# A Horn Logic Denotational Framework for Specification, Implementation, and Verification of Domain Specific Languages

**Gopal Gupta and Enrico Pontelli**

Laboratory for Logic, Databases, and Advanced Programming
Department of Computer Science
New Mexico State University
Box 30001/CS, Las Cruces
New Mexico, USA 88003
http://www.cs.nmsu.edu/lldap

**Abstract**

We present a *Horn logic denotational semantics* framework for specification, efficient implementation, and automatic verification of domain specific languages (DSLs). Our framework is based on using Horn logic (or pure Prolog), and eventually *constraints*, to specify denotational semantics of domain specific languages. Both the syntax as well as the semantic specification of the DSL in question are directly executable in our framework: the specification itself serves as an interpreter for the DSL. More efficient implementations of this DSL—a compiler—can be automatically derived via partial evaluation. Additionally, the executable specification can be used for automatic or semi-automatic verification of programs written in the DSL. The ability to verify DSL programs is a distinct advantage of our approach. In this paper we give a general outline of our approach, and illustrate it with practical examples.

## 1 Introduction

Domain specific languages are languages that are defined for handling problems that arise in a specific domain. Any software system that interacts with the outside world defines a domain specific language. This is because the input language that the users use to interact with this software can be thought of as a domain specific language. For instance, if we consider a file editor, then the command language of the file editor constitutes a domain specific language. Indeed this language-centric view can be quite advantageous to support the software development process. This is because the *semantic specification of the input language of a software system is also a specification of that software system*—we assume the semantic specification also includes the syntax specification of the input language. *If the semantic specification of the input language is executable, then we obtain an executable specification of the software system.* In this paper we use the preceding observations to design a language semantics based framework for specifying, (efficiently) implementing, and verifying (rather model checking) DSLs.

An obvious candidate framework for specifying the semantics of a domain specific language is denotational semantics [30]. Denotational semantics has three components: (i) syntax, which is typically specified using a BNF, (ii) semantic algebras, or value spaces, in terms of which the meaning is given, and, (iii) valuation functions, which map abstract syntax to semantic algebras. In traditional denotational definitions, syntax is specified using BNF, and the semantic algebra and valuation functions using $\lambda$-calculus. There are various problems with this traditional approach: (i) the syntax is not directly executable, i.e., it does not immediately yield a parser, (ii) the semantic specification cannot be easily used for automatic verification or model checking. Additionally, the use of separate notations for the different components of the semantics implies the need of adopting different tools, further complicating the process of converting the specification into an executable tool. Verification should be a major use of any semantics, however, this has not happened for denotational semantics; its use is mostly limited to studying language features, and (manually) proving properties of language constructs (e.g., by use of *fixpoint induction*). In [32] Schmidt makes a similar observation, and laments the lack of practical impact of denotational semantics, particularly in automatic verification and debugging of programs. Elsewhere we have argued that a major reason for this lack of use of denotational semantics is the very rich notation—the $\lambda$-calculus—that is traditionally used for specifying denotational semantics.[1] In this paper, we show how the switch to Horn logic for expressing denotational

[1] Contrary to intuition, notation has a great impact in ease of use. Two very significant examples are the use of high-level language *vs* assembly language in programming, and the use of the decimal arithmetic notation *vs* the Roman numerals.

semantics facilitates the specification, implementation, and automatic verification of DSLs.

Traditionally, operational semantics is supposed to be meant for implementors, denotational semantics for language designers, and axiomatic semantics for programmers. Thus, each major type of semantics not only has a different target audience, they all use different types of notation as well. One major reason that has impeded practical uses of semantics, in our opinion, is this use of different semantics and different notations for different uses. The switch to Horn logic (and eventually constraints) for expressing denotational semantics creates a *uniform* description framework and brings flavors of both operational semantics as well as axiomatic semantics in the denotational semantic definition. In switching the notation, we may sacrifice some of the declarative purity of the traditional denotational definition, but the number of applications that become possible are well-worth this change [11]. As a matter of fact, one can argue that not a whole lot of declarative purity is lost, and analogs of techniques such as fixpoint induction can still be used.

A Horn logic denotational specification for a language yields a parser and an interpreter automatically. Partial evaluation of the interpreter w.r.t. a program yields compiled code. The interpreter can be used in conjunction with preconditions and postconditions to verify/model-check DSL programs. Thus, the operational semantics flavor allows efficient implementation to be derived, while the axiomatic semantics flavor permits automatic verification.

We illustrate the Horn Logic denotational framework through two example DSLs: the command language of a file-editor and a language for specifying real-time systems called UPPAAL [21, 2]. The rest of the paper is organized as follows: Section 2 introduces the concept of Horn Logic Denotations; Section 3 presents a software engineering perspective of DSL and Horn Logic Denotations. Section 4 presents the derivation of a DSL for file editing. Section 5 discusses the issues of verification of properties of DSL and presents the derivation of the UPPAAL language for describing real-time systems using Horn Logic Denotations. Section 6 presents the conclusions of this work. The main contribution of our work is to present a framework in which software development is seen as the activity of defining a DSL, and in which this DSL can be easily specified and implemented and, most significantly, verified as well.

# 2 Horn Logic Denotations

Denotational semantics [30, 31, 16] of a language has three components:

- *syntax*: specified as a context free grammar
- *semantic algebras*: these are the basic domains along with associated operations; the meaning of a program is expressed in terms of these basic domains
- *valuation functions*: these are mappings from patterns of parse trees to values in the domains in the semantic algebra

Traditional denotational definitions express syntax in the BNF format, and the semantic algebras and valuation functions in $\lambda$-calculus. However, a disadvantage of this approach is that while the semantic algebra and the valuation functions could be easily made executable, syntax checking and generation of parse trees cannot. A parser has to be explicitly written or generated using a parser-generator. The parse trees generated by the parser will then be processed by the valuation functions to produce the program's denotation in terms of the semantic algebras. These parser and valuation functions constitute an interpreter for the language being defined. An interpreter for a language can be thought of as a specification of its operational semantics, however, using traditional notation (BNF and $\lambda$-calculus) it has to be obtained in a complex way.

In contrast, if we use logic programming—with its formal basis in Horn logic, a subset of first-order logic—an interpreter will be straightforwardly obtained from the denotational specification. The additional advantage that logic programming possesses, among others, is that even syntax can be expressed in it at a very high level—and uniformly in the same language used for the rest of the specification—and *a parser for the language is immediately obtained from the syntax specification*. Moreover, the generation of parse trees requires a trivial extension to the syntax specification. The parsing and parse tree generation facility of logic programming is described in almost every logic programming textbook, under the heading *Definite Clause Grammars (DCGs)*. The semantic algebras and valuation functions are also expressed in Horn logic quite easily, since Horn logic allows to define relations, which in turn subsume functions. The semantic algebra and valuation functions are executable, and can be used to obtain executable program denotations. A very significant consequence of this is that the fixpoint of a program's denotation can be computed bottom-up (assuming that it is finite or finitely expressible), and such fixpoint can then be used for automatic verification (or model-checking). This implies that verification can also be conveniently done in the framework of Horn Logic.

