Combating Infinite State using Ergo

P. Robinson[1] and C. Shankland[2]

[1] School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, QLD 4072, Australia
[2] Department of Computing Science and Mathematics, University of Stirling, Stirling, FK9 4LA, UK. email: ces@cs.stir.ac.uk

Abstract. Symbolic transition systems can be used to represent infinite state systems in a finite manner. The modal logic FULL, defined over symbolic transition systems, allows properties over infinite state to be expressed, establishing necessary constraints on data. We present here a theory and tactics for FULL, developed using Ergo, a generalised sequent calculus style theorem prover allowing interactive proofs. This allows exploitation of the underlying *symbolic* transition system and reasoning about symbolic values.

1 Introduction

Although formal methods are becoming more popular, their uptake is at least partially governed by the success of associated tools. Theorem provers give the most flexibility, but are often difficult to use, requiring much expertise on the part of the user. Model checking tools on the other hand are easy to use, but traditional model checking techniques do not deal effectively with large and infinite systems. For example, model checkers based on BDDs [1] can deal with state spaces of around 10^{20}. This is inadequate when systems with data are considered; each variable will add at least another order of magnitude to the state space. The solution has often been to restrict attention to finite systems, but that is recognised to be an untenable position. Infinite state systems arise regularly and therefore must be dealt with satisfactorily. Many researchers are working on this problem, and have developed further solutions to reducing the state space, including symmetry reduction, for example [2], bounded model checking [3], and integration of model checking and theorem proving techniques, such as in the tool Salsa [4].

We deal with such infinite systems by using a novel semantics which removes infinite branching to represent the model, namely *Symbolic Transition Systems* (STSs) [6]. This is based on the work of Hennessy and Lin [5]. Then, having defined a logic on top of that semantics (FULL [7]), we carry out the proof of "model satisfies formula" in a theorem proving environment, Ergo [8]. The advantages of using a theorem prover are threefold. Ergo can represent and reason about the symbolic values on which the semantics is based. Further, Ergo provides tactics to automate simplification of satisfaction. Finally, rules to simplify the data constraints resulting from the satisfaction rules may also be implemented in Ergo, providing a fully integrated, partially automated way of checking "model satisfies formula". The work is based on a symbolic interpretation of LOTOS [9], but the approach can be applied to any similar language.

The paper is organised as follows. We present the technical background in Section 2, describing symbolic transition systems and the logic FULL. This work has been presented elsewhere [6, 7] but is repeated here briefly to allow the Ergo implementation to be fully described. In Section 3 we present details of the Ergo implementation of these concepts, motivated by some simple examples. Finally, we make some conclusions and note further work.

2 Background

2.1 LOTOS and Symbolic Transition Systems

LOTOS [9] is a popular process description language that has been in use for well over a decade in a number of domains, including protocols and services [10], and distributed systems [11]. The success of these applications has been amplified by the use of a number of mature verification tools, including the CADP toolset [12].

LOTOS includes a rich set of operators for describing both process control *and* data. However, much of the foundational work on LOTOS, and subsequently the verification tools, has ignored all, or parts, of the data aspect of the language because of the problem of infinite state. In particular, the use of data introduces infinite branching into the underlying state transition systems. For example, the simple process `g?x:Nat; exit` results in an infinite choice, one for each member of `Nat`. This presents a serious obstacle to reasoning, particularly to approaches based on (finite) model-checking. For example, in CADP the underlying semantics is in terms of Binary Coded Graphs, and in order to make reasoning about data tractable all data types are limited to a maximum of 256 values.

The focus of our work has been to provide a complete approach to data in LOTOS including semantics, abstract properties expressed using logic, equivalence relations, and tools to support reasoning about behaviours. See Figure 1. At the heart of this framework is a *symbolic* treatment of data. A new *finitely branching* semantics of LOTOS in terms of symbolic state transition systems (STSs) [6] is shown on the right hand side of the diagram, together with an HML-like logic (FULL) [7] and various bisimulation relations [6]. While the main advantage of the symbolic semantics is a more elegant and compact semantics for LOTOS, a further advantage is that open terms, or parameterised processes, can be described.

