Towards Agent Programming in CHR

Edmund S. L. Lam and Martin Sulzmann

School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
{lamsoonl,sulzmann}@comp.nus.edu.sg

Abstract. We investigate an approach to the design and implementation of linear logic based agent systems via the linear logic semantics of Constraint Handling Rules (CHR). The intuition behind our approach is simple: Linear logic provides strong logical foundations to reason, verify and specify agent systems beyond the limitations of classical logics, while with CHR, one can implement and analyse agent systems in a concise and compact manner by executable inference rules. We discuss necessary refinements of the CHR semantics to allow for sequential computations of actions and the verification of action determinism. Our approach can possibly provide a seemless integration of the formal specification and implementation of agent programs via CHRs.

1 Introduction

The idea of autonomous intelligent agent in programming has become increasing prevalent since its introduction in the early days of Artifical Intelligence. The concept of intelligent agents have opened new possibilities to computerization, allowing complex tasks previously thought to be beyond the computational limits of programming, to be mechanized. From advanced web service applications, to telemetry control systems for space exploration vessels (NASA's Deep Space 1, DS1 [MNPW98]), agent systems have been successfully deployed in a wide range of applications. Yet in spite of active research and progress into formal methods for agent specification, development of agent systems is still a difficult and tedious process because of the 'gap' between the abstract agent specifications and the concrete implementations.

In this paper, we explore an approach to the design and implementation of linear logic based agent systems via Constraint Handling Rules [Frü95] (CHR). Linear logic [Gir95] provides strong logical foundations to reason, verify and specify agent systems beyond the limitations of classical logics, thus avoiding the frame problem [MH69] entirely. With CHR one can implement and analyse agent systems in a concise and compact manner by executable inference rules. Recent works in [BF05] have shown that CHRs have a natural linear logic declarative semantics, establishing a long awaited bridge between the two worlds. Hence we can *directly* program linear logic agents in CHRs. One major advantage is that many agent domain properties can be stated and checked via the executable CHR operational semantics. Thus, CHR encodings of agent systems can support an effective and seemless integratation of specification and implementation. Specifically, we make the following contributions:

– We explore the benefit of CHRs to describe linear logic agents (Section 2). For example CHRs provide a natural and direct solution to the frame problem.

– We introduce an agent oriented refinement of CHR semantics, Monadic Action CHR Semantics, to support action sequencing (Section 3).
– We explore the verification of action determinism via a refined confluence test with respect to the domain invariants of the agent domain (Section 4).

In Section 3 we provide background material on CHRs as well as discuss the various issues of our approach. We also highlight the necessary refinements to the CHR semantics. Following this, in Section 4 we discuss action determinism and confluence. We conclude in Section 5 where we also discuss related works.

2 A Motivating Example

We consider a simple example called the *Block world*, which is a commonly used example to illustrate planning and domain representation problems in artifical intelligence. Block world consist of several blocks stacked on several tables and a robotic arm capable of picking up and putting down blocks one at a time. We consider a simple instance of Block world consisting of 3 blocks($B1, B2, B3 \in Blocks$), 3 tables(Labelled $T1, T2, T3 \in Tables$) and the robotic arm. To describe the the block world domain, we define four fluents (properties of the domain); $On(x, y)$ states that a block x is on block or table y, $Clear(x)$ states that nothing is stacked on x, $Empty$ states that the robot arm is empty and $Holds(x)$ says that the robot arm is holding x. A state is represented by a set of fluents described above. Block world has 2 legal actions described by the following.

– ***get(x)***: Provided the robotic arm is empty, block x is clear and x is on some y, the arm can pick up x, causing x to be no longer clear and on y, with x in the arm and y clear.
– ***putOn(x,y)***: Block x can be placed on y provided that x is held in the robotic arm, and y is clear. This makes the robotic arm empty and y no longer clear, with x on y.

To complete the informal specification, we also require the following domain invariants, labelled $\mathcal{I}^{Bw}$ 1 to 5.

– $\mathcal{I}_1^{Bw}$: Robotic arm cannot be both empty and holding something.
– $\mathcal{I}_2^{Bw}$: Objects cannot be both clear and have something on it.
– $\mathcal{I}_3^{Bw}$: Objects can only be stacked on and with at most one other object.
– $\mathcal{I}_4^{Bw}$: Robotic arm can hold at most one object.
– $\mathcal{I}_5^{Bw}$: Objects held by robotic arm cannot be clear or stacked on or with other objects.

Notice that if we try to represent actions naively in classical logic, we get an inconsistency as they contradict the domain invariants. Propositions, in classical logic, are viewed as 'truths' and once asserted, they are persistently valid. Situation Calculus [MH69] provides a solution to the representation of dynamically changing domains in classical logics. In situation calculus, we parameterize each fluent' additionally with a *situation*. Situations represent the sequence of actions that has lead to it's truth. For example, the fluent $Empty(s_1)$ where $s_1 = putOn(B1, T3) : get(B1) : S_0$ (we use the operator ':' as a standard list concatenation operator) simply asserts that the robotic arm is empty in the

state after the actions $get(B1)$ followed by $putOn(B1, T3)$ are executed from the initial situation S_0. Situations act as 'time tags' that distinguishes fluents of different states. With our fluents now parameterized by the situation, we are able to formulate the block world actions in first order logic:

$$
\begin{array}{ll}
(get) & \forall x, y, S. \\
& Empty(S) \wedge Clear(x, S) \wedge On(x, y, S) \supset Holds(x, get(x) : S) \wedge Clear(y, get(x) : S) \\
(putOn) & \forall x, y, S. \\
& Holds(x, S) \wedge Clear(y, S) \supset Empty(putOn(x, y) : S) \wedge On(x, y, putOn(x, y) : S) \wedge \\
& \qquad\qquad\qquad\qquad Clear(x, putOn(x, y) : S)
\end{array}
$$

It would seems that we can correctly represent actions simply by introducing situation 'time tags'. However we have also unwittingly introduced a new problem, known as the *Frame problem*.

