Executable denotations for concurrent languages using Concurrent Transaction Logic

Marcus Vinicius Santos

Department of Computer Science,
Ryerson University
M5B 2K3, Toronto, Canada
p: (+1) 416.979.500 x 7062; f: (+1) 416.979.5064
m3santos@ryerson.ca

Abstract. This paper presents an approach based on a Horn fragment of Concurrent Transaction Logic ($\mathcal{CTR}$) for semantic description and execution of programming languages. The Horn notation is used in much the same way that plain Horn logic is used to specify semantics of programming languages. However, $\mathcal{CTR}$ extends that framework a deductive database language which provides a declarative, logic programming framework that naturally accommodates the notions of store, store updates, dataflow in declarative languages, data-driven concurrency, and message passing concurrency. The contributions of this paper are twofold: it shows how the semantics of concurrent programming languages can be fully specified in a Horn-based logic framework; and it demonstrates that $\mathcal{CTR}$-based logical denotations provide a unified formal semantics for such languages, which can also serve as a prototyping tool for the language developer.

keywords: Horn logic, Semantics, Concurrent programming languages

1 Introduction

It is known that semantic descriptions of programming languages based on Horn logic (Horn denotations, for short) provide specifications which are easy to read and have a natural "calculational clarity" [5]. Horn denotations are also known to provide executable specifications, thus yielding an interpreter for the language under study. Such executable specifications are useful for both pragmatists and theorists; pragmatists could instrument an executable specification to obtain debuggers and profilers for the programming language; theorists could automatically verify program properties by adding preconditions and postconditions to parts of the program denotation. These and other features of Horn denotations have made this sort of approach potentially more attractive then denotations based on λ-calculus.

This paper proposes a framework based on a Horn fragment of Concurrent Transaction Logic ($\mathcal{CTR}$) [2] for semantic description of concurrent programming languages. In this paper we address two types of concurrency: data-driven concurrency and message passing concurrency.

Moss presented in [9] an overview on how to completely specify a language in Prolog. Building on this idea, Slonneger [15] convincingly demonstrated that, for the specification of denotational semantics, Prolog can be regarded as superior to imperative languages. Gupta [5] explored how Horn logic denotations lead to some interesting practical applications, such as automatic program verification and automatic generation of compilers. More recently, Gupta [6] and Wang [16] used the logical denotational framework introduced in [5] in combination with software component development techniques to provide language-centric approaches to software engineering.

The semantic specification framework used here is similar to the one proposed in [5] in the sense that we also use definite clause grammar (DCG) rules to specify the syntax of the language, and Horn rules to specify the semantic algebras and valuation functions. One difference concerns the representation of the store and store updates; another concerns the use of a different logic, *viz.* $\mathcal{CTR}$, that supplies a syntax and a semantics by which elementary database operations can be combined serially and concurrently into complex programs. In Gupta's framework, the store is represented as lists, which are thread through the Horn rules via parameters. Gupta also shows in [5] an alternative approach to obtain less cluttered

predicates: he uses the database to model the store, and Prolog's primitive operators *assert* and *retract* to specify store updates. Unfortunately, one knows that these update operators have no logical semantics. Moreover, Prolog programs using these operators are often awkward and the most difficult to understand, debug and maintain. These problems are all aggravated by concurrency. In $\mathcal{CTR}$, however, the notion of database state and state update is part of the logical framework. In $\mathcal{CTR}$, predicates communicate via the database. Hence, a concurrent process can read what another has written. This form of communication leads to a programming style that is very different from that of existing concurrent logic programming languages (CLP) [14], where concurrent processes communicate via shared variables and unification. Therefore, we capitalize on this feature of the formalism to obtain a framework for logical denotations which not only produces specifications which are easier to read, but also scales well when dealing with different kinds of concurrency in programming languages.

There have been other approaches to provide "user-friendly" frameworks for semantic description of programming languages. Mosses presented in [10] a hybrid semantic framework called action semantics which aims to provide easy-to-read, useful semantics for realistic programming languages. He showed in [11] that action semantics can also be used for providing semantic descriptions of concurrent languages. Doh showed in [4] that such framework can be used as a basis for a method for composing a programming language by combining action semantics modules. Unquestionably, action semantics does have interesting pragmatic features. However, it is clear that it does not provide executable semantics, such as Horn-based denotations; also it lacks a complete formal system which would allow one to reason about programs. The action semantics framework is also not adequate to accommodate shared-state concurrent languages. We conjecture that our framework can be used to provide semantic descriptions of languages with this type of concurrency. However, due to space restrictions, we did not address this issue here. In this paper we show that data-driven concurrency and message passing concurrency can be naturally specified in our framework.

