Design and Implementation of the High-level Specification Language CSP(LP) in Prolog

Michael Leuschel

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
`mal@ecs.soton.ac.uk`
WWW: `http://www.ecs.soton.ac.uk/~mal`

Abstract. We describe practical experiences of using a logic programming based approach to model and reason about concurrent systems. We argue that logic programming is a good foundation for developing, prototyping, and animating new specification languages. In particular, we present the new high-level specification language CSP(LP), unifying CSP with concurrent (constraint) logic programming, and which we hope can be used to formally reason *both* about logical and concurrent aspects of critical systems.
Keywords: Specification, Verification, Concurrency, Implementation, and Compilation.

1 Introduction and Summary

This (position) paper describes work carried out within the recently started projects ABCD[1] and iMoc.[2] The objective of the ABCD project is to increase the uptake of formal methods in the business critical systems industry by lowering the cost of entry and increasing the benefits of using formal modelling. The focus is on support of system definition and architectural design so that the systems integrator can more easily model systems and validate proposed system architectures. A main point is to apply formal methods—such as model checking—early on in the software development cycle, i.e., on high-level specifications or high-level prototypes of the particular business critical system to be developed.

To effectively lower the cost of entry for industrial users, we want a specification and/or prototyping language which is

– powerful and high-level enough to specify business and safety critical systems in a natural and succinct manner. For example, we want sophisticated data-structures and do not want to force the architect to have to come up with artificial abstractions himself.

[1] "Automated validation of Business Critical systems using Component-based Design," EPSRC grant GR/M91013, with industrial collaborators IBM, ICL, Praxis Critical Systems, and Roke Manor Research.

[2] "Infinite state MOdel Checking using partial evaluation and abstract interpretation," EPSRC grant GR/N11667.

- expressive enough to easily express and reason about concurrency. Ideally, one would want to compose larger systems from smaller ones, e.g., by putting them in parallel and allowing various forms of interaction (synchronisation, asynchronous message passing, shared memory,...), allowing encapsulation (hiding,...) and re-use (renaming, parameterisation,...). Ideally, one may also want to consider timing aspects (timeouts, interrupts,...).
- usable by industrial partners (graphical notation) and reduces the potential of errors (types, consistency checking,...)

To increase the benefits, we want the systems architect to be able to probe and examine his specification or prototype, e.g., via *animation*, and *automatic verification*, such as model checking.

Currently, the specification language issue within ABCD is still under investigation. First case studies were carried out using the B-method [1] and CSP with functions and datatypes [15, 29, 9]. Other languages and formalisms, such as Petri nets, the C-like language Promela of Spin [16], the Alloy language of Alcoa [17], LOTOS, are being studied (see also [13]). However, the language issue is far from fixed, and we wish to experiment with various extensions and integrations ([4]) of the paradigms. One might also want to consider domain-specific specification languages (for the ABC project with IBM a domain specific compensation operator has been developed for reliable transaction processing [8]). As we will show in this paper, an approach based on logic programming and partial evaluation seems very promising for this problem:

- a lot of languages can be easily encoded in logic programming, because of features such as non-determinism, unification, co-routining, constraints,...
- one can get compilers using offline partial evaluators such as LOGEN [18]. One can do more sophisticated optimisations using online partial evaluation systems such as MIXTUS [30], SP [10], or ECCE [22]. One can also apply existing analysers to infer properties about the source program (see [26]).
- one can easily animate such languages in existing Prolog systems,
- one can do finite and infinite state model checking [27, 6],[7], [23], [25].

All of this points will be substantiated in this paper, through a non-trivial high-level specification language. The design of this new language CSP(LP) was heavily influenced by the possibilities of (constraint) logic programming. In particular, CSP(LP) supports, amongst others, complex datatypes, constraints, concurrency, message passing via channels à la CSP and CCS, and asynchronous message passing via streams. We hope that CSP(LP) is a contribution in its own right, and can be used to formally reason *both* about logical and concurrent aspects of critical systems.

2 CSP, CSP-FDR, and CSP(LP)

2.1 Elementary CSP

CSP is a process algebra defined by Hoare [15]. The first semantics associated with CSP was a denotational semantics in terms of traces, failures and (failure and) divergences. Elementary CSP (without datatypes, functions, or other advanced operators) can be defined as follows. Given Σ, a finite or enumerable set of actions (which we will henceforth denote by lower case letters $a, b, c, \ldots$), and $\mathcal{X}$, an enumerable set of variables or processes (which we henceforth denote by identifiers such as $Q, R, \ldots$, or $MYPROCESS$ starting with an uppercase letter), the syntax of a basic CSP expression is defined by the following grammar (where A denotes a set of actions):

$$\begin{array}{lll} P ::= & STOP \text{ (deadlock) } \mid & SKIP \text{ (success) } \mid \\ & a \rightarrow P \text{ (prefix) } \mid & P \sqcap P \text{ (internal choice) } \mid \\ & P \,\square\, P \text{ (external choice) } \mid & P \,[\![A]\!]\, P \text{ (parallel composition) } \mid \\ & P \backslash A \text{ (hiding) } \mid & Q \text{ (instantiation of a process)} \end{array}$$

Moreover, each process Q used must have a (possibly recursive) definition $Q = P$.