2.1 The Frame Problem

Let's consider the situation calculus representation presented in the previous section. Suppose from the initial state $\Delta_0 = Empty(S_0) \wedge On(B1, B2, S_0) \wedge On(B2, B3, S_0) \wedge On(B3, T1, S_0) \wedge Clear(B1, S_0) \wedge Clear(T2, S_0) \wedge Clear(T3, S_0)$, the robotic arm executes $get(B1)$. Thus, from Δ_0 using the formula $get(B1)$, we can entail that $Holds(B1, S_1) \wedge Clear(B2, S_1)$ where $S_1 = get(B1) : S_0$. This yields the next state, $\Delta_1 = \Delta_0 \wedge Holds(B1, get(B1) : S_0) \wedge Clear(B2, get(B1) : S_0)$, which is unfortunately still incomplete, since we should also have $Clear(T2, S_1)$, $Clear(T3, S_1)$, $On(B2, B3, S_1)$ and $On(B3, T1, S_1)$. The formulation of the actions given above do not propagate the non-effects (non-effects refers to fluents that are not involved in the action, which should remain the way they were in the next situation). The problem of explicitly representing these non-effects of actions is known as the frame problem [MH69].

There are many proposed solutions to the frame problem [BMR95,Thi98]. The *successor state axioms* [BMR95] is one such solution. In [Thi98], Fluent calculus is proposed, solving the frame problem with *state update axioms*, which describe actions via explicit state representation. No doubt each of these solutions solves the frame problem, yet the cost in complexity and effort would often make any attempt in specifying practical dynamic agent systems in such classical logic based calculi extremely tedious, time consuming and at times difficult to read.

2.2 Block World via Linear Logic & CHR

Linear logics [Gir95] is an extension of classical logics. In linear logics, propositions are treated as expendible resources rather than irrefutable truths. This corresponds more to properties of a dynamically changing physical domain. The block world actions can be accurately represented by the following *linear implications*:

$$
\begin{array}{rl}
(get) & !(\forall x, y.\ Empty \otimes Clear(x) \otimes On(x, y) \multimap Holds(x) \otimes Clear(y)) \\
(putOn) & !(\forall x, y.\ Holds(x) \otimes Clear(y) \multimap Empty \otimes On(x, y))
\end{array}
$$

This linear logic interpretation of the block world is consistent and correctly represents the intended dynamics of the block world. Informally speaking, the implication ($\multimap$) says replace the left-hand-side conjuncts ($\otimes$) with the right-hand-side. The application of *get* $B1$ will result in the removal of $Empty$ (among others) before introduction of $Holds(B1)$ (among others), thus avoiding inconsistency.

The semantics of linear logic implications are remarkably similar to the operational semantics of the CHR simplification rule. Works in [BF05] have concretized this link between CHR and linear logic, suggesting that we can *directly* encode such agent systems as a CHR program. Here is the straightforward encoding via CHRs.

$$\begin{array}{rcl} get & @ & Empty, Clear(x), On(x,y) \iff Holds(x), Clear(y) \\ putOn & @ & Holds(x), Clear(y) \iff Empty, On(x,y), Clear(x) \end{array}$$

Each action in the block world is associated with a CHR rule to which we refer to as an *action rule*. A state in block world is simply represented by a CHR execution state. Like their linear logic counterparts, these direct CHR translations of the actions naturally solve the frame problem. Yet they are still problematic as we cannot explicitly state which action rules should fire and which should not. Thus, when applied to the CHR operational semantics, all rules fire exhaustively and non-deterministically. A further problem is that the above CHRs are potentially non-terminating.

Our solution is to introduce explicit *action constraints*. Here is our actual encoding of the block world.

$$\begin{array}{rcl} get & @ & get(x), Empty, Clear(x), On(x,y) \iff Holds(x), Clear(y) \\ putOn & @ & putOn(x,y), Holds(x), Clear(y) \iff Empty, On(x,y), Clear(x) \end{array}$$

Each action rules is appended with an action constraint at the left-hand-side of the rule. We distinguish fluents and actions constraints by upper case and lower case terms respectively. Here is a CHR derivation after executing the actions $get(B1)$ followed by $putOn(B1, T3)$.

$$\begin{array}{ll} & \{get(B1), putOn(B1,T3), Empty, On(B1,B2), On(B2,B3), On(B3,T1), \\ & Clear(B1), Clear(T2), Clear(T3)\} \\ \rightarrowtail_{get} & \{putOn(B1,T3), Holds(B1), On(B2,B3), On(B3,T1), Clear(B2), \\ & Clear(T2), Clear(T3)\} \\ \rightarrowtail_{putOn} & \{Empty, On(B1,T3), On(B2,B3), On(B3,T1), Clear(B1), Clear(B2), \\ & Clear(T2)\} \end{array}$$

With explicit action constraints we avoid the infinite firing of rules. Though, there are still several technical issues that must be addressed:

Action Sequencing. The sequence of execution of actions according to some plan must be respected. Our solution is to introduce a monadic action CHR semantics which can neatly be explained in terms of monads [Wad95]. The CHR solver is a state monad, driven by monadic action operations which describe sequential computations of actions. Here is a possible Haskell implementation of an agent executing the action sequence $[a_1(\bar{t}_1), ..., a_n(\bar{t}_n)]$.

Agent Program	Encapsulated CHR Derivations
do	
initialize C_1	
$a_1^M(\bar{t}_1)$	$(\{a_1(\bar{t}_1)\} \uplus C_1) \rightarrowtail^*_{P_{a_1}} C_2$
$a_2^M(\bar{t}_2)$	$(\{a_2(\bar{t}_2)\} \uplus C_2) \rightarrowtail^*_{P_{a_2}} C_3$
....	....
$a_n^M(\bar{t}_n)$	$(\{a_n(\bar{t}_n)\} \uplus C_n) \rightarrowtail^*_{P_{a_n}} C_{n+1}$
end	

We assume that *initialize* C_1 initializes the CHR state monad to the initial store C_1, and $a_i^M(\bar{t}_i)$ are monadic operations associated to the action $a_i(\bar{t}_i)$. We assume that P_a are the CHRs belonging to action a. Each operation a^M has the effect of exhaustively fire the rules P_a on the current constraint store (as indicated by the CHR derivations on the right-hand side). There is clearly a design space to be explored; given that we have the action sequence $[a \mid A]$ to be executed from C and suppose that no rules in P_a can fire from C, then we must decide whether a should be 'skipped' and A executed, or computation be 'blocked' at a. The exact details of our monadic action CHR semantics are given in Section 3.2.