In regards to the communication paradigm used in $\mathcal{CTR}$, it is inspired by the π-calculus [8]. That calculus is an extension of the formalism introduced by Milner in [7], designed for expressing concepts of parallelism and mobility in computation. However, $\mathcal{CTR}$ is a logic for *programming*, while π-calculus is an algebra for *specifying* and *verifying* finite-state concurrent systems (which databases and logic programs are not).

This paper as organized as follows: in Section 2 we motivate the use of $\mathcal{CTR}$ (and its Horn fragment, called $\mathcal{CTR}^{\mathsf{H}}$) and explain the basic concepts underlying the logic; Section 3 illustrates the use of the framework by giving a semantic description of a simple stateless, sequential programming language; in Section 4 we extend that framework to address concurrency, *viz.* data-driven concurrency and message passing concurrency.

Our results show that the semantics of programming languages in general, and concurrent languages in particular, can be fully specified in the $\mathcal{CTR}$ logic framework. It has become apparent from our approach that the presence of concurrency does not affect the description of other constructs in the specification. This contrasts sharply with conventional denotational descriptions, where domains of higher-order functions used to model concurrency and nondeterminism are radically different from those used in the specification of sequential languages.

2 Overview of Horn Concurrent Transaction Logic

$\mathcal{CTR}$ is an extension of classical logic that seamlessly integrates concurrency and communication with queries and updates. It has a purely logic semantics, including a natural model theory and a sound-and-complete proof theory. Like classical logic, $\mathcal{CTR}$ has a Horn fragment, here called $\mathcal{CTR}^{\mathsf{H}}$, with a procedural interpretation, in which programs can be specified and executed. This section reviews the syntax of $\mathcal{CTR}^{\mathsf{H}}$ and its procedural interpretation, summarizing material from [2].

2.1 Syntax

An atomic formula is a *goal*. If ϕ_1 and ϕ_2 are goals, then so are the formulas $\phi_1 \otimes \phi_2$, $\phi_1 \mid \phi_2$, and $\odot\phi$. If ϕ is a goal and p is an atomic formula, then $p \leftarrow \phi$ is a rule. A finite set of rules is a rulebase. A rulebase with a goal is a *program*.

For convenience, we assume that $\otimes$ binds more tightly than $|$. Thus, the expression $p \mid q \otimes r$ is parsed as $p \mid (q \otimes r)$. Intuitively, goals are procedures, and rules are subroutine definitions (in the logic-programming tradition). In particular, if α and β are goals, then

- $\alpha \otimes \beta$ means "*First execute α, then execute β, and commit iff both α and β commit.*"
- $\alpha \mid \beta$ means "*Execute α and β concurrently, and commit iff both α and β commit.*"
- $\odot\alpha$ means "*Execute α in isolation, and commit iff α commits.*"
- $p \leftarrow \alpha$ means "*An execution of α is also an execution of p, where p commits iff α commits.*"

2.2 Elementary operations

In general, an elementary operation can be any activity that access a database. The precise set of elementary operations is somewhat arbitrary, and in this paper, four are provided. These operations are simple and they can be efficiently implemented. To represent these four operations, we use four types of expressions, to wit:

- p, "*commit iff atom p is in the database*";
- $empty.p$, "*commit iff the database contains no atoms of the form p*";
- $ins.p$, "*insert atom p into the database*; and
- $del.p$, "*delete atom p from the database, and commit.*"

2.3 Programs

Using the aforementioned four logical operators, a programmer combines elementary operations into complex processes, *i.e.*, programs. For instance,

$$del.p(a) \otimes del.p(b)$$

is a simple program that first deletes $p(a)$ from the database and then deletes $p(b)$. Likewise, the goal

$$del.p(a) \otimes del.p(b) \mid ins.q(a) \otimes ins.q(b)$$

is a program consisting of two sequential processes that execute concurrently. $\mathcal{CTR}^{\mathsf{H}}$ can be used to program database transactions. For example, the goal $\odot(p(b) \otimes del.p(b))$ represents a transaction program with a precondition, $p(b)$. This program first asks if $p(b)$ is in the database, and if so, it deletes $p(b)$. This program commits if $p(b)$ is in the database at the start of the execution, and aborts otherwise. Example 1 shows the definition of a subroutine for updating a bank account balance.