We suppose that all used processes are defined by at least one recursive definition (if there is more than one definition this is seen to be like an external choice of all the right-hand sides). In the following, we also suppose the alphabet Σ to be finite.

Intuitively, $a \rightarrow P$ means that the system proposes the action a to its environment, which can decide to execute it. The external choice is resolved by the environment (except when two branches propose the same action, where a nondeterministic choice is taken in case the environment chooses that action). Internal choice is made by the system without any control from the environment. $P \,[\![A]\!]\, Q$ is the generalized parallel operator of [29], and means that the process P synchronizes with Q on any action in the set of actions A. If an action outside A is enabled in P or Q, it can occur without synchronization of both processes. Pure interleaving $P \,[\![\emptyset]\!]\, Q$ is denoted by $P \,|||\, Q$. Pure synchronization $P \,[\![\Sigma]\!]\, Q$ is denoted by $P \,||\, Q$. The hiding operator $P \backslash A$ makes any visible action $a \in A$ of P invisible. In the the operational semantics [29] the latter (as well as the internal choice) is achieved using the internal action τ.

A major practical difference[3] with CCS [24] is that CSP allows for synchronisation of an arbitrary number of processes (while CCS only supports binary synchronisation). This makes CSP processes more difficult to implement, but on the other hand more suitable as the basis of a high-level specification language (which is what we are after in this paper).

2.2 CSP-FDR and CSP(LP)

The elementary CSP presented above is not very useful in practice, either as a programming or a specification language, because its lack of value passing and the absence of more refined operators. [29, 9, 31] hence presents an extension of CSP (which we henceforth call CSP-FDR), along with a machine-readable ASCII syntax, which is usable in practice. CSP-FDR and the tools FDR and PROBE are used by government institutions and companies, e.g., in the semiconductor,

[3] There are plenty of other differences of course. For example, CSP has been developed with denotational semantics in mind, while CCS is more tightly linked with bisimulation.

defence, aerospace, and security areas (for the latter, see [28]). In this extension of CSP one can:

- pass tuples of datavalues on channels. For example, $a!1 \rightarrow STOP$ will output the datavalue 1 on the channel a, while $a?x \rightarrow P(x)$ will read a datavalue on channel a and bind x to the value read. One can also put additional constraints on values that are received on a channel,as in $a?x : x > 2 \rightarrow P(x)$.
- use constructs such as the if-then-else, let-constructs, and operators such as interrupt and timeout
- use sets and set operations, sequences and sequence operations, integers and arithmetic operators, tuples, enumerated types, ranges, ...

This extension of CSP was heavily influenced by functional programming languages, and hence relies on pattern matching as a means to synchronise on channels. As a consequence, $a?x \rightarrow P(x)$ can *not* synchronise with $a?y \rightarrow Q(y)$. In the remainder of this paper we show how, by using logic rather than functional programming as our foundation, we can overcome this limitation, thus leading to a more powerful language CSP(LP) which reconciles CSP-FDR with (concurrent) logic programming languages.

The basic syntax of CSP(LP) is summarised in Figure 1. The semantics of CSP(LP) will become clear in the next section, where we show how it can be mapped to logic programming.

Operator	Syntax	Ascii Syntax
stop	$STOP$	`STOP`
skip	$SKIP$	`SKIP`
prefix	$a \rightarrow Q$	`a->P`
conditional prefix	$a?x : x > 1 \rightarrow P$	`a?x:x>1->P`
external choice	$P \Box Q$	`P [] Q`
internal choice	$P \sqcap Q$	`P |~| Q`
interleaving	$P \vert\vert\vert Q$	`P ||| Q`
parallel composition	$P \,[\![A]\!]\, Q$	`P [| A |] Q`
sequential composition	$P; Q$	`P ->> Q`
hiding	$P \backslash A$	`P \\ A`
renaming	$P[R]$	`P [[ R ]]`
timeout	$P \triangleright Q$	`P [> Q`
interrupt	$P \triangle_i Q$	`P /\ Q`
if then else	*if t then P else Q*	`if T then P else Q`
let expressions	*let* $v = e$ *in* P	`let V=E in P`
agent definition	$A = P$	`A = P;`

Fig. 1. Summary of syntax of CSP(LP)

3 CSP(LP) and logic programming

3.1 Implementation

In this section we present an operational semantics of CSP(LP). Rather than using natural deduction style rules (as in [29]) we immediately present the (declarative) Prolog code. This code implements a ternary relation *trans*, where *trans*(e, a, e') means that the CSP(LP) expression e can evolve into the expression e' by performing the action a. Our language is not committed choice, there can be multiple solutions for the same e, and the order of those solutions is not relevant. Apart from a few minor additions, the rules below basically implement the operational semantics of [29]; the main difference lies in the generalisation of the synchronisation mechanism.

