Mechanical Analysis of Reliable Communication in the Alternating Bit Protocol Using the Maude Invariant Analyzer Tool

Camilo Rocha[1] and José Meseguer[2]

[1] Escuela Colombiana de Ingeniería, Bogotá, Colombia
[2] University of Illinois at Urbana-Champaign, Urbana IL, USA

Abstract. The InvA tool supports the deductive verification of safety properties of infinite-state concurrent systems. Given a concurrent system specified as a rewrite theory and a safety formula to be verified, InvA reduces such a formula to inductive properties of the underlying equational theory by means of the application of a few inference rules. Through the combination of various techniques such as unification, narrowing, equationally-defined equality predicates, and SMT solving, InvA achieves a significant degree of automation, verifying automatically many proof obligations. Maude Inductive Theorem Prover (ITP) can be used to discharge the remaining obligations which are not automatically verified by InvA. Verification of the reliable communication ensured by the Alternating Bit Protocol (ABP) is used as a case study to explain the use of the InvA tool, and to illustrate its effectiveness and degree of automation in a concrete way.

1 Introduction

The late Amir Pnueli entitled his invited talk at FM'99 "Deduction is Forever" [22]. Pnueli, who had pioneered the use of temporal logic in computer science as well as many model checking techniques, wanted to remind us that algorithmic verification methods are not enough by themselves and should always be complemented by deductive verification methods. What is actually happening is that model checking and theorem proving methods are increasingly used in tandem, and the borderline between both is becoming more tenuous. Verification of temporal logic properties, particularly for infinite-state systems, is an area where both algorithmic and deductive methods can be used, sometimes together.

In the rewriting logic research program [16], model checking methods and tools have been extensively developed. However, deductive techniques, while well-supported for equational specifications with an initial algebra semantics, do not directly apply to temporal logic formulas. One important exception is the deductive verification approach with *proof scores* and the *OTS/CafeOBJ method* [9,19], pioneered by Kokichi Futatsugi and his collaborators, for the verification of *invariants* of concurrent systems. By using such an approach quite

sophisticated systems have been verified [20,21]. Work on the OTS/CafeOBJ method has stimulated our own work on deductive verification of concurrent systems. Indeed, the work presented here is a concrete example of the way in which we have responded to such an encouraging stimulus. Our main purpose has been to advance the following goals:

1. **Deductive Support for Temporal Logic**. Beyond invariants, deductive reasoning about other temporal logic properties should be supported. For the moment we have advanced this to support a useful subset of *safety properties*, but we hope to extend the methods to also support liveness properties.
2. **Reduction of Temporal Verification to Equational Verification**. As much as possible, temporal logic properties should be construed as "syntactic sugar" for inductive properties of an algebraic specification corresponding to the system's states and the predicates satisfied by such states. In this way, all the wealth of techniques and tools already available to verify properties of algebraic specifications with an initial algebra semantics can be leveraged.
3. **Increased automation**. To reduce the verification effort, the level of automation should be increased as much as possible, both at the level of reasoning about temporal properties, and after reducing such properties to inductive proof obligations in equational logic.

Our way to advance goals (1)–(3) has been to develop new deductive verification techniques, embody them in the InvA tool [23,24] as part of the Maude formal environment, and test the practical advancement of goals (1)–(3) through case studies. We can summarize our present advances as follows. Goals (1) and (2) have been advanced by (i) identifying a class of commonly used safety properties; and (ii) developing and proving correct a set of inference rules that reduce the verification of such safety properties to inductive equational reasoning. A key technique for this reduction has been the use of unification and narrowing to prove stability properties in an inductive way. The development of Goal (3) has been advanced by a combination of automation techniques including: (i) automation of narrowing and unification in the underlying Maude system; (ii) automation of certain conditional inferences; (iii) systematic use of equationally-defined equality predicates [12]; (iv) use of SMT solvers; and (v) use of proof tactics in the Maude ITP.

Although still work in progress and amenable to many subsequent improvements and extensions, it seems fair to say that the advances in goals (1)–(3) supported by the InvA tool have been significant. A good way to give a feeling for such advances is to explain how they have helped in automating a remarkable amount of proof tasks when verifying a well-known benchmark verified by other systems, and in particular by the OTS/CafeOBJ methodology and tool, namely, the reliable communication ensured by the Alternating Bit Protocol. This also makes it possible to compare our proof efforts with those in OTS/CafeOBJ and measure advances in Goal (3) in a concrete and meaningful way.

In summary, therefore, this work should be seen in the context of a very long and broader exchange of ideas with Kokichi Futatsugi and his collaborators in the CafeOBJ group, which has stimulated advances in both Maude and

CafeOBJ. In particular, it has been a pleasure to discuss our ideas about InvA with Kokichi Futatsugi and Kazuhiro Ogata, and to benefit from their experience in the deductive verification of invariants. This work is cordially dedicated to Kokichi Futasugi in the spirit of such a long term and very fruitful exchange of ideas.

2 Preliminaries

This paper follows notation and terminology from [15] for order-sorted equational logic and from [5] for rewriting logic.

An *order sorted signature* Σ is a tuple $\Sigma = (S, \leq, F)$ with finite poset of sorts $(S, \leq)$ and a finite S-index set of function symbols $F = \{F_{w,s}\}_{(w,s)\in S^* \times S}$. It is assumed that: (i) each connected component of a sort $s \in S$ in the poset ordering has a top sort, denoted by k_s, and (ii) for each operator declaration $f \in F_{s_1 \ldots s_n, s}$ there is also a declaration $f \in F_{k_{s_1} \ldots k_{s_n}, k_s}$. The collection $X = \{X_s\}_{s\in S}$ is an S-sorted family of disjoint sets of variables with each X_s countably infinite. The set of terms of sort s is denoted by $T_\Sigma(X)_s$ and the set of ground terms of sort s is denoted by $T_{\Sigma,s}$, which are assumed nonempty for each s. The expressions $\mathcal{T}_\Sigma(X)$ and $\mathcal{T}_\Sigma$ denote the respective term algebras. The set of variables of a term t is written $vars(t)$ and is extended to sets of terms in the natural way. A *substitution* θ is a sorted map from a finite subset $dom(\theta) \subseteq X$ to $T_\Sigma(X)$ and extends homomorphically in the natural way; $ran(\theta)$ denotes the set of variables introduced by θ and $t\theta$ the application of θ to a term t. Substitution $\theta_1\theta_2$ is the composition of substitutions θ_1 and θ_2. A substitution θ is called *ground* iff $ran(\theta) = \varnothing$.

A Σ-*equation* is a Horn clause $t = u$ **if** γ, where $t = u$ is a Σ-*equality* with $t, u \in T_\Sigma(X)_s$ for some sort $s \in S$, and the *condition* γ is a finite conjunction of Σ-equalities $\bigwedge_{i\in I} t_i = u_i$. An *equational theory* is a tuple (Σ, E) with order-sorted signature Σ and finite set of Σ-equations E. For φ a Σ-equation, $(\Sigma, E) \vdash \varphi$ iff φ can be proved from (Σ, E) by the deduction rules in [15] iff φ is valid in all models of (Σ, E); assuming $T_{\Sigma,s} \neq \varnothing$ for each $s \in S$, (Σ, E) induces the congruence relation $=_E$ on $T_\Sigma(X)$ defined for any $t, u \in T_\Sigma(X)$ by $t =_E u$ iff $(\Sigma, E) \vdash t = u$. The expressions $\mathcal{T}_{\Sigma/E}(X)$ and $\mathcal{T}_{\Sigma/E}$ denote the quotient algebras induced by $=_E$ over the algebras $\mathcal{T}_\Sigma(X)$ and $\mathcal{T}_\Sigma$, respectively; $\mathcal{T}_{\Sigma/E}$ is the *initial algebra* of (Σ, E). An E-*unifier* for a Σ-equality $t = u$ is a substitution θ such that $t\theta =_E u\theta$. A *complete* set of E-unifiers for a Σ-equality $t = u$ is written $CSU_E(t = u)$ and it is called *finitary* if it contains a finite number of E-unifiers. The expression $GU_E(t = u)$ denotes the set of ground E-unifiers of a Σ-equality $t = u$. A theory inclusion $(\Sigma, E) \subseteq (\Sigma', E')$ is *protecting* iff the unique Σ-homomorphism $\mathcal{T}_{\Sigma/E} \longrightarrow \mathcal{T}_{\Sigma'/E'|\Sigma}$ to the Σ-reduct of the initial algebra $\mathcal{T}_{\Sigma'/E'}$ is an isomorphism.