Example 1 (Updating a bank account balance) To update the balance of a client's account, "atomically" check the current account balance, then remove it, and then insert the new balance.

$$\begin{aligned} &updateAccBlc(Acc, Old, New) \leftarrow \\ &\qquad \odot(blnc(Acct, Old) \otimes del.blnc(Acct, Old) \otimes ins.blnc(Acct, New)) \end{aligned}$$

where $blnc(x, y)$ is a database predicate denoting the current balance, y, of bank account x. □

2.4 Inference system

Bonner introduced in [2] an inference system for executing $\mathcal{CTR}^{\mathsf{H}}$ programs using an SLD-style resolution mechanism. To facilitate the understanding of the approach presented here, we show this inference system in this section.

First, let us present the notion of *hot components*, *i.e.*, the formulas which are ready to execute in a goal. This notion is important for understanding the example shown in Table (1).

Let ϕ be a concurrent serial goal. Its *hot components*, denoted $hot(\phi)$, is defined recursively as follows:

- $hot(( )) = \{\}$, where $( )$ is the empty goal;

- $hot(b) = \{b\}$, if b is a atomic formula;
- $hot(\psi_1 \otimes \cdots \otimes \psi_k) = hot(\psi_1)$;
- $hot(\psi_1 \mid \psi_2 \mid \cdots \mid \psi_k) = hot(\psi_1) \cup \cdots \cup hot(\psi_k)$.
- $hot(\odot\psi) = \odot\psi$.

The inference system manipulates expressions called *sequents*, which have the form $\mathbf{P}, \mathbf{D}... \vdash (\exists)\, \phi$, where $\mathbf{P}$ is a program, $\mathbf{D}$ is any database state, and ϕ is a goal. The informal meaning of such a sequent is that, based on program $\mathbf{P}$, the formula $(\exists)\, \phi$ can be proved from state $\mathbf{D}$. Let the concurrent serial goal clause be the expression $\leftarrow G_0$, where G_0 is the sequent $\mathbf{P}, \mathbf{D}_1... \vdash (\exists)\, \phi$. A SLD-style refutation of $\leftarrow G_0$ is a sequence of goal clauses $\leftarrow G_0 \cdots \leftarrow G_n$ where G_n is the *empty clause*, *i.e.*, the sequent $\mathbf{P}, \mathbf{D}_n... \vdash ()$, where $\mathbf{D}_n$ is a database state, and $()$ denotes the empty formula. This sequent is an axiom of the inference system, and this axiom states that the empty formula is true on any database state. Each $\leftarrow G_{i+1}$ is obtained from $\leftarrow G_i$ by using the the axiom and inference rules introduced in [2].

Axiom: $\mathbf{P}, \mathbf{D}... \vdash ()$, for any state $\mathbf{D}$

Inference rules: In rules 1-3, σ is a substitution, ψ and ψ' are goals, and a is an element (randomly) selected from $hot(\psi)$.

1. *Applying rule definitions:* Suppose $b \leftarrow \beta$ is a rule in $\mathbf{P}$ whose variables have been renamed so that the rule shares no variables with ψ. If a and b unify with mgu σ, then

$$\frac{\mathbf{P}, \mathbf{D}... \vdash (\exists)\, \psi'\sigma}{\mathbf{P}, \mathbf{D}... \vdash (\exists)\, \psi}$$

   where ψ' is obtained from ψ by replacing a by β.
2. *Querying the database:* If $\mathcal{O}^d(\mathbf{D}_i) \models^c (\exists)a\sigma$, and $a\sigma$ and $\psi'\sigma$ share no variables, then

$$\frac{\mathbf{P}, \mathbf{D}... \vdash (\exists)\, \psi'\sigma}{\mathbf{P}, \mathbf{D}... \vdash (\exists)\, \psi}$$