Basic Operators First, the elementary processes *STOP* and *SKIP* are extremely straightforward to encode:

```
trans(stop,_,_) :- fail.
trans(skip,tick,stop).
```

The unconstrained prefix operator is not much more difficult. Observe that the first argument to *prefix* is a tuple of values (annotated by either "!", "?", or "."). The constrained prefix operator is slightly more complicated. The actual implementation of the predicate *test* actually makes use of the co-routining and constraints facilities of SICStus Prolog. Notably, we use, e.g., the SICStus *dif* predicate, which delays if the arguments are not sufficiently instantiated.

```
trans(prefix(V,Ch,X),  io(V,Ch) ,X).
trans(prefix(V,Ch,Constraint,X), io(V,Ch) ,X) :- test(Constraint).
```

The CSP choice operators $\sqcap$, $\Box$ are implemented exactly as in [29]. The rules for the external choice might seem a bit surprising at first, but they are needed to ensure that $\tau \rightarrow P$ behaves like P ($\tau \rightarrow P$ is equivalent to P in the CSP semantics).

```
trans(int_choice(X,_Y),tau,X).
trans(int_choice(_X,Y),tau,Y).

trans(ext_choice(X,_Y),A,X1) :- trans(X,A,X1),dif(A,tau).
trans(ext_choice(_X,Y),A,Y1) :- trans(Y,A,Y1),dif(A,tau).
trans(ext_choice(X,Y),tau,ext_choice(X1,Y)) :- trans(X,tau,X1).
trans(ext_choice(X,Y),tau,ext_choice(X,Y1)) :- trans(Y,tau,Y1).
```

One can also implement the CCS style choice operator + (which has a simpler operational semantics, but it treats $\tau \rightarrow P$ differently from P and hence neither the CSP failures-divergences semantics nor weak bisimulation [24] is a congruence for +):

```
trans(ccs_choice(X,_Y),A,X1) :- trans(X,A,X1).
trans(ccs_choice(_X,Y),A,Y1) :- trans(Y,A,Y1).
```

Implementing sequential composition is again pretty straightforward:

```
trans(seq(P,Q),A,seq(P1,Q)) :- trans(P,A,P1), dif(A=tick).
trans(seq(P,Q),tau,Q) :- trans(P,tick,_).
```

Finally, the timeout and interrupt operators $\triangleright$, $\triangle_i$ are implemented as follows:

```
trans(timeout(P,_Q),A,P1) :- dif(A,tau),trans(P,A,P1).
trans(timeout(P,Q),tau,timeout(P1,Q)) :- trans(P,tau,P1).
trans(timeout(_P,Q),tau,Q).

trans(interrupt(P,Q),A,interrupt(P1,Q)) :- dif(A,tick),trans(P,A,P1).
trans(interrupt(P,Q),tick,omega) :- trans(P,tick,_).
trans(interrupt(P,Q),i,Q).
```

Agent calls and recursion When implementing agent calls to recursive definitions one has to be extremely careful in the presence of divergence and multiple agent equations.

Suppose that we have a set of *agent*/2 facts which represent all agent definition equations. These facts are generated from the CSP(LP) Ascii syntax using a parser. A first prototype predictive recursive descent parser has been developed in SICStus Prolog itself using DCG's.

If there are multiple equations for the same agent, then the entire agent should be seen as having an *external* choice between all individual equations. Now, the following piece of code seems the natural solution:

```
trans(agent_call(X),A,NewExpr) :-
   evaluate_agent_call(X,EX),agent(EX,AE),trans(AE,A,NewExpr).
```

This implementation is rather efficient, amenable to techniques such as partial evaluation (cf. Section 4.2), and the scoping of parameters works as expected.[4] It is also possible to add delay declarations to ensure that a call is only unfolded if its arguments are sufficiently instantiated.[5] However, the above code is not always correct:

- when an infinite number of calls without visible action is possible the interpreter might loop, instead of generating a divergent transition system.
- it effectively treats all equations as being in a big *CCS* choice rather than being within an external choice. The interpreter is thus only correct for agent definitions which never produce a τ action as their first action.

[4] Unlike the CSP in FDR [9], where the `out!1` within `P(out) = out!1->STOP` does not refer to the parameter `out` if a global channel `out` exists.