A Σ-*rule* is a sentence $t \to u$ **if** γ, where $t \to u$ is a Σ-*sequent* with $t, u \in T_\Sigma(X)_s$ for some sort $s \in S$ and the *condition* γ is a finite conjunction of Σ-equalities. A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E, R)$ with equational theory $\mathcal{E}_\mathcal{R} = (\Sigma, E)$ and a finite set of Σ-rules R. A *topmost rewrite theory* is a rewrite

theory $\mathcal{R} = (\Sigma, E, R)$ such that for some top sort $\mathfrak{s}$ and for each $t \to u$ **if** $\gamma \in R$, the terms t, u satisfy $t, u \in T_\Sigma(X)_\mathfrak{s}$ and $t \notin X$, and no operator in Σ has $\mathfrak{s}$ as argument sort. For $\mathcal{R} = (\Sigma, E, R)$ and φ a Σ-rule, $\mathcal{R} \vdash \varphi$ iff φ can be obtained from $\mathcal{R}$ by the deduction rules in [5] iff φ is valid in all models of $\mathcal{R}$. For φ a Σ-equation, $\mathcal{R} \vdash \varphi$ iff $\mathcal{E}_\mathcal{R} \vdash \varphi$. A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ induces the rewrite relation $\to_\mathcal{R}$ on $T_{\Sigma/E}(X)$ defined for every $t, u \in T_\Sigma(X)$ by $[t]_E \to_\mathcal{R} [u]_E$ iff there is a *one-step* rewrite proof $\mathcal{R} \vdash t \to u$. The expressions $\mathcal{R} \vdash t \to u$ and $\mathcal{R} \vdash t \stackrel{*}{\to} u$ respectively denote a one-step rewrite proof and an arbitrary length (but finite) rewrite proof in $\mathcal{R}$ from t to u. The expression $\mathcal{T}_\mathcal{R} = (\mathcal{T}_{\Sigma/E}, \stackrel{*}{\to}_\mathcal{R})$ denotes the *initial reachability model* of $\mathcal{R} = (\Sigma, E, R)$ [5]. A Σ-sequent φ is an *inductive consequence* of $\mathcal{R}$ denoted $\mathcal{R} \Vdash \varphi$ iff $(\forall \theta : X \longrightarrow T_\Sigma)\, \mathcal{R} \vdash \varphi\theta$ iff $\mathcal{T}_\mathcal{R} \models \varphi$.

State predicates. A set of *state predicates* Π for $\mathcal{R} = (\Sigma, E, R)$ can be equationally-defined by an equational theory $\mathcal{E}_\Pi = (\Sigma_\Pi, E \uplus E_\Pi)$. Signature Σ_Π contains Σ, two sorts $Bool \leq [Bool]$ with constants $\top$ and $\bot$ of sort $Bool$, predicate symbols $p : \mathfrak{s} \longrightarrow [Bool]$ for each $p \in \Pi$, and optionally some auxiliary function symbols. Equations in E_Π define the predicate symbols in Σ_Π and auxiliary function symbols, if any; they protect (Σ, E) and the equational theory specifying sort $Bool$, constants $\top$ and $\bot$, and the Boolean operations. It is easy to define a state predicate $p \in \Pi$ as a Boolean combination of other already-defined state predicates $\{p_1, \ldots, p_n\}$ in Σ_Π. The reason why p has typing $p : \mathfrak{s} \longrightarrow [Bool]$ instead of $p : \mathfrak{s} \longrightarrow Bool$, is to allow partial definitions of p with equations that fully define the *positive* case by equations $p(t) = \top$ **if** γ, and either leave the *negative* case implicit or may only define some negative cases with equations $p(t') = \bot$ **if** γ' without necessarily covering all the cases.

LTL semantics. For $p \in \Pi$ and $[t]_E \in T_{\Sigma/E,\mathfrak{s}}$, $\mathcal{E}_\Pi$ defines the *semantics of* p in $\mathcal{T}_\mathcal{R}$ as follows: it is said that $p([t]_E)$ *holds* in $\mathcal{T}_\mathcal{R}$ iff $\mathcal{E}_\Pi \vdash p(t) = \top$. This defines a Kripke structure $\mathcal{K}^\Pi_\mathcal{R} = (T_{\Sigma/E,\mathfrak{s}}, \to_\mathcal{R}, L_\Pi)$ with labeling function L_Π such that, for each $[t]_E \in T_{\Sigma/E,\mathfrak{s}}$, the semantic equivalence $p \in L_\Pi([t]_E)$ iff $p([t]_E)$ holds in $\mathcal{T}_\mathcal{R}$. Then, all of LTL can be interpreted in $\mathcal{K}^\Pi_\mathcal{R}$ in the standard way [6], including the "always" ($\Box$), "next" ($\bigcirc$), and "strong implication" ($\Rightarrow$) operators.

Executability conditions. It is assumed that the set of equations of a rewrite theory $\mathcal{R}$ can be decomposed into a disjoint union $E \uplus B$, with B a collection of axioms (such as associativity, and/or commutativity, and/or identity) for which there exists a *matching algorithm modulo* B producing a finite number of B-matching substitutions, or failing otherwise. It is also assumed that the equations E can be oriented into a set of *ground sort-decreasing*, *ground confluent*, and *ground terminating* rules $\overrightarrow{E}$ modulo B. The expression $t \downarrow_{\Sigma,E/B} \in T_{\Sigma,s}(X)$ denotes the E/B*-canonical form* of $t \in T_\Sigma(X)$, which is guaranteed to exist under the executability conditions above mentioned. The rules R in $\mathcal{R}$ are assumed to be *ground coherent* relative to the equations E modulo B [28].

Free constructors. For $\mathcal{R} = (\Sigma, E \uplus B, R)$, the signature $\Omega \subseteq \Sigma$ is a signature of *free constructors* modulo B iff for each sort s in Σ and $t \in T_{\Sigma,s}$ there is $u \in T_{\Omega,s}$ satisfying $t =_{E \uplus B} u$, and $v \downarrow_{\Sigma,E/B} =_B v$ for any $v \in T_{\Omega,s}$. For the

development in this paper it is required that $t \in T_\Omega(X)$ for each $t \to u$ **if** $\gamma \in R$ (see [23,24] for more details).

3 The Maude Invariant Analyzer Tool: An Overview

The *Maude Invariant Analyzer Tool* (InvA) is a tool designed for *interactively* proving two key safety properties of executable Maude specifications, namely, inductive stability and inductive invariance, plus their combination by strengthening techniques. The tool mechanizes an inference system that, without assuming finiteness of the set of initial or reachable states, uses rewriting and narrowing-based reasoning techniques, in which all temporal logic formulas eventually disappear and are replaced by purely equational conditional sentences. The InvA tool provides a substantial degree of mechanization and can automatically discharge many proof obligations without user intervention. It is implemented in the Maude language and exploits rewriting logic's reflection capabilities, i.e., it is a Maude specification that takes, as part of its input, a meta-representation of a Maude specification.

