 > BPQ1 | BQ1 < B1' : NL1)
    = (true)
.)
--- add our lemmas
(add-axiom dropBPQ1 :
(N1 ... B1 > (nil).BitPacketQueue ... NL1) |
    (good-queues...) /\ (inv...) = (true) =>
([N2 ... (nil).BitPacketQueue ... NL2]) |
    (good-queues...) = (true)
    /\ (inv...) = (true) .)
(add-axiom dropBPQ2 : ...
       .)
(start-proof .)
--- use our lemmas on the initial goal
((use-axioms dropBPQ1 dropBPQ2 .
dropBPQ1 dropBPQ2 on 1 .)
--- case rule on the sender message queue
(case (N:iNat ... BPQ:BitPacketQueue ...)
    on BPQ:BitPacketQueue by
    ((nil).BitPacketQueue)
    U ((B':Bit, N':iNat) ; BPQ:BitPacketQueue)
.)
(auto .)
quit .
```

In addition to these lemmas, in order to simplify the proof, each rewrite rule was proved separately. This method was also applied in [13] to great effect where a whole browser specification was verified.

9 Challenges and Problems Encountered

Although CBRL has been around for a while since its inception, the implementation of the tool in Maude leaves much to be desired. The lack of QoL features such as readability, over the course of the semester there were some bugs encountered that were critical. For brevity we will only discuss the most pathological bugs here.

During the computation of unifiers used in matching, due to the unbounded and undecidable nature of associativity we ran into stack overflow error where the tool loops. This rendered proving anything impossible.

In the same vein, during the concatenation of elements into a list a similar issue occurred. We would also have a stack overflow crash. More specifically this occurred when we performed the following concatenation of non-empty lists.

```
NeList NeList -> NeList
```

There was a workaround to this by splitting our concatenation the following way

```
NeList List -> List
List NeList -> List
```

However, this means that it would be possible for a empty list constructor (`nil`) to present in the list which is undesirable. While this does not affect the correctness of the proofs on lists, but it does increase the difficulties and complexities of the proof.

The generalized case inference rule which uses pattern matching to split all relevant goals would also crash the tool. This is important when we have multiple goals (in the tens or even hundreds) that have the same pattern. For example, we could have multiple goals like below. Where all the rules can be matched to the first one (`N1`)

```
(N1) | (invariant(N1)) = (true) =>
    (M1) | (invariant(M1)) = (true)
(N2 + 1) | (invariant(N2)) = (true) =>
    (M2) | (invariant(M2)) = (true)
(N3 - 1) | (invariant(N3)) = (true) =>
    (M3) | (invariant(M3)) = (true)
...
```

If we had multiple rules like those shown above, we would have had to case on each one of them individually which would make proofs very verbose and almost impossible to do if the number of active goals is 3 digits.

Lastly, there is a most import bug we would need to overcome. Recall that a constrained pattern predicate must evaluates to true in order to be a valid goal. However, the tool failed to discharge vacuously false goals with $false = true$ during its T-Consistency check. It was not unusual to see goals of the form below.

```
(N1) | (true) = (false)
    /\ (invariant(N1)) = (true) =>
(M1) | (invariant(M2)) = (true)
```

This made proving anything impossible. In addition to this, there were no possible workarounds. Recall that CBRL has a subsumption inference rule. While we could subsume such a goal into itself (which in itself is a bug), this goal would be added to the list of axioms that the tool could apply on the next step. In essence allowing us to prove incorrect statements that have a constrained pattern predicate of $false = true$.

10 Comparison to Other Proof Systems

As mentioned earlier, I attempted to implement the Choice module from section 5 in two other proof systems, Isabelle and Coq. I spent around one month attempting to learn Coq and around 2 weeks learning Isabelle. In this time, I was unable to implement even the simplest system (Choice) I could fathom. While I understand this is not enough time to properly learn how to use such a complex ITP it did give critical insight for the comparison.

Isabelle and Coq are not intuitive to use in the proof of distributed systems. There is no built-in notion of a "rewrite rule" which allows one to easily mutate a system state. In Isabelle I found an example [8] of a consensus algorithm which was proved in section 6. It uses *stepFunction* and *execute* functions to represent the changes to a system. However, unlike in CBRL, these functions themselves needed proofs and supporting modules to complete a specification of just the system, **not** the protocol. As for Coq, there exists old proofs of the alternating bit protocol dating back to the nineties. However, even after reading the paper [6] and understanding the proof methodology in detail, understanding the Coq proof itself was beyond my current abilities.

While line count is not the most appropriate metric for comparison, I believe when we count both the proof and specification together it gives a cursory idea of how much more complex a proof would be in Isabelle and Coq.

System	CBRL	Coq	Isabelle
Consensus	<100 lines	N/A	˜400 lines
ABP	˜800 lines	>4000 lines	N/A

In addition to the number of lines of required for the specification and the proof, the sheer number of tactics available in both Coq and Isabelle (in the tens) pale in comparison to CBRL (<10). While both Coq and Isabelle provide their own "auto" tactic (auto/intuition in Coq and sledgehammer in Isabelle), they are nowhere near as simple and easy to understand than CBRL's auto.

11 Future Work

Due to the generic nature of CBRL and Maude, it is simple to prove more than just distributed systems. As shown in [1] there is already work on proving imperative programs with CBRL and other work on proving imperative languages such as Java [2] in rewrite theories.

However it should not be a big stretch to take this idea one step further and model automata and hybrid automata in a rewrite theory. With regards to hybrid automata the only missing link is a functional module specifying the semantics of real numbers. From there it would be simple to model a hybrid automata. To demonstrate this, we give a theoretical partial module with operators and rewrite rules which could represent the transitions and update rules in a hybrid automata.