[5] Although this can be semantically tricky: e.g., what is the logical meaning of a floundering derivation?

The easiest way to solve this problem is to impose restrictions on the equations to ensure that the above problems do not arise (similar to what is done in FDR [9]). If the restrictions are not satisfied one has to generate an explicit external choice and possibly explicit τ actions as well. This is not a practical problem, however, as the translation can be done automatically by the parser.

Let expressions and conditionals Because of the absence of recursion, let expressions and conditionals are fortunately much less problematic to implement (below \+ denotes negation):

```
trans(let(V,VExp,CExp),tau,CExp) :- evaluate_argument(VExp,Val),V=Val.

trans(if(Test,Then,_Else),A,X1) :- test(Test), trans(Then,A,X1).
trans(if(Test,_Then,Else),A,X1) :- \+(test(Test)), trans(Else,A,X1).
trans(if(Test,Then),A,X1) :- test(Test), trans(Then,A,X1).
```

Hiding, Renaming, and Restriction Hiding can be implemented by replacing visible actions by the special τ action:

```
trans(hide(Expr,CList), A, hide(X,CList) ) :-
    trans(Expr,A,X),dif(A,tick),not_hidden(A,CList).
trans(hide(Expr,CList), tau, hide(X,CList) ) :-
    trans(Expr,A,X),hidden(A,CList).
trans(hide(Expr,_CList), tick, omega) :- trans(Expr,tick,_X).
```

`hidden(A,CList)` checks whether the channels of `A` appear within the channel list `CList` and `not_hidden` is its sound negation, implemented using `dif`.

The following implements renaming. Note that, because of our logic programming foundation, we can quite easily implement relational renaming [29].[6]

```
trans(rename(Expr,RenList), RA, rename(X,RenList) ) :-
       trans(Expr,A,X), rename_action(A,RenList,RA).
```

One can also implement CCS-like restriction in much the same style.

Synchronisation Operators via Unification The essence of our implementation of the parallel composition operators can be phrased as: *"synchronisation is unification."* The classical definition of synchronisation in CSP-FDR is based on pattern matching: the synchronised values must either be ground and identical (as in $c!1 \rightarrow P \; [\![\{c\}]\!] \; c!1 \rightarrow Q$) or one value must be ground and the other a free variable (as in $c!1 \rightarrow P \; [\![\{c\}]\!] \; c?x \rightarrow Q(x)$). We can implement a generalisation of this scheme, which allows variables or even partially instantiated terms on both sides and where synchronisation results in unification. This leads to the following piece of code for the generalised parallel operator:

[6] E.g., the expression `(a->b->STOP) [[ a<-c, a<-d ]]` has the traces $\{c, d, cb, db\}$.

```
trans(par(X,CList,Y), io(V,Ch), par(X1,CList,Y1)) :-
     trans(X, io(V1,Ch), X1),trans(Y, io(V2,Ch), Y1),
     unify_values(V1,V2,V),hidden(io(V,Ch),CList).
trans(par(X,CList,Y), A, par(X1,CList,Y) ) :-
  trans(X,A,X1),dif(A,tick),not_hidden(A,CList)). /* covers tau */
trans(par(X,CList,Y), A, par(X,CList,Y1) ) :-
  trans(Y,A,Y1),dif(A,tick),not_hidden(A,CList)). /* covers tau */
trans(par(X,CList,Y), tau, par(omega,CList,Y) ) :- trans(X,tick,_).
trans(par(X,CList,Y), tau, par(X,CList,omega) ) :- trans(Y,tick,_).
trans(par(omega,CList,omega), tick, omega ).
```

The interleaving operator ||| is then defined as follows:

```
trans(interleave(P,Q),A,R) :- trans(par(X,[],Y),A,R).
```

One can also implement CCS style synchronisation:

```
trans(ccs_par(X,Y), tau, ccs_par(X1,Y1)) :- trans(X, io(V1,Ch), X1),
         trans(Y,io(V2,Ch), Y1), ccs_unify_values(V1,V2,_V).
trans(ccs_par(X,Y), A, ccs_par(X1,Y) ) :- trans(X,A,X1).
trans(ccs_par(X,Y), A, ccs_par(X,Y1) ) :- trans(Y,A,Y1).
```

As we will see below, this innocently looking generalisation adds a lot of power, resulting in a language CSP(LP) which unifies CSP-FDR, (constraint) logic programming and concurrent logic programming. Of course, from a semantical point of view we have to be very careful when using this extension. For example, when computing a trace for a CSP(LP) specification, we have to check that there are concrete values for the uninstantiated variables which satisfy all the constraints set up by the interpreter.

3.2 The Expressive Power of CSP(LP)

Obviously, CSP(LP) allows one to express basically all things expressible in CSP-FDR. In this section we illustrate the additional expressivity of CSP(LP).