The concept of inductive stability for $\mathcal{R} = (\Sigma, E, R)$ is intimately related to the notion of the set of states $t \in T_{\Sigma,\mathfrak{s}}$ of $\mathcal{T}_\mathcal{R}$ that satisfy a state predicate $p \in \Pi$ and is closed under $\to_\mathcal{R}$. More precisely, for $p \in \Pi$ and $x \in X_\mathfrak{s}$, the property p being *inductively stable* for $\mathcal{R}$ is the safety property:

$$\mathcal{K}^\Pi_\mathcal{R} \models p(x) \Rightarrow \Box p(x)$$

meaning that if $p(t)$ holds in a state $t \in T_{\Sigma,\mathfrak{s}}$, then $p(u)$ holds in any state $u \in T_{\Sigma,\mathfrak{s}}$ that is reachable from t.

Invariants are among the most important safety properties. Given a set of initial states characterized by $I \in \Pi$, a state predicate $p \in \Pi$ being *inductively invariant* for $\mathcal{R}$ from the set of initial states I is the safety property

$$\mathcal{K}^\Pi_\mathcal{R} \models I(x) \Rightarrow \Box p(x)$$

meaning that if $I(t)$ holds in a state $t \in T_{\Sigma,\mathfrak{s}}$, then $p(u)$ holds in any state $u \in T_{\Sigma,\mathfrak{s}}$ reachable from t. In other words, the invariant p holds for all states reachable from I. Since the set of initial states is defined in $\mathcal{E}_\Pi$ as a state predicate $I \in \Pi$, an equational definition of I can of course capture an infinite set of initial states.

3.1 Inference System Mechanized in the InvA Tool

Given an inductive stability or inductive invariance property φ, the InvA tool generates equational proof obligations such that, if they hold, then $\mathcal{T}_\mathcal{R} \models \varphi$. For a topmost rewrite theory $\mathcal{R}$ and a set of state predicates Π specified in Maude, the InvA tool mechanizes inference rules ST, INV, STR1, STR2, C⇒, NR1, and NR2 depicted in Figure 1. Soundness proofs for each one of these inference rules can be found in [23]. The application of inference rules ST, INV, STR1, and STR2 to

a given inductive stability or invariance LTL verification goal ultimately reduces such a goal to simpler inductive equational reasoning that can be handled by applying rules C⇒, NR1, and NR2.

Inference rule ST reduces the verification task of the inductive stability of a predicate p to the simpler condition $p \Rightarrow \bigcirc p$, which only involves 1-step search instead of arbitrary depth search. Inference rule INV reduces the verification task of inductive invariance to equational implication and inductive stability. Inference rules STR1 and STR2 are strengthening rules. Inference rule C⇒ handles equational implications, while rules NR1 and NR2 use 1-step narrowing modulo axioms to handle the symbolic 1-step search, for the temporal next operator, in formulae of the form $p \Rightarrow \bigcirc p$. Note that any inductive stability and invariance formula is ultimately reduced to equational reasoning. Thanks to the availability since Maude 2.6 of unification modulo commutativity (C), associativity and commutativity (AC), and modulo these theories plus identities (U), and to the narrowing modulo infrastructure, the InvA tool can handle modules with operators declared C, CU, AC, and ACU. Furthermore, since unification modulo the above theory combinations is decidable, and each one yields a *finite* set of complete unifiers, the set of proof obligations resulting from applying rules NR1 and NR2 is always finite.

Under the executability assumptions, $\mathcal{R}$ has a disjoint union $E \uplus B$ of equations, with B a collection of structural axioms on some function symbols in Σ such as associativity, commutativity, identity, etc., and E a set of ground sort-decreasing, ground confluent, ground terminating, and ground coherent (w.r.t. R) equations modulo B. Then, it is key to note that for rules NR1 and NR2 and for a combination of free and associative and/or commutative and/or identity axioms, except for symbols f that are associative but not commutative, a finitary B-unification algorithm exists. Instead, in general there is no finitary $E \uplus B$-unification algorithm, but for $\Omega \subseteq \Sigma$ a signature of free equational constructors modulo B and a Ω-equality $t = u$, $CSU_B(t = u)$ exactly characterizes as its ground instances the set $GU_B(t = u)$ (see [23, Lemma 2, Chapter 4] for more details).

3.2 Methodology and Commands Available to the User

The approach for proving inductive stability and invariance properties in the InvA tool is depicted in Figure 2.

Given a topmost rewrite theory $\mathcal{R}$, an equational specification $\mathcal{E}_{\Pi}$ for the state predicates Π, and an inductive safety property φ the InvA tool internally generates equational proof obligations according to the inference system in Figure 1 and tries to discharge as many of them as possible by using the heuristics described in Section 3.3. Any proof obligation that cannot be automatically discharged is output to the user so it can be handled interactively in an external tool such as Maude's Inductive Theorem Prover (ITP) [7,13] (an experimental interactive tool for proving properties of the initial algebra $\mathcal{T}_{\mathcal{E}}$ of an order-sorted equational theory $\mathcal{E}$ written in Maude).

$$\frac{\mathcal{R} \Vdash p(x) \Rightarrow \bigcirc p(x)}{\mathcal{R} \Vdash p(x) \Rightarrow \Box p(x)}\ \textsc{St}$$

$$\frac{\mathcal{R} \Vdash I(x) \Rightarrow p(x) \qquad \mathcal{R} \Vdash p(x) \Rightarrow \Box p(x)}{\mathcal{R} \Vdash I(x) \Rightarrow \Box p(x)}\ \textsc{Inv}$$

$$\frac{\mathcal{R} \Vdash I(x) \Rightarrow J(x) \qquad \mathcal{R} \Vdash J(x) \Rightarrow \Box q(x) \qquad \mathcal{R} \Vdash q(x) \Rightarrow p(x)}{\mathcal{R} \Vdash I \Rightarrow \Box p}\ \textsc{Str1}$$

$$\frac{\mathcal{R} \Vdash I(x) \Rightarrow p(x) \qquad \mathcal{R} \Vdash I(x) \Rightarrow \Box q(x) \qquad \mathcal{R} \Vdash q(x) \wedge p(x) \Rightarrow \bigcirc p(x)}{\mathcal{R} \Vdash I(x) \Rightarrow \Box p(x)}\ \textsc{Str2}$$

$$\frac{\bigwedge_{(q(v)=w \textbf{ if } \gamma') \in E_\Pi} \mathcal{E}_\Pi \Vdash p(v) = \top \textbf{ if } \gamma' \wedge w = \top}{\mathcal{R} \Vdash q(x) \Rightarrow p(x)}\ \textsc{C}\Rightarrow$$

$$\frac{\bigwedge_{\substack{(l \to r \textbf{ if } \gamma) \in R \\ (\theta, w, \gamma') \in \Theta_l^p}} \mathcal{E}_\Pi \Vdash p(r\theta) = \top \textbf{ if } \gamma\theta \wedge \gamma'\theta \wedge w\theta = \top}{\mathcal{R} \Vdash p(x) \Rightarrow \bigcirc p(x)}\ \textsc{Nr1}$$

$$\frac{\bigwedge_{\substack{(l \to r \textbf{ if } \gamma) \in R \\ (\theta, w, \gamma') \in \Theta_l^p}} \mathcal{E}_\Pi \Vdash p(r\theta) = \top \textbf{ if } \gamma\theta \wedge \gamma'\theta \wedge w\theta = \top \wedge q(l)\theta = \top}{\mathcal{R} \Vdash q(x) \wedge p(x) \Rightarrow \bigcirc p(x)}\ \textsc{Nr2}$$

where $\Theta_l^p = \bigcup_{(p(v)=w \textbf{ if } \gamma') \in E_\Pi} \{(\theta, w, \gamma') \mid \theta \in CSU_B(l = v)\}$.

