IMPLEMENTING A MODAL LOGIC OVER DATA AND PROCESSES USING XTL

Jeremy Bryans
Department of Computing Science and Mathematics,
University of Stirling, UK
jwb@cs.stir.ac.uk

Carron Shankland
Department of Computing Science and Mathematics,
University of Stirling, UK
ces@cs.stir.ac.uk

Abstract The modal logic FULL is designed to capture strong bisimulation over early symbolic transition systems (STSs) for full LOTOS. It provides a compact way of expressing and verifying properties involving both data and transitions. In this paper we present a restricted prototype implementation of a model checker for LOTOS for queries written using the FULL logic. The model checker is developed within the CADP package using XTL.

Keywords: Modal Logic, Symbolic LOTOS, CADP, XTL

1. INTRODUCTION

Model checking (Clarke et al., 1999) has proved to be a valuable verification method in recent years, and the size of systems which can realistically be verified is constantly increasing. However, there is still much room for improvement, and many techniques have been proposed to increase the state space of verifiable systems, including symmetry (Emerson and Sistla, 1993), abstraction (Clarke et al., 1994) and symbolic methods (Burch et al., 1994). A particular problem is the inclusion of data in systems, since this can often lead to infinite state space. Typically this is dealt with by restricting the size of the data type. An alternative approach is to deal with data symbolically (Hennessy and Lin, 1995; Calder and Shankland, 2001). The state space is reduced by grouping transitions according to the kind of data passed, or the properties of that data. Thus, instead of investigating a single transition for every data value, groups of transitions are formed and verification is performed at the level of the groups.

Previous work (Calder et al., 2001a; Calder et al., 2001b) has established a formal framework for the symbolic interpretation of LOTOS behaviours. The formal description technique LOTOS (ISO:8807, 1988) was chosen due to its popularity and applicability to a wide range of applications (e.g. protocols and services (Ajubi et al., 1989), distributed systems (Pecheur, 1992), and as a semantics for higher level languages such as feature descriptions (Turner, 1998) and use-case maps (Amyot et al., 2000)). Further, there are well established tools such as CADP (Fernandez et al., 1996) for reasoning about LOTOS behaviours. LOTOS is particularly amenable to the symbolic approach because it allows the description of process flow of control as well as allowing data to be passed between processes. Such data can affect the process flow of control and therefore cannot simply be ignored in verification.

On top of the symbolic interpretation of LOTOS an equivalence relation (Calder and Shankland, 2001) and a modal logic called FULL (Calder et al., 2001a) have been defined. The logic provides a way of expressing properties involving both data and transitions. Currently tools are being developed to support reasoning in all parts of the framework, including model checkers for the logic (Bryans et al., 2001b; Robinson and Shankland, 2001). As an interim step, a prototype of the model checker has been implemented within CADP using XTL (Mateescu and Garavel, 1998), a functional-type programming language allowing description of computation over graphs. This approach has limitations. The underlying semantics of CADP is not symbolic therefore the prototype is necessarily limited. The logic remains unchanged syntactically, but its expressive power is reduced since only properties over finite data types can be expressed. So, here we do not exploit the main advantage of our symbolic framework, which is to represent infinitely branching systems by finitely branching ones. However, the advantage of the approach is integration with a range of verification tasks already implemented in CADP, and the ability to experiment with the logic at an early stage.

The purpose of this paper is to present the XTL implementation of FULL, illustrate the sorts of symbolic properties (albeit over processes which have a concrete representation in CADP) which can be expressed and verified automatically, and discuss the ongoing plans for symbolic reasoning about LOTOS behaviours. We begin by introducing the CADP toolkit, summarising the main capabilities of interest to us. In Section 3 we present an overview of the logic FULL. We assume the reader is familiar with Full LOTOS, or at least process algebra. A more detailed introduction may be found in (Calder et al., 2001a), where we discuss further the effects on the logic of the restrictions imposed by CADP. The main section of the paper is devoted to the implementation of modal operators with data using XTL. Finally we evaluate the success of this experiment, and present directions of ongoing and future work.

2. CADP

The CAESAR/ALDEBARAN development package (CADP) is a versatile multi-functional tool offering many different formal verification techniques, from interactive simulation through to compositional verification based on refinement. It is based around a common format for explicit Labeled Transition Systems (LTSs), known as Binary Coded Graphs, or BCGs. In particular, it accepts Full LOTOS as an input formalism, and offers a model-checking algorithm for expressions written in eXecutable Temporal Logic, or XTL. The BCG basis of CADP is finite and therefore all processes handled by CADP must use finite datatypes of up to 256 values.