Thus, given a language, both its syntax and semantics can be directly and uniformly specified in logic programming. This specification is executable using any standard logic programming system. What is noteworthy is that different operational models will be obtained both for syntax checking and semantic evaluation by employing different execution strategies during logic program execution. For example, in the syntax phase, if left-to-right, Prolog style, execution rule is used, then *recursive descent parsing* is obtained. On the contrary, if a *tabling-based* [3] execution strategy is used then *chart parsing* is obtained, etc. Likewise, by using different evaluation rules for evaluating the semantic functions, strict evaluation, non-strict evaluation, etc. can be obtained. By using bottom-up or tabled evaluation, the fixpoint of a program's denotation can be computed, which can be used for verification and structured debugging of the program. Compiled code for a DSL program can also be obtained via partial evaluation.

This approach of using Horn logic for specifying syntax and semantics of a language and then using the specification for obtaining efficient implementation and verification can be applied to any language; however, we feel that it is impractical for general purpose langauges that are fairly large be impractical for in using it for the study of domain specific languages, and its use in software specification.

# 3 Software Engineering and Domain Specific Languages

We take a language-centric view of the software development process. A language-centric view allows one to apply language-semantics based techniques for specification, implementation and verification of software system. In particular, we can use the Horn logic denotational approach for specification, efficient implementation, and verification of the software system.

Any software system can be understood in terms of how it interacts with the outside world. Thus, to understand a software system, one has to understand its *input language*. The input language is of course nothing but a domain specific language. If we have a denotational specification of this DSL, and if this specification (both the syntax and semantics) happens to be executable, then this denotational specification is *also an executable specification of the software system*. In other words, an interpreter for the DSL of the software system is an implementation of the software system. If this executable denotational specification is written in the proper notation, then it can also be used for proving properties of the DSL (i.e., the software system) as well as the programs written in the DSL.

The Horn logic denotational semantics indeed makes all of the above possible. The syntax specification immediately yields a parser, with the help of the Definite Clause Grammar facility of logic programming systems. The semantic specification yields a back-end of the interpreter. Together with the syntax specification, the semantic specification yields a complete interpreter. Given a program, its denotation can be obtained with respect to the interpreter. This denotation can be run to execute the program. Additionally, appropriate structured queries can be posed to the program's denotation and used for checking properties of the DSL as well as the programs. Essentially, the bottom-up execution [23] of the denotation yields the fixpoint of the program. The fixpoint of the program can be used for model checking.

Denotational semantics expressed in a Horn Logic notation is executable, but so is denotational semantics expressed in the $\lambda$-calculus notation. However, semantics expressed via Horn Logic allow for fixpoints of programs to be computed much more intuitively, simply, and efficiently than, we believe, in the case of the $\lambda$-calculus. There is a whole body of literature and implemented systems (e.g., tabling based systems such as XSB [3]) for computing fixpoints of logic programs, because of their applicability to deductive databases [28, 7, 23]. Due to this reason, semantic-based verification (and program debugging) can be much more easily performed in the Horn Logic denotational framework than using the $\lambda$-calculus based denotational framework, as is shown later. This becomes even more prominent when we generalize Horn Logic to Constraint Logic Programming—i.e., by adding additional domains (e.g., Real numbers, Sets) and predefined predicates over such domains. This generalization makes specification and verification of very complex systems, e.g., domain specific languages for real-time systems, considerably easier [13].

# 4 A Domain Specific Language for File-Editor Commands

Consider the input language (a DSL) of a file-editor. We show how the logical denotational semantics of this DSL yields an executable specification of the file editor. We also show how DSL programs can be compiled for efficiency. Later, we show how we can verify certain properties that a file-editor should satisfy (e.g., modifying one file doesn't change other files).

Consider a simple file editor which supports the following commands: edit I (open the file whose name is in identifier I), newfile (create an empty file), forward (move file pointer to next record), backward (move file pointer to previous record), insert(R) (insert record whose value is in identifier R), delete (delete the current record), and quit (quit the editor, saving the file in the file system). Let us provide the logic denotational semantics for this language. The syntax of the input command language is shown in the BNF below:

```
Program    ::= edit Id cr Statements
Statement  ::= Command cr Statements | quit
Command    ::= newfile | moveforward | movebackward | insert Record | delete
```

Note that `cr` stands for a carriage return, inserted between each editor command. To keep the example simple we assume that the records consists of simply integers. This BNF can be straightforwardly expressed as a DCG in Figure 1. There is a one-to-one correspondence between the rules of the BNF and rules in the DCG. An extra argument has been added to the DCG in which the parse tree is recursively synthesized. The DCG specification, when loaded in a Prolog system, automatically produces a parser. We next give the semantic algebras (Figure 2) for each of the domains involved: the file store (represented as an association list of file names and their contents) and an open file (represented as a pair of lists; the file pointer is assumed to be currently on the first record of the second list). The semantic algebra essentially defines the basic operations used by the semantic valuation functions for giving meanings of programs.

```
program(session(I,S)) --> [edit], id(I),
      [cr], sequence(S).
sequence(seq(quit)) --> [quit].
sequence(seq(C,S)) -->
      command(C), [cr], sequence(S).
command(command(newfile)) --> [newfile].
command(command(forward)) -->
[moveforward].
command(command(backward)) -->
[moveback].
command(command(insert(R))) -->
      [insert],record(R).
command(command(delete)) --> [delete].
id(identifier(X)) --> atom(X)
record(rec(N)) --> integer(N)
```

**Figure 1:** DCG for File Editor Language

The semantic valuation predicates that give the meaning of each construct in the language are given next (Figure 3). These semantic functions are mappings from parse-tree patterns and a global state (the file system) to domains (file system, open files) that are used to describe meanings of programs. The above specification gives both the declarative and operational semantics of the editor.