Fig. 1: Inference rules mechanized in the InvA tool.

The user interacts with the InvA tool via commands; the commands available are the following:

- `(help .)` shows the list of commands available.
- `(analyze-stable <pred> in <module> <module> .)` generates the proof obligations for inference ST with inference NR1, for the given predicate. The first module equationally specifies the state predicate and the second one the topmost rewrite theory. This command tries to eagerly discharge the proof obligations; those that cannot be discharged are shown to the user.
- `(analyze-stable <pred> in <module> <module> assuming <pred> .)` generates the proof obligations for proving the third premise of inference STR2 with inference NR2, for the given predicate and the given modules. The first module equationally specifies the state predicates and the second one the topmost rewrite theory. This command tries to eagerly discharge the proof obligations; those that cannot be discharged are shown to the user.
- `(analyze <pred> implies <pred> in <module> .)` generates the proof obligations for proving the given implication in the given module, according to inference C⇒. This command tries to eagerly discharge the proof obligations; those that cannot be discharged are shown to the user.
- `(show pos .)` shows the proof obligations computed by the last `analyze` command that could not be discharged; those that were discharged are not shown.
- `(show-all pos .)` shows all the proof obligations computed by the last `analyze` command.

Observe that the analysis commands in InvA give direct tool support for deductive reasoning with *some* of the inference rules presented here, but not for all of them. For example, there is no command in InvA directly supporting deduction with inference rule INV. Nevertheless, deduction with *all* inference rules is supported by InvA via *combination of commands*. For example, deduction with inference rule INV can be achieved by combining the `analyze` and `analyze-stable` commands.

3.3 Proof-Search Heuristics in InvA

After applying rules ST, INV, STR1, STR2, C⇒, NR1, and NR2 according to the user commands, the InvA tool uses rewriting-based reasoning and narrowing procedures, and SMT decision procedures for automatically discharging as many of the generated equational proof obligations as possible. For an executable equational specification $\mathcal{E}_\Pi = (\Sigma_\Pi, E_\Pi \uplus B)$ and a conditional proof obligation φ of the form

$$t = u \textbf{ if } \gamma,$$

the InvA tool applies a proof-search strategy such that, if it succeeds, then the Kripke structure $\mathcal{K}^\Pi_\mathcal{R}$ associated to the initial reachability model $\mathcal{T}_\mathcal{R}$ satisfies

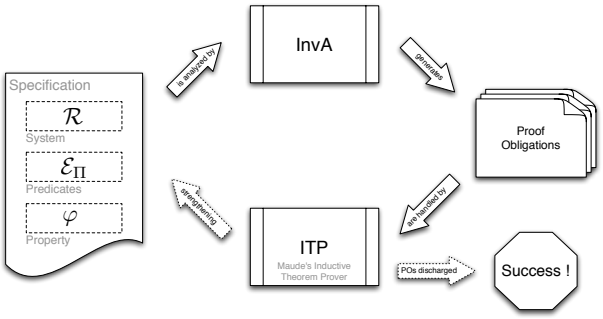

Fig. 2: Approach for checking inductive stability and invariance properties for rewrite theories.

φ. Otherwise, if the proof-search fails, the proof obligation φ (or a logically equivalent variant) is output to the user.

For the proof-search process, the InvA tool first tries to simplify Boolean expressions in φ. During the simplification process, the tool assumes that any operator '$\sim$' is an equationally defined equality predicate, i.e., an *equality enrichment*. Given an order-sorted signature $\Sigma = (S, \leq, F)$ and an order-sorted equational theory $\mathcal{E} = (\Sigma, E)$ with initial algebra $\mathcal{T}_\mathcal{E}$, an equality enrichment [17] of $\mathcal{E}$ is an equational theory $\mathcal{E}^\sim$ that extends $\mathcal{E}$ by defining a Boolean-valued equality function symbol '$\sim$' that coincides with '$=$' in $\mathcal{T}_\mathcal{E}$.

Definition 1. *An equational theory $\mathcal{E}^\sim = (\Sigma^\sim, E^\sim)$ is called an* equality enrichment *of $\mathcal{E} = (\Sigma, E)$, with $\Sigma^\sim = (S^\sim, \leq^\sim, F^\sim)$ and $\Sigma = (S, \leq, F)$, iff*

- *$\mathcal{E}^\sim$ is a protecting extension of $\mathcal{E}$;*
- *the poset of sorts of $\Sigma^\sim$ extends $(S, \leq)$ by adding a new sort Bool that belongs to a new connected component, with constants $\top$ and $\bot$ such that $\mathcal{T}_{\mathcal{E}^\sim, Bool} = \{[\top], [\bot]\}$, with $\top \neq_{E^\sim} \bot$; and*
- *for each connected component in $(S, \leq)$ there is a top sort $k \in S^\sim$ and a binary commutative operator $\_ \sim \_ : k\ k \longrightarrow Bool$ in $\Sigma^\sim$, such that the following equivalences hold for any* ground terms *$t, u \in T_{\Sigma,k}$:*

$$\mathcal{E} \vdash t = u \quad \Longleftrightarrow \quad \mathcal{E}^\sim \vdash (t \sim u) = \top,$$
$$\mathcal{E} \nvdash t = u \quad \Longleftrightarrow \quad \mathcal{E}^\sim \vdash (t \sim u) = \bot.$$

An equality enrichment $\mathcal{E}^\sim$ of $\mathcal{E}$ is called Boolean *iff it contains all the function symbols and equations making the elements of $\mathcal{T}_{\mathcal{E}^\sim, Bool}$ a two-element Boolean algebra.*

Using the information about '$\sim$', a Boolean transformation can be applied recursively to φ with the additional information of the equality enrichment, if any is defined.

The goal of the Boolean transformation process on a conditional proof obligation φ having the form $t = u$ **if** γ, is to obtain, if possible, an inductively equivalent proof obligation φ' for which the automatic search tests, explained below, have better chances of success. The following is a description of the Boolean transformations applied recursively by the InvA tool:

- If $t = u$ in φ is such that t is of the form $t_1 \sim t_2$ and u of the form $\bot$, then φ is transformed into $\top = \bot$ **if** $\gamma \wedge t_1 = t_2$.
- If $v_1 = v_2$, with $v_1, v_2 \in T_\Sigma(X)_{Bool}$, is any of the Σ-equalities in the condition γ of φ, then:
  - If v_1 is of the form $v_1^1 \sim v_1^2$ and v_2 of the form $\top$, then $v_1 = v_2$ is replaced by $v_1^1 = v_1^2$.
  - If v_1 is of the form $v_1^1 \sqcap \cdots \sqcap v_1^n$ and v_2 of the form $\top$, then $v_1 = v_2$ is replaced by $v_1^1 = \top \wedge \cdots \wedge v_1^n = \top$. Note that the v_1^i have sort *Bool*.
  - If v_1 is of the form $v_1^1 \sqcup \cdots \sqcup v_1^n$ and v_2 of the form $\bot$, then $v_1 = v_2$ is replaced by $v_1^1 = \bot \wedge \cdots \wedge v_1^n = \bot$. Note that the v_1^i have sort *Bool*.