   where $\mathcal{O}^d$ denotes a mapping from states to formulas that are true of the state (see [2]); ψ' is obtained from ψ by removing a.
3. *Executing elementary updates:* If $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c (\exists)a\sigma$, and $a\sigma$ and ψ' share no variables, then

$$\frac{\mathbf{P}, \mathbf{D}_2... \vdash (\exists)\, \psi'\sigma}{\mathbf{P}, \mathbf{D}_1... \vdash (\exists)\, \psi}$$

   where $\mathcal{O}^t$ is a mapping from pair of states to an atomic ground formula; ψ' is obtained from ψ by removing a.
4. *Executing atomic transactions:* If $\odot\alpha$ is a hot component in ψ, then

$$\frac{\mathbf{P}, \mathbf{D}... \vdash (\exists)\, (\alpha \otimes \psi')}{\mathbf{P}, \mathbf{D}_1... \vdash (\exists)\, \psi}$$

   where ψ' is obtained from ψ by removing a hot occurrence of $\odot\alpha$.

Each inference rule consists of two sequents, and has the following interpretation: if the upper sequent (G_{i+1}) can be inferred, then the lower sequent (G_i) can also be inferred.

To illustrate how the resolution mechanism operates, Table (1) shows a program and a deduction of a goal.

A program	A deduction for the goal s		
	Sequents	Inf. rule	Hot components
	$\mathbf{P}, \{\} \vdash s$	1	$\{s\}$
$s \leftarrow p \mid q$	$\mathbf{P}, \{\} \vdash p \,\vert\, q$	1	$\{p, q\}$
$p \leftarrow ins.r(a)$	$\mathbf{P}, \{\} \vdash p \,\vert\, ins.r(b)$	3	$\{p, ins.r(b)\}$
$q \leftarrow ins.r(b)$	$\mathbf{P}, \{r(b)\} \vdash p$	1	$\{p\}$
	$\mathbf{P}, \{r(b)\} \vdash ins.r(a)$	3	$\{ins.r(a)\}$
	$\mathbf{P}, \{r(a), r(b)\} \vdash ()$	$Axiom$	$\{\,\}$

(1)

3 Denotations for sequential languages using $\mathcal{CTR}^{\mathsf{H}}$

Denotational semantics of a language consists of three components:

- *syntax*: specified as a context free grammar;
- *semantic algebra*: defines the basic domains and the associated operations; meaning of a program is expressed in terms of these basic domains
- *valuation functions*: provide a mapping from abstract syntax trees of language constructs to values in the basic domains of the semantic algebra

Gupta presented in [5] a semantics specification framework, called Horn (logical) denotations, in which one uses DCG rules to specify the syntax of a source language, and Horn rules to specify the semantic algebras and valuation functions. For sequential languages, the most noticeable differences between our approach and that of [5] regard the framework used to specify semantic algebras and valuation functions.

Semantic algebras: Suppose we want to specify the semantics of a declarative, stateless programming language, called $\mathcal{M}$. We model the store using the elementary operations introduced in Section 2.2. To represent a store entity we use the database predicate $store(K, V)$, which associates a variable K with its current value V. For example, suppose the database includes the following predicates: $store(x_1, unbound)$, $store(x_2, 9)$. Then x_1 is unbound and x_2 is bound to the value 9. Moreover, we assume the following elementary operations are also defined: $empty.store(K, V)$, $ins.store(K, V)$ and $del.store(K, V)$.

Suppose the syntax of $\mathcal{M}$ specifies program blocks, *e.g.*, sequences of statements delimited by `begin`,`end`. Generally, this kind of control structure is used in programming languages for defining scopes. We model the environment using the database predicate $env(Id, X)$ to associate a variable identifier Id with a store entity X; and we use the elementary operations $empty.env(Id, X)$, $ins.env(Id, X)$, and $del.env(Id, X)$, which have meanings similar to the ones used to model the store.