Using a logic programming system, the above specification can serve as an interpreter for the command language of the editor, and hence serves as an implementation of the editor. Thus, this is an *executable specification* of an editor. Although editors are interactive programs, for simplicity, we assume that the commands are given in batches (interactive programs can also be handled by modeling the "unknown" commands through Prolog's unbound variables: we omit the discussion to keep the presentation simple). Thus, if the editor is invoked and a sequence of commands issued, starting with an unspecified file system (modeled as the unbound variable `Fin`), then the resulting file system (`Fout`) after executing all the editor commands will be given by the result of the query:

```
?- Comms = [edit,a,cr,newfile,cr,insert,1,cr,insert,2,cr,delete,
   cr,moveback,cr,insert,4,cr,insert,5,cr,delete,cr,quit]),
   program(Tree,Comms,[]),    %produce parse tree
   prog_val(Tree,Fin,Fout).  %execute commands
```

The final resulting file-system will be:

```
Fout = [(a,[rec(1),rec(4)])| _B ].
```

The output shows that the final file system contains the file `a` that contains 2 records, and the previously unknown input file system (represented by Prolog's anonymous variable `_B`, aliased to `Fin`). The key thing to note is that in our logical denotational framework, a specification is very easy to write as well as easy to

```
%Define Access and Update Operations
access(Id,[(I,File)|_],File).
access(Id,[(I,File)|Rest],File1) :-
        (I = Id -> File1 = File;
                access(Id,Rest,File1)).
update(Id,File,[],[(Id,File)]).
update(Id,File,[(Id,_)|T],[(Id,File)|T]).
update(Id,File,[(I1,F1)|T],[(I1,F2)|NT]) :-
       (Id=I1 --> F2 = File, NT = T;
            F2 = F1, update(Id,File,T,NT)).
```

```
%Operations on Open File representation
newfile(([],[])).
copyin(File,([],File)).
copyout((First,Second),File):-
         reverse(First,RevFirst),
         append(RevFirst,Second,File).
forwards((First,[X|Scnd]),([X|First],Scnd)).
forwards((First,[]),(First,[])).
backwards(([X|First],Scnd),(First,[X|Scnd])).
backwards(([],Scnd),([],Scnd)).
insert(A,(First,[]),(First,[A])).
insert(A,(First,[X|Y]),([X|First],[A|Y])).
delete((First,[_|Y]),(First,Y)).
delete((First,[]),(First,[])).
at_first_record(([],_)).
at_last_record((_,[])).
isempty(([],[])).
```

Figure 2: Semantic Algebras for File Editor

modify. This is because of the declarative nature of the logic programming formalism used and its basis in denotational semantics.

Given the executable implementation of the file-editor, and a program in its command language, we can partially evaluate it to obtain a more efficient implementation of the program. The result of partially evaluating the file-editor specification w.r.t. the previous command-language program is shown in Figure 4. Partial evaluation translates the editor command language program to a sequence of instructions that call operations defined in the semantic algebra. This sequence of instructions looks a lot like "compiled" code. More efficient implementations of the editor can be obtained by implementing these semantic algebra operations in an efficient way, e.g., using a more efficient language like C or C++, instead of using logic programming. Compilation may not make much sense in case of a file-editor command language, however, there are domain specific languages that have been designed for processing special types of file. For example, the DSL MidiTrans [17] has been designed to manipulate digital music files expressed in the MIDI format. The MidiTrans language can be thought of as an editor-command language, and MidiTrans programs are sequences of commands applied on a MIDI file. In this case compilation (via partial evaluation) is important, in order to achieve efficient execution of MidiTrans programs. Derivation of MidiTrans using Horn denotational descriptions is currently in progress [24].

# 5 Program Denotation and Verification

Axiomatic semantics is perhaps the most well-researched technique for verifying properties of programs. In Axiomatic Semantics [18] preconditions and postconditions are specified to express conditions under which a program is correct. The notation $(P)C(Q)$ states that if the predicate $P$ is correct before execution of command $C$, then $Q$ must be correct afterwards. $P$ and $Q$ are typically expressed in a well-defined form of logic. In this context we will explore the use of Horn logic for this purpose. The post-conditions of a program are theorems with respect to the denotation of that program and the program's pre-conditions [30, 31]. Given that the program's denotation is expressed in Horn logic, the pre-conditions can be uniformly incorporated into this denotation. The post-conditions can then be executed as queries w.r.t this extended program denotation, effectively checking if they are satisfied or not. In effect, symbolic *model checkers* [4] can be specified and generated automatically. By generalizing Horn logic to constraint logic programming, real-time systems can also be specified and implemented [13] and parallelizing compilers obtained [14].

One way to prove correctness is to show that given the set of all possible state-configurations, $S$, that can exist at the beginning of the command $C$, if $P$ holds for a state-configuration $s \in S$, then $Q$ holds for the state-configuration that results after executing the command $C$ in $s$. If the denotation is a logic program, then it is possible to generate all possible state-configurations. However, the number of such state-configurations may be infinite. In such a case, the precondition $P$ can be specified in such a way

```
prog_val(session(identifier(I),S),FSIn,FSOut) :-
  access(I,FSIn,File), copyin(File,OpenFile),
  seq_val(S,OpenFile,NewOpenFile),
  copyout(NewOpenFile,OutFile),
  update(I,OutFile,FSIn,FSOut).
seq_val(seq(quit),InFile,InFile).
seq_val(seq(C,S),InFile,OutFile) :-
  comm_val(C,InFile,NewFile),
  seq_val(S,NewFile,OutFile).

comm_val(command(newfile),_,OutFile) :-
  newfile(OutFile).
comm_val(command(moveforward),InFile,OutFile) :-
  (isempty(InFile) -> OutFile = InFile;
    (at_last_record(InFile) -> OutFile=InFile;
     forwards(InFile,OutFile))).
comm_val(command(moveback),InFile,OutFile) :-
  (isempty(InFile) -> InFile = OutFile;
    (at_first_record(InFile) ->
     InFile = OutFile; backwards(InFile,OutFile))).
comm_val(command(insert(R)),InFile,OutFile) :-
  record_val(R,RV), insert(RV,InFile,OutFile).
comm_val(command(delete),InFile,OutFile) :-
  (isempty(InFile) ->
   InFile = OutFile; delete(InFile,OutFile)).
record_val(R,R).
```

Figure 3: Valuation Predicates

```
access(a, Fin, C),
copyin(C, _),
newfile(D),
insert(rec(1), D, E),
insert(rec(2), E, F),
(   isempty(F) -> G=F
    ;   delete(F, G)),
(   isempty(G) -> H=G
    ;   at_first_record(G) ->
         H=G
      ;   backwards(G, H)),
insert(rec(4), H, I),
insert(rec(5), I, J),
(   isempty(J) -> K=J
  ;   delete(J, K)),
copyout(K, L),
update(a,L,Fin,Fout).
```

Figure 4: Compiled code

that it acts as a *finite* generator of all relevant state-configurations. A model checker is thus obtained from this specification. This model checker can be seen as a debugging aid, since a user can obtain a program's denotation, add preconditions to it and then pose queries to verify the properties that she thinks should hold.