XTL is a functional-type programming language designed to allow an easy, compact representation of common temporal logic operators. They are evaluated over LTSs encoded as BCGs. XTL provides low level operators which access the states, labels, and transitions of the LTS, and also operators to determine the successors and predecessors of states and transitions. A number of modal logics (eg. HML (Hennessy and Milner, 1985) and CTL (Clarke et al., 1999)) have already been successfully encoded within XTL. CADP is therefore an ideal tool within which to build a prototype model checker for FULL, although the finite basis of BCGs means that we will not be able to exploit the full expressive power of the logic. This is discussed in more detail in the next section.

3. THE LOGIC FULL

The FULL logic was designed as part of an ongoing research project (DIET website, 2000) to develop a framework for reasoning about Full LOTOS, i.e. the processes and the data. In this section we present an informal introduction to the symbolic logic FULL.

The syntax of FULL is based on a variant of HML (Stirling, 1989), with quantifiers over data added. It is made up of two parts. The first set of formulae, ranged over by Φ, applies to closed terms. The second set, ranged over by Λ, applies to terms with a single free variable, as would arise from a LOTOS process with a single parameter. (The extension to multiple free variables is straightforward.)

Definition 1 *Syntax of FULL*

$$\begin{array}{rcl}\Phi & ::= & b \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [a]\Phi \mid \langle a\rangle\Phi \\ & & \mid \langle\exists y\, g\rangle\Phi \mid [\exists y\, g]\Phi \mid \langle\forall y\, g\rangle\Phi \mid [\forall y\, g]\Phi \\ \Lambda & ::= & \exists y.\Phi \mid \forall y.\Phi\end{array}$$

where b is some Boolean expression, $a \in G \cup \{\mathbf{i}, \delta\}$, $g \in G \cup \{\delta\}$, G is the set of gate names, $\mathbf{i}$ is the internal event, δ is the exit event and y denotes a variable name.

The simple event operators are taken from HML (Stirling, 1989) and are well known.

$\langle a \rangle \Phi$: it is possible to do an `a` event (and then satisfy Φ), and

$[a]\Phi$: after every `a` event, Φ is satisfied. (May hold vacuously.)

In this logic, the novel operators are the four *quantified modal* operators. They are formed as combinations of the modes $\langle\ \rangle$ and $[\ ]$ and the quantifiers $\exists$ and $\forall$: $\langle \exists y\, g \rangle$, $[\exists y\, g]$, $\langle \forall y\, g \rangle$ and $[\forall y\, g]$. For each of these combination operators the meaning is determined using the quantifier to range over the value (y) being passed, and the mode to refer to the gate (g) at which it is being passed. For simplicity, we assume that a single data value is passed at a gate. The extension to multiple data offers at a gate is straightforward.

To introduce and illustrate these operators, we consider the example of the process `P` in Figure 1, which is taken from the introductory paper (Calder et al., 2001a). This illustrates a number of the capabilities of the FULL logic.

When encoding the process `P` within CADP, we used the library NUM10.lib, which gives the natural numbers from 1 to 10 and offers the operators `GT`, `LT` and `EQ` ($>, <$ and $=$). This allows the finite BCG representation of `P` given in Figure 2. As noted above, the data type may have up to 256 values, but 10 makes the example easier to present.

Informally, the formula $\langle \exists y\, G \rangle \Phi$ is satisfied by a process if it can perform some transition with data y at a `G` gate, and the subsequent process satisfies the formula Φ. For example, the formula $\langle \exists y\, G \rangle \langle K \rangle tt$ is satisfied by the process `P`, because there are values of y (in particular: 4, 5 and 9) which can lead to a state satisfying $\langle K \rangle tt$.

The formula $\langle \forall y\, G \rangle \Phi$ is satisfied by a process if all the values that y can take have at least one `G` transition that leads to a process that satisfies Φ. For example, the formula $\langle \forall y\, G \rangle \langle H \rangle tt$ is satisfied by the process `P`, because every y value has a possible `G` transition leading to a state satisfying $\langle H \rangle tt$.

The formula $[\exists y\, G]\Phi$ is satisfied by a process if there is some value that y can take, and every time the process performs a `G` transition with that value, the subsequent state satisfies Φ. For example, the formula $[\exists y\, G]\langle H \rangle tt$ is satisfied by the process `P`, because if $y = 1, 2, 3, 5, 6, 7, 8$ or 10 then it is always possible to perform an H action afterwards.

