A Preliminary Report on xMECH

A. Azurat and **I.S.W.B. Prasetya**
Institute of Information and Computing Sciences
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands
{*ade,wishnu*}*@cs.uu.nl*

Abstract

This document reports the *current development status* of xMECH. It is an implementation of the so-called skin or hybrid embedding approach [1] for HOL. Its purpose is to enhance HOL's power and interaction to do software verification. xMECH features languages and logics to describe and verify sequential and distributed programs, a reasonably rich expression language to write specifications, and optimized verification condition generators. It is available for public use, but it is still in a prototype phase, with limited features and user support. It comes with some simple demos, but doing a serious project with xMECH is not (yet) recommended for an inexperienced user.

1 Introduction

Despite the advance of computer aided technology, mechanical verification of realistic programs remains a challenge. Many programs are parameterized by unbounded constants, or require sophisticated data structures to specify. These are things that put them beyond the reach of even the most sophisticated model checkers. There are powerful mechanized logics such as HOL that can deal with infinite and sophisticated data structures, but they pose another restriction, namely that the actual programming logic we want to use has first to be embedded into them. Embedding, however, has its own problems (e.g. problems with readability and problems with embedded type system) [1] and are labour intensive.

1.1 The xMECH Concept

In the multimedia world, a skin is a program that runs on top of a primary application to change the outer look of the primary application. This is a good analogy for xMECH, which is a skin for a well established general purpose verification tool called HOL. However, the purpose of xMECH is not to give HOL a different physical look, but to give it various adds-on verification devices to make verification with HOL more convenient.

xMECH is built around HOL, thus giving xMECH user access to HOL's features:

1. HOL is based on a familiar logic (predicate logic).
2. HOL logic is higher order, so it is very expressive. It can easily express infinite state spaces and sophisticated data structures.
3. HOL has a rich collection of mathematical theories and proof tools.
4. HOL comes with a powerful proof scripting language (ML).

To circumvent problems encountered in (pure) embedding, xMECHuses an approach called Hybrid Embedding[1]. The approach has a more pragmatic view towards mechanical verification. So, parts of pure embedding which are problematic are simply not embedded. Instead, they are directly implemented. We implement them in ML, because it will make interfacing with HOL much easier, since HOL is also written in ML. Because some parts of the proof machinery are not embedded, xMECH cannot provide unbroken correctness guarantee to proofs the way HOL does [3]. In exchange, it strives to provide an environment that facilitates rapid composition of proofs of realistic programs.

1.2 Features

To express programs and their properties xMECH offers a number of programming and specification languages, built stratifiedly, together with some logics to reason about them:

1. EXPR: a quite rich language of expressions used to specify (the set of) possible states a program can be in a given moment. Currently, it has functions, tuples, arrays, and lists. The type system is simple but for many purposes sufficient, allowing polymorphism and higher order functions.

2. PLSEQ: a programming language and logic for sequential programs. Programs are specified using Hoare triples. PLSEQ uses EXPR to specify pre- and postconditions, so it is quite expressive in what it can specify.

3. PLGA: a programming language and logic of guarded actions. Guarded actions models concurrent actions. PLGA has lots of PROMELA[1] flavour in it. Programs are specified using UNITY temporal operators. The logic of PLGA is UNITY logic [2], which we favour because of its simplicity and its high level axiomatic style.

PLGA is intended to be intermediate. It is the core of PLPC, a programming language and logic of process classes, which is currently still unfinished. The language of PLPC extends PLGA with sections to define the structure of a distributed system and the interface of the system to its environment. The logic of PLPC is based on an extension of UNITY that includes necessary laws to deal with program composition [7].

As said, xMECH is a 'skin' built around HOL. The core semantics of the logics used by PLGA is embedded in HOL. The syntactic parts (grammars, parsers, and pretty printers) of the above languages provided by xMECH and their semantic functions (to translate to the HOL semantics) are part of the skin, and hence they are not embedded. In addition, the skin also provides utilities such as:

1. Type checking.

2. A verification condition generator for sequential programs (PLSEQ) specifications (based on wp calculation).

3. Some simplification algorithms, such as an algorithm to a DNF normalization on a formula and an algorithm to, given a PLGA program, remove as many actions as possible, which at a specified 'duration' during the program's executions are disabled (so these actions do not actually take part in the computation and hence can be safely removed).

1.3 Download

xMECH prototype is publicly available at `www.cs.uu.nl/~ade/xmech`.

[1]PROMELA is the input language for SPIN model checker. See [5]

```
expr  →     program-variable
        |   literal
        |   pair
        |   concrete-list
        |   function-application
        |   array-access-expression
        |   expr  infix expr
        |   existential-quantification
        |   universal-quantification
        |   ( expr )
```

Figure 1: The general syntax of EXPR

1.4 Using xMECH

Basically xMECH is working alongside with the HOL environment. Just like HOL, all instructions are executed interactively. A program and its specification is read by xMECH. In principle, once read, a specification can be manipulated interactively in xMECH. Currently we only provide a very limited form of xMECH interaction. We hope to provide more interactive transformation commands in future versions – until then a power user should program his own commands. In the current version, the user can only call the verification condition generator for either PLSEQ or PLGA. This will analyze the specification, apply some simplification algorithms, and generate EXPR level verification conditions. The latter are then translated to HOL and set as a HOL goal. From that point on the user will enter the HOL environment and finish the proof in HOL. Consequently, the user should already be familiar with the use of HOL.

In Sections 4 and 5, we will show examples of PLSEQ and PLGA specifications and the corresponding xMECH sessions –complete input files containing those examples and their proof script files can be found in the xMECH distribution.

EXPR is represented by a data structure which is more abstract than the λ calculus representation chosen by, for example, HOL. This makes, the programming of syntax driven analyses and transformations on EXPR sentences considerably easier.

2 EXPR

Unlike other tools, e.g. Spin or STeP, xMECH comes with a quite expressive language of expressions called EXPR. In a programming language, an expressions is typically used at the right hand side of assignment and as a guard of a conditional or a loop. Since xMECH is a verification tool, it also allows us to specify the properties of a program. EXPR is also used in specifications, for example to specify the pre- and post- condition of a sequential program. It is polymorphic and higher order. Using it, we can express quite complicated specifications. Note that the program being specified itself does not need to be polymorphic or higher order. It may however implement a property that requires a sophisticated expression language to describe.

The general syntax of EXPR is shown in Figure 1.