Symbols $\sqcap$ and $\sqcup$ are used to represent, respectively, the conjunction and disjunction function symbols used by the Boolean equality enrichment in Definition 1. Also note that Σ-equalities are unoriented, and thus in the Boolean transformation the order of terms in the equalities is immaterial.

After the Boolean transformation process is completed, some automatic search tests are applied to the resulting proof obligation following the strategy described below. In what follows, it is assumed that φ has been already simplified by the abovementioned Boolean transformations. Furthermore, let $\overline{t}$, $\overline{u}$, $\overline{\gamma}$ be obtained from t, u, and γ, respectively, by replacing each variable $x \in X$ by a new constant $\overline{x} \in \overline{X}$, with $\Sigma \cap \overline{X} = \varnothing$.

1. *Equational simplification.* The strategy checks if φ holds *trivially*, i.e., if

   $$t\downarrow_{\Sigma,E/B} =_B u\downarrow_{\Sigma,E/B}$$

   or there is $t_i = u_i$ in γ such that $t_i\downarrow_{\Sigma,E/B}, u_i\downarrow_{\Sigma,E/B} \in T_\Sigma$ but

   $$t_i\downarrow_{\Sigma,E/B} \neq_B u_i\downarrow_{\Sigma,E/B} .$$

   Some simplifications in the form of reduction to canonical forms can be made to φ, even if they do not yield a trivial proof of φ. In some cases, such canonical reductions are incorporated into φ and the Boolean transformation is used again.
2. *Context joinability.* It checks whether φ is *context-joinable* [8]. The proof obligation φ is context-joinable iff $\overline{t}$ and $\overline{u}$ are joinable in the rewrite theory $\mathcal{R}^{\varphi}_{\mathcal{E}} = (\Sigma(\overline{X}), B, \overrightarrow{E} \uplus \overrightarrow{\gamma})$, obtained by making variables into constants and by orienting the equations E as rewrite rules $\overrightarrow{E}$ and *heuristically* orienting each equality $t_i = u_i$ in γ as a sequent $\overline{t_i} \rightarrow \overline{u_i}$ in $\overrightarrow{\gamma}$.

3. *Unfeasability.* It checks if the proof obligation is *unfeasible* [8]. The proof obligation φ is unfeasible if there is a conjunct $\overline{t_i} \to \overline{u_i}$ in $\overrightarrow{\gamma}$ and $v, w \in T_\Sigma(X)$ such that $\mathcal{R}_{\mathcal{E}}^{\varphi} \vdash \overline{t_i} \to \overline{v} \wedge \overline{t_i} \to \overline{w}$, $CSU_B(v = w) = \varnothing$, and v and w are *strongly irreducible* with $\overrightarrow{E}$ modulo B, i.e., if v and w are such that each one of its ground instances is in E-canonical form modulo B.
4. *SMT Solving.* It checks if the proof obligation can be proved by an SMT decision procedure. The condition γ of the proof obligation φ is analyzed and, if possible, a subformula consisting only of arithmetic subexpressions is extracted. This subformula has the following property: if it is a contradiction, then γ is unsatisfiable. Therefore, if the SMT decision procedure answers that the input subformula is unsatisfiable, then, as in the previous test, φ is unfeasible.

Because of the admissibility assumptions on $(\Sigma, E \uplus B)$, the first test of the strategy either succeeds or fails in finitely many equational rewrite steps. For the second and third tests, the strategy is not guaranteed to succeed or fail in finitely many rewrite steps because the oriented sequents $\overrightarrow{\gamma}$ can falsify a termination assumption. So, for these last two checks, InvA uses a bound on the depth of the proof-search. For the fourth test, InvA offers support for integer linear arithmetic constraints, which is known to be decidable and for which there are decision procedures already implemented in the SMT solver of choice.

The code in InvA for tests (2) and (3) was borrowed and adapted from the Church-Rosser Checker Tool [8]. For the test (4), the InvA tool relies on an extension of Maude with the CVC3 theorem prover available from the Matching Logic Project [25].

4 The Alternating Bit Protocol

The *Alternating Bit Protocol* (ABP) [2] is a data layer protocol. It was designed to achieve reliable full-duplex data transfer between two processes over an unreliable half-duplex transmission line in which messages can be lost or corrupted in a detectable way. The data link layer, the second lowest layer in the OSI seven layer model, splits data into frames for sending on the physical layer and receives acknowledgment frames. It performs error checking and re-transmits frames not received correctly. It provides an error-free virtual channel to the network layer, the third lowest layer in the OSI layer model.

The overall structure of ABP is illustrated in Figure 3. The protocol comprises an input stream of data to be transmitted, a *sender* and a *receiver* process, each having a data buffer and a one *bit* state, a *data channel* for data-bit pairs called *bit-packets*, an *acknowledgment channel* for bit-packets consisting of a single bit, and an output data stream. Here is how the protocol works:

– The sender process starts by repeatedly sending bit-packets (b, d_1) into the data channel, where b is the sender's bit and d_1 is the first element of the input stream.

– The receiver process starts by waiting until it receives the bit-packet (b, d_1), and then it repeatedly sends b over the acknowledgment channel.
– When the source process receives b, it begins repeatedly sending the bit-packet $(\mathit{flip}(b), d_2)$, where d_2 is the second element of the input stream, which is what the receiver process is now waiting for.
– When the target receives $(\mathit{flip}(b), d_2)$, it begins sending packets containing $\mathit{flip}(b)$.
– At any moment either channel can duplicate or lose its oldest packet, if any.
– And so on ...

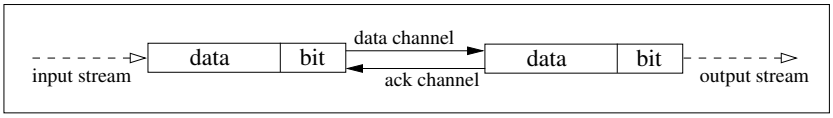

Fig. 3: The Alternating Bit Protocol.

The protocol is highly concurrent and non-deterministic because, for instance, it is unknown how long will it take before a bit-packet gets through. To guarantee progress, it must be assumed that the channels are fair, in the sense that if the sender persists, eventually a bit-packet will get through. The reason is that without this assumption the algorithm is not correct because data transmission might fail forever. However, this is a fairness assumption that is not needed for analyzing the reliable communication enforced by the protocol. Remember that a safety property assures that "nothing bad happens", even when nothing ever happens.

4.1 Formal Modeling

The ABP specification in Maude has 9 modules. This section gives an overview; the full specification can be found in [23].

At the top level, the state space is represented by the top sort `Sys` defined in module `ABP-STATE`, which is a 6-tuple:

```
sort Sys .
op _:_>_|_<_:_ : iNat Bit BitPacketQueue BitQueue Bit iNatList
                 -> Sys [ctor] .
```

The arguments of a state are the data from the input stream currently being transmitted by the sender (as `iNat`), the bit of the sender (as `Bit`), the data channel (as `BitPacketQueue`), the acknowledgment channel (as `BitQueue`), the bit of the receiver (as `Bit`), and the output stream (as `iNatList`).