Action Determinism. A deterministic action is one which produces a unique effect (output) from each current state (input). Given that an action is deterministic by definition, we wish to find a mechnical way to decide if our agent CHR programs correctly represent such deterministic actions. In our approach, we use a well established CHR property, Confluence, which intuitively describes determinism of CHR programs and has a decidable test [Abd97] that can be efficiently implemented. Yet confluence is too restrictive for agent programs as it considers all states, including states that are inconsistent with our agent domains. However, by simply providing standard confluence checking routines with a set of domain invariants, inconsistent states can be accurately eliminated from consideration. In Section 4 we present the details of our approach.

3 CHRs as a Agent Specification Language

In this section we give a formal introduction to the CHR semantics. Note that we only consider CHR simplification rules. Following this, we define the monadic action CHR semantics.

3.1 CHR Syntax and Semantics

In this paper, we explore implementation of agent systems under a variant of the *theoretical operational semantics* w_t of CHR [DSdlBH04] that only allows the use of simplification rules. We shall call this variant w_s. We begin by defining some of the basic notations we will use throughout the paper. $\bar{a}$ denotes an arbitral sequence of a's. We also use $[a_1, ..., a_n]$ to denote sequences if the arity n must be explicit. We at times use the notation $[H \mid T]$ also to denote a sequence with the first element being H and the rest of the sequence T. The empty sequence is denoted as ϵ. Similar to standard notations of multisets, $\uplus$ for multiset union and $\emptyset$ for the empty set. For simplicity, we will sometimes treat

multisets as sequences, only difference being that ordering of objects are choosen non-deterministically. Finally, we define a function $fv(S) = \bar{x}$ that takes a set S and returns $\bar{x}$,the set of all occurring free variables in S.

The following defines the syntax of terms, constraints of CHR:

Term	t	::=	x	$\mid$	$f(\bar{t})$
Builtin Constraint	B	::=	$b(\bar{t})$	$\mid$	$B \wedge B$
CHR Constraint Set	C	::=	$\{u(\bar{t})\}$	$\mid$	$C \uplus C$

A *term* t is a variable x (we may sometimes use u, v, w, y or z, but reserve i and n for integers) or a function $f(t_1, ..., t_n)$ where arity $n \geq 0$. We define 2 types of constraints, *builtin* and *CHR constraints*, in which the former is handled by the underlying constraint solver while the latter by the CHR program. In order not to lose generality, we model *builtin constraints* as a conjunction of logical constraints $b(t_1, ..., t_n)$, where b is an abstract logical operator of arity n. We assume that the underlying solver handles at least logical equality ($t_1 = t_2$) and the *true* and *false* predicates with their obvious meanings. The *constraint theory* CT represent this abstract logical system of the solver and we write $CT \models H \supset H'$ to denote that under the constraint theory CT, H entails H', where H and H' are builtin constraints. *CHR constraints* are defined by $u(t_1, ..., t_n)$, where u is a predicate symbol. We use CHR constraints to represent fluents and action constraints. We will sometimes represent a single CHR constraint with the symbol c. A multiset of CHR constraints is either a singleton set c or a multiset union $C_1 \uplus C_2$. We write $\{c_1, ..., c_n\}$ for short of $\{c_1\} \uplus ... \uplus \{c_n\}$.

Definition 1. (Execution State) *An execution state is a tuple of the form* $\langle G, S, B \rangle$ *where goal* G *and store* S *are multisets of CHR constraints and* B *is a conjunction of builtin constraints. We use the symbol* σ *to represent an execution state.*

A reader familiar with CHR semantics would realize that our definition of execution state differs from the original in 2 main ways: removal of the *propagation history* T and definition of the constraint store S as a multiset rather than a set of *numbered constraints*. These differences are the results of removing propagation rules. The goal G, contains all constraints to be executed, while constraint store S contains all constraints that can be matched with constraint handling rules. An initial state is defined as follows.

Definition 2. (Initial State) *Given a goal* G*, a multiset of CHR constraints, the initial state of* G *is* $\langle G, \emptyset, true \rangle$.

Constraint handling rules describe multiset term rewriting among constraint sets. In w_s, we consider only rules of one form, known as *simplification rules*:

$$r \ @ \ h \iff g \mid b$$

where r is the rule name, h is a multiset of CHR constraints known as the *head*, g is a conjunction of builtin constraints known as the *guard* and b is a multiset of constraints known as the *body*. Note that we at times omit r if the rule name is not required and g if the guard is trivial (ie. $g = true$). A *CHR program* consists of a set of constraint handling rules. For convinence, we define a

function $renamed(P) = P'$ that takes a CHR program P and returns a renamed variant such that $fv(P) \cap fv(P') = \emptyset$. The operational semantics of W_s is based on the following transitions which map execution states to excution states:

$$
\begin{array}{ll}
(Solve) & \langle \{c\} \uplus G, S, B\rangle \rightarrowtail_P \langle G, S, c \wedge B\rangle \\
 & \text{where } c \text{ is a builtin constraint.} \\
(Intro) & \langle \{c\} \uplus G, S, B\rangle \rightarrowtail_P \langle G, \{c\} \uplus S, B\rangle \\
 & \text{where } c \text{ is a CHR constraint.} \\
(Apply) & \langle G, h \uplus S, B\rangle \rightarrowtail_P \langle C \uplus G, S, \theta \wedge B\rangle \\
 & \text{where } \exists (r \ @ \ h' \Longleftrightarrow g \mid C) \in renamed(P) \text{ and} \\
 & \quad \exists \theta \text{ a substitution such that:} \\
 & \quad\quad \bullet\ h = \theta(h') \\
 & \quad\quad \bullet\ CT \models B \supset (\theta \wedge g)
\end{array}
$$

Definition 3. (CHR Derivations) *A CHR derivation of program* P*, denoted* $\rightarrowtail_P^*$*, represent a sequence of execution states connected by* w_s *transitions that leads to a final state* σ_n *where no* w_s *transitions are applicable. Given states* $\sigma = \langle G, S, B\rangle$ *and* $\sigma' = \langle G', S', B'\rangle$*, we write the derivation from initial state* σ *to final state* σ' *as* $\sigma \rightarrowtail_P^* \sigma'$*. For brevity, we sometimes present derivations with builtin constraints removed.*

For convinence, we define the function $State(\langle G, S, B\rangle) = C$ where $C = \theta(G \uplus S)$ such that θ is the most general unifier of G and S, and $CT \models B \supset \theta$. The multiset C represents an abstraction of $\sigma = \langle G, S, B\rangle$. We shall refer to states σ and abstract states C interchangeably. We also define derivations for abstract states.