Some of the syntax of EXPR is awkward, such as function application and array access –we do not have time yet to polish the syntax, but this should be fixed in the future. A similar remark holds for PLSEQ and PLGA.

- Boolean literals are `T` and `F`. Integer literals are, e.g., 1, 0, and -1. A string literal is written in quote, e.g. `"hello"`.
- An empty list is written `-[]-`. A concrete non-empty list is written, e.g., like `-[a,b,c]-`. Lists can be nested.
- A tuple is written, e.g., like `(x,y,z)`. Tuples can be nested.
- An array expression `a#n` returns the `n`-th element of the array `a`.

- A function application is written, e.g., like `f>>x`. In PLGA/PLPC we can also write it as `(f x)`.
- An existential quantification is written, e.g. like:

  ```
  (?i,j: 0<i /\ 0<j : i=j)
  ```

  A universal quantification is written in the same way, except that `!` is used instead of `?`.
- Table 2 lists the binary operators (infixes) that currently are recognized in EXPR. Some of the less common are, e.g.: `x::s` (the list consisting of `x` as the head and `s` as the tail); `s!!i` (the `i`-th element of the list `s`; `s++t` (the list `t` appended to `s`).
- A *meta identifier* is a constant that is not part of the program being specified by xMECH. Typically, this is a function such as the `length` function that returns the length of a list. The prefixes in Table 2 are the meta identifier currently known to EXPR.

  When parsing a program (PLSEQ, PLGA, or PLPC), an identifier which does not appear in the list of known meta identifiers will be looked upon by xMECH as a program variable.

EXPR basically allows expansion, but currently we have no support to do this conveniently. Not recommended for an inexperienced user.

2.1 Expanding EXPR

One can extend EXPR by adding new meta identifiers, for example, more list functions. It is important to realize that EXPR is just a language interface. So it does not care about the meaning of an expression, though it still needs to know, for example, the names of the meta identifiers. When xMECH needs to prove that a given expression asserts truth, it will send the expression to HOL and ask you to prove it there. In particular, HOL should know the meaning of all your meta identifiers. Therefore, when you add a new meta identifier to EXPR, make sure its meaning is already defined in HOL.

The file `exConst.sml` specifies the list of meta identifiers known to xMECH. To add a new meta identifier, you add a new entry in the list specifying the name of the meta identifier, its type, and its translation to HOL.

New infixes cannot currently be added without hacking into xMECHparsers.

It seems possible, by exploiting ML functor mechanism, to provide multiple type systems and to let the user chooses one at the run time.

3 TYPE

An expression (a sentence from EXPR) must be well typed. xMECH has a simple type system called TYPE which basically is similar to that of HOL. It supports polymorphism. TYPE has three primitive types: `bool`, `int`, and `string`. A type variable is written as any alpha-numeric sequence which is the name of a primitive type. It must start with a roman character. So, e.g., `a` or `ab` or `a1` can be used as type variables.

The following type constructors are available. The notations used by PLSEQ and PLGA differ slightly.

PLSEQ	PLGA/PLPC	meaning
`a->b`	`a->b`	function from `a` to `b`
`[a]`	`a list`	list over `a`
`[]a`	`a[n]`	array over `a` (in PLPC: of size `n`.)
`a#b#c`	`a#b#c`	the product type of `a`, `b`, and `c`

symbol	type (in PLSEQ notation)	fixity
`true`	bool	constant
`false`	bool	constant
`=`	a → a → bool	infix
`\/`	bool → bool → bool	infix
`/\`	bool → bool → bool	infix
`==>`	bool → bool → bool	infix
`~`	bool → bool → bool	prefix
`?`	string → bool → bool → bool	prefix
`!`	string → bool → bool → bool	prefix
`<`	a → a → bool	infix
`<=`	a → a → bool	infix
`>`	a → a → bool	infix
`>=`	a → a → bool	infix
`=`	a → a → bool	infix
`+`	a → a → a	infix
`*`	a → a → a	infix
`^`	a → int → a	infix
`/`	a → a → a	infix
`mod`	int → int → int	infix
`div`	int → int → int	infix
`-[]-`	[a]	constant
`hd`	[a] → a	prefix
`tl`	[a] → [a]	prefix
`::`	a → [a] → [a]	infix
`#`	[a] → int → a	infix
`++`	[a] → [a] → [a]	infix
`elem`	a → [a] → bool	prefix
`map`	(a→ b)→ [a] → [b]	prefix
`filter`	(a→ bool) → [a] → [a]	prefix
`concat`	[[a]] → [a]	prefix
`length`	[a] → int	prefix

Table 1: Predefined Symbols in EXPR

These are the types supported by xMECH parsers. The representation of TYPE actually allows arbitrary type primitives and constructors:

```
datatype Type  =  TyVar  of string
               |  TyCons of string * Type list
```

`TyVar a` represents a type variable called `a` and, for example, `TyCons "list" t` represents the type of list of `t`.

By constructing the representation directly the user can construct any type allowed by the representation. Since in verification expressions are sent to HOL, each type being used in the expressions must be a valid HOL type.

xMECH currently does not support in-line type information. So we cannot write something like `((x:int) = y) ==> b` as in ML and HOL. However, it has a type inference system, so you can write `x>0` and xMECH will recognize `x` as a program variable of type `int`.

When a PLPC program P is parsed, a type context for P is constructed. It lists the variables of P and their types. This context is needed by xMECH to type check P and its specifications. PLSEQ and PLGA do not currently have declaration sections, therefore, the user will have to construct the type context by hand, see Example 4.2:

4 PLSEQ

PLSEQ stands for Programming Language and logic for SEQuential programs. Currently, it is still a very simple language. It has sequential composition, conditionals, and loops. PLSEQ is originally intended to be used to specify actions in PLGA, but we can also use PLSEQ add-ons as a stand-alone environment for verifying PLSEQ programs. Supports for stand-alone use of PLSEQ is however still rather primitive, so some programmings to set up the environment is needed. This should be improved in the public release.

PLSEQ commands looks very much to C commands, though it allows a much richer expression language (EXPR). A rich expression language is needed to formulate a program's specification. Any EXPR sentence can also be used in the program itself, e.g. as guards of conditionals and as the right hand side of assignments. It means that PLSEQ programs can also be very abstract, and hence are also suitable to describe abstract models.

The types supported by PLSEQ are the same as the types of TYPE.

PLSEQ also supports array. Currently, only unbounded integer indexed arrays are supported. Accessing the `n`-th element of an array `a` is written as `a#n`.

No assignment to an array element is currently supported. Note that this will require the logic to be extended with rules to reason about aliasing.