The sort `iNat` is that of natural numbers in Peano notation, together with an equality enrichment. Natural numbers are used to represent packets in the potentially infinite input stream.

```
sort iNat .
op 0 : -> iNat [ctor] .
op s_ : iNat -> iNat [ctor] .
op _~_ : iNat iNat -> Bool [comm] .
```

Bits are defined in module `BIT` by sort `Bit` with two constructor constants, a 'flipping' operator, and an equality enrichment:

```
sort Bit .
ops on off : -> Bit [ctor] .
op flip : Bit -> Bit .
op _~_ : Bit Bit -> Bool [comm] .

eq flip(on)
 = off .
eq flip(off)
 = on .
```

Sort `BitPacketQueue` represents lists of bit-packets, sort `BitQueue` represents lists of bits, and sort `iNatList` represents lists of natural numbers. They are all lists defined in the usual way: an empty list is identified by the constructor constant `nil`, "cons" is a constructor binary symbol denoted by juxtaposition, and append is a defined binary symbol denoted by '`;`'. For instance, sort `BitQueue` defined in module `BIT-QUEUE` is specified as follows:

```
sort BitQueue .
op nil : -> BitQueue [ctor] .
op __ : Bit BitQueue -> BitQueue [ctor prec 61] .
op _;_ : BitQueue BitQueue -> BitQueue [prec 65] .

eq nil ; BQ:BitQueue
 = BQ:BitQueue .
eq B1:Bit BQ1:BitQueue ; BQ2:BitQueue
 = B:Bit (BQ1:BitQueue ; BQ2:BitQueue) .
```

Having covered the basic notation, consider the following ground term of sort `Sys` representing a state in the system:

```
s(0)  :  on  >  (off,0) nil  |  nil  <  off  :  (0 nil)
```

In this state, the packet from the input stream currently being sent is `s(0)`, the sender's bit is `on`, the data channel contains only the bit-packet `(off,0)`, the acknowledgment channel is empty, the receiver's bit is `off`, and the output stream consists only of the packet `0`.

Finally, module `ABP` specifies the operation of the protocol with 15 rewrite rules. These rewrite rules model the transmission of the bit-packets through the data channel, the reception of acknowledgments from the receiver, data duplication and loss, among other behaviors of the system. For instance, consider the following five rewrite rules:

```
rl [send-1] :
```

```
   N:iNat : B1:Bit > BPQ:BitPacketQueue
                   | BQ:BitQueue < B2:Bit : NL:iNatList
=> N:iNat : B1:Bit > BPQ:BitPacketQueue ; ((B1:Bit, N:iNat) nil)
                   | BQ:BitQueue < B2:Bit : NL:iNatList .

rl [recv-1b] :
   N:iNat : on     > BPQ:BitPacketQueue
                   | off BQ:BitQueue < B2:Bit : NL:iNatList
=> s(N:iNat) : off > BPQ:BitPacketQueue
                   | BQ:BitQueue < B2:Bit : NL:iNatList .

rl [recv-1c] :
   N:iNat : off    > BPQ:BitPacketQueue
                   | on BQ:BitQueue < B2:Bit : NL:iNatList
=> s(N:iNat) : on >  BPQ:BitPacketQueue
                   | BQ:BitQueue < B2:Bit : NL:iNatList .

rl [recv-2a] :
   N:iNat : B1:Bit > (on,N2:iNat) BPQ:BitPacketQueue
                   | BQ:BitQueue < on : NL:iNatList
=> N:iNat : B1:Bit > BPQ:BitPacketQueue
                   | BQ:BitQueue < off : (N2:iNat NL:iNatList) .

rl [dup-1] :
   N:iNat : B1:Bit > BP:BitPacket BPQ:BitPacketQueue
                   | BQ:BitQueue < B2:Bit : NL:iNatList
=> N:iNat : B1:Bit > BP:BitPacket (BP:BitPacket BPQ:BitPacketQueue)
                   | BQ:BitQueue < B2:Bit : NL:iNatList .
```

The effects of these rules in a state can be summarized as follows:

`[send-1]` models the "fifo" placement of the current bit-packet in the data channel (the acknowledgment channel behaves in the same way).

`[recv-1b]` models the reception of the acknowledgment the sender was waiting for and thus the sender process immediately updates the packet to be transmitted with the next available packet from the input stream and flips its communication bit.

`[recv-1c]` models the reception of an acknowledgment the sender was not waiting for and thus the acknowledgment is ignored.

`[recv-2a]` models the reception of a bit-packet whose contents are put in the output stream.

`[dup-1]` duplicates the first message in the data channel.

Note that because of rule `[recv-1c]`, for instance, the formal model of the ABP has potentially infinitely many reachable states: every time a packet is successfully transmitted, the sender's counter modeling the input stream is increased by one and then the whole sending process starts over again with the next packet.

5 Reliable Communication

The analysis that follows is based on the formal model explained in Section 4.1.

One of the main properties the ABP should enjoy is the reliable communication property. This means that the protocol makes possible to *reliably* communicate and deliver information from a source to a destination, even in the presence of unreliable channels of communication. The goal in this section is to report on the experience of using the InvA tool in the successful mechanical verification of this property.

5.1 Formal Specification of the Property

Reliable communication in `ABP` means that whenever n packets have been delivered, these were the first n packets sent in that particular order. Note that this is a property that must hold for each natural number n and that cannot be effectively checked by means of direct algorithmic techniques, such as model checking the `ABP` specification, even if the set of initial states is finite.

The reliable communication property is expressed by the state predicate `inv-main` and is defined as follows:

```
op inv-main : Sys -> Bool .

eq [inv-main-1] :
   inv-main(N:iNat : B:Bit  > BPQ:BitPacketQueue
                            | BQ:BitQueue < B:Bit : NL:iNatList)
 = (N:iNat NL:iNatList) ~ gen-list(N:iNat) .
ceq [inv-main-2] :
   inv-main(N:iNat : B1:Bit > BPQ:BitPacketQueue
                            | BQ:BitQueue < B2:Bit : NL:iNatList)
 = NL:iNatList ~ gen-list(N:iNat)
if B1:Bit ~ B2:Bit = false .

op gen-list : iNat -> iNatList .

eq gen-list(0)
 = (0 nil) .
eq gen-list(s N)
 = (s N) gen-list(N) .
```

State predicate `inv-main` is fully defined by two equations and uses the auxiliary function `gen-list`. Equation `[inv-main-1]` considers the case in which the parity of the sender and receiver bits coincides. In this case, the reliable communication property holds if and only if the delivered packets correspond to all but the last packet sent and they are all in order. Equation `[inv-main-2]` considers the case in which the parity of the sender and receiver bits does not coincide. In this case, the reliable communication property holds if and only if the delivered packets correspond to all packets sent and they are all in order.

Given a natural number n, function `gen-list` generates the list of the first n natural numbers in decreasing order.

Consider the rule `[recv-2b]` that models packet reception in `ABP` in order to motivate the correctness of the reliable communication property:

```
rl [recv-2b] :
   N:iNat : B:Bit > (off,N1:iNat) BPQ:BitPacketQueue
                  | BQ:BitQueue < off : NL:iNatList
=> N:iNat : B:Bit > BPQ:BitPacketQueue
                  | BQ:BitQueue < on : (N1:iNat NL:iNatList) .
```

Note that when a packet `N1:iNat` is received there is no assumption made about the relationship between `N1:iNat` and the current packet from the input stream `N:iNat` or the already delivered packets `NL:iNatList`. In this case, there is no obvious reason for the reliable communication property to hold, even if a state initially satisfies this property.

The goal is to prove the `ABP` `inv-main`-invariant from `init`. State predicate `init` defines the set of initial states as follows:

```
op init : Sys -> [Bool] .

eq [init-1] :
   init( 0 : on > nil | nil < on : nil)
 = true .
eq [init-2] :
   init( 0 : off > nil | nil < off : nil)
 = true .
```

The set of initial states for the verification task at hand, as defined by `init`, consists of exactly two states. Namely, those states where the packet to be transmitted is `0`, the sender and receiver bits coincide, the communication channels are empty, and no packet has been delivered.