Definition 4. (State Abstracted Derivations) *Given a multiset of CHR constraints* C*, we define the state abstracted derivation of* C *as* $C \rightarrowtail_P^* C'$ *where* C' *is a multiset of constraints, and there exists a CHR derivation* $\langle C, \emptyset, true\rangle \rightarrowtail_P^* \sigma'$ *such that* $State(\sigma') = C'$*.*

We use state abstracted derivations when we are more interested in the *apply* steps of CHR derivations, also when *solve* and *intro* steps can be abstracted.

3.2 Monadic Action CHR Semantics

In this section, we introduce the various agent oriented refinements on the CHR semantics w_s. We introduce a special type of CHR constraints known as *action constraints* which act as the explicit representation of actions. For simplicity, we restrict action rules to the following form,

$$r \ @ \ a, h \Longleftrightarrow g \mid b$$

where r is the name of the action, a is an action constraint, h (qualification conditions) is a multiset of CHR constraints, g (guard) is a conjunction of builtin constraint and b (body) is a multiset of constraints. Note that h and b strictly contain no action constraints. We call rules of this form *simplification action rules* or simply action rules.

Definition 5. (CHR Agent Program) *A CHR agent program* P *is a set of simplification action rules. We write* P_a*, such that* $P_a \subseteq P$*, to denote the set*

of all action rules $(r@a, h \Longleftrightarrow g \mid b) \in P$ *for some* r, h, g *and* b*. For short, we sometimes display CHR programs as sets of rule names.*

Lemma 1. (Termination) *Given a CHR agent program* P*, a set of action rules,* P *is terminating.*

Proof sketch: This proof depends on the restriction that action constraints cannot be found in the body of action rules and the head of actions must contain exactly an action constraint. Thus, the firing of action rules involve the 'comsumption' of an action constraint. Since there can only be finitely many action constraints, P is always terminating. □

Given an execution state $\sigma = \langle G, S, B\rangle$, an action a is said to be *active* in σ if $\exists a(\bar{t}) \in S$ or $\exists a(\bar{t}) \in G$. If this active action a has an associated rule variant $(r \ @ \ a(\bar{t}), h \iff g \mid b) \in renamed(P)$ such that r may eventually fire in σ (ie. $(\{a(\bar{t})\} \uplus h) \subseteq (S \uplus G)$ and $CT \models B \supset g$), then action a is *qualified* in σ. Otherwise, it is *unqualified* in σ. By introducing action constraints, block world CHR representation P^{Bw} allows explicit control of actions in w_s operational semantics, as only rules associated to active actions are allowed to fire.

However, we still cannot represent *sequences* of actions explicitly since for an execution state $\langle G, S, B\rangle$, action constraints in the goal G are introduced to S in a non-deterministic order. We introduce the *Monadic action CHR semantics*, which describes sequential computations of agent actions. Action Monads are structures that governs the order in which actions constraints are introduced to the underlying CHR solver. Rules associated to an active action must also be allowed to fire before new actions are activated, hence guaranteeing that at most one action constraint can exist in the constraint store at every CHR derivation step. We define *action execution state* as follows:

Definition 6. (Action Execution State) *An action execution state is a tuple of the form* $(A \mid C)$ *where* A *is a sequence of actions (also known as the* plan*) and* C *is a multiset of CHR constraints that contains no action constraints.*

We view the action execution states as a monad that describes the sequential computation of actions by encapsulating the CHR derivations of individual actions within each action monadic operation. Note that in this paper, we focus entirely on the domain representation problem of agent systems and assume that our agents possess 'black box' planning routines, possibly encapsulated within monadic operations as well. We define the *Monadic action CHR derivation* $\longrightarrow_P$ by the three following transitions that connects action execution states to action execution states.

$$\frac{C \rightarrowtail_P^* C'}{(\epsilon \mid C) \longrightarrow_P (\epsilon \mid C')} \qquad \text{(Empty Sequence)}$$

$$\frac{(\{a\} \uplus C) \rightarrowtail_P^* C' \quad a \notin C'}{([a \mid A] \mid C) \longrightarrow_P (A' \mid C')} \qquad \text{(Qualified Action)}$$

$$\frac{(\{a\} \uplus C) \rightarrowtail_P^* (\{a\} \uplus C')}{([a \mid A] \mid C) \longrightarrow_P ([a \mid A] \mid C')} \qquad \text{(Unqualified Action)}$$

The action monads represent the clear dividing line between the sequential computations of agent actions and the commited choice CHR derivations

describing effects of each individual actions, allowing the two unidentical semantics to coexist and function supportively. The first transition (Empty Sequence) describes the base case, where the action sequence is empty. The next two transitions describe cases where action a is active in abstract state C. For the former is the case where a qualifies in C and the exhaustive CHR derivation from the state $(\{a\} \uplus C)$ must result in the firing of a rule $r \in P_a$. The latter is the case where action a is unqualified in C, in which further computation of the action sequence A is not possible as computation *blocks* at the action a.

The choice of the *blocking semantics* of unqualified actions is a design decision that has been made with much considerations. No doubt for $[a \mid A]$ we could have modeled unqualified action a to be 'skipped' and have the rest of the actions A executed as usual, yet this semantics is often undesirable: Given an action execution state $\langle [a \mid A] \mid C \rangle$, the fact that action a is unqualified in C implies that either the agent's planning routines has constructed an inapplicable plan because of unforseen changes in the agent domain, or simply that the desirable qualification conditions of the action a has not been realised yet and the action should be attempted again at a later stage. Either way, the most intuitive response of the action monad is to return the last known action execution state that was successfully computed, from which the agent's planning routines can further deliberate appropriate responses. This corresponds exactly to the blocking semantics describe by the unqualified action transition above.