Classical Prolog in CSP(LP) We first show how one can do "classical" logic programming within CSP(LP), thus showing it to be a proper extension of both logic programming as well as of CSP-FDR. E.g. this is how one can encode the append and double-append predicates in CSP(LP).

```
App(nil,_Z,_Z) = SKIP;
App(cons(_H,_X),_Y,cons(_H,_Z)) = App(_X,_Y,_Z);
Dapp(_X,_Y,_Z,_R) = App(_X,_Y,_XY) ->> App(_XY,_Z,_R);
MAIN = Dapp(cons(a,nil),cons(b,nil),cons(c,nil),_R) ->> (cas!_R -> STOP);
```

The computed answer is output on channel cas. Note that sequential composition is used to encode conjunction and *SKIP* is used to encode success. This essentially mimics Prolog left-to-right execution. More flexible co-routining can be encoded using the interleaving operator:[7]

[7] But then, to prevent looping of the interpreter, one needs to generate a τ-action for every unfolding (as discussed in Section 3.1). One could also add delay declarations.

```
Dapp(_X,_Y,_Z,_R) = App(_X,_Y,_XY) ||| App(_XY,_Z,_R);
```

Constraints The following encodes the Petri net from [23] (controlling access to a critical section [cs] via a semaphore) and also shows how one can use constraints within CSP(LP).

```
Petri(P,Sem,CS,Y,RC) = enter:(Sem>0 and P>0)->Petri(P-1,Sem-1,CS+1,Y,RC);
Petri(P,Sem,CS,Y,RC) = exit:(CS>0) -> Petri(P,Sem+1,CS-1,Y+1,RC);
Petri(P,Sem,CS,Y,RC) = restart:(Y>0) -> Petri(P+1,Sem,CS,Y-1,RC+1);
MAIN = Petri(2,1,0,0,0);
```

Streams and Shared Memory As CSP(LP) can handle logical variables, we can basically use all the "tricks" of concurrent (constraint) logic programming languages [32] to represent asynchronous communication via streams (at the cost of complicating the semantical underpinning, as we have to use delay declarations to ensure that values are not read before they have been written). The example below illustrates this feature, where the process *INT* instantiates a stream (by generating ever larger numbers) and *BUF* reads from the stream and outputs the result on the channel *b*.

```
delay BUF(X) until nonvar(X);
BUF(nil) = STOP;
BUF(cons(_H,_T)) = b!_H -> BUF(_T);
INT(_X,cons(_X,_T)) = i!_X -> INT(+(_X,1),_T);
MAIN =  INT(0,_S) ||| BUF(_S);
```

One can also encode a limited form of (write-once read-many) shared memory. In that case variables are similar to pointers which can be passed around processes. In the simple example below, three processes share the pointer X to a memory location. When *RESET* sets this location to 0 the result becomes immediately visible to the 2 *REP* processes.

```
REP(X) = t.X -> REP(X);
RESET(0) = reset -> STOP;
MAIN = (REP(X) ||| REP(X) ||| RESET(X));
```

3.3 Adding Domain-Specific Features

Inspired by [8], we have added a domain specific compensation operator $\bowtie$. Intuitively, $P \bowtie S$ behaves like P until the special *abort* action is performed. At that point all the actions performed by P since the last *commit* action are compensated for in reverse order, according to the renaming specified by S. For example, for $P = a \rightarrow b \rightarrow abort \rightarrow STOP$ we have that $P \bowtie \{a \leftarrow c_a\}$ has the trace $\langle a, b, abort, c_a \rangle$. This compensation operator was used to arrive at a succinct specification of an electronic bookshop in Appendix A. For example, we use the construct $Shopper(id) \bowtie \{db.rem \leftarrow db.add\}$ to ensure that, if the shopper process $Shopper(id)$ terminates abnormally, books are re-inserted automatically into the main database *db*. One can encode more elaborate compensation mechanisms, which, e.g., distinguish between sequential and parallel compensations [8].

```
trans(compensate(P,RenLst,CP),A,compensate(P1,RenLst,prefix(V,CH,CP))) :-
  dif(A,tick),dif(A,io([],abort)),dif(A,io([],commit)),trans(P,A,P1),
  compensate_action(A,RenLst,RA), RA = io(V,CH).
trans(compensate(P,RenLst,CP),A,compensate(P1,RenLst,CP)) :-
  dif(A,tick),dif(A,io([],abort)),dif(A,io([],commit)),trans(P,A,P1),
  \+(compensate_action(A,RenLst,_)).
trans(compensate(P,_,_),tick,omega) :- trans(P,tick,_).
trans(compensate(P,_,CP),io([],abort),CP) :- trans(P,io([],abort),_).
trans(compensate(P,RenLst,_),io([],commit),compensate(P1,RenLst,skip)) :-
   trans(P,io([],commit),P1).
```

In the context of our work with industrial partners, we have also been able to write an interpreter for the high-level PROFORMA language, which is being used for clinical decision making. All this underlines our claim that logic programming is a very good basis to implement, and experiment with (domain specific) specification languages. Furthermore, once that implementation is complete, one can then use many existing, generic tools to obtain features such as animation, compilation, model checking, without further implementation effort. We demonstrate this in the following sections.