4.1 Program

Here is a simple PLSEQ program to search for an integer 0 in an array of N integers. It sets a boolean variable `b` to `T` if the search is successful, otherwise `b` is set to `F`. The program exits its loop as soon as a 0 is found.

Code 4.1 : Linear Search with Break

```
{ n=0                                    ;
  b=F                                    ;
  while (~b /\ ~(n=N))
        if (a#n=0)  b=T  else n=n+1      ;
}
```

□

1. The wps:

$$\begin{array}{lcl}
\mathsf{wp}\ (x \ \texttt{:=}\ E)\ Q & = & Q[E/x] \\
\mathsf{wp}\ (\texttt{if}\ g\ \texttt{then}\ A\ \texttt{else}\ B)\ Q & = & (g \Rightarrow \mathsf{wp}\ A\ Q) \wedge (\neg g \Rightarrow \mathsf{wp}\ B\ Q) \\
\mathsf{wp}\ (\texttt{inv}\ I\ \texttt{while}\ (\ g\ )\ A)\ Q & = & I \\
\mathsf{wp}\ \{\ \}\ \ Q & = & Q \\
\mathsf{wp}\ \{\ A\ ;\ B\ \}\ Q & = & \mathsf{wp}\ A\ (\mathsf{wp}\ \{\ B\ \}\ Q)
\end{array}$$

2. The inference rules:

   (a) For all PLSEQ statement A except loops we have:

$$\frac{[P \Rightarrow \mathsf{wp}\ A\ Q]}{\{P\}\ A\ \{Q\}}$$

   (b) For the `while` loop we have:

$$\frac{\begin{array}{l}[P \Rightarrow \mathsf{wp}\ (\texttt{inv}\ I\ \texttt{while}\ (\ g\ )\ A)\ Q] \\ \{I \wedge g\}\ A\ \{I\} \\ [I \wedge \neg g \Rightarrow Q]\end{array}}{\{P\}\ \texttt{inv}\ I\ \texttt{while}\ (\ g\ )\ A\ \{Q\}}$$

Figure 2: PLSEQ inference rules

Note that `;` is used as a terminator, not a separator.

See Appendix A for more language constructs supported by PLSEQ.

Currently PLSEQ supports no declaration part to specify the names and types of the variables of a program. These have to be specified by hand, e.g. as follow:

Code 4.2 : CONTEXT DECLARATION

```
val varContext_ofDemoProgram =
    let
    val p = TypePara.parseInLine ;
    in
    [("b",   p "bool") ,
     ("n",   p "int")  ,
     ("N",   p "int")  ,
     ("a",   p "[]int" )]
    end
```

□

4.2 Logic

PLSEQ logic is a simple Hoare logic. Currently, only partial correctness [4] is supported. PLSEQ inference rules are shown in Figure 2. If the program contains loops, xMECH requires the user to specify the invariant of each loop.

4.3 Specification

The following code shows the linear search program again, but now annotated with the pre/post-conditions specifying what the program is supposed to do. It also specifies the invariant of the loop in the program.

Code 4.3 : LINEAR SEARCH WITH BREAK –ANNOTATED CODE

```
{= 0<=N =}

{ n=0                                         ;
```

```
      b=F                                 ;

      inv    (! i: 0<=i /\ i < n : ~(a#i=0))
             /\  0<=n /\ n<=N
             /\ (b ==> (a#n=0)) /\ (b ==> n < N)

      while (~b /\ ~(n=N))
             if (a#n=0)  b=T  else n=n+1          ;
    }

    {= b = (? i: 0<=i /\ i < N : a#i=0) =}
```

□

4.4 Verification

Annotated code such as the one above is the input of xMECH. Before reading this input, the user must first start HOL. From HOL, execute the following ML command to load the required xMECH-PLSEQ add-ons (or it can be put in the script file):

Code 4.4 : PLSEQ Environment Loading

```
val xmech_sigobj_loc  = "./";
val env_loc  = "./EXAMPLES/PLGA/";

loadPath := xmech_sigobj_loc :: !loadPath ;
use (env_loc^"PLSEQ_environment.sml");
```

□

Now we can read the specification and verify it:

Code 4.5 : Script file for Linear Search With Break example

```
(*  File name : linearSearchWithBreakDemo.sml  *)

val varContext_ofDemoProgram =
        let val p = TypePara.parseInLine in
        [("n", p "int"), ("b", p "bool"),
         ("N", p "int"), ("a", p "[]int")] end ;

(* reading the program now.... *)
val theProgram = parseProgram varContext_ofDemoProgram
                 "linearSearchWithBreak.plseq" ;

(* generating the verification conditions *)
val generatedVCs = genVerificationConditions theProgram ;

(* Starting interactive HOL proof now ... *)
set_goal([], translate2HOL generatedVCs) ;

(* try to simplify the expression *)
e (RW_TAC std_ss [ARRAYREAD_def]
   THEN TRY (UNDISCH_ALL_TAC THEN COOPER_TAC)) ;

(* results in six goals... *)

        (* subgoal 1 *)
```

```
(0 <= N ==>
        ((((!i. 0 <= i /\ i < 0 ==> ~(ARRAYREAD i a = 0)) /\ 0 <= 0) /\
          0 <= N) /\ (F ==> (ARRAYREAD 0 a = 0))) /\ (F ==> 0 < N)) /\
        ((~b /\ ~(n = N)) /\
         ((((!i. 0 <= i /\ i < n ==> ~(ARRAYREAD i a = 0)) /\ 0 <= n) /\
           n <= N) /\ (b ==> (ARRAYREAD n a = 0))) /\ (b ==> n < N) ==>
         (((ARRAYREAD n a = 0) ==>
           ((((!i. 0 <= i /\ i < n ==> ~(ARRAYREAD i a = 0)) /\ 0 <= n) /\
             n <= N) /\ (T ==> (ARRAYREAD n a = 0))) /\ (T ==> n < N)) /\
          T) /\
         (~(ARRAYREAD n a = 0) /\ T ==>
          ((((!i. 0 <= i /\ i < n + 1 ==> ~(ARRAYREAD i a = 0)) /\
             0 <= n + 1) /\ n + 1 <= N) /\
           (b ==> (ARRAYREAD (n + 1) a = 0))) /\ (b ==> n + 1 < N))) /\
        (~(~b /\ ~(n = N)) /\
         ((((!i. 0 <= i /\ i < n ==> ~(ARRAYREAD i a = 0)) /\ 0 <= n) /\
           n <= N) /\ (b ==> (ARRAYREAD n a = 0))) /\ (b ==> n < N) ==>
         (b = ?i. (0 <= i /\ i < N) /\ (ARRAYREAD i a = 0))) /\ T
```

