Specification and Verification of Invariant Properties of Transition Systems

Daniel Găină
Kyushu University
Fukuoka, Japan
La Trobe University
Melbourne, Australia
Email: daniel@imi.kyushu-u.ac.jp

Ionuţ Ţuţu
Simion Stoilow Institute of Mathematics of the Romanian Academy, Bucharest, Romania
Royal Holloway University of London
Egham, United Kingdom
Email: ittutu@gmail.com

Adrián Riesco
Complutense University of Madrid
Madrid, Spain
Email: ariesco@fdi.ucm.es

***Abstract*—Transition systems provide a natural way to specify and reason about the behaviour of discrete systems, and in particular about the computations that they may perform. This paper advances a verification method for transition systems whose reachable states are described explicitly by membership axioms. The proof technique is implemented in the Constructor-based Inductive Theorem Prover (CITP), a proof management tool built on top of a variation of conditional equational logic enhanced with many modern features. This approach complements the so-called OTS method, a verification procedure for observational transition systems that is already implemented in CITP.**

***Index Terms*—algebraic specification; theorem proving; term rewriting; transition systems**

I. Introduction

Transition systems describe the behaviour of systems by means of states (which give a structural view of the system at a given time) and the possible transitions between them (which capture system dynamics). This simple approach allows those who specify concrete systems to abstract the implementation details and to focus instead on the high-level features that are critical for the correct functioning of the system. A prominent approach to modelling and reasoning about transition systems is the one based on *rewriting logic* [17]. This is a formalism of change, where the transitions between states correspond to *rewrite rules*, and where the static behaviour of the system is specified in a variant of *equational logic* [8]. The variations of rewriting logic that are relevant to the present study are implemented in Maude [4] and CafeOBJ [5].

Maude implements rewriting based on membership equational logic [18], which extends many-sorted equational logic with *membership axioms*. The expressiveness of such axioms is illustrated by the specification of *ordered lists*, which cannot be defined using plain many-sorted equational logic.[1] In the short example below, we use a *conditional membership axiom* to state that a list with at least two elements is ordered:

```
cmb N N' L : OList if N ≤ N' ∧ N' L : OList .
```

In this case, lists are constructed simply by juxtaposition, and `OList < List` is the subsort of ordered lists; the first condition indicates that the first two elements of the list must be ordered, while the second is a *membership condition* that requires the rest of the list to be ordered as well.

In this paper, we propose a formal method for analysing transition systems whose reachable states are described explicitly by membership axioms. The specification and verification technique is supported by the Constructor-based Inductive Theorem Prover (CITP) [14], which is implemented in Maude by using its meta-level – a reflective feature of Maude that allows specifiers to use modules and terms at the object level in a consistent way. The formal language of CITP is based on a variation of conditional equational logic with subsort relations, membership axioms, transitions, and constructor operators [12]. All this, combined with the efficient implementation of rewriting modulo axioms, makes Maude suitable for implementing CITP and for using its notation to define specifications.

The methodology described herein complements the one already supported by CITP [13] for analysing invariant properties of *Observational Transition Systems* (OTS) [21]. For the OTS method, the system states are regarded as 'black-boxes', and are distinguished only by *observational* functions. This approach was originally developed with the support of the CafeOBJ language. In OTS, one *principal* sort/type stands for the system, and the effects of transitions are obtained by observing – via functions – how the features of the system change with respect to the constructors of the principal sort.

Contributions. The contributions w.r.t. the methodology supported by CITP can be classified into two main categories.

1) Induction scheme: To prove properties of infinite-state transition systems, we need finite strategies for covering the infinite state space. By finite strategies we mean proof rules with a finite number of hypotheses for inductive reasoning. In the present methodology, each state is described by a collection of all data types involved in the behaviour of the system. This is achieved through the definition of a 'mixfix' operation:

```
op [_,...,_]: DataType1 ... DataTypeN → Sys .
```

The transition system has a finite non-empty set of initial states; in many cases, that set is a singleton. The issue is that not all collections of elements `[d1,...,dn]`, where `di` is an element of sort `DataTypei`, are reachable from initial states. In order to perform inductive reasoning on reachable states, or to define

[1] They can be defined using predicates, but not as a subsort of all lists.

properties that hold for all reachable states, we need a subsort `Reach < Sys`. In algebraic specification, the (sub)sorts can be described only by operations or membership axioms. For example, in the OTS method, the reachable states are described using constructor operations. In the present contribution, the reachable states are described by conditional membership axioms of the form **`cmb`** `t1 : Reach` **`if`** `t2 : Reach`, where `t1` and `t2` are terms of sort `Sys` (with universally quantified variables). These axioms play the role of constructors for the simultaneous induction scheme implemented in CITP.

2) Automated reasoning: The present methodology enjoys an increased level of automation due to the term-rewriting modulo associativity and/or commutativity supported by Maude. Our proof technique is intimately linked to the structure of sentences and takes advantage not only of the operational, but also of the denotational semantics of specifications in order to improve the users' interaction with the tool. Therefore, the present formal method is equipped with both:

a) *general tactics* derived from the proof rules of the calculus of the underlying logic of CITP [12], which are sound for all specifications (see Figure 1),

b) *specialized tactics* derived from the proof rules for the data types declared with initial-algebra semantics such as Booleans or sequences; those rules are sound only if the initial data types are *protected*, meaning that no "junk" (new elements) and no "confusion" (identification of existing elements) are added to the initial data types [19].

We argue on concrete examples of formal verification that the combination of the above tactics eases greatly the users' burden in discharging proof obligations. Algebraic-specification languages have libraries for defining the initial data types that are often used in specifying systems; examples include the Booleans, natural numbers, associative lists (or sequences), and sets. Due to Gödel's incompleteness theorem, the properties of initial data types are usually not recursively enumerable, which means that the general tactics are not sufficient for discharging properties that involve initial data types. For this reason, we need specialized tactics to complement the general ones in order to perform proofs in a more natural and efficient way. To the best of our knowledge, this combination of tactics is unique in the formal-methods literature.

NEW FEATURES. Compared to the first version of CITP, the new features implemented in the tool are:

- an induction scheme based on constructors given as membership axioms;
- a tactic for computing and joining critical pairs, i.e. for pairs of terms resulting from overlapping rewrite rules;
- improved decision procedures for automated reasoning,
- a new interface designed to improve the users' interaction, and implemented from scratch in Core Maude;
- the command-parsing component of the interface was upgraded to generate better error messages.