Consider the specification of the file editor. Under the assumption that the file system is finite and that the pool of possible records is also finite, we can verify, for instance, that every editing session consisting of an insertion followed by a deletion leaves the original file unchanged. Since the name space of file-names and record-names is infinite, we will use a precondition to restrict their size. The query for verifying this property, along with the appropriate pre-condition to assure finiteness is shown below. The precondition essentially restricts file names to either `a`, `b` or `c`, and the record names to `1`, `2` or `3` (we will show below how to make this query more general). The `member` predicate is the standard predicate for checking membership; when its first argument is unbound and the second argument is bound to a list, it acts as a generator of values.

```
?- member(X, [a,b,c]), member(Y, [1,2,3])      %pre-condition
   program(A,[edit,X,cr,insert,Y,cr,delete,cr,quit],[]),
   prog_val(A,F,G),
   F ≠ G.      %negated post-condition
```

The above query corresponds to verifying whether there exist values for `X` (file name) and `Y` (record value) such that inserting and deleting `Y` in `X` leads to a resulting file system different from the one we started from. This query should fail, if indeed the result of one insertion and one deletion leaves the file system unchanged. The backtracking mechanism of logic programming goes through all possible values for variables `X` and `Y` (finiteness is hence important), and finds that in every case `F = G` holds, and thus the whole query fails because the final call asserts that `F` $\neq$ `G`. In most practical cases, the restriction of having a finite generator can be readily dismissed as long as the fixpoint of the considered computation can be analyzed in a finite amount of time. The above example could be encoded more simply as:

```
?- program(A,   [edit,X,cr,insert,Y,cr,delete,cr,quit],[]),
                prog_val(A,F,G), F ≠ G.
```

If executed on a logic programming system which uses a fair search strategy (e.g., we tested it on the XSB system [28], which uses tabling), the query will produce a failed answer, thus verifying the desired property. More complex properties can be semi-automatically verified using this semantic definition. Let us study the following property: if an editing session is open for a certain file (let us say file a), then no other file is affected. Proving this property is not straightforward—as it requires being able to infer independence of arguments. This property can be easily tested using a combination of transformation and analysis techniques. We start by partially evaluating the editor specification with respect to the query:

```
?- program(X,[edit,a,cr|Rest],[]),
   prog_val(X,[(a,AIn),(b,BIn),(c,CIn)],[(a,AOut),(b,BOut),(c,COut)]).
```

(we are assuming that the file system contains only 3 files; this can be easily generalized). The entry point of the partially evaluated program is the following:

```
entry([edit,a,cr|A], [(a,B),(b,C),(c,D)], [(a,E),(b,F),(c,G)]) :-
        'entry.edit1'(A, B, C, D, E, F, G).

'entry.edit1'(A, B, C, D, E, F, G) :-
        sequence1(A, H),
        seq_val1(H, B, I),
        I=(J,K),
        reverse1(J, L),
        C=F,              %%% **
        D=G,              %%% **
        append3(L, K, E).
```

We have annotated the two unifications (with '**') to show how the partially evaluated program shows that the output files for b and c are indeed identical to the input ones—i.e., those two files are not affected by the editing session. Using a constraint-based execution, which allows to solve equality and inequality constraints, a query like

```
?- (C ≠ F; D ≠ G),
     entry([edit,a,cr|Rest],[(a,B),(b,C),(c,D)],[(a,E),(b,F),(c,G)]).
```

terminates with a failure. Thus, there is no computation which edits file a and modifies also file b or c. This property could be also determined without executing the last query but by simply performing *static analysis*—for *Independence* detection—on it [9].

## 5.1 DSLs: Verification with Constraints

When we generalize Horn Logic to Constraint Logic Programming [22] more interesting applications become possible. For example, domain specific languages for realizing real-time systems can be modeled and verified/model-checked using our approach. We illustrate this through an example. Consider the domain specific language UPPAAL [21, 2], designed by researchers at Uppsala University and Aalborg University, for specifying real-time systems. We specify the syntax and semantics of this DSL using our Horn logic denotational approach, and show how program denotations can be used for automatic verification.

UPPAAL is a domain specific language designed for specifying and verifying concurrent real-time systems. The concurrent real-time system is modeled as a collection of *timed automata* [1]. A timed automata is a finite state automata augmented with timers. These timers may be reset to zero on state transitions, and additionally, state transitions may also be conditioned on a given timer satisfying certain properties (e.g., the transition can be made only if a certain amount of time has elapsed on a particular timer). The UPPAAL language provides constructs for defining these automata, along with timing constraints that the timed automata imposes.

A real-time system is essentially a recognizer of a sequence of timed-event. A sequence of timed-event is correct if the individual events occur in a certain order (syntactic correctness) and the time at which these events occur satisfy the time-constraints laid out by the real-time system specification (semantic correctness). The syntax and semantics of a real-time system can be specified using constraint logic denotations [13]—the key insight is that the specification of a real-time system is a semantic specification of its corresponding timed-language [13]. Time constraints can be modeled as constraints over real numbers [22]. The semantic algebra models the state, which consists of the global time (wall-clock time), and the valuation predicates are maps from sequences of events to the global time. This constraint logic denotational specification is executable and can be used for verifying interesting properties of real-time systems, e.g., *safety* (for instance, if we design a real-time system for a railroad gate controller, we want to make sure that at the time a train is at the gate, the gate can never be open), and *bounded liveness* (for instance, the railroad gate is not closed forever). In the real-time systems modeled, we do not exactly know when an event is actually going to occur, all we know is the relationship between the time at which different events took place. Thus, the exact time at which each event took place cannot be computed from the constraint logic denotation. However, the constraints laid out in the denotation together with constraints that the safety property enforces can be solved to check for their consistency. Essentially, the constraints laid out by the real-time system should entail the constraints laid out by the properties to be verified, if the property indeed holds.

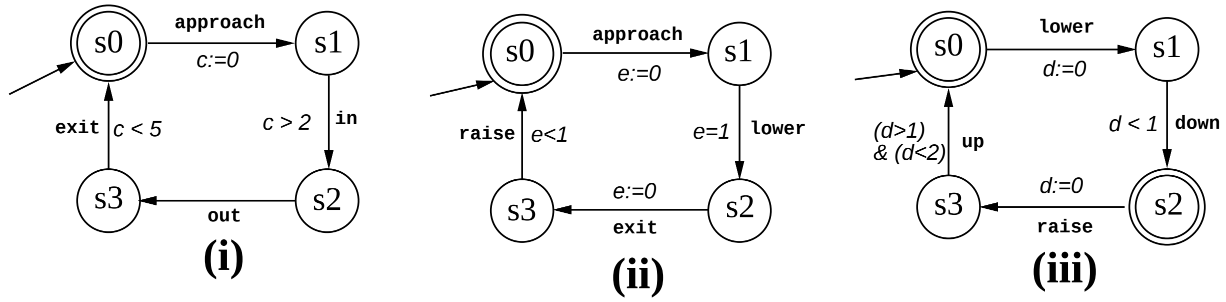


Figure 5: Train, Controller, and Gate automata

The real-time system that we wish to verify can be specified in the UPPAAL domain specific language. A real-time system that is to be modeled as a timed automata is expressed as an UPPAAL program. If we give the semantics of the UPPAAL language using constraint denotations, then an executable specification of a timed-automata system is easily obtained. This executable specification can be run on a constraint logic programming systems, such as $CLP(\mathcal{R})$ [22], and used for verifying properties of the real-time system. The BNF of the UPPAAL language can be found in [21, 2]. The DCG of the UPPAAL language is given in Appendix A. The Horn logic semantics of UPPAAL program is given as predicates that map the components of the abstract syntax of the program to a list of tuples of the form $\langle \alpha, \tau, States \rangle$, where $\alpha$ is a particular automata transition, $\tau$ is the time at which that transition took place, and *States* is the description of the states (of the different timed automata) from which the transition took place. The postconditions are queries that check for appropriate patterns in this semantics. The semantics is shown in Appendix B. Note that for reasons of practicality our semantic definition uses `assert`, a non-logical feature of Prolog. It is possible to give a completely declarative logical semantics, but this is avoided here for the sake of readability.