Figure 3: Generated verification conditions of the program in Code 4.3

```
25   val lemma1 = COOPER_PROVE (--`~((n:int)=N) /\ n<=N ==> n<N`--) ;
26   e(PROVE_TAC [lemma1]) ;
27
28           (* subgoal 2 *)
29   val lemma2 = COOPER_PROVE (--`(i:int)<n+1 = i<n \/ (i=n)`--) ;
30   e(PROVE_TAC [lemma2]) ;
31
32           (* subgoal 3 *)
33   val lemma3 = COOPER_PROVE (--`0<=(n:int) ==> 0 <= n+1`--) ;
34   e(PROVE_TAC [lemma3]) ;
35
36           (* subgoal 4 *)
37   val lemma4 = COOPER_PROVE (--`~((n:int)=N) /\ n<=N ==> n+1<=N`--) ;
38   e(PROVE_TAC [lemma4]) ;
39
40           (* subgoal 5 *)
41   e(PROVE_TAC []) ;
42
43           (* subgoal 6 *)
44   val lemma6 = COOPER_PROVE (--`~((N:int)<N)`--) ;
45   e(PROVE_TAC [lemma6]) ;
```

□

Line 9 loads the input file. Lines 3-6 load the context of the specification. This context is a piece of ML code specifying the names of the PLSEQ program variables used in the loaded specification along with their types. The use of this separated context description is temporary since we have not implemented a declaration section in PLSEQ yet. Line 13 calls a verification condition generator (VCG), resulting in a list of EXPR formulas which are then combined as a conjunction and translated to HOL and set as the proof goal. From this point on, the proof proceeds in HOL. The VCG exhaustively applies the inference rules of the Hoare Logic (Figure 2).

The resulting goal in HOL is displayed in Figure 3. The goal is quite large, but it does no longer contain any PLSEQ structure. In fact, it asserts a property purely over the data which are the object of the original program –in this case the data are all integers and booleans. HOL is very suitable for solving this kind of goal. First of all though, we want to simplify it. There are in fact many trivial sub-formulas which can be removed. Lines 19 from the interactive session in Code 4.5 first call HOL equational simplifier and then tries to solve the resulting new goals with HOL's solver for integer arithmetics (`COOPER_TAC`).

`COOPER_TAC` eliminates some sub-goals, but not all (all known decision procedures on number arithmetics are incomplete). We still have some remaining sub-goals (six). We can try to write a HOL script to try to automate the remaining proof, but in this case six sub-goals are few, and they are all also quite simple. We simply analyze them one by one, and decide which properties of integer arithmetic are needed to prove the goal. For example, lines 25-26 prove the first sub-goal, which looks like this:

```
   n < N
  ------------------------------------
   0.  ~b
   1.  ~(n = N)
   2.  !i. 0 <= i /\ i < n ==> ~(a i = 0)
   3.  0 <= n
   4.  n <= N
   5.  b ==> (a n = 0)
   6.  b ==> n < N
   7.  a n = 0
```

The goal can be proven if the following property is a theorem in HOL:

```
~(n=N) /\ n<=N ==> n<N
```

This property is quite simple. Assumptions number 1 and number 4 assert the above property, then call the COOPER prover to prove it, then use the resulting theorem to prove the goal quoted above.

The trick is repeated for the other sub-goals. They are proven in lines 29-45. The proof of each is very short, but notice that each sub-goal requires different help theorems. If more automation is desired, from HOL we have access to ML which serves as a powerful proof scripting language. For example, we can try to guess a list of lemmas which would enable `PROVE_TAC` to finish the remaining sub-goals after the step in lines 16. Code 4.6 shows an alternative proof script.

Code 4.6 : Alternative proof script

```
RW_TAC std_ss [ARRAYREAD_def]
THEN TRY (UNDISCH_ALL_TAC THEN COOPER_TAC)
THEN TRY (PROVE_TAC lemmas_selection)
```

□

`PROVE_TAC` is a first order predicate logic prover. It is quite powerful. The success of the above tactic depends on the selection of the lemmas. For example, using exactly the same lemmas as used in Code 4.5, the tactic will prove the initial goal without further assistance.

5 PLGA

PLGA stands for Programming Language of Guarded Actions. It is a language for specifying distributed systems. A PLGA program is a process in a distributed system. A system is specified by another language called PLPC, but currently the PLPC environment is not implemented yet. We can still use the logic of PLGA to PLGA programs in isolation. Moreover, PLGA allows concurrent actions and thus can be used to model a distributed system, though in a less abstract way than PLPC would.

PLGA allows types in TYPE and uses EXPR to describe expressions.

5.1 Program

Like PLSEQ, it has sequential composition, conditionals, and loops, but the style is very different. PLGA tries to be as close possible to PROMELA [5] because the latter is an established language to describe distributed systems and is supported by a powerful model checker called SPIN [5].
Here is a simple example of a PLGA program:

Code 5.1 :

```
do
:: (a>0)  -> x=x+1
:: (a=0)  -> atomic{x=x- 1; y=y+1;}
:: else block
od
```

□

This program goes into an infinite loop. If `a>0`, it performs the assignment `x=x+1`. If `a=0`, it performs the sequence `x=x-1; y=y+1` in a single atomic execution. If `a<0`, the loop simply blocks until either of the other two conditions becomes enabled. Since the above program does not write to `a`, if the loop blocks it cannot unblock itself. However, if put in a system with other PLGA programs, some other program may change the value of `a` and unblock the above loop.

5.2 Block and Exit Flags

The `else block` flag in the Code 5.1 means that the loop will block if no guards are enabled. Similarly, we can put the same flag in a conditional statement:

Code 5.2 :

```
if
:: (a>0)  -> x=x+1
:: (a=0)  -> atomic{x=x- 1; y=y+1;}
:: else block
fi
```

□

If none of the guards are true when the conditional is entered, it will block. If `a>0` becomes true the conditional *may* become unblocked. The corresponding action `x=x+1` will be executed, and then the conditional is exited. Note that in the existence of other processes, the condition `a>0` may not hold long enough and the above program may fail to see it. Only if the condition `a>0` remains to hold then the action `x=x+1` is guaranteed to be executed and can the conditional be exited.