Most of the improvements presented in this paper focus on reducing the user interaction. In fact, since inductive theorem provers are oftentimes interactive (e.g. Isabelle, Coq, HOL Light, HOL4; cf. [24]), they require trained users to direct the prover towards discharging goals which cannot be proved by automatic techniques. In many cases, the tool needs the users' help to perform even trivial proofs. The new induction scheme and the decision procedures pointed out above, as well as the improvements in the interface, try to alleviate this problem, passing the computational burden to the machine and allowing the user to focus on the overall proof strategy.

STRUCTURE OF THE PAPER. Section II presents the basic notions used in the paper. Section III deals with the methodology that the tool follows, and illustrates the verification of the Alternating Bit Protocol. Section IV discusses related work; and Section V concludes and outlines a few lines of future research. The tool is available on the first author's web page [9].

II. PRELIMINARIES

In what follows, by *specification* we mean a text that describes a *signature* (roughly, a collection of sorts and symbols), a set of possibly conditional *sentences* (built using those symbols) and a class of *models* (that assign meaning to the symbols). The sentences are Horn clauses of the form $(\forall \mathtt{X}) \wedge \mathtt{H} \Rightarrow \mathtt{C}$, where $\mathtt{H} \cup \{\mathtt{C}\}$ is a set of atomic formulas given as equations $\mathtt{t}_1 = \mathtt{t}_2$, membership axioms $\mathtt{t} : \mathtt{s}$, or transitions $\mathtt{t}_1 \Rightarrow \mathtt{t}_2$, where $\mathtt{t}$, $\mathtt{t}_1$, and $\mathtt{t}_2$ are terms and $\mathtt{s}$ is a sort. Such sentences also define the operational semantics of the specification, because they form a term-rewriting system that makes the specification executable by rewriting. Moreover, they are compatible with the denotational semantics of the specification (its class of models), in the sense that every model of the specification satisfies its sentences.

A *goal* is a pair $\langle \mathtt{SP} \vdash \mathtt{E} \rangle$, where SP is a specification and E is a set of formulas to prove. A *proof rule* is a mapping from a goal $\langle \mathtt{SP} \vdash \mathtt{E} \rangle$ to a list of goals $\langle \mathtt{SP}_1 \vdash \mathtt{E}_1 \rangle \ldots \langle \mathtt{SP}_n \vdash \mathtt{E}_n \rangle$. The *tactics* are obtained by canonically extending proof rules to mappings from lists of goals to lists of goals. In the current version of CITP, the user can give commands which may consist of lists of tactics rather than a single tactic. It follows that the *proof scripts* consist of sequences of lists of tactics instead of trees (whose nodes would be goals, and whose edges would be lists of tactics). This has the advantage of simplifying the design of the tool interface and increasing the automation level of the proof process by making all goals to be discharged available to the user simultaneously. However, if the user wants to apply a tactic list *tctList* only to the current goal, then the command `(.` *tctList*`)` can be issued to the tool. In addition, `(`**`select`** *no*`)` can be used to move the goal *no* to the top of the goal list, thus making it the current goal.

Apart from tactics, CITP is equipped with commands for managing the proof process, which have no effect on the list of goals to be proven (see Figure 2), but enable the user to have a better control and understanding of the actual proof.

Assume that the initial goal is a pair $\langle \mathtt{SP} \vdash \gamma \rangle$ given by a specification SP and a single sentence γ of the form $(\forall \mathtt{X}) \bigwedge \mathtt{H} \Rightarrow \mathtt{C}$, where X is a set of variables and $\mathtt{H} \cup \{\mathtt{C}\}$ is a set of atomic formulas. The idea is to develop tactics that decompose γ into

TACTIC	ABBREVIATION
Induction	(**ind on** *varList*)
Induction based on mb. ax.	(**indx on** *varList*)
Case analysis	(**ca**)
Theorem of constants	(**tc**)
Implication	(**imp**)
Reduction	(**red**)
Initialization	(**init** *sentence* **by** *substitution*)
Critical-pairs left	(**cp-l** *sentence1* $\gg$ *sentence2*)
Critical-pairs right	(**cp-r** *sentence1* $\gg$ *sentence2*)
Select	(**select** *no*)
Dot	(. *tctList*)

Fig. 1: General tactics

COMMAND	DESCRIPTION
(**rollback**)	returns the proof process to the state before applying the last list of tactics
(**show goals**)	displays the goals to discharge
(**show proof**)	shows the sequence of lists of tactics applied so far
(**redTerm** *t*)	reduces the term *t* to its normal form w.r.t. the specification of the current goal

Fig. 2: CITP commands

simpler, basic constituents, e.g. equational formulas, which can be discharged using term rewriting. There are two tactics designed to eliminate universal quantification: induction and the theorem of constants. While the latter is applicable to all variables in X, the former can eliminate only variables of *constrained sort*.[2] It is the users' responsibility to separate variables of constrained sorts, which will be dealt with by induction, from the rest of the variables, which will be handled by the theorem of constants. Typically, induction is applied first and then the theorem of constants comes into play. The goals $\langle \mathtt{SP}_1 \vdash \gamma_1 \rangle \ldots \langle \mathtt{SP}_n \vdash \gamma_n \rangle$ obtained by applying induction and the theorem of constants have only quantifier-free formulas, i.e. γ_i is of the form $\bigwedge \mathtt{H}_i \Rightarrow \mathtt{C}_i$ for all $i \in \{1, \ldots, n\}$. The tactic (**imp**) is designed to eliminate the logical implication. For example, by applying implication to a goal $\langle \mathtt{SP}_i \vdash \gamma_i \rangle$ as above, the result is $\langle \mathtt{SP}_i + \mathtt{H}_i \vdash \mathtt{C}_i \rangle$, where $\mathtt{SP}_i + \mathtt{H}_i$ is the specification obtained from $\mathtt{SP}_i$ by adding the axioms in $\mathtt{H}_i$. The goals $\langle \mathtt{SP}_i + \mathtt{H}_i \vdash \mathtt{C}_i \rangle$ are discharged using term rewriting.

Example 1: Consider the following specification of natural numbers with addition. We first define sorts and subsort relations using the keywords **sorts** and **subsorts**, respectively. This allows us to define Peano natural numbers (PNat) and non-zero Peano natural numbers (PNzNat). We can then define operations on these sorts using the keywords **op**; some of these operations are constructors (**ctor**), such as 0 and successor, s_, where the underscore is a placeholder. In the same way, we define an operation symbol _+_ for the addition of natural numbers, but without marking it as a constructor. Finally, we define the behaviour of addition by means of equations (**eq**), where the **metadata** attribute allows the users to provide additional information to the tool, as we will see later.

```
fmod PNAT is
 sorts PNat PNzNat .
 subsorts PNzNat < PNat .
 op 0  : → PNat              [ctor].
 op s_ : PNat → PNzNat       [ctor].

 op _+_ : PNat PNat → PNat .
 vars M N : PNat .
 eq 0 + N = N                [metadata "1"].
 eq s M + N = s(M + N)       [metadata "2"].
endfm
```

Since the above specification starts with **fmod** and ends with **endfm**, its semantics is initial, meaning that the class of models Mod(PNAT) of the specification PNAT consists of all models that are isomorphic with the model of natural numbers.