4 Applying Existing Tools

4.1 Animation

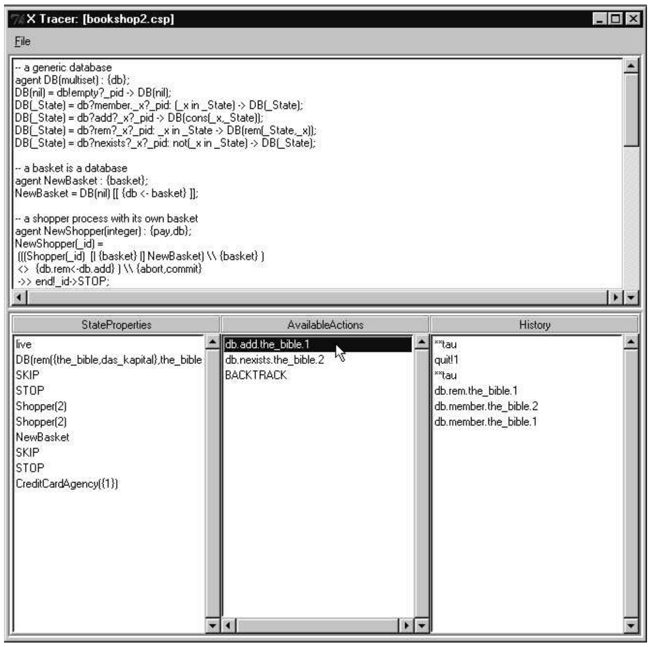
An animator was developed using the Tcl/Tk library of SICStus Prolog 3.8.4. This turned out to be very straightforward, and the tool depicted in Figure 2 was basically developed in a couple of days (and can be grafted on top of other interpreters). Part of the code looks like this, where `translate_value` converts terms into a form readable by Tcl/Tk:

```
tcltk_get_options(Options) :- current_expression(CurState),
  findall(BT, (trans(CurState,B,_NE),translate_value(B,BT)), Acts),
  (history([])-> (Options=Acts) ; (append(Acts,['BACKTRACK'],Options))).
```

The tool was inspired by the ARC tool [14] for system level architecture modelling and supports (backtrackable) step-by-step animation of the CSP specification as well as an iterative deepening search for invalid states. We plan to provide a graphical notation for CSP(LP) and also link the tool with the partial evaluators and model checkers described below.

4.2 Compiling using Partial Evaluation

As our interpreter has been written purely declaratively, we can apply many existing specialisation and analysis tools without much hassle.



```
-- a generic database
agent DB(multiset) : {db};
DB(nil) = db!empty?_pid -> DB(nil);
DB(_State) = db?member._x?_pid: (_x in _State) -> DB(_State);
DB(_State) = db?add?_x?_pid -> DB(cons(_x,_State));
DB(_State) = db?rem?_x?_pid: _x in _State -> DB(rem(_State,_x));
DB(_State) = db?nexists?_x?_pid: not(_x in _State) -> DB(_State);

-- a basket is a database
agent NewBasket : {basket};
NewBasket = DB(nil) [[ {db <- basket} ]];

-- a shopper process with its own basket
agent NewShopper(integer) : {pay,db};
NewShopper(_id) =
 (((Shopper(_id)  [| {basket} |] NewBasket) \\ {basket} )
 <>  {db.rem<-db.add} ) \\ {abort,commit}
 ->> end!_id->STOP;
```

```
live
DB(rem({the_bible,das_kapital},the_bible
SKIP
STOP
Shopper(2)
Shopper(2)
NewBasket
SKIP
STOP
CreditCardAgency({1})
```

```
db.add.the_bible.1
db.nexists.the_bible.2
BACKTRACK
```

```
**tau
quit!1
**tau
db.rem.the_bible.1
db.member.the_bible.2
db.member.the_bible.1
```

Fig. 2. Screenshot of the Animator for CSP(LP)

One potential application is the compilation of CSP(LP) code into ordinary Prolog code using partial evaluation. For example, the following is a compilation of the CSP(LP) process *BUF* (in the context of a tracing predicate *trace*) from the previous section using the ECCE online partial evaluator [22]:

```
/* Specialised program generated by Ecce 1.1 */
/* Transformation time: 183 ms */
trace(agent_call(a_BUF(keep(A))),B) :- trace__1(A,B).
trace__1(A,[]).
trace__1([A|B],[io([A],b)|C]) :- trace__1(B,C).
```

This compilation is very satisfactory, and has completely removed the overhead of CSP(LP). Further work will be required to achieve efficient, predictable compilation for all CSP(LP) programs, e.g., by using an offline specialiser such as [18]. It also seems that partial evaluation could be used to compute the so-called symbolic operational semantics of CSP-FDR (see, e.g., [20]).