The monadic action CHR semantics guarantees an important property which we denote *action isolation*. Action isolation states implies that our agent CHR solver is never put in a situation where more than one action constraint is in the constraint store.

Lemma 2. (Action Isolation) *Given an action sequence A_0 and an execution state C_0 such that C_0 contains no action constraints, for all action derivation sequences $(A_0 \mid C_0) \longrightarrow_P .... \longrightarrow_P (A_n \mid C_n)$, all C_i contains no action constraints.*

The proof sketch of lemma 2 is as follows: We proof by induction on action execution states. Assume that C_0 contains no action constraints and CHR derivation $C_0 \rightarrowtail_P^* C_1$, for the empty sequence case, we know the state C_1 that contains no action constraints, since all rules in P never introduce new action constraints. For the qualified action case, suppose the active action is a, an action rule associated to a must have fired and consumed a, thus C_1 contains no action constraints. For unqualified action case, C_1 indeed contains exactly one action constraint a, (ie. $C_1 = \{a\} \uplus C_2$) . However, output of the unqualified action transition is C_2 which contains no action constraint. □

Note that action isolation guarantees that any monadic action derivation $([a_1, ..., a_n] \mid C_1) \longrightarrow_P^* (\epsilon \mid C_{n+1})$ effectively produces the CHR derivation sequence $(\{a_1\} \uplus C_1) \rightarrowtail_{P_{a_1}}^* C_2$,..., $(\{a_n\} \uplus C_n) \rightarrowtail_{P_{a_n}}^* C_{n+1}$.

4 Action Determinism

In this section we introduce a framework to reason about properties of CHR agent systems. In particular, we examine *determinism of actions* defined by the following.

Definition 7. (Action Determinism) *An action a of an agent domain is* Deterministic *iff executing the action a from any agent domain state results in a unique resultant state.*

Both block world actions (*get* and *putOn*) are deterministic according to the block world specifications, thus any formal specification or implementation of these actions must correctly reflect this property. We introduce a practical way to verify the determinism of agent actions via CHR *confluence*. For the purpose of capturing cases of actions with multiple rules, we consider an extension of the block world, denoted *block world prime* $P^{Bw'}$.

$$\begin{array}{rcl} get1 & @ & get(x), Empty, Clear(x), On(x,y) \iff Holds(x), Clear(y) \\ get2 & @ & get(x), Holds(z), Clear(x), On(x,y) \iff Holds(x), Clear(y), Thrown(z) \\ putOn & @ & putOn(x,y), Holds(x), Clear(y) \iff Empty, Clear(x), On(x,y) \end{array}$$

Block world prime extends block world by allowing the *get* action to be executed even if the robotic arm is holding some other object z. The side effect is that z will be thrown away, denoted by the fluent $Thrown(z)$.

4.1 Confluence

We now define *state variants*, *joinability* and *confluence* under the theoretical semantics w_s.

Definition 8. (Variants) *Two execution states* $\sigma = \langle G, S, B\rangle$ *and* $\sigma' = \langle G', S', B'\rangle$ *are said to be variants* ($\sigma \approx \sigma'$) *if there exists a variable rename* ρ*, such that* $\rho(G) = G'$*,* $\rho(S) = S'$ *and* $CT \models B \leftrightarrow B'$*.*

Definition 9. (Joinable) *Given a CHR program P and two execution states,* σ_1 *and* σ_2*, are joinable if there exists 2 states* σ_1' *and* σ_2' *such that* $\sigma_1 \rightarrowtail^* \sigma_1'$ *and* $\sigma_2 \rightarrowtail^* \sigma_2'$ *and* σ_1' *and* σ_2' *are variants.*

Definition 10. (Confluence) *A CHR program P is confluent iff given the states* σ_0*,* σ_1 *and* σ_2*, we have the following: if* $\sigma_0 \rightarrowtail_P^* \sigma_1$ *and* $\sigma_0 \rightarrowtail_P^* \sigma_2$ *then* σ_1 *and* σ_2 *are joinable.*

Confluence provides a more suitable notion of determinism for CHR programs. In [Abd97], an efficient way of testing for confluence is introduced. This *confluence test* is based on the testing of all *critical pairs* between each rules of the CHR program. We define critical pairs and confluence test with respect to the w_s semantics.

Definition 11. (Critical Pairs & Origins) *Given a CHR program P, for any 2 rules* $(r_1 \ @ \ H_1 \iff g_1 \mid B_1)$ *and* $(r_2 \ @ \ H_2 \iff g_2 \mid B_2)$ *such that* $r_1, r_2 \in P$*. Let* $H_1' \subseteq H_1$ *and* $H_2' \subseteq H_2$ *and* θ *be the most general unifier of* H_1' *and* H_2'*. Also define* S, S_1 *and* S_2 *such that* $S = \theta(H_1) \uplus (H_2 - H_2')$*,* $S_1 \subseteq S$ *and* $S_1 = \theta(H_1)$*, and* $S_2 \subseteq S$ *and* $S_2 = \theta(H_2)$*, then the states*

$$\sigma_1 = \langle B_1, S - S_1, g_1 \wedge g_2\rangle \qquad \sigma_2 = \langle B_2, S - S_2, g_1 \wedge g_2\rangle$$

forms a critical pair (σ_1, σ_2) *between rules* $r1$ *and* $r2$*. We say that the critical pair* σ_1 *and* σ_2 *originates from* S*.*

Definition 12. (Confluence Test) *Given a terminating CHR program P, the confluence test on P is as follows: if all critical pairs of P are joinable, then P is confluent.*

Note that the confluence test is applicable only for terminating CHR programs. However, this will not be a problem, since we are interested only in terminating CHR agent programs. It would seems confluence test is perhaps a candidate for testing determinism of actions, yet $P^{Bw'}$ is not confluent even though it is obviously deterministic. Consider the state $\sigma_0 = \langle\{\}, \{get(x), Empty, Holds(w), Clear(x), On(x,y), On(x,w)\}, true\rangle$. The following shows derivations resulting to a non-joinable critical pair (σ_1, σ_2), between the rules $get1$ and $get2$.