Suppose one wants to prove the associativity of addition. The goal is introduced by the following command:

```
(goal PNAT ⊢ eq (X:PNat + Y:PNat)+ Z:PNat =
                X:PNat + (Y:PNat + Z:PNat);)
```

The variable X is chosen for induction and the tactic (**ind on** X:PNat) is applied. The tool then generates two subgoals, one for each constructor: zero (0) and successor (s_). By the theorem of constants (**tc**), the variables Y and Z are introduced as constants to the specifications in the subgoals generated by induction. Since logical implication does not occur in any of the formulas to prove, the goals can be discharged by reduction (**red**), which replaces each term in the equations to prove with its normal form. The proof of associativity thus consists of the tactic list (**ind on** X:PNat **tc red**). See [9] for a detailed presentation of the proof.

In the case of large specifications, it is undesirable to display all their sentences during the proof process. Therefore, only sentences with the attribute **metadata** "n" will be displayed, where n is a natural number. The axioms introduced in the proof process will be assigned automatically a natural number as an attribute, and therefore will always be displayed.

Example 2: Consider the following specification of two functions on natural numbers:[3]

```
fmod FG-FUN is
 protecting NAT .
 op F : Nat → Nat .
 op G : Nat → Nat .
 vars X Y : Nat .

 ceq F(X) = 5 if X ≤ 7 [metadata "CA-1"].
 ceq F(X) = 1 if 8 ≤ X [metadata "CA-2"].

 ceq G(Y) = 2 if Y ≤ 4 [metadata "CA-a"].
 ceq G(Y) = 7 if 5 ≤ Y [metadata "CA-b"].
endfm
```

[2]We distinguish between *constrained* sorts, which have constructors explicitly described in the signature, and *loose* sorts, with no such constructors.

[3]To benefit from the usual, decimal representation of natural numbers, we import the predefined Maude module NAT instead of PNAT from Example 1.

The purpose of this example is to show that in formal verification one can easily reach subgoals that have inconsistent specifications, despite the fact that the proof started with a goal whose underlying specification is consistent, so we can automatically discharge the corresponding subgoal. Note that:

1) either `X ≤ 7` or `8 ≤ X`, for all natural numbers `X`, and
2) the left-hand sides of the conditional equations labelled with `CA-1` and `CA-2` are equal.

Therefore, the conditional equations labelled with `CA-1` and `CA-2` should be regarded as one sentence. The labels `CA-1` and `CA-2` are used by CITP to perform case analysis. This tactic is sound if and only if:

1) the conditions of `CA-1` and `CA-2` are disjointly true, and
2) the left-hand sides of `CA-1` and `CA-2` are equal.

A similar remark holds for the sentences labelled with `CA-a` and `CA-b`. The goal we are interested in is the following:

```
(goal FG-FUN
      ⊢ eq 9 ≤ G(F(X:Nat))+ G(X:Nat)= true ;)
```

By the theorem of constants (`tc`), the variable `X:Nat` is added to the body of the specification `FG-FUN` as a fresh constant. The new goal is the following:

```
⟨ FG-FUN + {op X#1 : → Nat}
         ⊢ eq 9 ≤ G(F(X#1))+ G(X#1)= true ; ⟩
```

In order to apply case analysis (`ca`), the tool selects the ground term `9 ≤ G(F(X#1))+ G(X#1)` from the formula to prove `eq 9 ≤ G(F(X#1))+ G(X#1)= true`. Note that:

1) `CA-1` and `CA-2` match the subterm `G(F(X#1))` of the term `9 ≤ G(F(X#1))+ G(X#1)` via the substitution `X:Nat ← X#1`, and there are no proper subterms of `G(F(X#1))` that can be matched by any `CA`-axioms;
2) `CA-a` and `CA-b` match the subterm `G(X#1)` of the term `9 ≤ G(F(X#1))+ G(X#1)` via the substitution `Y:Nat ← X#1`, and there are no proper subterms of `G(X#1)` that can be matched by any `CA`-axioms.

Case analysis splits the goal above into four subgoals by adding to the specification `FG-FUN + {op X#1 : → Nat}` one condition from each of the two groups of sentences below:

```
1) ceq F(X#1) = 5 if X#1 ≤ 7 .
   ceq F(X#1) = 1 if 8 ≤ X#1 .

2) ceq G(X#1) = 2 if X#1 ≤ 4 .
   ceq G(X#1) = 7 if 5 ≤ X#1 .
```

The new list of goals is as follows:

```
1) ⟨ FG-FUN + {op X#1 : → Nat,
               eq X#1 ≤ 7,
               eq X#1 ≤ 4}
   ⊢
   eq 9 ≤ G(F(X#1:Nat))+ G(X#1:Nat)= true ; ⟩

2) ⟨ FG-FUN + {op X#1 : → Nat,
               eq X#1 ≤ 7,
               eq 5 ≤ X#1}
   ⊢
   eq 9 ≤ G(F(X#1:Nat))+ G(X#1:Nat)= true ; ⟩

3) ⟨ FG-FUN + {op X#1 : → Nat,
               eq 8 ≤ X#1,
               eq X#1 ≤ 4}
   ⊢
   eq 9 ≤ G(F(X#1:Nat))+ G(X#1:Nat)= true ; ⟩

4) ⟨ FG-FUN + {op X#1 : → Nat,
               eq 8 ≤ X#1,
               eq 5 ≤ X#1}
   ⊢
   eq 9 ≤ G(F(X#1:Nat))+ G(X#1:Nat)= true ; ⟩
```

Note that the specification of the third goal above is inconsistent as we cannot have a natural number `X#1` that is greater than or equal to `8` and less than or equal to `4`. Formally, since the specification `NAT` is imported in **`protecting`** mode, no new elements of sort `Nat` are introduced, and none of the existing elements can be made equal (say, by equations); this implies that `X#1` is a natural number; also, the preorder relation ≤ is interpreted in the obvious way; it follows that the equations `{eq 8 ≤ X#1, eq X#1 ≤ 4}` create confusion w.r.t. the initiality of `NAT` as there is no natural number (in place of `X#1`) satisfying both equations. CITP recognizes this contradiction and discharges the third goal automatically.