The UPPAAL language specifies a real-time system in terms of a collection of timed automata. Each automata is specified using the `process` construct. The `process` construct identifies the name of the automata, its states, the initial state (using the `init` statement), and the transitions (using the `trans` construct). The actions associated with each transition—i.e., testing a conditions on clocks, resetting clocks, communicating with other processes—are specified with each transition.

We next specify a real-time system for controlling a gate at a railroad crossing using the UPPAAL Domain Specific Language. The system is composed of three processes, a gate-controller, the gate itself, and the train. The train is modeled by the *timed automata* shown in figure 5(i). It has four different transitions that are labeled `approach` `in`, `out`, `exit`: (i) The `approach` edge corresponds to the train approaching the gate; (ii) the edge labeled `in` denotes that the train is at the gate; (iii) the edge `out` denotes that the train has just left the gate; and, (iv) and `exit` denotes that the train has left the gate area. The controller is modeled by the timed-automata in figure 5(ii); it synchronizes with the train process on the `approach` and `exit` edges described above. The controller has two other edges labeled: (i) `lower` denoting starting of the lowering of the gate; and, (ii) `raise` denoting starting of the raising of the gate. Finally, the gate is modeled by the timed-automata in figure 5(iii). Time constraints force the train to employ at most 5 units of time to cross the gate area, with a speed which should allow 2 units of time to the gate to lower before the arrival of the train. The controller should be able to react in less than one unit of time to the approach of the train, and the gate should employ no more than one unit of time to completely lower or raise the gate. The UPPAAL program describing this system is presented in Figure 6.

```
// Global Declarations
clock c,e,d;
chan approach,exit,
     lower,raise;

//
// Processes Section
//

process train {
    state s0, s1, s2, s3;
    init s0;
    trans s0 -> s1 {
             sync approach!;
             assign c := 0;
             },
          s1 -> s2 {
             guard c > 2;
             },
          s2 -> s3,
          s3 -> s0 {
             guard c < 5;
             sync exit!;
             };
}
```

```
process cntl {
    state s0, s1, s2, s3;
    init s0;
    trans s0 -> s1 {
             sync approach?;
             assign e := 0;
            },
          s1 -> s2 {
             guard e == 1;
             sync lower!;
            },
          s2 -> s3 {
             sync exit?;
             assign e := 0;
            },
          s3 -> s0 {
             guard e < 1;
             sync raise!;
            };
}
```

```
process gate {
    state s0, s1, s2, s3, s4;
    init  s0;
    trans s0 -> s1 {
             sync lower?;
             assign d := 0;
            },
          s1 -> s2 {
             guard d < 1;
            },
          s2 -> s3 {
             sync raise?;
             assign d := 0;
            },
          s3 -> s4 {
             guard d > 1;
            },
          s4 -> s0 {
             guard d < 2;
            };
}

// System Description

system train,cntl,gate;
```

Figure 6: UPPAAL Program

This program can be fed to the executable syntax/semantics specification shown in Appendices A and B. Partial evaluation of this program (massaged for readability) yields the following program:

```
train(s0,approach,s1,T1,T2,T3) :- T3 = T1.
train(s1,epsilon,s2,T1,T2,T2) :- T1 - T2 > 2.
train(s2,epsilon,s3,T1,T2,T2).
train(s3,exit,s0,T1,T2,T2) :- T1 - T2 < 5.
train(X,lower,X,T1,T2,T2).
train(X,raise,X,T1,T2,T2).
```

```
gate(s0,lower,s1,T1,T2,T1).
gate(s1,epsilon,s2,T1,T2,T2)
     :- T1 - T2 < 1.
gate(s2,raise,s3,T1,T2,T1).
gate(s3,epsilon,s4,T1,T2,T2)
     :- T1-T2 > 1, T1-T2 < 2.
gate(s4,epsilon,s0,T1,T2,T2)
     :- T1-T2 < 2.
gate(X,approach,X,T1,T2,T2).
gate(X,exit,X,T1,T2,T2).
```

```
cntl(s0,approach,s1,T1,T2,T1).
cntl(s1,lower,s2,T1,T2,T2)
        :- T1 - T2 = 1.
cntl(s2,exit,s3,T1,T2,T1).
cntl(s3,raise,s0,T1,T2,T2)
        :- T1-T2 < 1.
cntl(X,epsilon,X,T1,T2,T2).
```

```
driver([],S0,S1,S2,T,T0,T1,T2,[]).
driver([X|S],S0,S1,S2,T,T0,T1,T2,[(X,T,[S0,S1,S2])|R]):-
   train(S0, X, S00, T, T0, T00),
   gate(S1, X, S10, T, T1, T10) ,
   cntl(S2, X, S20, T, T2, T20) , TA > T,
```

```
driver(S,S00,S10,S20,TA,T00,T10,T20,R).
```

The above constraint denotation of the railroad crossing controller is an executable specification of the composite real-time system. The composite specification can be used for verifying various global properties. For example, we may want to verify the safety property that when the train is at the gate, the gate is always closed. These properties are specified by the designer, and ensure correctness of the real-time system specification. We use the axiomatic semantics based framework discussed earlier to perform this verification. We use pre-conditions to put restrictions on the events list to ensure finiteness of the computation, and then the properties of interest (post-conditions) can be verified. The net effect obtained is that of (deductive) model-checking. Thus, our queries to the program will be of the form:

```
pre_condition(X),
driver(X, ...),
not post_condition(X)
```

where `post_condition` is the verification condition, while `pre_condition` is the condition imposed on the input—e.g., characterization of the input states of interest and conditions to ensure finiteness. This query should fail, if the `post_condition` holds true. The `pre_condition` should be such that it generates all sentences satisfying it. For example, if we want to check that the event "the train is in the gate area" (state `s2` of the train process) never occurs before the event "the gate is down" (state `s2` of process gate), then a possible pre-condition is that the transitions to state `s2` for the train or to `s2` for the gate must occur between two `approach` events. This pre-condition can be expressed so that it can act as a generator of all possible strings that start with an `approach` and end in `approach` (i.e., thus focusing only on one train at the time). Integrating this pre-condition in the semantics and partially evaluating it again, we get a modified `driver` as follows:

```
driver(_,[],_,_,_,_,_,_,_,[]) :-
driver(N,[X|S],S0,S1,S2,T,T0,T1,T2,[(X,T,[S0,S1,S2])|R]) :-
     train(S0,X,S00,T,T0,T00),
     gate(S1,X,S10,T,T1,T10) ,
     contr(S2,X,S20,T,T2,T20) ,
     TA > T, (X = approach ->
                (N = 0 -> M = 1; Rest = []);
                 M = N),
     driver(M,S,S00,S10,S20,TA,T00,T10,T20,R).
```