We can also have a `else exit` flag. In this case the loop or conditional *may* be exited if all the guards are false. Only if the guards remain false then the loop (conditional) is guaranteed to be eventually exited. The use of `else` flags is obligatory and it is always the last branch of a `do` or an `if` constructs.

5.3 Atomic Actions and Fairness

An atomic action a is a statement or command the execution of which cannot be interfered (at least not by other programs accessing the same set of variables accessed by a). A distributed system is a concurrent system. Execution of such a system can be modelled by interleaved executions of the system's atomic actions.

This is a quite standard model of concurrency, e.g. as in [2]. In this model, we can think that the execution of a PLGA program, in the context of a system, can be interrupted at any time by other programs. There is of course a certain fairness constraint put on the underlying implementation.

An assignment is atomic. PLSEQ can be used to describe a more complicated atomic action. It should be preceded by the `atomic` keyword. If the PLSEQ program contains loops, they are assumed to terminate.

Currently, termination proofs of PLSEQ programs are still unsupported by xMECH.

An atomic action A can be guarded. The syntax is g `->` A. If g holds, the action is enabled and A is executed. If g does not hold the action is not enabled, and thus cannot be executed. We also say that in the latter case the action is *blocked* or *disabled*. An action cannot be continually enabled without eventually being executed. This is to impose fairness in the execution of a system of PLGA programs. This model of fairness is the same as used by UNITY [2] and is usually called *weak fairness*.

In case that A is not atomic, for example if it is:

```
{x=x-1 ; y=y+1 ;}
```

then the meaning of:

```
(a=0) -> {x=x- 1 ; y=y+1 ;}
```

is the same as:

```
{(a=0) -> skip ; x=x- 1 ; y=y+1;}
```

5.4 Labels and Jumps

Because the execution of a PLGA program can be interrupted, it is very useful to be able to specify the state of a program when it is at a certain control location (program location). For this purpose PLGA statements in a program can be labelled to mark the control location of the program when the corresponding statement is entered.

With a `goto` statement we can also jump directly to a given control location. Jumps are useful in implementing, for example, automata. Here is an example of a program with labels and jumps:

Code 5.3 :

```
{[start_label]:
 do
 :: (a>0)  -> x=x+1
 :: (a=0)  -> {x=x-1 ; y=y+1 ;}
 :: (x=y)  -> goto exit_label
 :: else block
 od ;
 [exit_label]: done = T ;
}
```

□

The above program will block if `a<0 /\ ~(x=y)`. It may jump to the label `exit`, and thus exiting the loop. But this only happens if `x=y`.

5.5 Internal Concurrency

When used to specify a component of a distributed system, we typically use a sequential PLGA program. However, a PLGA program can also be concurrent. This is useful to specify a sequential component with limited internal concurrency. We can also use a PLGA program to model an entire distributed system. Here is an example:

Code 5.4 :

```
do
:: (x>y)  -> x=x- 1
:: (x>y)  -> y=y* 2
:: (x<y)  -> x=x* 2
:: (x<y)  -> y=y- 1
:: else block
od
```

□

For example, when `x>y` two guards are enabled, and hence there are two possible branches to continue. One will be picked non-deterministically. In the interleaving model of concurrency, the resulting execution of the above `do` statement is the same as the execution of a distributed system with four processes with process i does an eternal loop in which at each iteration the i-th (guarded) action from the above `do` is attempted.

5.6 Logic

PLGA uses UNITY logic to reason about programs. It is a simple and elegant logic by Chandy and Misra [2] to reason about distributed systems. The logic works at the level of atomic actions. For example, in UNITY we say that a predicate J is a stable in a program P if it cannot be falsified by any action in P. So to verify the stability of J in P we must verify the Hoare triple $\{J\}\ a\ \{J\}$ for any action a in P. The verification condition generator (VCG) of PLGA will generate the set of atomic actions of a given program and produce the corresponding Hoare triples. Because an atomic action is either a simple assignment or a PLSEQ structure, we can subsequently call the VCG of PLSEQ to further process the results of the VCG of PLGA. The VCG of PLGA also performs an analysis to try to eliminate disabled actions from its consideration.

UNITY provides some operators to specify the (temporal) property of a program, namely `stable`, `unless`, `ensures`, `invariant`, and `leadsto`. The logic for the last two operators are not fully implemented yet. The inference rules for the other operators are shown in Figure 4.

5.7 Example: a Simple One Bit Protocol

As a more involved example, consider the following protocol. It is a simpler variant of the one bit sliding window protocol.

The idea is to send the value of `x` from the sender side to the receiver side. The model consists of three distributed actions, which are `update`, `send` and `receive`. The action `update` updates the value of `x`. The action `send` copies `x` to `c`. The latter models the data channel between the sender and receiver. The `receive` copies `c` to `y` which then completes a protocol cycle. These actions are synchronized by two booleans variables: `senderReady` and `recReady`. Roughly speaking, the sender is

1. Stable Rule:

$$\frac{\{J\}\ a\ \{J\} \quad \text{, for } \textit{all} \text{ action } a \text{ in } \mathbf{ac}P}{\texttt{|--}\ \texttt{stable}\ J}$$

2. Unless Rule:

$$\frac{\{p \wedge \neg q\}\ \ a\ \ \{p \vee q\} \quad \text{, for } \textit{all} \text{ action } a \text{ in } \mathbf{ac}P}{\texttt{|--}\ p\ \texttt{unless}\ q\ \texttt{using}\ V}$$

3. Ensures Rule:

$$\frac{\begin{array}{l}\texttt{|-}\ p\ \texttt{unless}\ q\ \texttt{using}\ V \\ \{p \wedge \neg q\}\ \ a\ \ \{p \vee q\} \quad \text{, for } \textit{some} \text{ action } a \text{ in } \mathbf{ac}P\end{array}}{\texttt{|--}\ p\ \texttt{ensures}\ q\ \texttt{using}\ V}$$

where

- **ac**P denotes the set of atomic actions of the program P.
- The V parameter is a list of variable names. It specifies variables which may be written by the program during the specified behaviour. This parameter is currently ignored –it is needed in an extension of the UNITY logic as described in [7]. The parser still requires it to be present. Just write a non-empty dummy V, for example `{a}`.