The remaining three goals are discharged by the reduction tactic (`red`). Therefore, the proof of the goal

```
(goal FG-FUN ⊢
      eq 9 ≤ G(F(X:Nat))+ G(X:Nat)= true ;)
```

consists of the list of tactics (`tc ca red`); see [9] for the full specification and proof script. Here it is also worth noting that case analysis (`ca`) plays an essential role in the OTS method; see [13] for a formalization of (`ca`) and its applications.

III. Formal Method

The verification method described in Example 1 forms the basis for two main directions through which one can prove invariant properties of transition systems:

1) One technique is designed for OTS, where the structure of the states is not accessible to the users; instead, the states are distinguished by so-called observation functions [13].
2) The other approach is built for transition systems whose states are described explicitly using membership equations.

Clearly, the two methods share common tactics like the theorem of constants, implication, or reduction. The implementation of the tool is modular in that regard, allowing users to reuse the code of the shared tactics. In this paper, we focus on the latter methodology; cf. [13], which deals with the former.

A. Alternating Bit Protocol

The specification and verification methodology supported by the tool is presented through a case study. The example chosen is of the *Alternating Bit Protocol (ABP)*, a protocol operating at the data-link layer of networks that ensures the reliability of the communication between a sender and a receiver [1]. The overall structure of the ABP is illustrated in Figure 3.

The protocol consists of two processes, *Sender* and *Receiver*, each having a data buffer and a one-bit state. Sender and

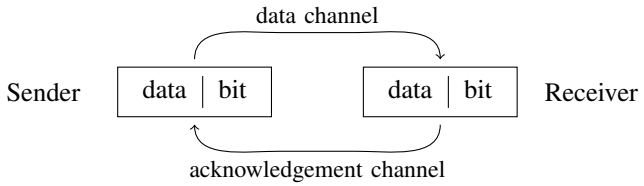

Fig. 3: The Alternating Bit Protocol

Receiver use channels to communicate with each other, as they do not share any common memory: *(a)* a *data channel* from Sender to Receiver for sending *bit-packet* pairs, and *(b)* an *acknowledgement channel* from Receiver to Sender for sending single confirmation bits. The protocol works as follows:

- Initially both channels are empty and Sender's bit is different from Receiver's bit.
- Sender repeatedly writes bit-packet pairs $\langle bit, pac \rangle$ on the data channel, where *bit* is the Sender's bit and *pac* is the Sender's data to transmit. If the Sender reads *bit* from Receiver over the acknowledgement channel, then it is a confirmation that the packet has been delivered. In that case, Sender flips its bit and selects the next packet to send.
- Receiver writes its bit on the acknowledgement channel repeatedly. When it reads a pair $\langle bit, pac \rangle$ such that *bit* is different from its bit, it stores the package and flips its bit.

B. Specification

This section gives an overview of the ABP specification. The complete specification can be found in [10]. The state space is explicitly described by tuples that consist of all the data types that define the protocol. Here, the state space is represented by the sort `Sys`, which has only one constructor:

```
sorts Reach Sys . subsort Reach < Sys .
op [_,_,_,_,_,_] : Bit PNat Channel1
                   Bit Data Channel2 → Sys [ctor].
```

The first two arguments are the Sender's bit and packet, which, for simplicity, we identify by a natural number; `Channel1` corresponds to the data channel; the arguments four and five are the Receiver's bit and packet list, respectively; `Channel2` corresponds to the acknowledgement channel. As explained in Section I, all terms built using this constructor have the sort `Sys`; but when reasoning about the ABP, we are actually interested in the subset of terms that denote reachable states (i.e. states of sort `Reach`[4]). Hence, the strategy is to use membership axioms in order to define these particular terms:

1) In the initial state, Sender's bit is `true`, the packet number is `0`, and both communication channels are empty, Receiver's bit is `false`, and Receiver's packet list is empty.

```
mb [t,0,empty,f,empty,empty]: Reach
                        [metadata "ctor-init"].
```

2) Sender writes the current bit-packet pair `⟨ B1,pac(N) ⟩` to the data channel `Ch1`.

```
cmb [B1,N,(⟨B1,pac(N)⟩,Ch1),B2,D,Ch2]: Reach
if [B1,N,Ch1,B2,D,Ch2]: Reach
                        [metadata "ctor-send1"].
```

3) Receiver writes its current bit `B2` to the acknowledgement channel `Ch2`.

```
cmb [B1,N,Ch1,B2,D,(B2,Ch2)]: Reach
if [B1,N,Ch1,B2,D,Ch2]: Reach
                        [metadata "ctor-send2"].
```

4) Sender acknowledges that the package has been delivered when its bit `B1` is the last bit of the acknowledgement channel `(Ch2,B1)`.[5] It immediately flips its bit `B1` and updates the number `N` of the packet to be transmitted.

```
cmb [not B1,s N,Ch1,B2,D,Ch2]: Reach
if [B1,N,Ch1,B2,D,(Ch2,B1)]: Reach
                        [metadata "ctor-rec1"].
```

5) Receiver gets a new packet `P` and stores it in the packet list `D` when the last bit `B2` from the data channel `Ch1,⟨ B2,P ⟩` is different from its own bit (`not B2`).

```
cmb [B1,N,Ch1,B2,(P,D),Ch2]: Reach
if [B1,N,(Ch1,⟨ B2,P ⟩),not B2,D,Ch2]: Reach
                        [metadata "ctor-rec2"].
```

6) The data channel is not reliable: at any time, any element of the channel (from an arbitrary position) might be lost.

```
cmb [B1,N,(Ch1,Cn1),B2,D,Ch2]: Reach
if [B1,N,(Ch1,BP,Cn1),B2,D,Ch2]: Reach
                [metadata "ctor-drop1" nonexec].
```

7) Dropping one element from the acknowledgement channel is modelled similarly to item 6 above.

8) In addition to losing data, the elements of the data channel (once again, from arbitrary positions) can also be duplicated.

```
cmb [B1,N,(Ch1,BP,BP,Cn1),B2,D,Ch2]: Reach
if [B1,N,(Ch1,BP,Cn1),B2,D,Ch2]: Reach
                [metadata "ctor-dup1" nonexec].
```

9) Duplication in the acknowledgement channel is modelled similarly to item 8 above.

The constructors of a given sort can be either functions or membership axioms. In this case, the sort `Reach` is described by the nine membership axioms listed above. The prefixes `ctor-` from the metadata attributes, e.g. `ctor-init` or `ctor-send1`, are necessary for the tool to recognize that these membership axioms are constructors for `Reach`, and to generate appropriate induction schemes for formal verification.