4.3 Model Checking

One can directly link the interpreter either with logic programming based CTL model checking as in [27, 6], [7], [23, 21] [25] to achieve finite state model checking at no extra implementation effort.

For instance, when combining[8] our CSP interpreter with the CTL model checker of [23], we were able to check CTL formulas such as $\Box(\neg deadlock)$, $\Diamond deadlock$, $\Diamond restart$, $\Box \Diamond restart$ for the *Petri* process from Section 3.2 (without the place RC to make the state space finite):

```
check(F) :- sat(agent_call(a_MAIN),F).
| ?- check(ag(not(p(deadlock)))).
yes
| ?- check(ef(p(deadlock))).
no
| ?- check(ef(p(io([],restart)))).
yes
| ?- check(ag(ef(p(io([],restart))))).
yes
```

This task again required very little effort, and it would have been much more work to generate, e.g., a Promela encoding of CSP(LP) for use with model checker SPIN. In future work, we aim to combine our interpreter directly with an infinite state model checker, as described in [23], thus providing more powerful verification and hopefully limiting the need for users to manually abstract the system to be analysed.

5 Related Work

We are not the first to realise the potential of logic programming for animating and/or implementing high-level specification languages. See for example [3], where an animator for VERILOG is developed in Prolog, or [2] where Petri nets are mapped to CLP. Also, the model checking system XMC contains an interpreter for value-passing CCS [27,6]. Note, however, that CCS [24] uses a more low-level communication mechanism than CSP and that we provide a much richer specification language with sophisticated datatypes, functions, etc.

The idea of automatically generating compilers from interpreters by partial evaluation is not new either. Its potential for domain specific languages has been identified more recently (e.g., [5]) but has not yet made a big impact in the logic programming area. A similar approach is advocated in [11] to systematically generate correct compilers from denotational semantics specifications in logic programming. This approach was applied in [19] to verify an Ada implementation of the" Bay Area Rapid Transit" controller. The approach in [11,19] has its root in denotational semantics, while we focus on operational semantics with associated techniques such as model checking. Also, [11,19] tries to verify implementations while we try to apply our techniques earlier in the software cycle, to specifications expressed in a new higher-level language (possibly with domain specific features) but whose verification is (arguably) more tractable than for the final implementation.

[8] For this we had to slightly adapt the interpreter, as the CTL checker currently runs only in XSB Prolog.

Also, extending logic programming for concurrency is not new. Many concurrent (constraint) logic programming languages have been developed (see, e.g., [32]). Compared to these languages, we have added synchronous message passing communication via named channels, as well as many CSP-specific operators and features (such as the distinction between internal and external choice). A particular, more recent language worth mentioning is Oz [33, 12], which also integrates (amongst others) concurrency with logic programming. All of the above are real programming languages, whereas we are interested in a *specification* language suitable to rigorous mathematical inspection and verification. In other words, we want to be able to reason about distributed and concurrent systems, and not necessarily develop efficient distributed programs.

Acknowledgements

My thanks go to Laksono Adhianto for helping me set up the Tcl/Tk interface of the animation tool. I also want to thank Michael Butler, Michael Goldsmith, Ranko Lazic, Bill Roscoe, and anonymous referees for their extremely useful comments, feedback, and explanations about CSP.

References

1. J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.
2. B. Bérard and L. Fribourg. Reachability analysis of (timed) petri nets using real arithmetic. In *Proceedings of Concur'99*, LNCS 1664, pages 178–193. Springer-Verlag, 1999.
3. J. Bowen. Animating the semantics of VERILOG using Prolog. Technical Report UNU/IIST Technical Report no. 176, United Nations University, Macau, 1999.
4. M. Butler. csp2B: A practical approach to combining CSP and B. *Formal Aspects of Computing.* To appear.
5. C. Consel and R. Marlet. Architecturing software using : A methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of ALP/PLILP'98*, LNCS 1490, pages 170–194. Springer-Verlag, 1998.
6. B. Cui, Y. Dong, X. Du, N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of ALP/PLILP'98*, LNCS 1490, pages 1–20. Springer-Verlag, 1998.
7. G. Delzanno and A. Podelski. Model checking in clp. In R. Cleaveland, editor, *Proceedings of TACAS'99*, LNCS 1579, pages 223–239. Springer-Verlag, 1999.
8. C. Ferreira and M. Butler. A process compensation language. In *Proceedings IFM'2000*, LNCS 1945, pages 61–76. Springer-Verlag, 2000.
9. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement — FDR2 User Manual.*
10. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
11. G. Gupta. Horn logic denotations and their applications. In *The Logic Programming Paradigm: A 25 year perspective*, pages 127–160. Springer-Verlag, April 1998.
12. S. Haridi, P. Van Roy, P. Brand, and C. Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.