The following verification commands can be given to the InvA tool in order to check if state predicate `inv-main` is an inductive invariant from `init`:

```
(analyze init(S:Sys) implies inv-main(S:Sys) in ABP-PREDS .)

(analyze-stable inv-main(S:Sys) in ABP-PREDS ABP .)
```

It is assumed that module `ABP-PREDS` contains the state predicates and their corresponding auxiliary function symbols, and module `ABP` contains the specification of ABP, as explained in Section 4.1 and documented in [23].

When the above-mentioned commands, the InvA tool generates the following output:

```
Checking ABP-PREDS ||- init(S:Sys) => inv-main(S:Sys) ...
Proof obligations generated:  2
Proof obligations discharged: 2
Success!
```

```
Checking ABP-PREDS |- inv-main(S:Sys) => 0 inv-main(S:Sys) ...
Proof obligations generated:  30
Proof obligations discharged: 22
The following proof obligations need to be discharged:
8. from inv-main-2 & recv-2b : pending
    inv-main(#7:iNat : #8:Bit > #10:BitPacketQueue
      | #11:BitQueue < on :(#9:iNat #12:iNatList)) = true
 if off ~ #8:Bit = false
 /\ #12:iNatList = gen-list(#7:iNat).
...
```

The tool generates 32 proof obligations and automatically discharges 24 of them. The remaining 8 proof obligations are returned to the user; in the snapshot, only one proof obligation for ground stability that was not automatically discharged is shown and it is identified by label 8.

Upon inspection of the InvA's output, it is relatively easy to observe that `inv-main` is not an inductive invariant for `ABP`. Indeed, consider the proof obligation identified by label 8, as show in the snapshot above, and a ground interpretation where `#8:Bit` is `on`, `#7:iNat` and `#9:iNat` are `0`, and `#12:iNatList` is the singleton list `0 nil`. For this particular ground instantiation, the condition in the proof obligation is satisfied because `on ~ off` reduces to `false` and the value returned by `gen-list` on input `0` is the ground list `0 nil`. However, by equation `[inv-main-2]` in the definition of predicate `inv-main`, this proof obligation is false because the lefthand side of the conclusion reduces to the Boolean term `0 nil ~ 0 0 nil`, which ultimately reduces to `false`. This is evidence of the fact that a stronger predicate is needed, that is, `inv-main` needs to be strengthened.

5.2 Strengthening the Invariant

The first observation to make is that the InvA tool would be able to automatically discharge more proof obligations and also return simpler ones if there were some mechanism for achieving case analysis on the sort `Bit`. Since the InvA internals do not yet offer this feature, a practical approach is to include the case splitting as part of the predicate's equational definition (similarly to what was done in the definition of state predicate `init`). For instance, state predicate `inv` is a finer-grained version of `inv-main` that exhibits the idea of case splitting on the sort `Bit` for the case of the bits in the sender and receiver.

```
op inv : Sys -> Bool .

eq [inv-1a] :
   inv(N:iNat : on  > BPQ:BitPacketQueue
                    | BQ:BitQueue < on : NL:iNatList)
 = (N:iNat NL:iNatList) ~ gen-list(N:iNat) .
eq [inv-1a] :
   inv(N:iNat : off > BPQ:BitPacketQueue
                    | BQ:BitQueue < off : NL:iNatList)
```

```
  = (N:iNat NL:iNatList) ~ gen-list(N:iNat) .
eq [inv-2a] :
   inv(N:iNat : on  > BPQ:BitPacketQueue
                    | BQ:BitQueue < off : NL:iNatList)
 = NL:iNatList ~ gen-list(N:iNat) .
eq [inv-2a] :
   inv(N:iNat : off > BPQ:BitPacketQueue
                    | BQ:BitQueue < on : NL:iNatList)
 = NL:iNatList ~ gen-list(N:iNat) .
```

Since the case analysis on the sort `Bit` is already implemented in predicate `inv`, and this is potentially useful for automation in the overall proof, this predicate is preferred over predicate `inv-main`. The idea is then to strengthen `inv` instead of `inv-main`. Within the overall context of the verification task, the change of predicate `inv-main` for `inv` requires a formal proof of the following implications:

$$\mathtt{ABP} \Vdash \mathtt{init} \Rightarrow \mathtt{inv} \qquad \text{and} \qquad \mathtt{ABP} \Vdash \mathtt{inv} \Rightarrow \mathtt{inv\text{-}main}.$$

These two proof obligations can be analyzed with the help of inference rule C⇒ in Section 3.1. The InvA's mechanization of this inference rule can automatically discharge the implications:

```
Checking ABP-PREDS ||- init(S:Sys) => inv(S:Sys) ...
Proof obligations generated:  2
Proof obligations discharged: 2
Success!

Checking ABP-PREDS ||- inv(S:Sys) => inv-main(S:Sys) ...
Proof obligations generated:  4
Proof obligations discharged: 4
Success!
```

Finding a strengthening for `inv` is not an easy task at first sight. The non-obvious relationships between the channels and the alternating bits, and the many rules that can concurrently apply to a state make this harder. But it is the deep understanding of these relationships that guides the proof effort for obtaining a useful, yet succinct and elegant, strengthening for `inv`.

The key to it all is that the channels behave under some sort of uniformity that is parametric on the sender and receiver bits. This notion of uniformity can be precisely captured with the help of some auxiliary predicates for the two communication channels. Indeed, consider the following auxiliary predicates `all-packets` and `good-packet-queue`:

```
op all-packets : BitPacketQueue Bit iNat -> Bool .

eq [ap-1] :
   all-packets(nil,B:Bit,N:iNat)
 = true .
eq [ap-2] :
```

```
    all-packets(BP:BitPacket BPQ:BitPacketQueue,B:Bit,N:iNat)
  = BP:BitPacket ~ (B:Bit,N:iNat) and
    all-packets(BPQ:BitPacketQueue,B:Bit,N:iNat) .

 op good-packet-queue : BitPacketQueue Bit iNat -> Bool .

 eq [gpq-1] :
    good-packet-queue(nil,B:Bit,N:iNat)
  = true .
ceq [gpq-2] :
    good-packet-queue((B1:Bit,N1:iNat) BPQ:BitPacketQueue,
                      B:Bit,N:iNat)
  = N:iNat ~ s(N1:iNat) and
    good-packet-queue(BPQ:BitPacketQueue,B:Bit,N:iNat)
 if B1:Bit = flip(B:Bit) .
 eq [gpq-3] :
    good-packet-queue((B:Bit,N1:iNat) BPQ:BitPacketQueue,
                      B:Bit,N:iNat)
  = N:iNat ~ N1:iNat and
    all-packets(BPQ:BitPacketQueue,B:Bit,N:Nat) .
```

Predicate `all-packets` on input `BPQ:BitPacketQueue` and `(B:Bit,N:iNat)` is true if and only if all bit-packets in `BPQ` have the form `(B,N)`. Predicate `good-packet-queue` on input `BPQ:BitPacketQueue` and `(B:Bit,N:iNat)` is true if and only if `BPQ` can be split into two parts, one of them possibly empty, where in the initial part of the channel all packets are of the form `(flip(B),N-1)` and in the second part of the form `(B,N)`. For example:

```
good-packet-queue((on,3) (off,4) (off,4) nil, off, 4) = true
good-packet-queue((on,3) (on,3) nil, off, 4) = true
good-packet-queue((off,4) nil, off, 4) = true
good-packet-queue((off,4) (on,4) nil, off, 4) = false
```