The formula $[\forall y\, G]\Phi$ is satisfied by a process if for every possible value of y, every `G` transition leads to a state satisfying Φ. For example, the formula $[\forall y\, G](\langle H \rangle tt \vee \langle K \rangle tt)$ is satisfied by the process `P`, because every possible transition leads to a state which can either perform an `H` or a `K` action.

```
process P [G,H,K] : exit :=
      G?x:Num10 [x LT 5]; H; exit
   [] G!4; K; exit
   [] G?x:Num10 [x EQ 5]; (H; exit [] K; exit)
   [] G!5; H; exit
   [] G?x:Num10 [x GT 5]; H; exit
   [] G!9; K; exit
endproc
```

Figure 1. process `P`

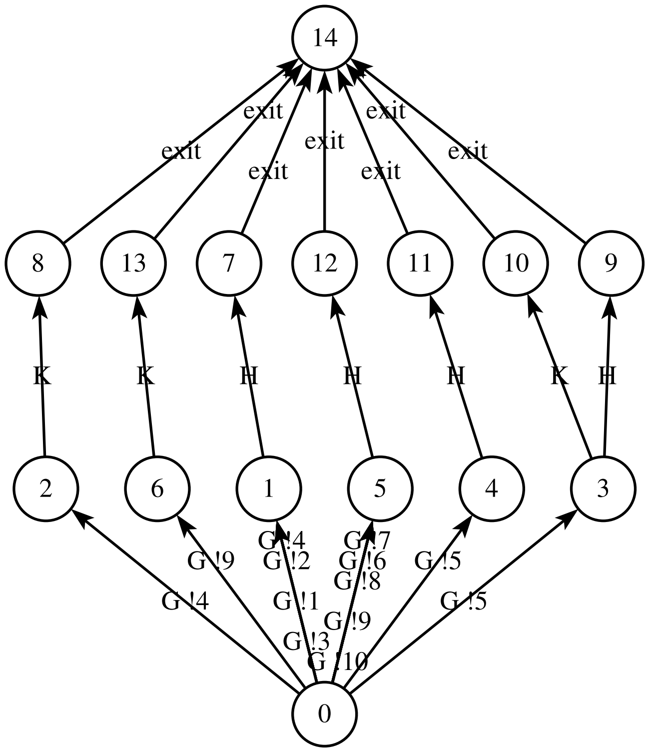

Figure 2. Binary Coded Graph for process `P`

The logic is presented more completely in (Calder et al., 2001a), where it is defined formally in terms of symbolic transition systems as derived from LOTOS (Calder and Shankland, 2001). In order to determine the meaning of the logic (normally defined over symbolic transition systems) over Binary Coded Graphs, an equivalent interpretation of the logic over labelled transition systems is used. (See Appendix.) There is a direct mapping between BCGs and LTSs. The equivalence of the two definitions of the logic is shown in (Bryans et al., 2001a).

The expressivity of the logic may be considered to be parameterised by the number of values of the data types used. For this prototype, CADP restricts us to using data types with only 256 values, therefore the expressivity of the logic is clearly restricted, since when interpreted over a symbolic transition system an infinite number of data types may be considered.

In practice, this restriction may not affect the verification process. It may be that only 256 distinct values are required to distinguish processes and properties. If this where not a possibility, then CADP would be almost useless as a verification tool.

As soon as more than 256 values may be used in one branching point the logic of the prototype can no longer be used to distinguished processes which can be distinguished by FULL.

Another consideration is efficiency. A symbolic transition system allows many (possibility infinitely many) transitions to be represented finitely (possibility by just one transition). If a data variable can have 256 values, CADP will construct a BCG with 256 corresponding transitions, and all of these transitions must be explored when verifying properties. Therefore more work has to be done than in the symbolic case, in which the same transitions may be represented by just one transition.

4. IMPLEMENTING FULL WITHIN XTL

We show in this section how to implement the simple modal operators ($\langle g \rangle$ and $[g]$) and the quantified modal operators ($[\exists y\, g]$, $\langle \exists y\, g \rangle$, $[\forall y\, g]$ and $\langle \forall y\, g \rangle$) within the XTL language. The boolean operators $\wedge$, $\vee$ and $\neg$ have direct translations within XTL and are not shown here.

In XTL, a formula is associated with the largest set of states which satisfy it. Since the BCGs are finite these sets of states are necessarily finite.

To illustrate some useful XTL constructs, we begin with the simple modal operators, involving no data.