The standard semantics [9] are represented in the leftmost portion, where labelled transition systems give meaning to LOTOS specifications. The arrows between the components represent relationships. For example, we have proved elsewhere that the standard, concrete and symbolic bisimulations are all equivalent for closed processes (i.e. those with no free variables), and further, that the same equivalence relation is induced by the modal logic FULL. These results are

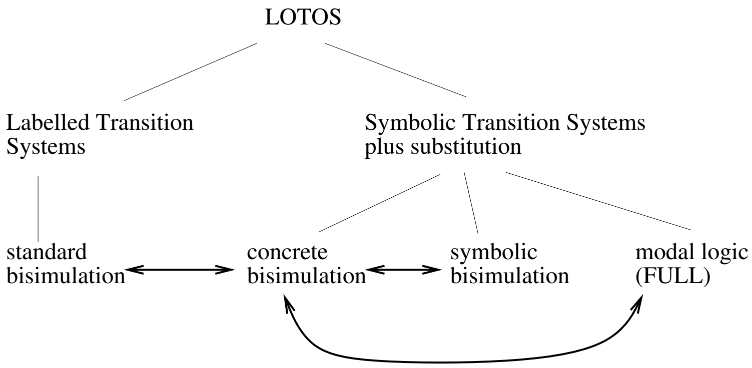


Fig. 1. Symbolic Framework for LOTOS

all essential to showing the strength and self-consistency of our symbolic framework, and its consistency with the standard semantics over closed behaviours (i.e. without free variables).

Symbolic Transition Systems are essentially labelled transition systems augmented by a *transition condition* representing the conditions under which that transition is available and free variables in the data label and state. This approach was first introduced for value passing CCS by Hennessy and Lin [5]. We have adapted it for the LOTOS setting.

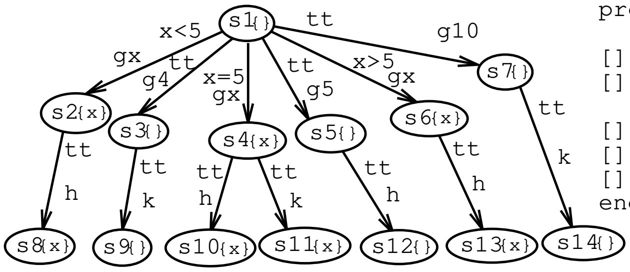
The set of events, denoted *Act*, ranged over by α, comprises *SimpleEv* and *StructEv*. The set of simple events, *SimpleEv*, ranged over by a, is defined as $G \cup \{\mathbf{i}, \delta\}$, where G is the set of possible gate labels, e.g. `in, out, g, send, receive, ack` and so on. $\mathbf{i}$ is the internal event and δ is the exit event. The set of structured events, *StructEv*, combines gate labels and data expressions. *StructEv* contains all gate-expression combinations gE, as well as all combinations δE. LOTOS allows multiple free variables in the data label of a transition; however, for simplicity we treat only the single variable case here. The extension to multiple free variables is straightforward but tedious and is therefore omitted.

Definition 1. *Symbolic Transition Systems*
A symbolic transition system consists of:

- *A (nonempty) set of states.*
  Each state T is associated with a set of free variables, denoted $fv(T)$.
- *A distinguished initial state, T_0.*
- *A set of transitions $T \xrightarrow{b \quad \alpha} T'$ where b is a Boolean expression and α is an action, such that $fv(T') \subseteq fv(T) \cup fv(\alpha)$ and $fv(b) \subseteq fv(T) \cup fv(\alpha)$ and $\#(fv(\alpha) - fv(T)) \leq 1$.*

The definition of the function $fv(\,)$ may be found in [6].

The rules presented in [6] define how a symbolic transition system may be constructed from LOTOS process syntax. The resulting transition system is typically a cyclic graph (if recursive processes are involved) and is always of finite width (since only a finite number of branches may be described in a LOTOS behaviour). We do not repeat those rules here, but illustrate STS by an example, Figure 2, where state `s1` corresponds to the behaviour `P`. The processes `Q, R, S, T, U, V, W` are place-holders, i.e. we assume there are more actions described by these processes, but the details are unnecessary.



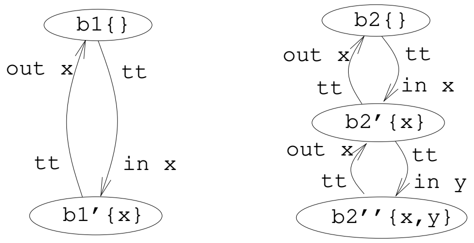

```
process P [g, h, k] :=
   g?x [x<5]; h; Q
[] g!4; k; R
[] g?x [x=5];
    (h; S [] k; T)
[] g!5; h; U
[] g?x [x>5]; h; V
[] g!10; k; W
endproc
```

Fig. 2. A Simple Symbolic Transition System

The set of free variables associated with each node, deduced syntactically from the LOTOS behaviour, is shown in braces at each state in Figure 2. For example, the appearance of `{x}` in state `s8` of Figure 2 implies that `s8` stands for a process `Q` in which `x` appears (i.e. `x` is a free variable in `Q`). The same is true of states `s10`, `s11` and `s13`.