Based on the above considerations, we present in Figure 1 the semantic algebra for $\mathcal{M}$. Because of space restrictions, we have omitted the definitions of helper predicates *newVar*, *getVal*, *getVar*, and *bindInStore*. The reader may find in [12] more information regarding the design of single-assignment stores. For the purposes of this paper, it should suffice to know that predicate *newVar* instantiates its parameter with a new variable; *getVal* gets the value bound to a given variable; *getVar* gets the variable to which a given variable identifier is mapped; and *bindInStore* updates the store to reflect bindings which are made between variables or between a variable and a value.

$$
\begin{aligned}
&initializeStoreEnv \leftarrow initEnv \otimes initStore\\
&initEnv \leftarrow empty.env(I, X)\\
&initEnv \leftarrow env(Id, X) \otimes del.env(Id, X) \otimes initEnv\\
&initStore \leftarrow empty.store(I, V)\\
&initStore \leftarrow store(Id, V) \otimes del.store(Id, V) \otimes initStore\\
&addVarToEnv(VarId) \leftarrow createVar(Var, unbound) \otimes\\
&\qquad rmBindingFromEnv(VarId) \otimes ins.env(VarId, Var)\\
&createVar(Var, Content) \leftarrow newVar(Var) \otimes ins.store(Var, Content)\\
&access(VarId, Val) \leftarrow env(VarId, Var) \otimes getVal(Var, Val)\\
&bind(VarId1, VarId2) \leftarrow getVar(VarId1, Var1) \otimes\\
&\qquad getVar(VarId2, Var2) \otimes bindInStore(Var1, Var2)\\
&valCreate(VarId, Val) \leftarrow createVar(NewVar, Val) \otimes\\
&\qquad getVar(VarId, Var) \otimes bindInStore(Var, NewVar)
\end{aligned}
$$

Fig. 1. Semantic algebra for $\mathcal{M}$

If the reader compares the approach we have used to obtain the semantic algebra presented in Figure 1 with the approach Gupta used in [5], you will notice we do not resort to lists to represent the store and the environment. Instead, both notions are defined in terms of database elementary operations, combined

with typical logic programming techniques. Obviously, one may use Prolog's database operators $assert$ and $retract$ to do the same thing. However, it is known such operators have no logical semantics.

Valuation predicates: In a denotational semantics framework, valuation functions (predicates) impart meaning to the syntactic structures of the language. Below we present the grammar rule (in BNF notation) and the valuation predicates for a binding statement in $\mathcal{M}$. Notice that a predicate associates a given abstract syntax tree to a procedure, *i.e.*, the tail of its respective Horn rule, which defines the semantics of the syntactic structure.

$$\langle binding \rangle \ \texttt{::=} \ \langle id \rangle = \langle Expr \rangle \left| \begin{array}{l} stmtvp(bind(V1, V2)) \leftarrow expr(V2, Val) \otimes \\ \qquad Val = unbound \otimes bind(V1, V2) \\ stmtvp(bind(V1, V2)) \leftarrow expr(V2, Val) \otimes \\ \qquad Val <> unbound \otimes valCreate(V1, Val) \end{array} \right.$$

Again, when compared to the approach presented in [5], the only novelty introduced in the specification presented in the valuation predicates above is a more clean formulation, which does not use extra parameters, thread through the Horn rules, to represent the store and the environment.

4 Denotations for concurrency

The logical denotations presented in Section 3 provide a framework to specify the semantics of stateless, sequential programming languages. This section extends that framework to address concurrency in programming languages.

4.1 The Data-driven concurrent language $\mathcal{M}_1$

In this section we show a logical denotation based on $\mathcal{CTR}^{\mathsf{H}}$ for a simple data-driven concurrent programming language called $\mathcal{M}_1$. Data-driven concurrency is found in declarative, "stateless" programming languages.

The variables of $\mathcal{M}_1$ support *dataflow execution*, *i.e.*, an operation waits until all arguments are bound before executing. We assume dataflow execution without exception handling. That is, if an argument is required for a computation but a value is never bound to it, then the execution will result in failure.