$$\begin{array}{l} \langle\{\}, \{get(x), Empty, Holds(w), Clear(x), On(x,y), On(x,w)\}, true\rangle \\ \rightarrowtail_{get2} \langle\{Holds(x), Clear(z), Thrown(w)\}, \{Empty, On(x,y)\}, true\rangle \end{array}$$

$$\begin{array}{l} \langle\{\}, \{get(x), Empty, Holds(w), Clear(x), On(x,y), On(x,w)\}, true\rangle \\ \rightarrowtail_{get1} \langle\{Holds(x), Clear(y)\}, \{Holds(w), On(x,w)\}, true\rangle \end{array}$$

On closer observation, we notice that the origin (σ_0) of the critical pairs σ_1 and σ_2, describes a state that is inconsistent with block world domain invariant $\mathcal{I}_1^{Bw'}$ and should never be reachable (Robotic arm cannot be both $Empty$ and $Holds(w)$). In fact, all other non-joinable critical pairs of $P^{Bw'}$ either originates from states that do not conform to the block world specifications or are critical pairs between rules of different actions (action isolation of monadic action CHR derivation guarantees these never occur). Due to space constraints, we will not enumerate all such critical pairs here, but invite the reader to replicate this observation.

Confluence test is too restrictive in testing for action determinism because it considers all critical pairs, including those which may not be reachable from any initial state. What we need is a more refined confluence test that tests only *reachable* critical pairs for joinability and ignores unreachable ones.

Definition 13. (Reachability) *Given a CHR program P, a state C is reachable iff either C is an initial state or there exists a CHR derivation $C' \rightarrowtail_P^* C$ for some initial state C'.*

We define a weaker notion of confluence, called $\mathcal{I}$-confluence and the $\mathcal{I}$-confluence test as follows:

Definition 14. ($\mathcal{I}$-Confluence) *A CHR program P is confluent iff given the states σ_0, σ_1 and σ_2 such that σ_0 is reachable, we have the following: if $\sigma_0 \rightarrowtail_P^* \sigma_1$ and $\sigma_0 \rightarrowtail_P^* \sigma_2$ then σ_1 and σ_2 are joinable.*

Definition 15. ($\mathcal{I}$-Confluence Test) *Given a terminating CHR program P the $\mathcal{I}$-confluence test on P is as follows: For all critical pairs (σ_1, σ_2) of P, if origin of (σ_1, σ_2) is reachable and σ_1 and σ_2 are joinable, then P is $\mathcal{I}$-confluent.*

The $\mathcal{I}$-confluence of a program P immediately implies the determinism of all action rules of P. However, we still need a more concrete test for reachable state. For this, we look to domain invariants.

4.2 Domain Invariants

Domain invariants of block world prime ($\mathcal{I}_{1-5}^{Bw'}$) were briefly mentioned informally in section 2 (We assume that $\mathcal{I}_{1-5}^{Bw'} = \mathcal{I}_{1-5}^{Bw}$). These domain invariants can be generalized to the 2 following forms.

Definition 16. (Mutual Exclusion) *Given two CHR constraints $P(\bar{x})$ and $Q(\bar{y})$, and an state C, $P(\bar{x})$ and $Q(\bar{y})$ are mutually exclusive in C iff*

$$\forall \bar{z}.(P(\bar{x}) \in C) \supset (Q(\bar{y}) \notin C) \wedge (Q(\bar{y}) \in C) \supset (P(\bar{x}) \notin C) \text{ where } \bar{z} = fv(\bar{x}) \cup fv(\bar{y})$$

For brevity, we denote mutual exclusion of p_1 and p_2 in C as $\forall \bar{t}\ MX(p_1, p_2, C)$ where $\bar{t} = fv(p_1) \cup fv(p_2)$.

Definition 17. (Functional Dependency) *Given a CHR constraint $P(x_1, .., x_n)$, and an state C, the functional dependency $\{x_{i_0}, .., x_{i_j}\} \rightsquigarrow \{x_1, .., x_n\}$ where $\{x_{i_0}, .., x_{i_j}\} \subseteq \{x_1, .., x_n\}$ exists in C, iff*

$$\begin{array}{l} \forall x_1, .., x_n, y_1, .., y_n. \\ \quad (P(x_1, .., x_n) \in C \wedge P(y_1, .., y_n) \in C \wedge x_{i_0} = y_{i_0} \wedge .. \wedge x_{i_j} = y_{i_j}) \\ \qquad \supset (x_1 = y_1 \wedge .. \wedge x_n = y_n) \end{array}$$

For brevity, we denote function dependency, $\bar{x}' \rightsquigarrow \bar{x}$ where $\bar{x}' \subseteq \bar{x}$, of $P(x_1, .., x_n)$ in S as $\forall \bar{x}\ FD(P(\bar{x}), \bar{x}', C)$.

A third form of domain invariant *uniqueness* is required for a more technical reason that will be clear soon. Uniqueness of a CHR constraint $P(\bar{t})$ in C dictates that we can only have exactly one copy of $P(\bar{t})$ in the multiset C.

Definition 18. (Uniqueness) *Given a CHR constraint $P(\bar{x})$, and an state C, $P(\bar{x})$ is unique in C iff*

$$\begin{array}{l} \forall \bar{x}, n, m.(C' = label(C) \wedge (P(\bar{x})\#n) \in C' \wedge (P(\bar{x})\#m) \in C') \supset m = n \\ \text{where } label(\{c_1, .., c_i, .., c_n\}) = \{(c_1\#1), .., (c_i\#i), .., (c_n\#n)\} \end{array}$$

For brevity, we denote uniqueness of $P(\bar{x})$ in C as $\forall \bar{x}\ UN(P(\bar{x}), C)$.

We formally define domain invariants $\mathcal{I}$ as a conjunction of first order logic formulas of the above forms. We write $\mathcal{I}(C)$ to denote that the state C satisfies the domain invariant $\mathcal{I}$. We define block world initial states as follows.