2.2 Terms

If recursive processes are considered, e.g. as in Figure 3, symbolic transition systems on their own are clearly not enough to accurately describe behaviour.




```
process b1 :=
  in?x; out!x; b1
endproc
```

```
process b2 := in?x; b2'(x) endproc
process b2'(x) :=
  out!x; b2 [] in?y; out!x; b2'(y)
endproc
```

Fig. 3. The One Place Buffer and the Two Place FIFO Buffer

For example, if the value `3` is read at gate `in` in process `b1` then that value should propagate to the rest of the process. However, if we substitute `3` for `x` in the whole process then we no longer have a general buffer. Substitutions must be added to states in such a way as to allow repeated binding of a name to different values. In the two place buffer the substitution is essential, so that when moving from `b2''{x,y}` to `b2'{x}`, `y` is substituted for `x` in the remainder.

The foundation of our symbolic framework is in fact the pair (STS state, substitution) called a *Term*. Formally, a *substitution* is a partial function from Variables to Variables ∪ Values and a *term* T_σ consists of an STS state, T, paired with a substitution, σ, such that $domain(\sigma) \subseteq fv(T)$. Transitions on terms are derived from those over STS, as shown in Definition 2. It can be seen that the main difference between these and transitions in the STS is the presence of the substitution and the piecewise application of that substitution in each transition.

Definition 2. *Transitions on Terms*

$$\begin{array}{llll}
T \xrightarrow{b \quad a} T' & \textit{implies} & T_\sigma \xrightarrow{b\sigma \quad a} T'_{\sigma'} \\
T \xrightarrow{b \quad gE} T' & \textit{implies} & T_\sigma \xrightarrow{b\sigma \qquad\qquad gE\sigma} T'_{\sigma'} \\
 & & \textit{where } fv(E) \subseteq fv(T) \\
T \xrightarrow{b \quad gx} T' & \textit{implies} & T_\sigma \xrightarrow{b\sigma[z/x] \qquad gz} T'_{\sigma'[z/x]} \\
 & & \textit{where } x \notin fv(T) \textit{ and } z \notin fv(T_\sigma)
\end{array}$$

In all cases, $\sigma' = fv(T') \lhd \sigma$, *that is, the restriction of* σ *to include only domain elements in the set* $fv(T')$.

In the remainder of the paper we use t to denote terms, and the notation $t_{[v/z]}$ to denote the term $T_{\sigma[v/z]}$, i.e. adding $[v/z]$ to the substitution σ, overriding any previous substitution for z in σ.

2.3 The modal logic FULL

In order to describe abstract properties of symbolic transition systems a modal logic called FULL [7] was developed. We give an overview of FULL here, so the reader can better understand the Ergo definitions. FULL is able to describe simple properties ordering events with data and is adequate with respect to symbolic bisimulation [13].

FULL is based on the classic modal logic for concurrent systems known as Hennessy Milner Logic (HML) [14]. In addition to the usual operators of $\langle\ \rangle$ and $[\ ]$, which can be thought of as existential and universal quantification over transitions, we add quantifiers over data which may be communicated by those transitions. We refer to these as the quantified modal operators.

The syntax of FULL is as follows:

Definition 3. *Syntax of FULL*

$$\begin{array}{lll}
\Phi & ::= & b \mid \Phi_1 \ \wedge\ \Phi_2 \mid \Phi_1 \ \vee\ \Phi_2 \mid [a]\Phi \mid \langle a\rangle\Phi \\
 & & \mid \langle \exists x\ g\rangle\Phi \mid \langle \forall x\ g\rangle\Phi \mid [\exists x\ g]\Phi \mid [\forall x\ g]\Phi
\end{array}$$

Here b is a Boolean expression, $a \in G \cup \{\mathbf{i}, \delta\}$, $g \in G \cup \{\delta\}$ and x denotes a variable name. We have deliberately left b unspecified, as it depends on the language of data as described in the LOTOS specification from which the STS is generated. We assume that it at least includes the usual Boolean constants. Note that it is possible in LOTOS to defined a data type in which evaluation of data expressions is undecidable. All of the following rules assume only decidable data types are used.

The semantics of FULL is given over transitions on terms. We define $t \models \Phi$, denoting that a term t satisfies a modal formula Φ (Definition 4). The first five rules are standard.

Definition 4. *Semantics of FULL: Closed Terms*
Given any closed term t, the semantics of $t \models \Phi$ is given by:

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

For each quantified modal operator there are two cases to consider when matching transitions: either the expression has the closing substitution applied, yielding a value, or a new variable is bound. For example, if a process P satisfies $\langle \exists x\ g \rangle \Phi$ then there is some value v for which we can find a matching transition (labelled by g and either the same v or a symbolic value z consistent with v). Note that z is the variable from the transition system. We are free to choose whichever name we like (thanks to the rules for transitions on terms, Definition 2), but we must replace z by v in the rest of the transition system to correctly evaluate the rest of the logical formula $\Phi[v/x]$. Note that substitutions are used in a restricted way here; we only ever add a substitution of the form [value for variable], and that although some transitions may introduce new variables, the states and formulae remain closed because of the substitutions applied.