Concurrency is denoted in $\mathcal{M}_1$ via the concurrent block delimiters `cobegin`, `coend`. The BNF rule for a concurrent block is given below:

$$stmt \ \texttt{::=} \ \texttt{cobegin}\, stmtList\, \texttt{coend}$$

In programming languages with this sort of control structure, it is expected that each statement in the concurrent block runs using its own environment. Therefore, we cannot use a semantic algebra which defines a *global* environment as the one specified for $\mathcal{M}$ (see Figure 1). To obtain the semantic algebra for $\mathcal{M}_1$, we simply represent the environment as a list of mappings $env(I, X)$, thread through the valuation predicates (see valuation predicates below). We should also define predicates for inserting and removing bindings from the environment. Notice, however, that the store can still be a global data structure. Below we show the valuation predicates for a concurrent block:

$$stmtvp(coBlock(C), E) \leftarrow stmtvp(C, E)$$
$$stmtvp(coBlock(stmts(C1, C2)), E) \leftarrow stmtvp(C1, E) \mid stmtvp(coBlock(C2), E)$$

The intended meaning provided by the above valuation predicates is very clear: the statements within a concurrent block should run concurrently. Moreover, each statement runs using its own environment.

Example 2 (Dataflow synchronization) This example illustrates how concurrent instructions communicate through dataflow synchronization.

```
begin var B;
  cobegin
    B=1;
    if B then
      output B
    else
      output 0
  coend
end
```

□

To analyze Exampe 2, let us skip the initial computation steps and go directly to the situation when the binding (denoted by the '=' operator) and `if` statements are the two hot components of the execution.

As seen in Section 2.4, at each resolution step, any element of the hot components set can be selected for execution. If the `if` statement is selected, then the deduction of the `if` statement will fail, because the value of `B` is undetermined. $\mathcal{CTR}^{\mathsf{H}}$'s inference system would then backtrack and attempt another path by picking the statement `B=1` for execution. This amounts to waiting until the argument of the `if` is bound before executing.

4.2 The message passing concurrent language $\mathcal{M}_2$

In this section we present a logical denotation for a simple message passing concurrent language called $\mathcal{M}_2$. In this language, a program is a set of named communicating processes, *i.e.*, processes that send and receive messages amongst themselves. Example 3 illustrates one of such processes.

Example 3 (A message passing process) Definition of a process, sum, which receives two numbers from a process p, adds them up, and sends the result to process q.

```
process sum(in p, out q);
  beginp
    begin local a, b;
      receive a from p;
      receive b from p;
      send a+b to q
    end
  endp
```

□

Figure 2 shows the BNF grammar of $\mathcal{M}_2$.

Semantic algebra for $\mathcal{M}_2$: In Figure 2, the names listed in the heading of a process, under $\langle connections \rangle$, after the words in and out, define the set of processes with which it communicates. To represent that, we make slight changes to the definition of store and environment introduced in Section 3. To model the store, we also use the following elementary operations: $proc(I, C)$, $empty.proc(I, C)$, $ins.proc(I, C)$, and $del.proc(I, C)$, where $proc(I, C)$ is a database predicate denoting a store entity which associates a process identifier, I, with its *connections* $C = conn(In, Out)$, where In, out are the lists of processes from which I receives input and sends output, respectively. We also assume an environment includes the term $rproc(P)$, which denotes a running process P.

In this kind of programming language, processes communicate via communication channels, using $send$ and $receive$ statements such as the ones shown in Figure 2. To model channels we use the run-of-the-mill insert, delete, and empty primitives applied on the database predicate $channel(Q, Msg)$, which

$\langle program \rangle$ ::= $\langle processList \rangle$.
$\langle processList \rangle$ ::= $\langle process \rangle \mid \langle process \rangle ; \langle processList \rangle$
$\langle process \rangle$ ::= $process \langle id \rangle \langle connections \rangle ;$
$beginp$
$\langle stmtList \rangle$
$endp$
$\langle connections \rangle$ ::= $\langle empty \rangle \mid (\langle inputs \rangle \langle outputs \rangle)$
$\langle inputs \rangle$ ::= $\langle empty \rangle \mid in \langle idList \rangle$
$\langle outputs \rangle$ ::= $\langle empty \rangle \mid out \langle idList \rangle$
$\langle idList \rangle$ ::= $\langle id \rangle \mid \langle id \rangle, \langle idList \rangle$
$\langle stmtList \rangle$ ::= $\langle statement \rangle \mid \langle statement \rangle ; \langle stmtList \rangle$
$\langle statment \rangle$ ::= $\langle send \rangle \mid \langle receive \rangle \mid \langle assignment \rangle$
$\langle send \rangle$ ::= $send \langle expression \rangle to \langle procRef \rangle \mid send to \langle procRef \rangle$
$\langle receive \rangle$ ::= $receive \langle id \rangle from \langle procRef \rangle \mid receive from \langle procRef \rangle$
$\langle assignment \rangle$ ::= $\langle id \rangle := \langle expression \rangle$

Fig. 2. DCG rules for $\mathcal{M}_2$

means: "the channel Q contains message Msq." In the semantic algebra of $\mathcal{M}_2$ we implement $send$ and $receive$ in terms of predicates $sendC$ and $receiveC$.