The specification `ABP` obtained in this way [10] can be enhanced with transitions `rl T1 ⇒ T2 [metadata "trans"]` for each conditional membership axiom `cmb T2 : Reach if T1 : Reach [metadata "ctor-trans"]`. For example, the corresponding transition for the conditional membership axiom `ctor-send1` would be:

```
rl [B1,N,Ch1,B2,D,Ch2] ⇒
   [B1,N,(⟨ B1,pac(N)⟩,Ch1),B2,D,Ch2]
                        [metadata "send1"].
```

[4] `Sys` and `Reach` are the standard names for such sorts. However, they are not predefined; users can use any other names they consider to be suitable.

[5] Notice the convenience of using the associative list-concatenation operation (and pattern matching) instead of the left-associative append operation.

This makes it possible to apply model-checking techniques for formal verification. However, model checking is outside the scope of this paper. It is worth pointing out though that conditional membership axioms and transitions are in one-to-one correspondence. Because of this, one may be tempted to eliminate membership axioms and keep only the transitions. But, denotationally, in the absence of membership axioms, there are no terms of sort `Reach`, and one cannot formalize invariant properties that hold for all reachable states. Moreover, transitions such as `send1` do not play any role in the deductive verification described in the following section, hence our methodological preference for membership axioms.

In the present contribution, dropping elements from arbitrary positions in the communication channels is modelled faithfully. This is possible thanks to *(a)* the matching equations provided by Maude for the conditions of the Horn sentences, and *(b)* the support for the associative data types that are used to describe the communication channels. The formal verification involving the matching of associative data types is facilitated by the tactic (**cs**) – case analysis for sequences and sets [13] – which breaks a goal into subcases that are easier to analyse.

C. Formal verification

The methodology described in Section II is developed further in order to support the verification of safety properties for transition systems whose states are described explicitly. For that purpose, the new features added to the proof strategy presented in Section II can be summarized as follows:

- Induction is replaced by a simultaneous induction scheme (**indx on** *varList*) based on constructors given as membership axioms, where *varList* is a list of variables.
- In order to preserve the confluence property, after each application of a tactic it is necessary to reduce to the normal form each term occurring in the formulas to prove. This is achieved by applying the tactic (**red**).
- In addition, it is often necessary to apply case analysis (**cs**) on the sequence data type. In our case study, this initial data type models the communication channels. This tactic is applied after induction. So far, the verification strategy consists of the following tactic list:

  (**indx on** *varList* **red cs red tc red imp red**)

  However, that alone is not sufficient for discharging all the goals generated by the induction scheme.
- Even if the specification of the initial goal is confluent, after applying the tactic list above, some specifications of the goals generated by the verification strategy proposed may lose the confluence property. For this purpose, we have developed tactics for joining critical pairs:

  (**cp-l** *sentence1* $\gg\!<$ *sentence2*), and
  (**cp-r** *sentence1* $\gg\!<$ *sentence2*).

- Last but not least, extra care is needed for proving formulas that are not term-rewriting rules. Consider the invariant:

```
ceq B = B1
if [B1,N,(Ch1,⟨ B,P ⟩,Cn1),B1,D,Ch2]: Reach
                    [metadata "inv2" nonexec].
```

  which states that, if the Receiver's bit is equal to Sender's bit, `B1`, then all the bits in the data channel are equal to `B1`. Notice that `inv2` is not a term-rewriting rule, because there are variables in the condition `[B1,N,(Ch1,⟨ B,P ⟩,Cn1),B1,D,Ch2]: Reach` such as `N` that do not occur in the left-hand side of the rule. This is the reason for using the attribute **nonexec**. The operational semantics of a specification is therefore given by all sentences of the specification that do not have the attribute **nonexec**. After applying the tactic list

  (**indx on** *varList* **red cs red tc red imp red**)

  one may need to initialize non-executable sentences in order to complete the verification process. According to our experience, the tactic list above followed by the initialization of such sentences and the joining of critical pairs is sufficient for completing the verification process.

The proof strategy for proving the properties of the specification `ABP` consists in the following sequence of tactic lists:

(**indx on** *varList* **red cs red tc red imp red**)
(**cp-l** *sentence1* $\gg\!<$ *sentence2* **red**)
(**init** *sentence* **by** *substitution* **red**)

Most of the tactics in the above proof strategy were defined for the OTS method [13]. The new tactics implemented for the present verification method are the simultaneous induction scheme based on constructors given as membership axioms (**indx on** *varList*) and the joining of critical pairs (**cp-l** *sentence1*$\gg\!<$*sentence2*) and (**cp-r** *sentence1*$\gg\!<$*sentence2*).

We employ the above strategy to prove that the `ABP` specification satisfies the following safety property: all the packets sent are received in the same order in which they were sent. This property is formalized by the two sentences below.

```
ceq mk(N)= D if [B,N,Ch1,B,D,Ch2]:= S
                                [metadata "goal1"].
ceq mk(N)= pac(N),D if[B,N,Ch1,not B,D,Ch2]:= S
                                [metadata "goal2"].
```

Here, the function `mk : PNat → Data` is defined by two equations: *(a)* **eq** `mk(0) = pac(0)` and *(b)* **eq** `mk(s N) = pac(s N) mk(N)`. That is, for every natural number `N`, `mk(N)` is the list of packets that correspond to the natural numbers `N`, `N-1`, ..., `0`. The sentence `goal1` states that, if Sender's bit is equal to Receiver's bit, then `N` packets delivered were received in the same order in which they were sent. The sentence `goal2` states that, if Sender's bit is different from Receiver's bit, then the last packet delivered was not received.

In order to prove the two goals described above, we need five lemmas. The first four are proved by simultaneous induction, while the fifth lemma is derived from the others. The proofs of these lemmas are similar to the proofs of `goal1` and `goal2`. Because of this, we focus only on proving the two goals. More information about the lemmas can be found at [10].

INDUCTION. After applying (**indx on** `S:Reach`) to `goal1` and `goal2`, the tool generates nine subgoals, each subgoal corresponding to one of the nine constructors. If

```
cmb T2 : Reach if T1 : Reach
                    [metadata "ctor-trans"].
```

is one of the constructors, then the corresponding subgoal is

```
⟨ ABP + {mb T1 : Reach [metadata "trans"],
         goal1(S ← T1) [metadata "trans"],
         goal2(S ← T1) [metadata "trans"]}
 ⊢ {goal1(S ← T2),goal2(S ← T2)} ⟩
```

The underlying specification of the subgoal above is obtained from ABP by adding:

1) the variables in ctor-trans as constants,
2) the membership axiom **mb** T1 : Reach, and
3) the induction hypothesis
{goal1(S ← T1), goal2(S ← T1)}.

The label trans is for the user to recognize the induction case to prove. There are nine subgoals, and each subgoal has two formulas to prove. This means eighteen cases to discharge.