The definition of the logic may seem clumsy, especially the way each operator $\langle \exists x\ g \rangle$, $\langle \forall x\ g \rangle$, ..., is dealt with explicitly. This structure is necessary because of the chosen syntax (which echoes somewhat the LOTOS syntax for events), and because the logic is designed to be adequate with respect to symbolic bisimulation. The latter requires that for each quantified modal operator the data quantifier must be dealt with before the transition quantifier. An alternative approach, maintaining adequacy, would be to use the syntax $\exists x \langle g \rangle$ instead of $\langle \exists x\ g \rangle$ which would allow the rules for data quantification to be split from the rules for transition quantification. This is in fact the approach taken in the Ergo definition of Section 3 since it simplifies the Ergo axiomatisation.

2.4 Ergo

Ergo [8] is an interactive proof tool, designed and implemented at the Software Verification Research Centre. Ergo has many useful features, but the ones most relevant here are:

- A generalised sequent calculus proof style.
- Support for schematic theorems and answer extraction.
- A query language for searching for theorems.
- A pre-defined Gumtree proof interface and tactic language.
- Many supplied tactics that encode common proof patterns including support for transformational reasoning (window inference).
- An Emacs graphical user interface for proof browsing and construction.
- Support for storing, browsing and replaying proofs.
- Sophisticated theory construction facilities, including theory interpretation.
- Separate name space and syntax for each theory.
- A large library of standard theories. Currently, this library contains over 50 theories and 1500 axioms/theorems.

Ergo is implemented in Qu-Prolog, which is a high-level language designed primarily for rapid prototyping of interactive theorem provers. The object-level terms of Qu-Prolog include quantified terms, substitutions and object variables.

3 Implementing STS and FULL in Ergo

In this section we present an overview of an implementation of STS and FULL in the Ergo theorem prover. One of our key reasons for using a theorem prover is its ability to handle arbitary data values on transitions. This is one advantage theorem proving has over model checking, which can typically handle only (small) finite sets of data values, cf. CADP and its 256 values. On the other hand, one of the major reasons for using a model checker is its ability to find counterexamples. We illustrate how Ergo can be used in this way.

The implementation described in this section provides an approach that allows the construction of both proofs and counterexamples in a uniform way.

This is done by using a form of transformational reasoning called window inference [15]. In this approach, the formula of interest is transformed via logical equivalence to a simpler formula. Here, this means a formula b, only involving expressions over data. If this simplified formula is true then we have proved that the original formula is a theorem. On the other hand, if the simplified formula is not true, then it describes all the counterexamples preventing the original formula from being a theorem. Note that counterexamples produced in this way can describe an infinite number of counterexamples (e.g. $x > 5$) which is not possible using model checkers.

Our goal is to reduce sentences of the form "model satisfies formula", where *model* is expressed as an STS, and *formula* is a FULL formula, and *satisifies* is as described in Definition 4. Therefore, there are three components of the prototype implementation of STS and FULL in Ergo:

- A generic Ergo theory for STS and FULL, discussed in Section 3.1. This involves representation of *models* and *formulae*, and the *satisfies* relation.
- Support for automatically constructing Ergo theories that model specific STSs, discussed in Section 3.2. That is, given a specific system, converting it to the correct format for Ergo.
- Ergo tactics for automatically simplifying formulae in the STS theory, discussed in Section 3.3. That is, reducing the satisfies relation automatically, and dealing with data expressions (for particular models).

The key motivation in the development of the axiomatisation is to keep it as simple as possible, echoing the FULL syntax while exploiting existing Ergo constructs. The axiomatisation directly implements the concepts of *models*, *formulae* and *satisfies*. The axiomatisation is discussed in more detail in Section 3.1 and some illustrative examples relating to Figures 2 and 3 are presented in Section 3.4.

3.1 An Ergo theory for STS

In modelling any system in Ergo we need to model the syntax and the semantics of the system. The syntax is modelled using declared constants and the semantics is modelled via the axioms (primitive inference rules). The Ergo theory described here models STSs and terms as described in Definitions 1 and 2, and the semantics of FULL described in Definition 4.

Ultimately, for STS and FULL we aim to translate expressions such as

$$s1_{[]} \models \langle \forall y\ g \rangle \langle h \rangle \mathrm{tt}$$

so we need to declare constants to model the major components of such expressions. An example of a declaration of a constant is

```
declare models(_,_)
```

which declares `models` as a two place function symbol. Expressions such as

```
models(State,FULLform)
```

are true if and only if the FULL formula `FULLform` is true at the state given by `State`. For example,

```
models(state(s1,[]), ex x:ints dia(g, x, x > 0))
```

is true if and only if there is a `g` transition from state `s1` (with the empty substitution) with data value `x` satisfying `x > 0`.