Figure 4: PLGA inference rules

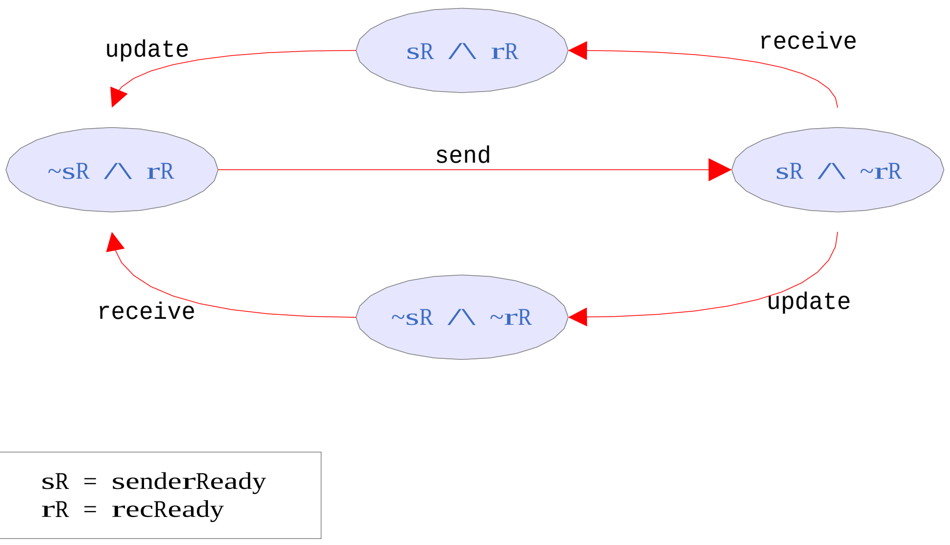

Figure 5: State Diagram of One Bit Protocol without assertion label

allowed to update `x` if `senderReady` holds, whereas `recReady` means the receiver is ready to accept a new data package –see Figure 5.

When PLPC is ready, we can model the sender and the receiver as two separate PLPC process. We use another PLPC process to specify the system that consist of the sender and receiver.

Notice that the protocol is sequential in most situations, except when `senderReady` and `~recReady` hold, where the protocol is non-deterministic.

Below is a PLGA program modelling of the protocol[2]. Notice that the program is a direct translation of the automaton graphically displayed in Figure 5.

Code 5.5 : One Bit Protocol

```
Process OneBitProtocol ()
private: bool senderReady,recReady ;
main:
{ [root] :
do
:: (* UPDATE --it is modelled as non-atomic *)
        { (senderReady) -> x = x + 1            ;
          [mid] : senderReady=F                 ;
        }
:: (* SEND action --atomic*)
        (recReady /\ ~senderReady)
        ->
        atomic {c = x ; recReady=F ; senderReady=T ; }
:: (* RECEIVE --atomic*)
        (~recReady)  -> atomic {y = c ; recReady=T ; }
:: else block
od                                     ;
(* this statement will never be executed *)
[unreachable] : y = y - 1              ;
}
```

□

Notice that the `send` and `receive` actions are modelled as atomic actions, whereas the `update` action is not. Modelling all protocol's actions as atomic will only simplify its verification. However, we deliberately break the atomicity of the `update` action, as an extra challenge to investigate whether the protocol will still work correctly.

To demonstrate some features of the VCG of PLGA we also add the action `y=y-1`. This action is actually unreachable, and thus always disabled. The user will see that in this case the VCG is smart enough to detect the situation, and thus will discard the action and hence reducing the number of generated verification conditions.

5.8 Specifying Safety

Suppose we want to verify that whenever both `sender` and `receiver` are ready, then the value of `x` has been successfully copied to `y`. This can be specified as follows:

```
J0: stable     senderReady /\ recReady  ==> (x=y)
```

We will however soon discover that the UNITY logic will need more information regarding the values of related variables at different points in the program. xMECH

[2]To be more precise, the program is actually a PLPC program with no sibling. PLGA only describes the body of a PLPC program. Since PLPC environment is not fully ready, for now it is sufficient to know that a PLPC program is just a single PLGA body plus variable declaration.

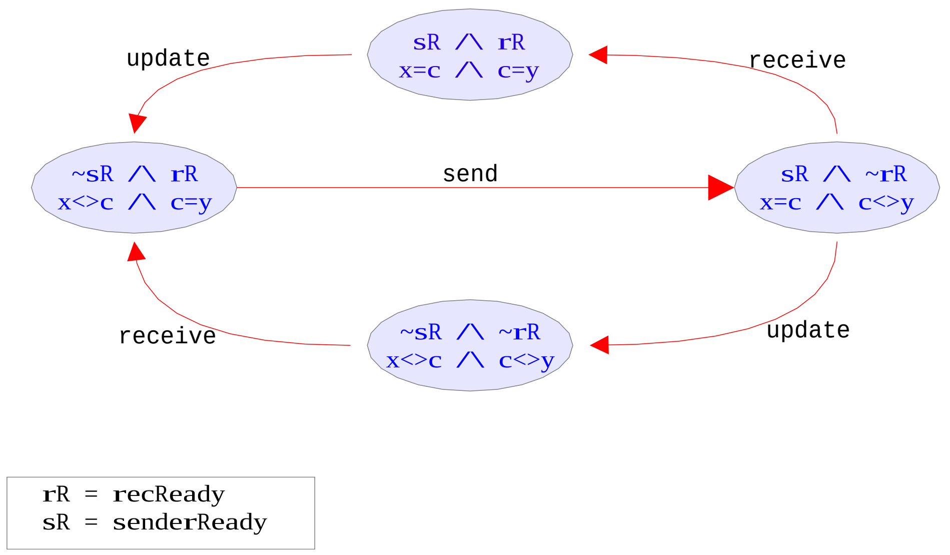

Figure 6: State Diagram of One Bit Protocol with assertion label

has no built-in insight capability to construct the required information, so we will have to help it. Figure 6 shows the same diagram as in Figure 5, but this time, it is annotated with predicates relating the states of `x`, `y`, and `c`. Each predicate specify what we think should hold in each of the combinations of the two boolean variables that control the flow of the protocol. Notice that the new diagram implies our original specification, namely that the values of `x` and `y` are equal when both `sender` and `receiver` are ready.

The new diagram in Figure 6, which now serves as our specification, can be translated into a `stable` specification[3] (Code 5.6). The translation is quite straight forward.

Code 5.6 : Safety Specification of One Bit Protocol

```
J:

|-- stable

(SYSLAB.P.root ==>
        ((senderReady  /\  recReady  ==>  (x=c) /\  (y=c)) /\
         (~senderReady /\  recReady  ==> ~(x=c) /\  (y=c)) /\
         ( senderReady /\ ~recReady  ==>  (x=c) /\ ~(y=c)) /\
         (~senderReady /\ ~recReady  ==> ~(x=c) /\ ~(y=c))))
/\
(SYSLAB.P.mid ==> senderReady)
/\
(SYSLAB.P.mid ==>
        (( recReady ==> ~(x=c) /\  (y=c)) /\
         (~recReady ==> ~(x=c) /\ ~(y=c))))
```

□

[3]We can read it as an invariant, since a stable predicate which also holds initially is an invariant.