CITP performs several other transformations; all are sound w.r.t. the semantics and help the user discharge the goals easier. Take, for example, the subgoal corresponding to ctor-init:

```
⟨ ABP ⊢
  {ceq mk(N) = D
   if [B,N,Ch1,B,D,Ch2]:=
      [t,0,empty,f,empty,empty]
                        [metadata "goal1"];
   ceq mk(N) = pac(N),D
   if [B,N,Ch1,not B,D,Ch2] :=
      [t,0,empty,f,empty,empty]
                        [metadata "goal2"];} ⟩
```

Based on the initiality of tuples, the goal is refined into:

```
⟨ ABP ⊢
  {ceq mk(N) = D
   if B := t ∧ N := 0 ∧ Ch1 := empty
      ∧ B := f ∧ D := empty
      ∧ Ch2 := empty    [metadata "goal1"];

   ceq mk(N) = pac(N),D
   if B := t ∧ N := 0
      ∧ not B = f ∧ D := empty
      ∧ Ch2 := empty    [metadata "goal2"];} ⟩
```

For each matching equation V := T in the condition, where V is a variable and T is term, the tool transforms the goal by removing the matching equation and substituting the term T for the variable V in the formula to prove. This procedure is performed sequentially for all matching equations V := T, from left to right. Then all terms are reduced to their normal forms. The user can see only the final result:

```
⟨ ABP ⊢
  {ceq pac(0)= empty if t := f
                        [metadata "goal1"];
   ceq pac(0)= pac(0) if nil
                        [metadata "goal2"];} ⟩
```

This splitting of conditions based on tuple matching, and followed by the application of substitutions V ← T, and then by reductions to normal forms are integrated into the tactic (**indx on** S:Reach) to ease the user's task. It avoids complicated proofs of obvious properties. We argue that a solid background in algebraic specification allows one to combine general tactics that are sound for all specifications with specialized tactics that are sound for initial data types – and often used in practice to increase the performance of the verification method. This is one example of decision procedure that increases the automation level of the current methodology. All nine subgoals can be handled in this manner.

CONTRADICTION. Since there are no matching equations involving sequences in the goals obtained after the application of (**indx on** S:Reach), the tactic (**cs**) just splits each goal into two subgoals. Therefore, the result is a list of eighteen subgoals. For example, the subgoals corresponding to ctor-init are as follows:

```
⟨ ABP ⊢ ceq pac(0)= empty if t := f
                        [metadata "goal1"]; ⟩
⟨ ABP ⊢ ceq pac(0)= pac(0) if nil
                        [metadata "goal2"]; ⟩
```

The tactic (**tc**) has no effect on the eighteen goals, because there are no variables in the formulas to prove. The tactic (**imp**) adds the conditions of the formulas to the specification. For example, the goals corresponding to **init** become:

```
⟨ ABP + {eq f = t} ⊢ eq pac(0)= empty
                        [metadata "goal1"]; ⟩
⟨ ABP ⊢ eq pac(0)= pac(0)
                        [metadata "goal2"]; ⟩
```

Since the second goal is a tautology, it is discharged automatically by the tool. We thus focus only on the first goal. For that purpose, we start with the specification of bits:

```
fmod BIT is
 sort Bit .
 op f : → Bit [ctor].
 op t : → Bit [ctor].
 op not_ : Bit → Bit .
 eq not f = t .
 eq not t = f .
 ceq true = false if t = f
                    [metadata "contradict"].
endfm
```

The specification BIT is declared with initial semantics and it is imported in the specification ABP in **protecting** mode. This means that there are exactly two (distinct) values for the bits, t and f. The addition of **eq** f = t to ABP gives rise to an inconsistency. From a model-theoretic point of view, the class of models of the specification ABP + {**eq** f = t} is empty. It follows that ABP + {**eq** f = t} entails any formula. In particular, ABP + {**eq** f = t} entails **eq** pac(0)= empty. But the only inconsistency recognized by CITP is on the sort Bool. In other words, if CITP can prove

```
⟨ ABP + {eq f = t} ⊢ eq true = false ; ⟩
```

then the goal

```
⟨ ABP + {eq f = t} ⊢ eq pac(0) = empty ; ⟩
```

is discharged. Here, the conditional equation contradict from the specification BIT comes into play. This conditional

equation states that any inconsistency created on the sort `Bit` will determine an inconsistency on the sort `Bool`. Notice that `contradict` has no role in the denotational semantics of the specification `BIT`. It is used by the tool to discharge goals with specifications that satisfy sentences in contradiction with the initial semantics. It follows that CITP discharges the goal

```
⟨ ABP + {eq f = t} ⊢ eq pac(0) = empty ; ⟩
```

because the specification `ABP + {eq f = t}` trivially entails the equation `eq f = t`, which is in contradiction with the initiality of the specification `BIT`. This tactic may look strange for engineers with no training in the theory of algebraic specification; but in practice it often happens to reach subgoals with inconsistent specifications (see Example 2); the approach described above is simple and efficient in such situations.

CRITICAL PAIRS. What we have presented above is only one part of the proof corresponding to the constructor `ctor-init`. By applying the tactic list

```
(indx on S:Reach red cs red tc red imp red)
```

to `goal1` and `goal2`, the simultaneous induction scheme generates nine goals – expanded to eighteen subgoals, as each of the nine goals contains two formulas. Fifteen goals are discharged by reduction or contradiction as we have described above. We are left with three goals to prove, as follows:

1) the constructor `ctor-rec1` and the formula `goal1`,
2) the constructor `ctor-rec2` and the formula `goal1`, and
3) the constructor `ctor-rec2` and the formula `goal2`.

The tool provides sufficient information for the users to understand the current state of the proof. By looking at the attributes of the formulas to prove and the induction hypotheses one can recognize each case mentioned above. All three of these subgoals are discharged by joining critical pairs and by initializing one of the five lemmas previously mentioned (and proved). The theorem-proving technique presented in this paper is interactive, which means that the user needs insight into ABP to discharge the remaining cases. We will present the proof of the goal corresponding to `ctor-rec1` and `goal1`:

```
⟨ mod ABP-L is
  ...
  mb [B1#1,N#2,Ch1#3,B2#4,D#5,(Ch2#6,B1#1)]:
     Reach      [metadata "1" metadata "rec1"].

  eq B2#4 = not B1#1 [metadata "4"].

  ceq mk(N#2)= D#5 if B1#1 := B2#4
           [metadata "2"
            metadata "goal1" metadata "rec1"].

  ceq mk(N#2) = pac(N#2),D#5
  if not B1#1 := B2#4
           [metadata "3"
            metadata "goal2" metadata "rec1"].
  ...
  endm
  ⊢ eq pac(s N#2),pac(N#2),D#5 = D#5
                          [metadata "goal1"]; ⟩
```

Notice that the following configuration is reachable

```
[B1#1,N#2,Ch1#3,B2#4,D#5,(Ch2#6,B1#1)]
```

where:

1) `B1#1` is a `Bit`-constant representing Sender's bit,
2) `N#2` is a natural number representing the index of the next packet to be delivered by Sender,
3) `Ch#3` is a `Channel1`-constant for the data channel,
4) `B2#4` is a `Bit`-constant representing Receiver's bit,
5) `D#5` is a `Data`-constant for packets delivered to Receiver,
6) `Ch2#6` is a constant of sort `Channel2` such that `(Ch2#6,B1#1)` represents the acknowledgement channel.