`models` depends on the constants for terms and FULL formulae. A term (state in the STS decorated by substitutions) is constructed using

```
declare state(_,_).
```

where the first component is a STS state, and the second is a value list, modelling substitution in a positional way (more on this at the end of this subsection).

Clearly, since Ergo is a sequent calculus theorem prover, certain logical operators are already built in, such as `true`, `false`, `and`, `or`, and the quantifiers `ex` and `all`. The particular modal operators for FULL are modelled by the declared constants

```
declare dia(_,_,_).
declare box(_,_,_).
```

The semantics for both of these is in terms of their arguments: `Gate`, data, and `FULLform`. A minor difference exists between the syntax of closed terms for quantified modal operators given in Definition 4 and the corresponding terms in Ergo; namely, data quantifiers sit outside the modal quantifiers. We use a syntax that is consistent and easily modelled with Ergo, but without changing the semantics of such terms.

After the declarations of the required constants we declare the primitive inference rules (called axioms in Ergo). These allow `models` to be interpreted for each type of FULL formula. Some of these are presented below to illustrate the basic principles of the use of Ergo. The full set of inference rules are available, together with the examples, from the website [16].

The following three rules come from the STS theory. First, the obvious rule for logical "and".

```
axiom models_and_iff ===
    models(State, Exp1 and Exp2)
    <=> models(State, Exp1) and models(State, Exp2).
```

Second, a rule to deal with the modal operator $\langle Gate \rangle$. This corresponds very closely to the format of the FULL semantics of Definition 4, except that the format is more general, allowing data labels. This makes the extension to quantified modal operators straightforward (see below). Essentially, this rule says that a transition with the right gate label and a satisfiable transition condition has to be found. Moreover, the destination state of that transition should be one which satisfies the remainder of the formula, `Form`.

```
axiom models_dia_iff ===
    models(State, dia(Trans, V, Form))
    <=>
    ex x is_trans(State, Trans, x)
            and (V = V1) and formula(x, State, V1)
            and models(to(x, State, V1), Form)
       provided x not_free_in State,
                x not_free_in Trans,
                x not_free_in Form,
                x not_free_in V,
                x not_free_in V1.
```

In the Ergo theory for convenience we add an artificial distinguishing identifier for transitions. For example, in Figure 2 there are several `g` transitions from state `s1` and so we give different identifiers to the different transitions (although they are all uniquely determined by the elements of the transition). The formula `is_trans(State, Gate, Id)` is true if the transition `Id` is a `Gate` transition from state `State`. The expression `to(Id, State, Value)` represents the `State` reached by taking the transition `Id`. The formula `formula(Id, State, Value)` is the condition on the transition `Id` with data value `Value`. Note that in the semantics of Definition 4 we write `tt` for the transition condition, assuming there is some automatic simplification of ground terms. In Ergo the formulation is more generic, and the simplification is implemented explicitly (see Section 3.3). The "from state" is required because the value could be determined from the substitution. The special constant `nd` is used for a transition with no data value.

Third, the rule above can be very neatly augmented to handle the modal operator $\langle \exists x\ gate \rangle$, by adding a rule to first extract the data variable of the transition (which can be dealt with by standard rules for `ex`, existential quantification).

```
axiom models_ex_dia_iff(x) ===
    models(State, ex x:Type dia(Trans, V, Form))
    <=>
    ex x:Type models(State, dia(Trans, V, Form))
       provided x not_free_in Type,
                x not_free_in State,
                x not_free_in Trans.
```

The only important difference between the Ergo theory and the FULL semantics is in the positional management of term substitutions. The Ergo theory could have modelled each term substitution as a mapping from names to values. Updates to, or application of, a substitution would then be achieved by a corresponding update or application of the mapping. This approach, however, is somewhat inefficient, particularly in the application of a mapping to a term, which would be achieved by a combination of rules that did the replacement.

We therefore decided on an approach that takes much more advantage of pattern matching in rule application, thereby giving a much more efficient approach to the application of substitutions. In this approach substitutions are represented as lists of values, where the position in the list determines the STS variable with which the value is associated. This approach works for STS because the way substitutions change on a transition is simple and uniform as can be seen in the examples given later. This approach is both adequate and efficient.

3.2 Specific STS theories

A specific STS theory contains its own constant declarations and axioms. The constant declarations relate to the specific labels and names used for the given system. The axioms describe the structure of the given STS. Even for a small STS, the number of constant declarations and axioms can be quite large. For this reason we have written a preprocessor that takes a "user-friendly" representation of the system and automatically generates an Ergo theory that represents this system. The preprocessor carries out some simple checks on the input and produces error output for some obvious mistakes such as multiple uses of the same state identifier.