The `driver` thus acts as a generator, generating all possible strings that begin with `approach` and end in `approach` and that will be accepted by the automata. Now a property can be verified by calling the driver with uninstantiated input, and checking that the negated property does not hold for every possible meaning of the automata. Suppose, we want to verify that when the train is at the crossing, the gate must be down. This boils down to the following fact: in every possible run of the real-time system, the transition of the train to `s2` must occur after the transition of the gate to state `s2`. The negated property will be that the train gets to `s2` before the gate gets to `s2`. Thus, for example the query

```
?- driver(0,X,s0,s0,s0,0,0,0,0,R),
   append(_,[(_,_,[s2,s1,_])|_],R).
```

will fail when run on a constraint logic programming system (we used the CLP($R$) [22] system). The `append` program is the standard logic program for appending two lists.

Likewise, if we want to verify that the gate will be down at least 4 units of time—i.e., the time between the gate transition to `s1` and the transition to `s0` is at least 4 time units—then we will pose the following query:

```
?- driver(0,s0,s0,s0,0,0,0,0,X,R),
        append(A, [(_,T2,[_,s4,_]),(_,_,[_,s0,_]) |_], R), % Trans. to s0
        append(_, [(_,_,[_,s0,_]), (_,T1,[_,s1,_])|_], A), % Trans. to s1
        T2 - T1 < 4.
```

The above query will fail. Using our system one can also find out the minimum and the maximum amount of time the gate will be closed by posing the following query:

```
?- driver(0,s0,s0,s0,0,0,0,0,X,R),
   append(A, [(_,T2,[_,s4,_]) , (_,_,[_,s0,_]) |_], R),
   append(_, [(_,_,[_,s0,_]), (_,T1,[_,s1,_])|_], A),
   N < T2 - T1 < M, M > 0, N > 0.
```

We obtain the answer `M < 7, N > 1`. This tells us that the minimum time the gate will be down is 1 units, and the maximum time it will be down is 7 units. Other properties of the real-time system can similarly be tested. The ability to easily compute values of unknowns is a distinct advantage of a logic programming based approach, and considerable effort is required in other approaches used for verifying real-time systems to achieve similar behavior.

### 5.2 Discussion

A major weakness of our approach is that we have to make sure that the verification of properties leads to finite computations. The use of constraint handling and the use of tabling in the underlying execution model often guarantee such property (as seen in the examples in Section 5). If it is not, then we have to impose pre-conditions that ensure this. A popular approach to verifying an infinite state system is to *abstract* it (so that it becomes finite) while making sure that enough information remains in the abstraction so that the property of interest can be verified. The technique of abstraction can be easily adapted in our logical denotational approach: (i) one can give an abstract (logical denotational) semantics for the language, and then run tests on the resulting abstract denotation obtained, using the approach described above. (ii) we can use abstract interpretation tools built for logic programming to abstract the concrete denotation and use that for verifying the properties; in fact, work is in progress in to use non-failure analysis of constraint logic programs [8, 10] to verify properties of real-time systems. A third approach that can be used to remove the finiteness restriction is to use first-order theorem proving. The logical denotation of a program provides an axiomatization w.r.t. the language's semantics. These axioms can then be fed to a theorem prover, along with the preconditions and postconditions, and other additional axioms that may be needed in the postcondition, to perform verification [12].

However, we feel that our approach based on preconditions and postconditions is a pretty good compromise. While we do not verify the system completely, we do verify the program subject to the preconditions, and thus gain more confidence regarding software correctness.

Note that verification can be done more efficiently by inserting preconditions and postconditions in the program's denotation, and then partially evaluating the interpreter obtained for the language with respect to this annotated program denotation. We essentially obtain a compiled version of the program annotated with preconditions and postconditions. We have adopted this technique in both the domain specific languages considered in our examples. It is a lot more efficient to execute this annotated partially evaluated program rather than the original annotated denotation.

Note also that while the examples that we have included in this paper use only *direct semantics*, *continuation semantics* can also be easily modeled using Horn logic and constraints [11].

## 6 Related Work and Conclusions

In this paper we presented a Horn logic and constraint based framework for using denotational semantics for specifying, efficiently implementing, and verifying Domain Specific Language programs. The two distinct advantage that our Horn logic approach offers are: (i) the syntax specification is conveniently expressed using the Definite Clause Grammar facility of Logic Programming to obtain a parser for the language with minimal effort; (ii) a uniform framework is created for the specification of all components of the denotational description; (iii) verification of programs written in the DSL. In this regard, our framework is more advantageous than the frameworks for DSLs developed using the $\lambda$-calculus, e.g., in [5].

Our work shares some similarities with Consel's framework [5] for specification of DSLs. Consel's framework relies on the traditional $\lambda$-calculus approach. Thus, the parser has to be supplied separately, and verification cannot be done automatically. In contrast, in our approach not only the semantic specification can be used for deriving efficient implementations, it can also be used for verification.

Our framework can also be used for software development, by taking a language-centric view of the software development process. In the language-centric view of software development, we think of the software system as a processor of programs written in the software system's input language (a domain specific language). An executable semantic specification of this input language is also an executable specification of the software system. This executable specification can be made more efficient by using partial evaluation. The executable specification can also be used for verification and model checking purposes.

In this paper, we illustrated our framework by considering two domain specific languages: a file editor command language and a language for specifying real-time systems.

Our framework provides a rapid way of obtaining an executable specification of a domain specific language or a software system. For example, it took us only 2-3 hours to produce the syntax and semantic specification of the UPPAAL language shown in appendix A and B, which we could then use to verify properties on different examples. Work is in progress to use our framework for specification, implementation, and verification of other DSLs.

# Acknowledgments

The authors wish to thank N. Jones and M. Hermenegildo for the helpful comments on this work.

The authors have been supported by NSF grants CCR 96-25358, CDA-9729848, EIA 98-10732, HRD 98-00209, HRD 96-28450, and INT 95-15256, and by a grant from the US-Spain Research Program.

# References

[1] R. Alur and D. Dill. The Theory of Timed Automata. *Theoretical Computer Science*, 126, 1994.

[2] J. Bengtsson, et al. *UPPAAL: A Tool Suite for Validation and Verification of Real-time Systems.* http://www.docs.uu.se/rtmv/uppaal, 1997.

[3] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. In *Journal of the ACM*, 43(1):20-74, 1996.