By the membership axiom `1`, the last bit of the acknowledgement channel is equal to Sender's bit. By the equation `4`, the Sender's bit `B1#1` is different from the Receiver's bit `B2#4`.

The third lemma in [10], labelled `inv3` below, states that, if the acknowledgement channel contains a bit that is equal to Sender's bit, then the Receiver's bit is also equal to Sender's bit, which is a contradiction with the equation `4`.

```
ceq B2 = B1
if [B1,N,Ch1,B2,D,(Ch2,B1,Cn2)]: Reach
                    [nonexec metadata "inv3"].
```

The proof script for the goal corresponding to `ctor-rec1` and `goal1` is as follows:

```
(. cp-l 1 >< 4)
(. init inv3 by B1:Bit ← B1#1 ;
 N:PNat ← N#2 ; B2:Bit ← B2#4 ;
 D:Data ← D#5 ;
 Ch1:Channel1 ← Ch1#3 ;
 Ch2:Channel2 ← Ch2#6 ;
 Cn2:Channel2 ← (empty).Channel2 ; )
```

Remember that the dot prefix means that the tactics are applicable only to the current goal. The tactic `(. cp-l 1 >< 4)` thus joins the critical pairs generated by the axioms:

```
mb [B1#1,N#2,Ch1#3,B2#4,D#5,(Ch2#6,B1#1)]:
   Reach                          [metadata "1"].
eq B2#4 = not B1#1                [metadata "4"].
```

More concretely, `(. cp-l 1 >< 4)` tries to unify a subterm of the membership axiom `1` with the left-hand side of equation `4`. If the unification is successful, then it adds to the specification of the current goal the membership axiom obtained from `1` by substituting the right-hand side of the equation `4` for the subterm of the membership axiom `1` used for unification. Otherwise, `cp-l` leaves the goal unchanged. Therefore, the following membership axiom is added to the specification:

```
mb [B1#1,N#2,Ch1#3,not B1#1,D#5,(Ch2#6,B1#1)]:
   Reach
```

The tactic `init` instantiates `inv3` by the given substitution and reduces all terms of the newly obtained sentence to their normal forms. The result is a goal obtained from the current goal by adding the following equation to the specification:

```
eq not B1#1 = B1#1 .
```

Finally, the goal obtained in this way is discharged automatically by CITP because its underlying specification is inconsistent; see [10] for the full proof.

IV. Related Work

The Alternating Bit Protocol is a well-established benchmark in the area of formal methods, being used for more than four decades to test the strength and capabilities of various verification technologies. The safety property of interest in this case is that of reliable communication: all messages from Sender are successfully delivered to Receiver, in the correct order, even though the communication channels can lose or duplicate messages. The result advanced in this paper is a fully mechanized proof with a high degree of automation, meaning that most of the routine proofs are performed automatically.

Several variants of the ABP have been studied in the formal-verification literature, among which [2], [3] are some of the earliest verification efforts (based on process algebra and branching-time temporal logic). In many cases, in order to cope with the infinite state-space of the ABP, the solution relies on imposing bounds or restrictions on communication channels, reducing in this way the generality of the protocol.

A number of related studies use a combination of theorem-proving and model-checking techniques in order to reason about safety properties of the ABP. For example, in [20] the authors use the Isabelle theorem prover to show that a cleverly designed finite system is a suitable abstraction of the protocol, thus enabling the verification of safety properties of the ABP by means of model checking. In that case, theorem proving is used for representing both the finite and the infinite-state systems, and for verifying the soundness of the abstraction, meaning that the finite-state system has all the traces of the ABP. Similarly, [16] shows how ACL2 can be used to prove that a variant of the protocol is stuttering bisimilar to a finite-state system, which can then be model-checked as well, while in [15], [25] the authors use the PVS theorem prover for analysing the ABP by integrating static analysis, theorem proving, and other comparable abstraction techniques.

Another prominent approach to verifying the ABP is based on the use of co-algebraic languages for describing processes. For instance, in [6] this has allowed the development of formal correctness proofs for the protocol by representing Prasad's Calculus of Broadcasting Systems in the Coq theorem prover. The proof presented in [7] is also co-algebraic, and it is based on the circular co-inductive rewriting algorithm from BOBJ.

What distinguishes the tool and technique that we have proposed in this paper is that:

1) It makes use of a single, general algebraic framework for reasoning about the ABP, without the need to encode within the framework a specialized language for describing the protocol or to develop a finite-state abstraction of it; in particular, in contrast to [16], [20], the verification is performed entirely by theorem proving.

2) There are no compromises in the specification of the ABP; for instance, at any time, both the data and the acknowledgement channel can lose or duplicate information, not only from one of their end-points, but from arbitrary positions; since the technique implicitly supports associative data types, such properties can be specified in a straightforward way.

3) It combines general tactics, which are universal in the sense that they correspond to the underlying logic of CITP (and can be used for all specifications), with specialized tactics, which are tailored to the data types declared with initial semantics (and can be used only for specifications that protect such data types).

The closest related work on proving safety properties of the ABP is that on the framework of observational transition systems [13], [21], where the focus is on proving the reliable communication between Sender and Receiver based on simultaneous induction. In [22], [23], the same property is proved using a technique based on narrowing, which supports partially the unification of associative data types. Because of this, the specification of the ABP from [22] models the dropping of elements only from the first position of the communication channels. The same compromise w.r.t. the specification of the ABP was made in [21] as well.

It is worth mentioning that the specification `ABP` that we have described in this paper (available at [10]) is much more compact than the specification of the ABP that relies on the OTS method [11]. On the other hand, the OTS method generates no critical pairs for the case studies that we have analysed so far. This means that the OTS method has a higher level of automation. Figure 4 presents a comparison of the formal proof described here and the ones from [21], [22], and [13].