For example, from the simple example in Figure 2 we can extract the transitions in a standard format. For example the six transitions from state `s1` with the empty substitution can be represented by the following Prolog terms.

```
trans(g1, X < 5, g, X, state(s1, []), state(s2, [X])).
trans(g2, X = 4, g, X, state(s1, []), state(s3, [])).
trans(g3, X = 5, g, X, state(s1, []), state(s4, [X])).
trans(g4, X = 5, g, X, state(s1, []), state(s5, [])).
trans(g5, X > 5, g, X, state(s1, []), state(s6, [X])).
trans(g6, X = 10, g, X, state(s1, []), state(s7, [])).
```

The components of a transition are: the transition identifier (as described earlier), the transition condition, the gate label, the data label (including the *no data* label, `nd`), the originating state (sts state plus substitution), and the destination state. This is the user-friendly notation.

The preprocessor generates declarations for all the constants (such as `g1`) and axioms such as the ones below.

```
axiom is_trans_s1_g === is_trans(state(s1, []), g, A) <=>
  (A = g1) or (A = g2) or (A = g3) or (A = g4) or (A = g5) or (A = g6).
axiom is_trans_s1_h === is_trans(state(s1, []), h, A) <=> false.
axiom is_trans_s1_k === is_trans(state(s1, []), k, A) <=> false.
axiom to_g1 === to(g1, state(s1, []), A) = state(s2, [A]).
axiom formula_g1 === formula(g1, state(s1, []), A) = (A < 5).
```

The first rule declares that the only `g` labelled transitions from state `s1` are those enumerated (and no others), while the second and third rules declare that there are no `h` or `k` labelled transitions from state `s1`. Lastly, we have rules about the other components of the state (the destination state, and the formula). The complete file produced for this example can be seen on our website [16].

3.3 Ergo Tactics

In initial proofs, the simple axioms described above were applied to carry out transformational reasoning, but one of the advantages of using Ergo is the powerful tactic language. This allows common sequences of rule application to be applied automatically, thereby automating large parts of proofs.

All the STS rules are stated as equivalences so that they can be used as part of a rewrite system using transformational reasoning. Tactics were developed that use the STS rules as well as inherited classic logic rules such as one-point rules and formula simplification rules. These tactics automatically transform `models` formulae into classical formulae involving the datatypes appearing on transitions. This is the extent of simplification possible using Definition 4.

In developing STS and FULL the main concern was to separate reasoning about data from reasoning about processes. It can be seen in the FULL semantics of Definition 4 that all rules deal with reasoning about process evolution (transitions) while information about data (transition conditions) is simply collected. The assumption made in developing this theory was that an additional reasoning system exists to evaluate data conditions. A further advantage of using Ergo is that not only can Ergo be used to implement such a reasoning system, but that rules to reason about data can be added to the same framework as those reasoning about processes.

For example, if the data values were integers then we would require rules and tactics that could be used to simplify arithmetic expressions, and to simplify formulae such as

```
all x:ints x < 5 or x = 5 or x > 5
```

which might arise, for example, in a proof about the STS of Figure 2. Note that, in Ergo, we can declare rules that are interfaces to proof techniques outside of Ergo. For example, we could include a rule that interfaces to an off-the-shelf arithmetic simplifier/prover.

Similar sets of rules and tactics, or interfaces to external simplifiers, require to be implemented for every data type used. Some of these may have already been developed for Ergo, while others will have to be developed for the particular application under investigation. LOTOS data types are expressed using ACT ONE, an algebraic data type language, therefore the equations defining the type may be used directly in Ergo. Of course, this may not give the most efficient method of reducing terms of the data type. Recall also that it may not be possible to fully automate the reduction of terms since not all data types in LOTOS can be decidably evaluated.

3.4 Examples

To illustrate the formulation of properties in Ergo, we provide here some simple examples. These have all been proved to hold (or not hold) as appropriate, using Ergo. Firstly, for the example in Figure 2 we can prove the following formulae, given below with their FULL and Ergo representations for comparison.

$s1_{[]} \models \langle g4 \rangle \mathrm{tt}$

```
models(state(s1,[]), dia(g,4,formula(true))).
```

$s1_{[]} \models \langle \exists y\ g \rangle (y = 4)$

```
models(state(s1,[]),
    ex y:ints dia(g,y,formula(y = 4))).
```

$s1_{[]} \models [\forall y\ g](\langle h \rangle \mathrm{tt} \vee \langle \mathrm{k} \rangle \mathrm{tt})$

```
models(state(s1,[]),
    all y:ints box(g,y,
        dia(h,nd,formula(true)) or
        dia(k,nd,formula(true)))).
```

$s1_{[]} \models [k]\mathrm{tt}$

```
models(state(s1,[]), box(k,nd,formula(true))).
```

$s1_{[]} \models [\exists y\ g][k]\mathrm{ff}$

```
models(state(s1,[]),
    ex y:ints box(g,y,box(k,nd,formula(false)))).
```

The first two formulae say it is possible to take a `g 4` transition. These are equivalent. The third formula says that after all `g` transitions it is possible to do either a `h` transition or a `k` transition. It is not, however, possible to do a `k` transition straight away, so formula four is true vacuously. Finally, for some value, it is possible to take a `g` transition and end up in a state where a `k` transition is not possible.