Simple modal operators

The $\langle a \rangle \Phi$ operator has two parameters, the simple gate name a and the formula Φ. To represent the gate a we use the labelset A, and the formula Φ is represented by the set of states Phi. Any state S will satisfy this operator, provided at least one of the outgoing edges of S is labelled a, (i.e. `exists T : edge among out (S) in (label (T) among A)`), and the resultant state of that edge satisfies the formula Φ (i.e. `(target (T) among Phi)`). We define $\langle a \rangle \Phi$ as `DIA(A,Phi)` within XTL by

```
def DIA (A : labelset, Phi : stateset) : stateset =
    { S : state where
    exists T : edge among out (S) in
        (label (T) among A) and (target (T) among Phi)
    end_exists }
end_def
```

Note here that any labelset A and any stateset Phi will be acceptable to `DIA`, but that to remain true to the FULL definition, A should be a single label representing the simple event a, and the stateset Phi should represent the FULL formula Φ. This stateset can be supplied as the result of a nested function. For example, the formula $\langle a \rangle \langle b \rangle tt$ would be implemented as `DIA(a, DIA(b, tt))`, where `tt` represents all states that satisfy *true*, i.e. all states.

The $[a]\Phi$ operator has the same two parameters, the gate name a and the formula Φ. A state S will satisfy this operator, provided for any appropriately labelled outgoing edges of S, the resultant state of that edge satisifies the formula Φ. This may be vacuously true if no edges are appropriately labelled. The differences from the implementation of $\langle a \rangle \Phi$ are therefore the use of `forall` instead of `exists`, and `implies` instead of `and`. Again, although BOX takes a set of labels A and a set of states Phi, A should be a single label representing the simple event a, and the stateset Phi should represent the FULL formula Φ.

We define $[a]\Phi$ as `BOX(A,Phi)` within XTL by

```
def BOX (A : labelset, Phi : stateset) : stateset =
    { S : state where
          forall T : edge among out (S) in
              ((label (T) among A) implies (target (T) among Phi))
          end_forall }
end_def
```

Quantified modal operators

The quantified modal operators are the ones which allow us to refer to the data part of Full LOTOS.

To understand the implementation of a particular quantified modal operator, we begin by considering a general formula $OP(y, g)\Phi$, where OP is one of the four quantified modal operators, y is a variable of type *ytype* and g is a gate name, of type *ytype*. We deconstruct Φ into a form $\Phi_y \wedge \Phi_1$, where Φ_y contains all the restrictions on the variable y, and Φ_1 contains none of them.

The interpretation of each operator over a BCG can be understood as matching a particular pattern in the matrix of reachable states, as shown below.

$\Phi_1 = \langle H \rangle tt$	Outgoing transitions									
	G!1	G!2	G!3	G!4	G!5	G!6	G!7	G!8	G!9	G!10
P1	T	T	T	T	o	o	o	o	o	o
P2	o	o	o	F	o	o	o	o	o	o
P3	o	o	o	o	T	o	o	o	o	o
P4	o	o	o	o	T	o	o	o	o	o
P5	o	o	o	o	o	T	T	T	T	T
P6	o	o	o	o	o	o	o	o	o	F

Figure 3. Matrix representing the process `P` and $\langle H \rangle tt$

The matrix in Figure 3 is formed from the BCG (Figure 2) of process `P`, where the headers of the columns are the labels of the outgoing transitions from the initial state, and the labels on the rows are all the reachable states from the initial state.

For a particular label and state, the relevant position in the matrix can have one of three labels:

- T: a transition marked with this label can reach this state, and Φ_1 holds,
- F: a transition marked with this label can reach this state, but Φ_1 does not hold, or
- o: a transition marked with this label cannot reach this state.

The pattern on the matrix is then determined solely by Φ_1, i.e. the part of the formula which is not concerned with the variable y. Similar matrices can be built for other process states and other FULL formulae Φ.

The formula Φ_y, which contains all the restrictions on y, determines which columns need to be considered when evaluating the complete formula $\Phi_y \wedge \Phi_1$. The validity of different FULL operators can be related to different patterns within the matrix.

As an example, let us consider the formula $(y = 4) \wedge \langle H \rangle tt$, so Φ_y is $(y = 4)$ and Φ_1 is $\langle H \rangle tt$. We will consider prefixing this by each of the four modal operators in turn.

- $\langle \exists y\, G \rangle \Phi$ needs just *one place* in the matrix to be labelled T. But since our example is $(y = 4) \wedge \langle H \rangle tt$ then this place must be in the $G!4$ column, because in every other column the formula $(y = 4)$ is not true.
- $[\exists y\, G]\Phi$ needs *one column* in the matrix which contains only the letters T and o in order to hold. But our specific example restricts this to the

$G!4$ column, and it therefore does not hold. If instead we had Φ defined as $(y = 5) \wedge \langle H \rangle tt$ the formula $[\exists y\, G]\Phi$ would be true.

- $\langle \forall y\, G \rangle \Phi$ needs *at least one T in every column* in order to hold. Allowing Φ to restrict y, although syntactically possible, makes the whole formula immediately false because the quantifier ranges over all possible values of y. Clearly, formulae such as $\langle \forall y\, G \rangle (y = 4)$ must be false, since y can have values other than 4. However, the formula $\langle \forall y\, G \rangle \Phi_1$ holds.