Definition 19. (Block world Initial States) *Given a state C, C is a block world initial state if $\mathcal{I}(C)$.*

Note that we omit the obvious that initial state C must contain only block world fluents. An important property of the invariants $\mathcal{I}$ is as follows:

Lemma 3. (Invariance of Subsets) *Given a state C and invariants $\mathcal{I}$, if $\mathcal{I}(C)$ and $C' \subseteq C$, then $\mathcal{I}(C')$*

We omit a formal proof of lemma 3 because it follows almost immediately from the definition of the 3 forms of invariants. Given invariants $\mathcal{I}$, we are interested if a CHR program *preserves* $\mathcal{I}$. We define this by the following

Definition 20. (Preserving Invariants) *Given the invariants $\mathcal{I}$ and a CHR program P, P preserves $\mathcal{I}$ iff for any state C such that $\mathcal{I}(C)$ and we have the derivation $C \rightarrowtail_P C'$ for some state C', then $\mathcal{I}(C')$.*

We can now formally define the block world domain invariants $\mathcal{I}_{1-5}^{Bw'}$. The following shows that formal representation of the domain invariants:

$$
\begin{aligned}
&\mathcal{I}_1^{Bw'}(C) = \forall x.\ MX(Empty, Holds(x), C)\\
&\mathcal{I}_2^{Bw'}(C) = \forall x, y.\ MX(On(x,y), Clear(y), C)\\
&\mathcal{I}_3^{Bw'}(C) = \forall x, y.\ FD(On(x,y), \{x\}, C) \wedge \forall x, y.\ FD(On(x,y), \{y\}, C)\\
&\mathcal{I}_4^{Bw'}(C) = \forall x.\ FD(Holds(x), \emptyset, C)\\
&\mathcal{I}_5^{Bw'}(C) = \forall x, y.\ MX(Holds(x), On(x,y), C) \wedge \forall x, y.\ MX(Holds(y), On(x,y), C) \wedge\\
&\qquad\qquad \forall x.\ MX(Holds(x), Clear(x), C)
\end{aligned}
$$

We introduce the following axilliary invariants:

$$
\begin{aligned}
&\mathcal{I}_U^{Bw'}(C) = \forall x.\ UN(Holds(x), C) \wedge \forall x, y.\ UN(On(x,y), C) \wedge\\
&\qquad\qquad \forall x.\ UN(Clear(x), C) \wedge UN(Empty, C)\\
&\mathcal{I}_A^{Bw'}(C) = \forall x, y.\ MX(get(x), get(y), C) \wedge \forall x, y, z.\ MX(get(x), putOn(y,z), C) \wedge\\
&\qquad\qquad \forall x, y, w, z.\ MX(putOn(x,y), putOn(w,z), C)
\end{aligned}
$$

Things can go wrong if a block world state C is allowed to have multiple copies of certain fluents. Thus $\mathcal{I}_U^{Bw'}$ ensures all fluents of block world must be unique (Note that in general, we allow multisets). Also, recall the *action isolation* (section 3.2) property that the monadic action CHR derivations guarantee. We express this with $\mathcal{I}_A^{Bw'}$, which states mutual exclusion of all action constraints. The complete block world invariants are represented by:

$$\mathcal{I}^{Bw}(C) = \mathcal{I}_1^{Bw'}(C) \wedge \mathcal{I}_2^{Bw'}(C) \wedge \mathcal{I}_3^{Bw'}(C) \wedge \mathcal{I}_4^{Bw'}(C) \wedge \mathcal{I}_5^{Bw'}(C) \wedge \mathcal{I}_U^{Bw'}(C) \wedge \mathcal{I}_A^{Bw'}(C)$$

Lemma 4. ($P^{Bw'}$ Invariance) *Block world program $P^{Bw'}$ preserves $\mathcal{I}^{Bw'}$*

The proof of lemma 4 involves proving that each rule $r \in P^{Bw'}$ preserves each atomic invariant $\mathcal{I}_i^{Bw'}$ of $\mathcal{I}^{Bw'}$. We shall sketch the proof for $get1$, which will provide the general recipe of the proof. given a state C such that $\mathcal{I}^{Bw'}(C)$, by lemma 3, we can infer that removing constraints from C will not violate $\mathcal{I}^{Bw'}$, thus for $get1$ rule, we only need to check if invariants containing references of $Holds(x)$ and $Clear(y)$ is violated. Since we consider the derivation where $get1$ has fired, we must have $C \rightarrowtail_{get1} C'$ such that $C = \{get(X), Empty, Clear(X), On(X,Y)\} \uplus C''$ and $C' = \{Holds(X), Clear(Y)\} \uplus C''$ for some C'' and skolem constants X and Y. Now we prove $\mathcal{I}^{Bw'}(C')$. The enumeration of the proof is as follows, recalling for each we have $\mathcal{I}^{Bw'}(C)$:

- Prove $\mathcal{I}_1^{Bw'}(C')$. Since $C = \{get(X), Empty,\ Clear(X), On(X,Y)\} \uplus C''$ and $UN(Empty, C)$, then $Empty \notin C''$. Thus, $Empty \notin C'$, implying $\mathcal{I}_1^{Bw'}(C')$.

- Prove $\mathcal{I}_2^{Bw'}(C')$. Similar to above, we have $On(X,Y) \notin C''$, Thus $On(X,Y) \notin C'$ and $\mathcal{I}_2^{Bw'}(C')$.
- Prove $\mathcal{I}_4^{Bw'}(C')$. We need $\forall z.\ Holds(z) \notin C''$. Since $Empty \in C$, $\mathcal{I}_1^{Bw'}(C)$ dictates that $\forall z.\ Holds(z) \notin C''$.
- Prove $\mathcal{I}_5^{Bw'}(C')$. First we need $\forall z.\ MX(Holds(X), On(X,z), C')$ which we get from $On(X,Y) \notin C''$ and $\mathcal{I}_3^{Bw'}(C)$. Next, we get $\forall z.\ MX(Holds(X), On(z,X), C')$ from $Clear(X) \in C$ and $\forall z.\ MX(On(z,X), Clear(X), C)$. Finally $MX(Holds(X), Clear(X), C')$ is simply obtained from $Clear(X) \notin C''$.
- Prove $\mathcal{I}_U^{Bw'}(C')$. We need $UN(Holds(X), C')$ and $UN(Clear(Y), C')$. Since $Empty \in C, \forall z. Holds(z) \notin C''$, by $\mathcal{I}_1^{Bw'}(C)$, hence $UN(Holds(X), C')$. Since $On(X,Y) \in C$ $Clear(Y) \notin C''$ as $\mathcal{I}_2^{Bw'}(C)$. Therefore $UN(Clear(Y), C')$.