5.8.1 Note

One may want to verify that when the `receiver` is ready (`recReady` holds) but the `sender` is not (`~ senderReady` holds), then the previous value[4] of `x` has been successfully copied to `y`.

5.9 Specifying Progress

Notice that when the protocol is in either of the following states:

1. Both the `sender` and the `receiver` are ready.

2. The `receiver` is ready but the `sender` is not.

then we know that the value of `x` has been successfully copied to `y`[5]. If this is the specification, then it is not strong enough since a protocol that does nothing and becomes stuck also satisfies it. We still need to show that the protocol will always make progress towards either of the above situations.

In UNITY, progress property is specified by the `leadsto` operator. The implementation of the inference rules for `leadsto` is still unfinished, but it is not too difficult to code once the VCG for `ensures` is implemented. The latter is done. An `ensures` property specifies progress which can be guaranteed by a single action. Progress by `leadsto` is actually the transitive and disjunctive closure of progress by `ensures`. For example, notice that the progress property from the previous paragraph is established if each group of outgoing arrows in Figure 5 makes progress. Each of those groups of outgoing arrows can be specified by an `ensures` property.

For example the following property asserts the progress made by the `send` arrow:

Code 5.7 : Progress of the send action

```
PROGRESS1:

|--

SYSLAB.P.root /\ ~senderReady /\ recReady

ensures

SYSLAB.P.root /\ senderReady /\  ~recReady

using

{senderReady, recReady, c}
```

□

5.10 Verification

A PLGA program and specifications are stored in separate files. A PLGA program file should have a `.pc` extension. Its specification file should have the same name with an `.spc` extension. A PLGA program file can only contain the text of a single program, but a specification file may contain a list of UNITY properties.

[4]UNITY has no past time operator nor next operator. An auxiliary variable can be used to record previous values of some variables.

[5]For the first situation, this is implied by the invariant proposed in Code 5.6. The invariant needed to represent the specification in Sub-subsection 5.8.1 would confirm the fact for the second situation.

To start verification HOL must be loaded first. Then, run the following command from HOL to load the xMECH-PLGA environment:

Code 5.8 : PLGA Environment Loading

```
val xmech_sigobj_loc  = "./";
val env_loc  = "./EXAMPLES/PLGA/";

loadPath := xmech_sigobj_loc :: !loadPath ;
use (env_loc^"PLGA_environment.sml");
```

□

Interaction is done through a PLGA proof manager –currently, it is still very simple. It has the following basic instructions:

- `loadProcess` *dir name*: load the files (program and specification which have the extension of `.pc` and `.spc`) needed for proving *name*. The *dir* is the directory where the files are located. Example:

  ```
  loadProcess "./" "oneBitProtocol"
  ```

- `chooseSpec` *specificationName*: choose the specification to be proven. Example : `chooseSpec "PROGRESS1"`.
- `useHol`: generate proof obligations that will be fetched into the HOL proof system and continue proving in the HOL environment.
- `confirmProven`: confirm whether the current specification has been completely proven.
- `newSpec`: continue with the next unproven specification in the system.

Below is the script to prove the `ensures` property in Code 5.7 and the invariant in Code 5.6 of the One Bit Protocol example (Code 5.5).

Code 5.9 : One Bit Protocol Proof Script

```
(* Begin *)

loadProcess "./" "oneBitProtocol";

chooseSpec "PROGRESS1" ;
useHol();

e (DEL_ALL_TAC THEN PROVE_TAC []) ;
confirmProven();

newSpec();
useHol();

e(DEL_ALL_TAC); (* removing unnecessary assumptions *)
e(REPEAT CONJ_TAC THEN TRY (PROVE_TAC [])) ;

e(RW_TAC std_ss [] THEN TRY (PROVE_TAC [])) ;
(* two left, which requires some lemmas  *)

```

```
20   (* first subgoal *)
21   e(COOPER_TAC) ;
22
23   (* second subgoal *)
24   e(PROVE_TAC [COOPER_PROVE (--`~((x:int)+1 = x)`--)]) ;
25
26   confirmProven();
27
28   (* DONE *)
```

□

Line 3 loads the program file and the specification file that begins with the name `oneBitProtocol`. These files are included in the distribution of xMECH.

In Line 5, we choose to prove a property called `PROGRESS1` first –it is the `ensures` property from Code 5.7.

The `useHol()` command in Line 6 generates the verification conditions that correspond to this property. They are subsequently combined into one formula, translated to HOL, and set up as a goal. The verification conditions are computed in two phases. First, the UNITY logic in Figure 4 is applied. This generates verification conditions in the form of Hoare triples. Next those Hoare triples are piped to the VCG of PLSEQ. The final result is a list of plain expressions (sentences of EXPR) which can be easily translated to HOL. After the call to `useHol` the proof proceeds in HOL.

The resulting goal of the property `PROGRESS1` turns out to be very easy to prove. Line 8 essentially call HOL's prover for the first order predicate logic (`PROVE_TAC`) and it is sufficient to prove the goal.

For the current version of xMECH, the command `confirmProven()` (Lines 9 and 26) is necessary to inform the xMECH proof manager that the current specification has been proven.

In Line 11 the command `newSpec()` will ask xMECH proof system to move to the next unproven specification, which in this case is the property `J` of Code 5.6. This can be easily proven with xMECH too.

After generating the verification conditions and translating them to HOL (line 12), we apply some combinations of HOL's prover for first order predicate logic prover and its equational simplifier to get rid of trivial sub-formulas. After this, there are only two goals left. With a properly selected lemma, both can be easily proven by HOL's prover for integer arithmetic (Cooper).

6 Some Closing Remarks

xMECH offers lots of flexibility. Some of its components are also highly reusable, such as its EXPR language which many languages can reuse. This gives a possibility to extend xMECH by building more logics on top or along side its existing logics.

xMECH represent programs and specifications as parse trees, so it is very good for implementing syntax-driven analyses or transformations. For example we use this capability to efficiently implement the calculation of the weakest pre-condition of a PLSEQ program. We also use it to implement a control location analysis for PLGA programs distributed program to detect disabled actions and subsequently discard them from our reasoning.