```
sorts Real State Automata .
subsort State < Automata .
--- Value, update rate: x, xDot
op _,_ : Real Real : -> State [ctor] .
--- currentState | State1 | State2
op _|_|_| : Nat State State -> Automata [ctor] .
--- variable declaration
vars N1 N2 M1 M2 : Real .
--- Rules
--- Increment state1's variable
crl [update-1] :
    1 | N1,M1 | N2,M2 | =>
    1 | N1+M1,M1 | N2,M2 | .
--- Increment state2's variable
rl [update-2]
    2 | N1,M1 | N2,M2 | =>
    2 | N1,M1 | N2+M2,M2 | .
--- Transition from state1 -> state2
crl [trans1-2] :
    1 | N1,M1 | N2,M2 | =>
    2 | N1,M1 | N2,M2 |
    if transCond1 = true .
--- Transition from state2 -> state1
crl [trans2-1] :
    2 | N1,M1 | N2,M2 | =>
    1 | N1,M1 | N2,M2 |
    if transCond2 = true .
```

12 Conclusion

CBRL is a very powerful and young invention. It differs from other proof systems by being theory

generic, and when paired with Maude, it allows the user to create very powerful rewrite theories to prove. This allows a user to very easily model system changes and steps in a distributed system via conditional and unconditional rewrite rules. However, the current implementation of the tool leaves much to be desired. From critical implementation bugs, quality of life issues, and lack of documentation, the tool currently is only usable by a very small amount of people. Namely the creator and those who have access to the creator of the tool.

However, while the problems listed above are very real, with the help from users such as myself, the tool has developed a lot over the past few months. Bugs have been ironed out and quality of life improvements have been made. The current implementation is already much more usable than before. With the changes, as we have shown here, distributed system proofs of varying complexity can be very easily done in CBRL and Maude.

References

[1] M. Abir, M. Saxena, https://github.com/mickyabir/cs576.

[2] F. Chen, M. Hills, G. Rosu, A rewrite logic approach to semantic definition, design and analysis of object-oriented languages.

[3] N.-S. Chen, H.-P. Yu, S.-T. uang, A self-stabilizing algorithm for constructing spanning trees.

[4] M. Clavel, F. Duran, S. Eker, S. Escobar, P. Lincoln, N. Marti-Oliet, J. Meseguer, C. Talcott, Maude manual (version 2.7.1).

[5] F. Duran, J. Meseguer, Chc 3:a coherence checker tool for conditional order-sorted equational maude specifications.

[6] E. Gimenez, An application of co-inductive types in coq: verification of the alternating bit pro31.464mstocol.

[7] R. Gutierrez, C. R. Jose Meseguer, Order-sorted equality enrichments modulo axioms.

[8] M. Kleppmann, https://gist.github.com/ept/b6872fc541a68a321a26198b53b3896b.

[9] J. Meseguer, Twenty years of rewriting logic.

[10] J. Meseguer, Variant-based satisfiability in initial algebras.

[11] H. C. R. Nino, Symbolic reachability analysis for rewrite theories.

[12] D. Plyukhin, G. Agha, Concurrent garbage collection in the actor model.

[13] S. Skeirik, J. Meseguer, C. Rocha, Verification of ibos browser security properties in reachability logic.

[14] A. S. Stephen Skeirik, J. Meseguer, A constructor-based reachability logic for rewrite theories.

[15] H. Zhang, Contextual rewriting in automated reasonin.