The human proof effort is significantly higher in [21] and [22] than in [13] and the present work. In the present case study, there is no need to create a new module to define extra predicates necessary for proving the desired properties. Therefore, in the second row of the table in Figure 4, the number of lines of code for these extra predicates in the present work is 0. The number of state predicates in [22] is less than the number of invariants in the present work, but the definition of predicates in [22] takes almost the same number of lines of code as the specification. The specifications in [13] and the present contribution are closer to a real description than the specifications in [21] and [22] since they model the dropping and duplication of elements from arbitrary positions of the channels. There is no significant difference in length between the proof presented in [13] and the proof described here; but the specification in [13] is significantly longer than the one given in this paper, and the difference would be more visible for larger examples. In addition, the present specification can be enhanced with transitions, which makes it possible to apply model-checking techniques as well for verification.

V. Conclusions and Ongoing Work

In this paper, we have presented a proof technique for transition systems that is supported by CITP. The tool is implemented in Maude and presents important advantages over previous versions of CITP and other state-of-the-art theorem provers, including a new induction scheme based on membership axioms, a tactic for joining critical pairs, and improved decision procedures. We have exemplified these

CRITERIA	MEASURE	[21]	[22]	[13]	This work
Model	LOC	286	208	195	114
Model + Extra predicates	LOC	286 + 63	208 + 200	195 + 0	114 + 0
State predicates/invariants	#	11	3	7	7
Lemmata	#	7	10	1	0
Proof scripts	LOC	5189	213	80	93
Proof scripts / # predicates	LOC	471	71	11	13

Fig. 4: Comparison with similar case studies

features in a case study on the verification of the ABP; the results obtained encourage us to continue this line of research.

As future work, we are interested in developing new commands for performing bounded rewriting, hence preventing infinite computations, and also for applying specific equations or rules to a goal. We are also conducting more case studies to assess the strong points of the current implementation and provide a solid example database for future users.

From the point of view of the interface, we aim to provide a graphical user interface that includes options for saving the current proof and editing commands. Moreover, a web interface would allow users to use the tool without installing it, while providing an easy way to access the proofs developed so far.

REFERENCES

[1] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12(5):260–261, May 1969.

[2] J. A. Bergstra and J. W. Klop. Verification of an alternating bit protocol by means of process algebra. In W. Bibel and K. P. Jantke, editors, *Mathematical Methods of Specification and Synthesis of Software Systems '85, Proceedings of the International Spring School, Wendisch-Rietz, GDR, April 22–26, 1985*, volume 215 of *LNCS*, pages 9–23. Springer, 1985.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude – A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.

[5] R. Diaconescu and K. Futatsugi. *CafeOBJ Report – The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[6] E. Giménez. An application of co-inductive types in Coq: Verification of the alternating bit protocol. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5–8, 1995, Selected Papers*, volume 1158 of *LNCS*, pages 135–152. Springer, 1995.

[7] J. A. Goguen and K. Lin. Behavioral verification of distributed concurrent systems with BOBJ. In *3rd International Conference on Quality Software (QSIC 2003), 6–7 November 2003, Dallas, TX, USA*, page 216. IEEE Computer Society, 2003.

[8] J. A. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *SIGPLAN Notices*, 17(1):9–17, 1982.

[9] D. Găină. Constructor-based Inductive Theorem Prover. http://imi.kyushu-u.ac.jp/~daniel/citp.html, 2018. [Online; accessed July 2018].

[10] D. Găină. The Alternating Bit Protocol – Explicitly described states. http://imi.kyushu-u.ac.jp/~daniel/examples/abpr/abp.html, 2018. [Online; accessed July 2018].

[11] D. Găină. The Alternating Bit Protocol – The OTS/CafeOBJ method. http://imi.kyushu-u.ac.jp/~daniel/examples/abp/abp.html, 2018. [Online; accessed July 2018].

[12] D. Găină, K. Futatsugi, and K. Ogata. Constructor-based Logics. *J. UCS*, 18(16):2204–2233, 2012.

[13] D. Găină, D. Lucanu, K. Ogata, and K. Futatsugi. On Automation of OTS/CafeOBJ Method. In S. Iida, J. Meseguer, and K. Ogata, editors, *Specification, Algebra, and Software*, volume 8373 of *LNCS*, pages 578–602. Springer, 2014.

[14] D. Găină, M. Zhang, Y. Chiba, and Y. Arimoto. Constructor-based Inductive Theorem Prover. In R. Heckel and S. Milius, editors, *Algebra and Coalgebra in Computer Science – 5th International Conference, CALCO 2013*, volume 8089 of *LNCS*, pages 328–333. Springer, 2013.

[15] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In M. Gaudel and J. Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, Co-Sponsored by IFIP WG 14.3, Oxford, UK, March 18–22, 1996, Proceedings*, volume 1051 of *LNCS*, pages 662–681. Springer, 1996.

[16] P. Manolios, K. S. Namjoshi, and R. Summers. Linking theorem proving and model-checking with well-founded bisimulation. In N. Halbwachs and D. A. Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6–10, 1999, Proceedings*, volume 1633 of *LNCS*, pages 369–379. Springer, 1999.

[17] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Second Edition, Volume 9*, pages 1–87. Kluwer Academic Publishers, 2002. First published as SRI Technical Report SRI-CSL-93-05, August 1993.

[18] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97*, volume 1376 of *LNCS*, pages 18–61. Springer, 1997.

[19] J. Meseguer and J. A. Goguen. Initiality, induction, and computability. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, New York, NY, USA, 1986.

[20] O. Müller and T. Nipkow. Combining model checking and deduction for i/o-automata. In E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS '95, Aarhus, Denmark, May 19–20, 1995, Proceedings*, volume 1019 of *LNCS*, pages 1–16. Springer, 1995.

[21] K. Ogata and K. Futatsugi. Simulation-based Verification for Invariant Properties in the OTS/CafeOBJ Method. *Electr. Notes Theor. Comput. Sci.*, 201:127–154, 2008.

[22] C. Rocha. *Symbolic Reachability Analysis for Rewrite Theories*. PhD thesis, University of Illinois at Urbana-Champaign, 2012.

[23] C. Rocha and J. Meseguer. Proving safety properties of rewrite theories. In A. Corradini, B. Klin, and C. Cîrstea, editors, *Algebra and Coalgebra in Computer Science – 4th International Conference, CALCO 2011, Winchester, UK, August 30 – September 2, 2011. Proceedings*, volume 6859 of *LNCS*, pages 314–328. Springer, 2011.

[24] D. Rosén and N. Smallbone. TIP: Tools for Inductive Provers. In M. Davis, A. Fehnker, A. McIver, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning – 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24–28, 2015, Proceedings*, volume 9450 of *LNCS*, pages 219–232. Springer, 2015.

[25] V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In R. Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22–28, 1999, Proceedings*, volume 1579 of *LNCS*, pages 178–192. Springer, 1999.