Syntax driven transformations are often very powerful in simplifying our problems. However, it is true that by implementing them directly they are less trustworthy than by implementing them by inclusion in the embedding. However, for complicated transformations, embedding may be too costly (remember that embedding will require those transformations to be proven first). This is just a fact of life which we have to live with. Though less secure, directly implemented transformations mechanize and automate a large part of our proofs, therefore reducing the chance of human errors in the proofs, and improving the speed with which we produce proofs. Having said this, it is also true that some transformations may be highly reusable. In this case attempting their verification may be worth our investment.

Another possibility to improve the safety of a directly implemented transformation is to let it generate a HOL proof script along with the actual result. We can subsequently ask HOL to run the script. If it is successful, then this can be seen as some sort of certificate of the correctness of the result of the transformation.

7 Development

In addition to finishing and cleaning the current version, below are some future work in the development of xMECH we plan to do:

xMECH Phase 1 – Modularity and Extendibility

We want xMECH to be customizable. It should be customizable in two ways. The first way is by adding or replacing transformation modules (simplifiers)[6], and the second way is by adding or replacing languages modules.

The current version of xMECH is not modular and hence also difficult to extend. In this phase we will re-structure it to provide an architecture that enables high modularity. The use of ML-functors[8] and parser combinators will be investigated.

The implementation of a complicated transformation may involve a series of non-trivial recursions, each trying to combine various recursion parameters in a non-trivial way. Extending such an implementation requires a quite involved recoding work which is also error prone. We want to investigate the use of the AG-system[9] which provides a separate layer on which syntax driven transformations can be elegantly specified and treated in a modular way by separating various aspects of a given transformation.

xMECH Phase 2 – Building Application Layers

We plan to add more layers of languages and logics. Each layer should provide a better abstraction than the layers below it. For example, we have mentioned PLPC, which extends PLGA with a mechanism to specify the structure of a distributed system.

xMECH Phase 3 – Meta Verification

We want to investigate techniques to make unembedded xMECH transformations more secure. Two options have been mentioned: including them in embedding (which is very costly), and generating proof scripts. We want to investigate the use of the AG system to facilitate the latter. There is also a third option that we want to investigate. We can use the AG system to produce a HOL model of a transformation specified with it. The model can subsequently be verified in HOL and can be regarded as certifying the transformations they represent. In principle, it is possible to program the AG system to produce a highly abstract model which is feasible to prove.

[6]xMECH simplifier set is more like decision procedure interface or plug-in of external tools rather than the notion of simplifier sets that HOL and Isabelle[6] have.

References

[1] Ade Azurat and I.S.W.B. Prasetya.
A survey on embedding programming logic in a theorem prover.
Technical Report UU-CS-2002-007, Institute of Information and Computing Sciences Utrecht University, P.O.Box 80.089 3508 TB Utrecht The Netherlands, January 2002.

[2] K.M. Chandy and J. Misra.
Parallel Program Design.
Addison-Wesley, Austin, Texas, May 1989.

[3] M.J.C. Gordon and T.F. Melham.
Introduction to HOL.
Cambridge University Press, 1993.

[4] C.A.R. Hoare.
An axiomatic basis for computers programs.
Commun. Ass. Comput. Mach., 12:576–583, 1969.

[5] G.J. Holzmann.
An overview of the spin model checker.
Technical report, Computing Principles Research Department, Bell Laboratories 2C-521, Murray Hill, New Jersey, USA, 1996.

[6] Lawrence C. Paulson and Tobias Nipkow.
Isabelle tutorial and user's manual.
Technical Report 189, Computer Laboratory, University of Cambridge, January 1990.

[7] I.S.W.B. Prasetya.
Mechanically Supported Design of Self-stabilizing Algorithms.
PhD thesis, Utrecht University, 1995.

[8] Sergei Romanenko, Claudio Russo, and Peter Sestoft.
Moscow ML Owner's Manual.
Available at http://www.dina.kvl.dk/~sestoft/mosml.html.

[9] S. D. Swierstra, P. R. A. Alcocer, and J. Saraiva.
Designing and implementing combinator languages.
Lecture Notes in Computer Science, 1608:150–??, 1999.

Appendix

A More PLSEQ Statements

1. Increment and decrement, e.g. like `x++` and `x--`. Note that these are statements. They are not allowed to appear in an expression.

2. `if` without else, e.g `if (x=0) then x++`. PLSEQ logic treats this as if it has an `else skip` branch.

3. PLSEQ supports a Java style `switch` statement, e.g.:

```
switch (x+y)
  {0       : c = 0
   z+1     : c = c+z
   default : c++
  }
```

   PLPC logic treats in the same way it treats if-then-else. The alternatives do not need to be disjunct. The logic will simply assume that if there are multiple two alternatives possible, then one will be selected non-deterministically. An implementation may always select the first alternative possible, but the logic is currently not strong enough to expose this.

4. PLSEQ supports `for` statements, e.g.:

```
inv ...
for (i = 0; 10; i++)
  { a = a + i   ;
    s = s + x#i ; }
```

   PLSEQ logic will simply transform a `for` loop into the corresponding `while` loop.

B Channels in PLGA

PLGA also supports asynchronous channels. The syntax is:

Send action	:	*channel-variable* `!` *expr*
Receive action	:	*channel-variable* `?` *program-variable*

Currently, xMECH parsers do not allow `!` and `?` inside atomics. An atomic communication involving multiple channels will have to be coded by hand using PLSEQ.

For example, `c!(x+1)` sends the value of `x+1` into the channel `c`, whereas `c?y` removes the foremost element of `c` and copies it to `y`.

Send and receive actions are treated as atomics in PLGA. PLGA logic translates a send action `g -> c!e` to the following (PLGA) code:

```
g /\ ~ chanFull c
->
atomic { c = send>>c>>e ; SYS.HIST.c = SYS.HIST.c ++ -[e]- }
```

where `SYS.HIST.c` is an auxiliary variable that records all values ever sent to `c`. This variable is typically not implemented. However, it often useful for the purpose of specification.

A receive action `g -> c?x` is translated to the following code:

```
g /\ ~ chanEmpty c
->
atomic { x = fstMsg>>c ; c = flushFstMsg>>c ; }
```

xMECH currently performs no further simplification on send and receive actions, nor does it come with a special HOL theory about channels. If the user want to use these channel features, then he must define the constants `send`, `fstMsg`, and `flushFstMsg` in HOL and add to the file `exConst.sml` so that xMECH will recognize them as them as meta identifiers.