[4] E. M. Clark, E. A. Emerson and A. P. Sistla. Automatic Verification of finite-state Concurrent Systems Using Temporal Logic Specification. In *ACM TOPLAS*, 8(2), 1986.

[5] C. Consel. Architecturing Software Using a Methodology for Language Development. In *Proc. 10th Int'l Symp. on Prog. Lang. Impl., Logics and Programs (PLILP)*, Springer LNCS 1490, pp. 170-194, 1998.

[6] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Model for Static Analysis of Programs for Construction or Approximation of Fixpoints. In *Proceedings of POPL,* pp. 238-252, 1977.

[7] S. K. Das. *Deductive Databases and Logic Programming.* Addison-Wesley. 1992.

[8] S. Debray, P. Lopez-Garcia, and M. Hermenegildo. Non-failure Analysis for Logic Programs. In *International Conference on Logic Programming.* MIT Press, 1997.

[9] M. Garcia de la Banda et al. Independence in Constraint Logic Programming. In *Procs. Int. Symposium on Logic Programming*, MIT Press, 1993.

[10] M. Garcia de la Banda, M. Hermenegildo, et al. Global Analysis of Constraint Logic Programs. In *ACM Trans. on Prog. Languages and Systems*, Vol. 18, Num. 5, pages 564-615, ACM, 1996.

[11] G. Gupta. Horn Logic Denotations and Their Applications. In *Proc. Workshop on Strategic Research Directions in Logic Programming*, Springer Verlag, (to appear).

[12] G. Gupta. *A Horn Logic Denotational Approach to Verification based on First Order Theorem Proving.* Internal report. Dec. 1998.

[13] G. Gupta and E. Pontelli. A Constraint-based Denotational Approach to Specification and Verification of Real-time Systems. In *Proc. Real-time Systems Symposium*, IEEE pp. 230-239, 1997.

[14] G. Gupta, E. Pontelli, R. Felix-Cardenas, A. Lara, Automatic Derivation of a Provably Correct Parallelizing Compiler, In *Proceedings of International Conference on Parallel Processing*, IEEE Press, pp. 579-586, 1998.

[15] G. Gupta, E. Pontelli. *Horn Logic Denotational Framework for Abstract Interpretation.* New Mexico State University. Working paper, 1999.

[16] C. Gunter. *Programming Language Semantics.* MIT Press. 1992.

[17] R. Hartley. *MidiTrans: a MIDI File Transform Language.* Tech. Report, NMSU, 1998.

[18] C. Hoare. An Axiomatic Basis for Computer Programming. In *Comm. of the ACM.* Vol. 12. 1969.

[19] N. Jones. Introduction to Partial Evaluation. In *ACM Computing Surveys.* 28(3):480-503, 1996.

[20] A. Karshmer, G. Gupta, S. Geiger and C. Weaver. A Framework for Translation of Braille Nemeth Math to LaTeX: The MAVIS Project. In *Proc. Conf. on Assistive Technologies*, ACM Press, 1998.

[21] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. In *Software Tools for Technology Transfer.* 1997.

[22] J. L. Lassez and J. Jaffar. Constraint logic programming. In *Proc. 14th ACM POPL*, 1987.

[23] J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag. 2nd ed. 1987.

[24] E. Pontelli, G. Gupta, and A. Lara. *Horn Logic Denotations for MidiTrans.* Internal Report, New Mexico State University, 1999.

[25] C. Ramming. *Proceedings of the Usenix Conference on Domain-Specific Languages.* Usenix, 1997.

[26] M. Raskovsky and P. Collier. From Standard to Implementational Denotational Semantics. In *Semantics Directed Compiler Generation.* Springer Verlag. pp. 94-139, 1994.

[27] Y.S. Ramakrishnan, C.R. Ramakrishnan, I.V. Ramakrishnan et al. Efficient Model Checking using Tabled Resolution. In *Proceedings of Computer Aided Verification (CAV'97)*, 1997.

[28] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. SIGMOD International Conf. on Management of Data*, 1994.

[29] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* Ph.D. Thesis, Royal Inst. of Techn. Sweden, 1994.

[30] D. Schmidt. *Denotational Semantics: a Methodology for Language Development.* W.C. Brown Publishers, 1986.

[31] D. Schmidt. Programming language semantics. In *ACM Computing Surveys*, 28-1, 265-267, 1996.

[32] D. Schmidt. On the Need for a Popular Formal Semantics. *Proc. ACM Conf. on Strategic Directions in Computing Research,* Cambridge, MA, June 1996. ACM SIGPLAN Notices 32-1 (1997) 115-116.

[33] L. Sterling and S. Shapiro. *The Art of Prolog.* MIT Press, 1996.

[34] P. Van Hentenryck. *Constraint Handling in Prolog.* MIT Press, 1988.

# Appendix A: Definite Clause Grammar for UPPAAL

```
ita(ita(V,P,G)) -->  varlist(V), proclist(P), globals(G).
varlist([])     -->  { true }.
varlist([X|Y])  -->  channel(X), varlist(Y).
varlist([X|Y])  -->  variable(X), varlist(Y).

proclist([P])   -->  proc(P).
proclist([P|Q]) -->  proc(P), proclist(Q).

globals(system(List)) -->  ['system'], idlist(List), [';'].
channel(chan(L)) --> [chan], idlist(L), [';'].
variable(var(clock,L)) --> [clock], idlist(L), [';'].
proc(proc(Id,States,Trans)) --> [process], id(Id),
                              ['{'],statedecls(States), transdecls(Trans) , ['}'].
idlist([I])   -->  id(I).
idlist([I|L]) -->  id(I), [','], idlist(L).

statedecls(state(Initial,States)) -->
     [state], idlist(States), [';'], [init], id(Initial), [';'].

transdecls(trans(L)) -->  [trans], translist(L) , [';'].
translist([A])   --> trans(A).
translist([A|B]) --> trans(A), [','],translist(B).
trans(t(I1,I2,G,S,A)) -->
     id(I1), ['->'], id(I2), ['{'], opg(G), ops(S), opa(A), ['}'].

opg(noguard)  -->  {true}.
opg(guard(L)) -->  [guard], guardlist(L), [';'].
ops(nosync)          -->  {true}.
ops(sync(send,I))    -->  [sync], id(I), ['!'],[';'].
ops(sync(receive,I)) -->  [sync], id(I), ['?'], [';'].
opa(noassign)  -->  {true}.
opa(assign(L)) -->  [assign], assignlist(L),[';'].
```

```
guardlist([L])   -->  guard(L).
guardlist([L|R]) -->  guard(L), [','], guardlist(R).
assignlist([A])   -->  assign(A).
assignlist([A|B]) -->  assign(A), [','], assignlist(B).
assign(A) -->  clockassign(A).

guard(compare(I,N,Op)) -->  id(I), relop(Op), nat(N).
guard(ccompare(I1,I2,N,Op1,Op2)) -->
     id(I1), relop(Op1), id(I2), oper(Op2), nat(N).

clockassign(assign(I,N)) --> id(I) , [':='], nat(N).

relop('<')  -->   ['<'].
relop('<=') -->   ['<='].
relop('>')  -->   ['>'].
relop('>=') -->   ['>='].
relop('=')  -->   ['=='].
oper(plus)  -->   ['+'].
oper(minus) -->   ['-'].
nat(X) -->  [X], {integer(X)}.
id(X)  -->  [X], {atom(X)}.
```