- $[\forall y\, G]\Phi$ needs *no Fs anywhere* in the table in order to hold. Again, restricting y in Φ immediately forms a false formula. In this case, since the table contains Fs, no formula $[\forall y\, G]\Phi$ holds. But if we constructed the table $\Phi_1 = \langle H \rangle tt \vee \langle K \rangle tt$ then $[\forall y\, G]\Phi_1$ would hold.

IMPLEMENTATION In order to interpret $OP(y, G)\Phi$ within XTL, we will break Φ into $\Phi_y \wedge \Phi_1$ as described above. There will be no other variables in Φ because any variables will have been bound in a previous step.

Each of the implementations of the quantified modal operators will receive two parameters: a labelset A which (loosely) corresponds to the combination of the gate g and the formula Φ_y, and a stateset Phi which corresponds to the formula Φ_1. The implementation will return the set of all states which satisfy the operator. A formula is satisfied by a process if the initial state is in this returned set.

Whereas for the simple modal operators we assumed the labelset was in fact a singleton (to conform with FULL syntax), for the quantified modal operators the labelset will typically not be a singleton. Although in FULL we write a symbolic expression such as $[\exists y\, G]\Phi$ this has to be expanded by XTL into a concrete set of labels. We use the (built in) XTL macro `EVAL_A(A)` to generate the labelset of the actions which satisfy the predicate `A`. For example, all y actions possible at a `G` gate are therefore evaluated as `EVAL_A(G?y:ytype)`, yielding the set `{"G!1", "G!2", "G!3", ..., "G!10"}`. This labelset is then supplied to the implementation of a quantified modal formula.

This is the point at which the restrictions placed on our implementation of the FULL model checker by the BCG implementation become evident. If we were adhering strictly to the semantics as expressed in the Appendix, then the labelset could be infinite (ranging across all values of the data type). Using the BCG representation we quantify over finite types.

The labelset may also encode Φ_y. To restrict y to values which satisfy Φ_y we use `EVAL_A(G?y:ytype where` Φ_y`)`. The type `ytype` can be a simple XTL type, or one of the types created for the LOTOS process. If we wish to use a created type we need to be careful to provide C implementations for the constructors and operators within the type definition, and to reiterate these within the `.xtl` file. In our example the CAESAR compiler identifies the

NUM10 values as integers, and we can use `integer` in the XTL queries. But if we used the strings `one`, `two`, ... etc. instead, then the compiler would implement these as character strings, and we would be forced to provide a C function to convert these back to NUM10 values within the `.xtl` file.

$\langle \exists \boldsymbol{y}\, \boldsymbol{G} \rangle \boldsymbol{\Phi}$ A state S will satisfy this operator, provided there is some outgoing edge in A which hits the target set Phi (i.e. `exists T : edge among out (S)`). In matrix terms, finding one T anywhere would be sufficient, provided A made no restrictions on y. Otherwise the T would have to be in one of the columns to which y was restricted. This additional constraint is implemented wholly by the use of the `EVAL_A(G?y:ytype where` Φ_y`)` expression in generating a restricted labelset. See Section 5 for an example of this sort of formula. We can define $\langle \exists y\, G \rangle \Phi$ within XTL as

```
def DIAEVAL (A : labelset, Phi : stateset) : stateset =
          { S : state where
              exists T : edge among out (S) in
            (if label (T) among A
              then (target(T) among (Phi))
              else false
             end_if)
              end_exists }
end_def
```

The XTL definition above is a reasonably direct translation of the semantics of $\langle \exists y\, G \rangle \Phi$ as given in the Appendix. The `if` statement corresponds to finding one T in the matrix (with the right sort of label).

$[\exists \boldsymbol{y}\, \boldsymbol{G}] \boldsymbol{\Phi}$ A state S will satisfy this operator, provided there is some label L in A (`exists L : label among A`) such that every outgoing edge labelled with L hits the target set Phi (the `forall` expression with nested `if`). That is, there is a whole column in the matrix containing only the letters T and o. The `else true` implements the fact that this operator can be vacuously true if there is a label L in A which is not assigned to any outgoing edge, i.e. the corresponding column contains only the letter o. We define $[\exists y\, G] \Phi$ within XTL as

```
def BOXEVAL (A : labelset, Phi : stateset) : stateset =
          { S : state where
              exists L : label among A in
                  (forall T : edge among  out (S) in
                    if   label (T) = L
                     then target (T) among (Phi)
                     else true
```

```
                          end_if
                        end_forall)
                   end_exists }
end_def
```

It is harder to see a correspondence between the semantics of this operator as expressed in the Appendix and the XTL implementation above. The finite BCG semantics means that we effectively translate an expression of the form $[\exists y\, G]\Phi$ into $[g1]\Phi \;\vee\; [g2]\Phi \;\vee\; \ldots \vee [g10]\Phi$, where each value of the data type yields a corresponding hardwired [ ] operator.