13. P. Hartel, M. Butler, A. Currie, P. Henderson, M. Leuschel, A. Martin, A. Smith, U. Ultes-Nitsche, and B. Walters. Questions and answers about ten formal methods. Technical report, Trento, Italy, July 1999. Extended version as technical report DSSE-TR-99-1, University of Southampton.
14. P. Henderson. Modelling architectures for dynamic systems. Technical report, University of Southampton, December 1999.
Available at `http://www.ecs.soton.ac.uk/~ph/arc.htm`.
15. C. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
16. G. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.
17. D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proc. International Conference on Software Engineering*, Limerick, Ireland, June 2000.
18. J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 238–262, Schloß Dagstuhl, 1996. Springer-Verlag.
19. L. King, G. Gupta, and E. Pontelli. Verification of Bart controller: An approach based on horn logic and denotational semantics. submitted, 2000.
20. R. S. Lazic. *A semantic study of data-independence with applications to the mechanical verification of concurrent systems.* PhD thesis, Oxford University, 1997.
21. M. Leuschel and H. Lehmann. Solving coverability problems of Petri nets by partial deduction. In M. Gabbrielli and F. Pfenning, editors, *Proceedings of PPDP'2000*, pages 268–279. ACM Press, 2000.
22. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.
23. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, September 1999.
24. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
25. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model checking. In J. Lloyd, editor, *Proceedings of CL'2000*, LNAI 1861, pages 384–398, London, UK, 2000. Springer-Verlag.
26. J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. In G. Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 246–261, Pisa, Italy, September 1998. Springer-Verlag.
27. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In O. Grumberg, editor, *Proceedings of CAV'97*, LNCS 1254, pages 143–154. Springer-Verlag, 1997.
28. A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *IEEE Symposium on Foundations of Secure Systems*, 1995.
29. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1999.
30. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
31. J. B. Scattergood. *Tools for CSP and Timed-CSP.* PhD thesis, Oxford University, 1997.
32. E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

33. G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, LNCS 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

A Simple E-bookshop in CSP(LP)

```
-- a generic database
agent DB(multiset) : {db};
DB(nil) = db!empty?_pid -> DB(nil);
DB(_State) = db?member._x?_pid: (_x in _State) -> DB(_State);
DB(_State) = db?add?_x?_pid -> DB(cons(_x,_State));
DB(_State) = db?rem?_x?_pid: _x in _State -> DB(rem(_State,_x));
DB(_State) = db?nexists?_x?_pid: not(_x in _State) -> DB(_State);

-- a basket is a database
agent NewBasket : {basket};
NewBasket = DB(nil) [[ {db <- basket} ]];

-- a shopper process with its own basket
agent NewShopper(integer) : {pay,db};
NewShopper(_id) =
 (((Shopper(_id)  [| {basket} |] NewBasket) \\ {basket} )
 <>  {db.rem<-db.add} ) \\ {abort,commit}
 ->>> end!_id->STOP;

agent Shopper(integer) : {pa,db,basket,checkout,quit};
Shopper(_id) = db!member!_x!_id ->
    ((db!rem._x!_id -> basket!add._x!_id -> Shopper(_id))   []
     (db!nexists._x!_id -> Shopper(_id))  );
Shopper(_id) = checkout!_id -> Payer(_id);
Shopper(_id) = quit!_id -> abort -> SKIP;

agent Payer(integer) : {pay,db,basket};
Payer(_id) =
 ((pay!_id?ok -> commit -> SKIP)  []
  (pay!_id?ko -> db!add!_x -> abort -> SKIP)  );
Payer(_id) = basket?empty -> SKIP;

-- a simple credit card agency
agent CreditCardAgency(multiset) : {pay};
CreditCardAgency(_DB) = pay?_id!ok:(_id in _DB) -> CreditCardAgency(_DB);
CreditCardAgency(_DB) =
        pay?_id!ko:not(_id in _DB) -> CreditCardAgency(_DB);

-- a test system with 2 shoppers and 2 books
agent MAIN : {db,pay};
MAIN = (DB(cons(the_bible,cons(das_kapital,nil)))
       [| {db} |]
       ( NewShopper(1) |||  NewShopper(2)
       )) [| {pay} |] CreditCardAgency(cons(1,nil));
```