# Appendix B: Logical Denotation of UPPAAL

```
semita( ita([var(clock,Vars),chan(Chan)] , Procs , System) ) :-
     semprocs(Procs,Vars,Chan,Initials), semglobals(System,Initials,Vars,Chan).

semglobals(system(List),_,Vars,_) :-
     length(List,N), length(States,N), length(Vars,M), length(Clocks,M),
     append(States,[C|Clocks],Args1),  append(Args1, [[(X,C)|Remainder]], Args),
     Head =.. [driver,[X|Y]|Args],
     generate_body(States,List,Clocks,C,X,Body,NewStates,NewClocks),
     append(NewStates,[T1|NewClocks],NewArgs1),  append(NewArgs1,[Remainder],NewArgs),
     RecCall =.. [driver,Y|NewArgs],
     assert((Head :- Body, (T1>C) , RecCall)).

generate_body([State],[Name],Clocks,GClock,Symbol,C,[NState],Clocks1):-
     gen_new_clocks(Clocks,Clocks1,ClocksCode),
     C =.. [Name,State,Symbol, NState,GClock|ClocksCode].
generate_body([State|OtherStates], [Name|OtherNames], Clocks, GClock,
               Symbol, (Call,OtherCalls), [NState|NewStates], NewClocks) :-
     gen_new_clocks(Clocks,Clocks1,ClocksCode),
     Call =.. [Name,State,Symbol,NState,GClock|ClocksCode],
     generate_body(OtherStates,OtherNames,Clocks1,GClock,Symbol,OtherCalls,NewStates,NewClocks).

gen_new_clocks([],[],[]).
gen_new_clocks([X|Y],[Z|W],[X,Z|Rest]) :-    gen_new_clocks(Y,W,Rest).

semprocs([],_,_,[]).
semprocs([proc(Name,States,Trans)|Rest],Vars,Chan,[In1|In2]) :-
     semprocedure(Name,States,Trans,Vars,Chan,In1), semprocs(Rest,Vars,Chan,In2).

semprocedure(Name,state(Init,_),trans(List),Vars,Chan,Init) :-
     semtransitions(List,Name,Vars,Chan).

semtransitions([],_,_,_).
semtransitions([t(From,To,Guard,Sync,Assign)|Rest],Name,Vars,Chan) :-
     (Sync = sync(_,C) ->  generate_fact(Name,From,To,C,GClock,Vars,Fact) ;
                           generate_fact(Name,From,To,epsilon,GClock,Vars,Fact) ),
     semguardassign(Fact,Guard,Vars,Assign,GClock),
     semtransitions(Rest,Name,Vars,Chan).

semguardassign(Head,noguard,_,noassign,_) :-
     generate_equalities(Head,Eqs), assert((Head :- Eqs)).
semguardassign(Head,noguard,Vars,assign(AList),GClock) :-
     semopa(AList,Vars,Head,Equalities,GClock,Used),
```

```
     difflist(Vars,Used,Remaining), additional_equalities(Remaining,Vars,Head,Others),
     assert((Head :- Equalities,Others)).
semguardassign(Head,guard(List),Vars, noassign,GClock) :-
     semopg(List,Vars,Head,Constraints,GClock),
     generate_equalities(Head,Equalities),
     assert((Head :- Constraints,Equalities)).
semguardassign(Head,guard(List),Vars,assign(AList),GClock) :-
     semopg(List,Vars,Head,Constraints,GClock), semopa(AList,Vars,Head,Equalities,GClock,Used),
     difflist(Vars,Used,Remaining), additional_equalities(Remaining,Vars,Head,Others),
     assert((Head :- Constraints,Equalities,Others)).

semopg([],_,_,true,_).
semopg([compare(Clock,Nat,Op)|Rest],Vars,Head,(C1,C2),GClock) :-
     semguard(Op,Clock,Nat,Vars,Head,C1,GClock),
     semopg(Rest,Vars,Head,C2,GClock).

semopa([],_,_,true,_,[]).
semopa([assign(Var,Nat)|Rest],Vars,Head,(C1,C2),GClock,[Var|Used]) :-
     semassign(Var,Nat,Vars,Head,C1,GClock),
     semopa(Rest,Vars,Head,C2,GClock,Used).

semassign(Var,Nat,Vars,Head,C1,GClock) :-
     extract_variable1(Var,Vars,Head,V),
     C1 = ((GClock - Nat) = V).

semguard(Op,Clock,Nat,Vars,Head,C1,GClock) :-
     extract_variable(Clock,Vars,Head,V),
     C1 =.. [Op,(GClock-V),Nat].

%%%%%%%% Auxiliary predicates

difflist([],_,[]).
difflist([X|Y],Used,Z)     :- member(X,Used), !, difflist(Y,Used,Z).
difflist([X|Y],Used,[X|Z]) :- difflist(Y,Used,Z).

additional_equalities([],_,_,true).
additional_equalities([X|Y],Vars,Head,((V1=V2),Rest)) :-
     extract_variable(X,Vars,Head,V1),
     extract_variable1(X,Vars,Head,V2),
     additional_equalities(Y,Vars,Head,Rest).

generate_equalities(Head,Eqs) :-  Head =..[_,_,_,_,_|List],
                                  get_equalities(List,Eqs).

get_equalities([],true).
get_equalities([X1,X2|Rest],((X1=X2),C)) :- get_equalities(Rest,C).

extract_variable(Clock,Vars,Head,V) :-      Head =.. [_,_,_,_,_|Clocks],
                                            search_variable(Vars,Clock,Clocks,V).

search_variable([X|_],X,[Y|_],Y).
search_variable([X|Y],Z,[_,_|Rest],V) :-    X \== Z,
                                            search_variable(Y,Z,Rest,V).

extract_variable1(Clock,Vars,Head,V) :-     Head =.. [_,_,_,_,_|Clocks],
                                            search_variable1(Vars,Clock,Clocks,V).

search_variable1([X|_],X,[_,Y|_],Y).
search_variable1([X|Y],Z,[_,_|Rest],V) :-   X \== Z,
                                            search_variable1(Y,Z,Rest,V).

generate_fact(Name,StartState,EndState,Symbol,GlobClock,Clocks,FACT) :-
     generate_rest(Clocks,Rest),
     FACT =.. [Name,StartState,Symbol,EndState,GlobClock|Rest].

generate_rest([],[]).
generate_rest([_|Y],[_,_|Rest]) :-          generate_rest(Y,Rest).
```