The above formulae are all shown to be true, but of course we can also use Ergo to prove that formulae are false. For example, it is not the case that for all values it is possible to take a `g` transition and end up in a state where `k` is possible:

$s1_{[]} \models \langle \forall y\ g \rangle \langle k \rangle \mathrm{tt}$

```
models(state(s1,[]),
    all y:ints dia(g,y,dia(k,nd,formula(true)))).
```

After the application of automatic rewrite tactics in Ergo we get

```
all x:ints x = 4 or x = 5 or x = 10
```

This indicates the counterexamples for this particular property.

To illustrate the use of substitutions and terms, we also show some simple properties about the Buffer examples of Figure 3. The one place buffer satisfies some obvious properties about values in matching values out:

$b1_{[]} \models \langle \exists x1\ in \rangle \langle out\ x1 \rangle \mathrm{tt}$

```
models(state(b1,[]),
    ex x1:ints dia(in,x1,dia(out,x1,formula(true)))).
```

$b1_{[]} \models \langle \forall x1\ in \rangle [out\ x1] \mathrm{tt}$

```
models(state(b1,[]),
    all x1:ints dia(in,x1,box(out,x1,formula(true)))).
```

Moreover, the substitutions on terms ensure that a sequence of `in/out` actions operate correctly, i.e. although the substitution at first maps the x of the transition system to the value 3, it can override that with a later mapping to 42.

$b1_{[]} \models \langle in\ 3\rangle\langle out\ 3\rangle\langle in\ 42\rangle\langle out\ 42\rangle\mathrm{tt}$

```
models(state(b1,[]), dia(in,3,dia(out,3,
    dia(in,42,dia(out,42,formula(true)))))).
```

The two place buffer satisfies these properties, and additionally it is possible to permute the order of `in/out` actions since there are two places:

$b2_{[]} \models \langle in\ 3\rangle\langle in\ 4\rangle\langle out\ 3\rangle\langle out\ 4\rangle\mathrm{tt}$

```
models(state(b2,[]), dia(in,3,dia(in,4,
    dia(out,3,dia(out,4,formula(true)))))).
```

The input files for these examples are available from our website [16].

4 Conclusions and Further Work

We have developed an Ergo theory and a collection of tactics for automatically simplifying STS formulae. This approach tackles infinite state in two ways. Firstly, the use of symbolic transition systems yields a compact, finitely branching representation of the semantics of the system being analysed. Secondly, we use the theorem proving environment provided by Ergo to model those symbolic semantics in a truly symbolic fashion. Tactics for the system have been developed, allowing proofs to be completed automatically, resulting in a formula expressing data constraints on the system. We make no claims as to the efficiency of the tactics developed. Rather, the efficiency of our method lies in the selection of a more compact representation for behaviours. Further work may proceed in several directions.

Firstly, we are carrying out larger scale case studies, based on benchmark studies in the literature. Work has begun on the RPC (Remote Procedure Call) study [17]. This will not only serve to demonstrate the applicability of our semantic framework and Ergo theory described herein, it will also inform further refinement and development of tactics for Ergo. In particular, the only data tactics developed so far are for simple Booleans and integers. While users of the system can easily develop their own tactics for data, it would be convenient to build up a library of these.

Secondly, although the semantics of processes here is finitely branching, it is still possible to get infinite depth transition systems (these result from parameterised recursive processes). Related work [18] has developed symbolic transition systems with assignment to deal with this in certain cases. We have yet to extend this work to the logic, and to the Ergo implementation.

Thirdly, three prototype tools for model checking FULL have been implemented as part of the DIET project [16]. Initially, a prototype implementation was made in CADP [19]. The advantage of using CADP is the possibility to integrate with an established suite of LOTOS tools; however, the symbolic transition system semantics is not accurately represented since CADP does not handle symbolic values. In fact, we actually implemented a variant of the logic defined over standard labelled transition systems.

To create a truly symbolic reasoner we used Ergo, as described here, and Maude [20, 21]. Each approach exploits a different implementation paradigm (sequent calculus versus rewriting logic) and a further goal is to compare and contrast the different approaches. This can only be meaningfully carried out once substantial case studies stretching each tool to its limit have been completed.