$\langle \forall y\, G\rangle \Phi$ To implement the universally quantified operators we rely on the duality of $\exists$ and $\forall$, and $\langle\ \rangle$ and [ ], as pointed out in (Calder et al., 2001a). In XTL, `not (Phi)` is the complement of the set `Phi`. We define $\langle \forall y\, G\rangle$ as

```
def DIAAVAL (A : labelset, Phi : stateset) : stateset =
  not (BOXEVAL (A, not (Phi)))
end_def
```

taking advantage of the fact that $neg(\langle \forall y\, G\rangle \Phi)$ is $[\exists y\, G] neg(\Phi)$.

$[\forall y\, G]\Phi$ We define $[\forall y\, G]\Phi$ as

```
def BOXAVAL (A : labelset, Phi : stateset) : stateset =
 not (DIAEVAL (A, not (Phi)))
end_def
```

taking advantage of the fact that $neg([\forall y\, G]\Phi)$ is $\langle \exists y\, G\rangle neg(\Phi)$.

5. THE MODEL CHECKER

To model check a LOTOS process within CADP using the FULL logic, we use the macro `SAT`, defined as

```
macro SAT (Phi) =
   PRINT_FORM( if Phi includes init then (TRUE)
                                     else (FALSE)
               end_if)
end_macro
```

This accepts a formula Phi and prints TRUE if the initial state of the process is included in the stateset representing Phi, and FALSE otherwise.

We can then model check a process using

```
$xtl test.xtl process.bcg
```

The file `process.bcg` is the output from the `caesar.adt` and `caesar` components of the CADP tool. These take a LOTOS specification and produces a

Binary Coded Graph(BCG). The file `test.xtl` contains a series of `SAT` commands. For convenience, SAT commands can also take a text parameter, which is returned with the model checker output. This is optional, but such comments can serve as a reminder of the formula being tested.

Examples

Here we use again the process `P` from Section 3.

1 The query *is it possible for process* `P` *to perform a G!4 action?* which is captured as $P \models \langle \exists y\, G \rangle (y = 4)$ using FULL, is written

```
SAT( "<E y:G>(y=4)tt: ",
             ( DIAEVAL(EVAL_A(G?y:integer where y=4),TRUE)))
```

and because `(EVAL_A(G?y:integer where y=4))` evaluates to `"G !4"` the formula becomes

```
if DIAEVAL({"G !4"}, TRUE) includes INIT then TRUE
     else FALSE
end_if
```

and for `DIAEVAL({"G !4"}, TRUE)` to be true for process `P`, the initial state simply needs one outgoing *G!4* action. This is clearly true (see Figures 2 and 3), so the output is

```
<E y:G>(y=4)tt:  TRUE
```

2 The query *for all values y, is it possible to do a G!y action followed by an H action?* which is written as $P \models \langle \forall y\, G \rangle \langle H \rangle tt$ using FULL, is expressed as

```
SAT( "<V y:G><H>tt: ",
   (DIAAVAL(EVAL_A(G?y:integer), (DIA(EVAL_A(H),TRUE)))))
```

The parameter `(EVAL_A(G?y:integer)` evaluates to the set

```
{"G !1", "G !2", "G !3", "G !4", "G !5",
 "G !6", "G !7", "G !8", "G !9", "G !10"}
```

the restriction to ten elements coming from the process `P`, which is limited to the library NUM10.lib. This means every transition possible for `P` must be considered. The operator $\langle \forall y\, G \rangle$ requires that each label is on a transition leading to at least one state where the formula Φ is true. This is true here (see Figure 3, one T in every column), and so the output is

```
<V y:G><H>tt: TRUE
```

3 The query *for all values y, after any G!y action is it possible to do an H action?* is written within FULL as $P \models [\forall y\, G]\langle H\rangle tt$ and is expressed as

```
SAT( "[V y:G]<H>tt:  ",
     ( BOXAVAL(EVAL_A(G?y:integer),(DIA(EVAL_A(H),TRUE))))
```

The parameter `EVAL_A(G?y:integer)` generates the full list of possible transitions again, but this time the query fails, because, for example, it is possible to take a transition labelled $G!4$ to a state which only permits a *K* action. It doesn't matter that there is a successful *G!4* transition which can subsequently perform an H action.

The reasonably direct translation from FULL to XTL can be seen from these simple examples. The only tricky part is the need to provide requirements on a value Φ_y to the point at which that value is introduced.