To accommodate the above and obtain the semantic algebra for $\mathcal{M}_2$, we modify the semantic algebra presented in Figure 1 as follows:

$$initializeStoreEnv([\,]) \leftarrow empty.store(I,V) \otimes empty.rProc(P)$$
$$initializeStoreEnv(E) \leftarrow store(Id,Val) \otimes del.store(Id,Val) \otimes initializeStoreEnv(E)$$
$$initializeStoreEnv(E) \leftarrow rProc(P) \otimes del.rProc(P) \otimes initializeStoreEnv(E)$$
$$addProcToEnv(ProcId,E,NewE) \leftarrow NewE = [rproc(ProcId)|E]$$
$$sendC(Q,Msg) \leftarrow ins.channel(Q,Msg)$$
$$receiveC(Q,Msg) \leftarrow \odot(channel(Q,Msg) \otimes del.channel(Q,Msg))$$

Valuation predicates: Figure 3 presents the valuation predicates for $\mathcal{M}_2$. Notice that $\mathcal{M}_2$ does not include procedure definition or procedure call. Instead, it includes the $send$ and $receive$ statements to provide process input and output operations. Notice that predicates *receivevp* and *sendvp*, along with $\mathcal{CTR}$'s resolution mechanism, specify the intuitive semantics for statements $receive$ and $send$, which is: $receive$ waits for a value to be sent to it by the process referred to in the statement. Then the value is assigned to the variable identified in the statement. If no variable is specified, then $receive$ simply waits until the process referred to sends it a value and then discards it. This allows processes to synchronize their activities without updating variables. $send$ simply sends a message to a process.

5 Discussion

An approach which provides a denotational semantics based on a Horn fragment of Concurrent Transaction Logic ($\mathcal{CTR}$) was presented.

We have showed that the semantics of a class of concurrent programming languages can be fully specified in a Horn-based logic framework. The proposed framework seamlessly integrates with the notions of environment, store, store updates, dataflow in declarative languages, data-driven concurrency, message passing concurrency, and synchronization between concurrent processes. It has become apparent from our approach that the presence of concurrency does not affect the description of other constructs in the specification. This contrasts sharply with conventional denotational descriptions, where domains of higher-order functions used to model concurrency and nondeterminism are radically different from those used in the specification of sequential languages.

$$
\begin{array}{l}
progEval(P) \leftarrow initializeStoreEnv(E) \otimes processList(X, E).\\
processList(P, E) \leftarrow process(P, E).\\
processList(par(P1, P2), E) \leftarrow process(P1, E) \,|\, process(P2, E).\\
process(proc(Id, Conn, nodecl, Bdy), E) \leftarrow\\
\quad addProcToEnv(Id, E, NE) \otimes ins.proc(Id, Conn) \otimes\\
\quad stmtvp(Bdy, NE).\\
receivevp(receive(noVal, Proc), E) \leftarrow\\
\quad \odot(checkHeader(in, Proc, E) \otimes receiveC(Proc, \_)).\\
receivevp(receive(Proc, VarId), E) \leftarrow VarId <> noVal \otimes\\
\quad \odot(checkHeader(in, Proc, E) \otimes receiveC(Proc, Val) \otimes\\
\quad valCreate(VarId, Val, E)).\\
sendvp(send(Proc, noVal), E) \leftarrow\\
\quad \odot(checkHeader(out, Proc, E) \otimes sendC(Proc, unit)).\\
sendvp(send(Proc, Expr), E) \leftarrow Expr <> noVal \otimes\\
\quad \odot(expr(Expr, Val, E) \otimes checkHeader(out, Proc) \otimes\\
\quad sendC(Proc, Val)).\\
checkHeader(in, Proc, E) \leftarrow\\
\quad procInScope(E, PId) \otimes proc(Proc, conn(\_, Out)) \otimes\\
\quad checkProcId(PId, Out).\\
checkHeader(out, Proc) \leftarrow\\
\quad procInScope(E, PId) \otimes proc(Proc, conn(In, \_)) \otimes\\
\quad checkProcId(PId, In).\\
procInScope([proc(PId)|\_], PId).\\
procInScope([X|R], PId) \leftarrow X <> rproc(\_) \otimes procInScope(R, PId).\\
checkProcId(Proc, identif(Proc)).\\
checkProcId(Proc, vList(iddentif(Proc), R)).\\
checkProcId(Proc, vList(\_, R)) \leftarrow checkProcId(Proc, R).
\end{array}
$$