Thus, we have $\mathcal{I}^{Bw'}(C')$ for the *get*1 rule. Notice that the uniqueness of fluents of block world are necessary for the proof. The proof for the *get*2 and *putOn* rules can be easily replicated from the above recipe. Note that the reasoning involved in the proof above is rather simple and mechanical, suggesting possibility for efficient implementations if domain invariants are well represented. □

Lemma 5. (Block world Soundness) *With respect to program $P^{Bw'}$, given a state C, if C is reachable, then $\mathcal{I}^{Bw'}(C)$.*

Proof Sketch of lemma 5 is as follows: there are 2 cases of C. First, if C is a block world initial state, then we get this by definition 4.2. Otherwise we have $C' \rightarrow^* C$ for some initial state C, which means $\mathcal{I}^{Bw'}(C')$ By lemma 4, we get $\mathcal{I}^{Bw'}(C)$. □

4.3 $\mathcal{I}$-Confluence of Block World

With the domain invariants of block world $\mathcal{I}^{Bw'}$ and the soundness of block world with respect to the invariants, we can prove the $\mathcal{I}$-confluence of block world.

Lemma 6. ($\mathcal{I}$-Confluence of Block World) *$P^{Bw'}$ is $\mathcal{I}$-confluent under $\mathcal{I}^{Bw'}$.*

Proof Sketch of lemma 6: We begin by setting up the use of the invariants $\mathcal{I}^{Bw'}$ to identify reachable states of block world. We use lemma 5 which states that C is reachable implies that $\mathcal{I}^{Bw'}(C)$ Given a critical pair (σ_1, σ_2) that originates from a state S, we test reachability of (σ_1, σ_2) by testing $\mathcal{I}^{Bw'}(S)$. If $\mathcal{I}^{Bw'}(S)$ then we test if σ_1 and σ_2 are joinable. Otherwise (σ_1, σ_2) is unreachable: Domain invariants $\mathcal{I}_{1-5}^{Bw'}$ prunes away states that are inconsistent with block world, while $\mathcal{I}_U^{BW'}$ prunes away states with duplicate block world fluents and $\mathcal{I}_A^{Bw'}$ prunes away states violating action isolation (lemma 2). The second part of the proof involves the enumeration of all critical pairs of $P^{Bw'}$ and pruning away critical pairs that are unreachable, and testing all others for joinability. We omit this part of the proof but insist that it is tedious but straight forward. □

Note that this testing method is in fact only an approximate for $\mathcal{I}$-confluence test: Passing the test guarantees $\mathcal{I}$-confluence, but failing does not guarantee non $\mathcal{I}$-confluence. In general, the set of all states S such that $\mathcal{I}(S)$ is a super set of the 'ideal' set of all reachable states of the CHR program.

5 Conclusion

We have laid out the foundations for an approach towards programming of linear logic agents via CHRs. Linear logics provide a strong and sound logical foundation for the formal description and reasoning of dynamic agent systems, while CHRs provide an executable framework for testing and running simulations. We have investigated the necessary refinements of the standard CHR semantics such as action constraints and the monadic action CHR semantics.

Our approach in using monads to combine computation paradigms shares similarities with the monadic concurrent linear logic programming proposed in [LPPW05] called LolliMon. LolliMon has a natural forward chaining, committed choice operational semantics inside the monad, and an asynchronous linear logic based backward chaining opertional semantics outside the monad. The $\mathcal{I}$-confluence we have defined for determinism test is closely related to observable confluence explored in [DSS06]. These resemblances are yet to be studied in detail.

In future work, we plan to develop a general $\mathcal{I}$-confluence test for CHR programs with reachable states approximated by program annotations denoting domain invariants. We also intend to extend the framework by introducing propagation rules to the framework.

References

[Abd97] S. Abdennadher. Operational Semantics and Confluence of Constraint Propagation Rules. In *Proc. of CP'97*, pages 252–266, 1997.

[BF05] H. Betz and T. Frühwirth. A Linear-Logic Semantics for Constraint Handling Rules. In *Proc of CP'2005*, volume 3709 of *LNCS*, pages 137–151. Springer-Verlag, 2005.

[BMR95] A. Borgida, J. Mylopoulos, and R. Reiter. On the Frame Problem in Procedure Specifications. *IEEE Trans. Software Eng.*, 21(10):785–798, 1995.

[DSdlBH04] G. J. Duck, P. J. Stuckey, M. J. García de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *Proc of ICLP'04*, pages 90–104, 2004.

[DSS06] G. J. Duck, P. J. Stuckey, and M. Sulzmann. Observable Confluence for Constraint Handling Rules, 2006. In this volume.

[Frü95] T. Frühwirth. Constraint Handling Rules. *Lecture Notes in Computer Science*, (910):90–107, 1995.

[Gir95] J. Y. Girard. Linear Logic: Its Syntax and Semantics. In *Proc. of Workshop on Linear Logic, Cornell University*, number 222. Cambridge University Press, 1995.

[LPPW05] P. López, F. Pfenning, J. Polakow, and K. Watkins. Monadic Concurrent Linear Logic Programming. In *Proc. of PPDP*, pages 35–46, 2005.

[MH69] J. McCarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[MNPW98] N. Muscettola, P. Pandurang Nayak, B. Pell, and B. C. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[Thi98] M. Thielscher. Introduction to the Fluent Calculus. *Electron. Trans. Artif. Intell.*, 2:179–192, 1998.

[Wad95] P. Wadler. Monads for Functional Programming. In *Advanced Functional Programming*, pages 24–52, 1995.