6. CONCLUSIONS AND FURTHER WORK

We have successfully implemented a model checker for the FULL logic over LOTOS (with data), and we are planning to extend this work in a number of directions.

We have modelled the Remote Procedure Call case study (Broy et al., 1996; Hardy, 2000) using LOTOS within CADP, and performed some simple queries on it using FULL. We have yet to fully explore this example.

The FULL logic cannot as yet express mu-calculus type queries (Kozen, 1983) with infinitely repeating patterns, but we are currently working on extending the logic in this direction. Further, for ease of presentation, the FULL logic is currently limited to single data values being passed at gates, and for practical use it would be preferable to allow multiple data values. For example, if `G?y1:bool?y2:integer` was a gate, we would like to be able to identify all labels that matched `G!true?y2:integer`. Implementing the mu-calculus operators in XTL should be straightforward, but currently expressing pattern matching queries within XTL is not feasible.

The CADP toolkit provides a very accessible way of building a prototype model checker, and we are very pleased with the results obtained. However the BCG graphs are only ever finitely branching, because the LOTOS processes are restricted by the datatypes used. In order to reason about symbolic aspects of

FULL allowing infinite data types we are considering two other approaches: theorem proving (using the Ergo tool) (Robinson and Shankland, 2001) and rewriting logic (Bryans et al., 2001b). Much interesting work on combining theorem proving and model checking is available (Rajan et al., 1995; Joyce and Seger, 1993), and rewriting logic, while a comparatively recent technique, has already been successfully used to implement a number of simple model checkers (Verdejo and Marti-Oliet, 2000). These techniques also have the advantage of allowing integration of reasoning about data and reasoning about processes.

Acknowledgments

Thanks to Radu Mateescu for the help provided for our work with CADP. The authors would like to thank the Engineering and Physical Sciences Research Council for supporting this research under the project award "Developing Implementation and Extending Theory: A Symbolic Approach to Reasoning about LOTOS".

References

Ajubi, I., Scollo, G., and van Sinderen, M. (1989). Formal Description of the OSI Session Layer. In van Eijk, P., Vissers, C., and Diaz, M., editors, *The Formal Description Technique LOTOS*, pages 89–210. Elsevier Science Publishers B.V. (North-Holland).

Amyot, D., Charfi, L., Gorse, N., Gray, T., Logrippo, L., Sincennes, J., Stepien, B., and Ware, T. (2000). Feature Description and Feature Interaction Analysis with Use Case Maps and LOTOS. In Calder, M. and Magill, E., editors, *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press.

Broy, M., Merz, S., and Spies, K., editors (1996). *Formal Systems Specification*, volume 1169 of *Lecture Notes in Computer Science*. Springer-Verlag.

Bryans, J., Calder, M., Maharaj, S., Ross, B., and Shankland, C. (2001a). Report on Developing a Symbolic Framework for Reasoning about Full LOTOS. Unpublished.

Bryans, J., Verdejo, A., and Shankland, C. (2001b). Using Rewriting Logic to implement the modal logic FULL. In Nowak, D., editor, *AVoCS'01: Workshop on Automated Verification of Critical Systems*. Oxford University Computing Laboratory technical report PRG-RR-01-07.

Burch, J. R., Clarke, E. M., Long, D. E., McMillan, K., and Dill, D. L. (1994). Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer Aided Design of Integrated Circuits*, 13(4):401–424.

Calder, M., Maharaj, S., and Shankland, C. (2001a). A Modal Logic for Full LOTOS based on Symbolic Transition Systems. *The Computer Journal*. In press.

Calder, M., Maharaj, S., and Shankland, C. (2001b). An Adequate Logic for Full LOTOS. In Oliveira, J. and Zave, P., editors, *Formal Methods Europe'01*, LNCS 2021, pages 384–395.

Calder, M. and Shankland, C. (2001). A Symbolic Semantics and Bisimulation for Full LOTOS. Technical Report TR-2001-77, University of Glasgow.

Clarke, E. M., Grumberg, O., and Long, D. E. (1994). Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542.

Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. The MIT Press.

DIET website (2000). Developing Implementation and Extending Theory: A Symbolic Approach to Reasoning about LOTOS. `http://www.cs.stir.ac.uk/diet/diet.html`.

Emerson, E. and Sistla, A. P. (1993). Symmetry and model checking. In Courcoubetis, C., editor, *Proceedings of the 5th Workshop on Computer Aided Verification*, pages 463–478.

Fernandez, J.-C., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, R., and Sighireanu, M. (1996). CADP(CAESAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Alur, R. and Henzinger, T., editors, *CAV'96*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag.