Fig. 3. Valuation predicates for $\mathcal{M}_2$

For future work we intend to further experiment with this framework to test the adequacy of the communication paradigm of $\mathcal{CTR}$ for description of other flavours of concurrency, *e.g.*, shared-state and soft real-time. We conjecture shared-state concurrency should naturally and easily follow from our approach. To address this type of concurrency, one would need to first define the notion of a mutable store based on the notion of single-assignment store presented here. Then define a suitable semantic algebra that operates on this store. After that, use logic programming techniques in $\mathcal{CTR}^{\mathsf{H}}$ to implement valuation predicates which specify coarse-grained atomic actions, *e.g.*, locks, monitors, or transactions.

Bonner and Kifer introduced in [3] how to reason about state changes in Transaction Logic [1]. There they provide a declarative semantics for axioms specifying the preconditions and effects of state updates. Drawing on [3], Santos presented in [13] a computational method to reason about robot actions specified as Transaction Logic formulas. Based on these works, we intend to explore methods for doing program verification and reasoning in our framework.

References

1. A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, University of Toronto, November 1995. `http://www.cs.toronto.edu/~bonner/transaction-logic.html`.
2. A.J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *Joint Intl. Conference and Symposium on Logic Programming*, pages 142–156, Bonn, Germany, September 1996. MIT Press.
3. A.J. Bonner and M. Kifer. Results on reasoning about action in transaction logic. In B. Freitag, H. Decker, A. Voronkov, and M. Kifer, editors, *Transactions and Change in Deductive Databases*, Lecture Notes in Artificial Intelligence. Springer Verlag, 1998.
4. Kyung-Goo Doh and Peter D. Mosses. Composing programming languages by combining action-semantics modules. *Sci. Comput. Program.*, 47(1):3–36, 2003.

5. G. Gupta. Horn logic denotations and their applications. In *Workshop on Current trends and Future Directions in Logic Programming Research*, 1998.
6. Gopal Gupta. A language-centric approach to software engineering: Domain specific languages meet software components. In *Proceedings of the CoLogNet Workshop Series on Component-based Software Development and Implementation Technology for Computational Logic Systems*, Madri, Spain, 2002.
7. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
8. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, September 1992.
9. Chris D. Moss. How to define a language using prolog. In *Acm Symposoium on Lisp and Functional Programming*, pages 67–73, 1982.
10. P. D. Mosses. *Action semantics*. Cambridge University Press, 1992.
11. Peter D. Mosses. On the action semantics of concurrent programming languages. In *Proceedings of the REX Workshop on Semantics: Foundations and Applications*, pages 398–424, London, UK, 1993. Springer-Verlag.
12. P. Van Roy and S. Haridi. *Concepts, techniques, and models of computer programming*. MIT Press, 2004.
13. Marcus V. Santos. Specifying and reasoning about actions in open worlds using transaction logic. In *ECAI: Workshop on Cognitive Robotics*, August 2000.
14. E. Shapiro. A family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3), 1989.
15. Ken Slonneger. Implementing denotational semantics with logic programming. In *CSC '92: Proceedings of the 1992 ACM annual conference on Communications*, pages 337–344, New York, NY, USA, 1992. ACM Press.
16. Qian Wang and Gopal Gupta. A logic programming-based environment for integrating architecture description languages. In *Proceedings of the 3rd International Workshop on Multiparadigm Constraint Programming Languages MultiCPL'04*, Saint-Malo, France, 2004.