Lastly, given that we have three different tools for analysing STSs, a useful objective is to integrate them. A student project at Stirling developed a translator to take a standard ASCII version of FULL expressions and produce the forms required by Ergo and the CADP based tool.

Acknowledgements The Engineering and Physical Sciences Research Council (grant GR/M07779/01, DIET: Developing Implementation and Extending Theory, A Symbolic Approach to Reasoning about LOTOS), and the Faculty of Management, University of Stirling Internal Research Fund provided financial assistance to this study, which is duly acknowledged with gratitude.

References

1. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic Model Checking: 10^{20} States and Beyond. Information and Computation **98** (1992) 142–170
2. Clarke, E., Filkorn, T., Jha, S.: Exploiting Symmetry In Temporal Logic Model Checking. In Courcoubetis, C., ed.: Proceedings of the 5th Workshop on Computer-Aided Verification, Springer-Verlag (1993) 450–462
3. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In Cleaveland, W., ed.: Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer-Verlag (1999)
4. Bharadwaj, R., Sims, S.: Salsa: Combining constraint solvers with BDDs for automated invariant checking. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000). (2000)
5. Hennessy, M., Lin, H.: Symbolic Bisimulations. Theoretical Computer Science **138** (1995) 353–389
6. Calder, M., Shankland, C.: A Symbolic Semantics and Bisimulation for Full LOTOS. In Kim, M., Chin, B., Kang, S., Lee, D., eds.: Proceedings of FORTE 2001, 21st International Conference on Formal Techniques for Networked and Distributed Systems, Kluwer Academic Publishers (2001) 184–200
7. Calder, M., Maharaj, S., Shankland, C.: A Modal Logic for Full LOTOS based on Symbolic Transition Systems. The Computer Journal **45** (2002) 55–61
8. Utting, M., Robinson, P., Nickson, R.: A Generic Proof Engine that uses Prolog Proof Technology. London Mathematical Society Journal of Computation and Mathematics **5** (2002)
9. International Organisation for Standardisation: Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. (1988)

10. Sighireanu, M., Mateescu, R.: Verification of the Link Layer Protocol of the IEEE-1394 Serial Bus (FireWire): an Experiment with E-LOTOS. Springer International Journal on Software Tools for Technology Transfer (STTT) **2** (1998) 68–88
11. Pecheur, C.: Using LOTOS for specifying the CHORUS distributed operating system kernel. Computer Communications **15** (1992) 93–102
12. Fernandez, J.C., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, L., Sighireanu, M.: CADP (CAESAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Alur, R., Henzinger, T., eds.: Proceedings of CAV'96. Number 1102 in Lecture Notes in Computer Science, Springer-Verlag (1996) 437–440
13. Calder, M., Maharaj, S., Shankland, C.: An Adequate Logic for Full LOTOS. In Oliveira, J., Zave, P., eds.: Formal Methods Europe'01. LNCS 2021, Springer-Verlag (2001) 384–395
14. Hennessy, M., Milner, R.: Algebraic Laws for Nondeterminism and Concurrency. Journal of the Association for Computing Machinery **32** (1985) 137–161
15. Robinson, P., Staples, J.: Formalising a hierarchical structure of practical mathematical reasoning. Logic and Computation **3** (1993) 47–61
16. Shankland, C., Calder, M.: Developing Implementation and Extending Theory: A Symbolic Approach to Reasoning about LOTOS. EPSRC project GR/M07779/01. http://www.cs.stir.ac.uk/diet/ (2002)
17. Broy, M., Merz, S., Spies, K., eds.: Formal Systems Specification: The RPC-Memory Specification Case Study. Number 1169 in Lecture Notes in Computer Science. Springer-Verlag (1996)
18. Ross, B.: Computing Symbolic Bisimulations. PhD thesis, University of Glasgow (2002)
19. Bryans, J., Shankland, C.: Implementing a modal logic over data and processes using XTL. In Kim, M., Chin, B., Kang, S., Lee, D., eds.: Proceedings of FORTE 2001, 21st International Conference on Formal Techniques for Networked and Distributed Systems, Kluwer Academic Publishers (2001) 201–218
20. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.: Maude: Specification and Programming in Rewriting Logic. SRI International, Jan. 1999, revised Aug. 1999. (1999) http://maude.csl.sri.com.
21. Bryans, J., Verdejo, A., Shankland, C.: Using Rewriting Logic to implement the modal logic FULL. In Bharadwaj, R., ed.: AVIS'01: First International Workshop on Automated Verification of Infinite-State Systems, Naval Research Laboratory Technical Memorandum (2001) Also in D. Nowak, editor, AVoCS'01: Workshop on Automated Verification of Critical Systems, Oxford University Computing Laboratory technical report PRG-RR-01-07.