Hardy, R. A. (2000). RPC Specification in LOTOS. Unpublished manuscript.

Hennessy, M. and Lin, H. (1995). Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389.

Hennessy, M. and Milner, R. (1985). Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161.

ISO:8807 (1988). Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Organisation for Standardisation.

Joyce, J. J. and Seger, C.-J. H. (1993). Linking BDD-based symbolic evaluation to interactive theorem proving. In *Proceedings of the 30th Design Automation Conference*. Association for Computing Machinery.

Kozen, D. (1983). Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354.

Mateescu, R. and Garavel, H. (1998). XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In *Proceedings of the International Workshop on Software Tools for Technology Transfer*.

Pecheur, C. (1992). Using LOTOS for specifying the CHORUS distributed operating system kernel. *Computer Communications*, 15(2):93–102.

Rajan, S., Shankar, N., and Srivas, M. K. (1995). An integration of model checking with automated theorem proving. In Wolper, P., editor, *Proceedings of the 1995 Workshop on Computer-Aided Verification*, volume 939 of *LNCS*, pages 84–97. Springer-Verlag.

Robinson, P. and Shankland, C. (2001). Implementing the modal logic FULL using Ergo. In Nowak, D., editor, *AVoCS'01: Workshop on Automated Verification of Critical Systems*. Oxford University Computing Laboratory technical report PRG-RR-01-07.

Stirling, C. (1989). Temporal Logics for CCS. In de Bakker, J., de Roever, W.-P., and Rozenberg, G., editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 66–672, REX School/Workshop, Noordwijkerhout, The Netherlands. Springer-Verlag.

Turner, K. J. (1998). An architectural description of intelligent network features and their interactions. *Computer Networks*, 30(15):1389–1419.

Verdejo, A. and Marti-Oliet, N. (2000). Implementing CCS in Maude. In *Proceedings of FORTE/PSTV 2000*, pages 351–366.

Appendix: The FULL logic definition

We give here the semantics of the logic, defined inductively over concrete labelled transition systems. It is assumed that *vars*$(\Phi) \cap$ *vars*$(t) = \{\}$, i.e. the variables of the transition system and the variables of the logical formula are disjoint.

First we define $t \models \Phi$, denoting that a closed term t satisfies a closed modal formula Φ. The relation is defined inductively over the syntax of the logic by the equations of Definition 2.

The last two rules of the definition where t is an open term relate the (single) free variable of the parameterised transition system to the (single) free variable of a quantified logical property. The remaining part of the property is evaluated using the earlier $t \models \Phi$ definitions.

Definition 2 *Semantics of FULL over a labelled transition system*
Given any closed term t*, the semantics of* $t \models \Phi$ *is given by:*

$$
\begin{array}{lcl}
t \models b & = & b \equiv tt \\
t \models \Phi_1 \wedge \Phi_2 & = & t \models \Phi_1 \textit{ and } t \models \Phi_2 \\
t \models \Phi_1 \vee \Phi_2 & = & t \models \Phi_1 \textit{ or } t \models \Phi_2 \\
t \models \langle a \rangle \Phi & = & \textit{there is a } t' \textit{ s.t. } t \xrightarrow{a} t' \textit{and } t' \models \Phi \\
t \models [a]\Phi & = & \textit{whenever } t \xrightarrow{a} t' \textit{ then } t' \models \Phi \\
t \models \langle \exists x\, g \rangle \Phi & = & \textit{for some value } v\textit{, for some } t'\textit{, } t \xrightarrow{g\, v} t' \textit{ and } t' \models \Phi[v/x] \\
t \models \langle \forall x\, g \rangle \Phi & = & \textit{for all values } v\textit{, for some } t'\textit{, } t \xrightarrow{g\, v} t' \textit{ and } t' \models \Phi[v/x] \\
t \models [\exists x\, g]\Phi & = & \textit{for some value } v\textit{, whenever } t \xrightarrow{g\, v} t' \textit{ then } t' \models \Phi[v/x] \\
t \models [\forall x\, g]\Phi & = & \textit{for all values } v\textit{, whenever } t \xrightarrow{g\, v} t' \textit{ then } t' \models \Phi[v/x]
\end{array}
$$

Given any term t *with one free variable* z *the semantics of an open formula,* $t \models \Lambda$*, is given by:*

$$
\begin{array}{lcl}
t \models \exists x.\Phi & = & \textit{for some value } v\textit{, } t_{[v/z]} \models \Phi[v/x] \\
t \models \forall x.\Phi & = & \textit{for all values } v\textit{, } t_{[v/z]} \models \Phi[v/x]
\end{array}
$$

For a more complete explanation of the logic, consult (Calder et al., 2001a).