Auxiliary predicates `all-bits` and `good-bit-queue` are similar to the auxiliary predicates just discussed for channels of bit-packets, but they are about channels of bits.

```
 op all-bits : BitQueue Bit -> Bool .

 eq [ab-1] :
    all-bits(nil,B:Bit)
  = true .
 eq [ab-2] :
    all-bits(B1:Bit BQ:BitQueue,B:Bit)
  = B1:Bit ~ B:Bit and all-bits(BQ:BitQueue,B:Bit) .

 op good-bit-queue : BitQueue Bit -> Bool .

 eq [gbq-1] :
    good-bit-queue(nil,B:Bit)
```

```
  = true .
ceq [gbq-2] :
    good-bit-queue(B1:Bit BQ:BitQueue, B:Bit)
  = good-bit-queue(BQ:BitQueue,B:Bit)
 if B1:Bit = flip(B:Bit) .
 eq [gbq-3] :
    good-bit-queue(B:Bit BQ:BitQueue, B:Bit)
  = all-bits(BQ:BitQueue,B:Bit) .
```

The strengthening for `inv` is the state predicate `good-queues` that uses the auxiliary predicates above-mentioned:

```
op good-queues : Sys -> Bool .

eq [good-queues-1a] :
   good-queues(N:iNat : on > BPQ:BitPacketQueue |
               BQ:BitQueue < on : NL:iNatList)
 = all-bits(BQ:BitQueue,on) and
   good-packet-queue(BPQ:BitPacketQueue,on,N:iNat) .
eq [good-queues-1b] :
   good-queues(N:iNat : off > BPQ:BitPacketQueue |
               BQ:BitQueue < off : NL:iNatList)
 = all-bits(BQ:BitQueue,off) and
   good-packet-queue(BPQ:BitPacketQueue,off,N:iNat) .
eq [good-queues-2a] :
   good-queues(N:iNat : on > BPQ:BitPacketQueue |
               BQ:BitQueue < off : NL:iNatList)
 = good-bit-queue(BQ:BitQueue,off) and
   all-packets(BPQ:BitPacketQueue,on,N:iNat) .
eq [good-queues-2b] :
   good-queues(N:iNat : off > BPQ:BitPacketQueue |
               BQ:BitQueue < on : NL:iNatList)
 = good-bit-queue(BQ:BitQueue,on) and
   all-packets(BPQ:BitPacketQueue,off,N:iNat) .
```

State predicate `good-queues` is fully defined by four equations. It characterizes the patterns observed on the communication channels, and their relationship with the alternating bits, in four cases. For example, equation `[good-queues-1a]` states that a state in which both bits are `on` satisfies predicated `good-queues` if and only if all bits in the receiver's queue are `on` and the sender's channel can be split into two parts, where in the initial part of the channel all packets are of the form `(off,N-1)` and in the second part of the form `(on,N)`.

As it will be shown, the strengthening `good-queues` of `inv` is enough to prove the correctness of ABP. Figure 4 depicts the full proof-tree for the inductive invariance of `inv-main` from `init` that uses state predicates `inv` and `good-queues`.

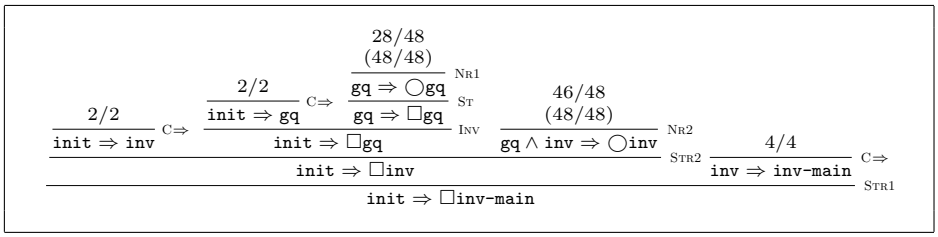

Fig. 4: Correctness proof of the Alternating Bit Protocol (`gq` stands for `good-queues`). The expression d/g denotes the number g of proof obligations generated and the number d of proof obligations automatically discharged by the InvA tool; the same expression in parenthesis has the same meaning but includes the use of the ITP and/or some auxiliary lemmata. Some trivial inferences have been omitted.

The next step in the proof is to check

$\texttt{ABP} \Vdash \texttt{good-queues} \wedge \texttt{inv} \Rightarrow \bigcirc\texttt{inv}$ and
$\texttt{ABP} \Vdash \texttt{init} \Rightarrow \Box\texttt{good-queues}$,

since the following two properties have been already proved:

$\texttt{ABP} \Vdash \texttt{init} \Rightarrow \texttt{inv}$ and $\texttt{ABP} \Vdash \texttt{inv} \Rightarrow \texttt{inv-main}$.

When checking $\texttt{good-queues} \wedge \texttt{inv} \Rightarrow \bigcirc\texttt{inv}$, the following is the output given by the InvA tool:

```
rewrites: 97315 in 348ms cpu (346ms real) (279623 rewrites/second)
Checking ABP-PREDS ||- inv(S:Sys) => O inv(S:Sys)
  assuming good-queues(S:Sys) ...
Proof obligations generated:  48
Proof obligations discharged: 46
The following proof obligations could not be discharged:
8. from inv-1a & recv-2b : pending
    gen-list(#5:iNat)~(#6:iNat #9:iNatList) = true
 if #5:iNat = #6:iNat
 /\ all-bits(#8:BitQueue,off) = true
 /\ all-packets(#7:BitPacketQueue,off,#5:iNat) = true
 /\ gen-list(#5:iNat) = #5:iNat #9:iNatList .
46. from inv-1a & recv-2a : pending
    gen-list(#5:iNat)~(#6:iNat #9:iNatList) = true
 if #5:iNat = #6:iNat
 /\ all-bits(#8:BitQueue,on) = true
 /\ all-packets(#7:BitPacketQueue,on,#5:iNat) = true
 /\ gen-list(#5:iNat) = #5:iNat #9:iNatList .
```

The tool generates 48 proof obligations and automatically discharges 46 of them. The remaining two proof obligations are about properties of lists of natural numbers. Note that the Boolean transformation internally implemented by the InvA

tool (explained in Section 3.1) splits the Boolean conjunctions in the specification of `good-queues` into conditions and the equality predicate '$\sim$' into '=', whenever it was possible. A proof script for proof obligations 8 and 46, that automatically discharges these proof obligations, can be given to the ITP as follows:

```
(goal po8 : ABP-PREDS |- A{ #5:iNat ; #6:iNat ; #9:iNatList ;
                            #8:BitQueue ; #7:BitPacketQueue }
 (
  (#5:iNat) = (#6:iNat) &
  (all-bits(#8:BitQueue,off)) = (true) &
  (all-packets(#7:BitPacketQueue,off,#5:iNat)) = (true) &
  (gen-list(#5:iNat)) = (#5:iNat #9:iNatList)
  =>
  (gen-list(#5:iNat) ~ (#6:iNat #9:iNatList)) = (true)
  )
.)
(auto .)

(goal po46 : ABP-PREDS |- A{ #5:iNat ; #6:iNat ; #9:iNatList ;
                             #8:BitQueue ; #7:BitPacketQueue }
 (
  (#5:iNat) = (#6:iNat) &
  (all-bits(#8:BitQueue,on)) = (true) &
  (all-packets(#7:BitPacketQueue,on,#5:iNat)) = (true) &
  (gen-list(#5:iNat)) = (#5:iNat #9:iNatList)
  =>
  (gen-list(#5:iNat) ~ (#6:iNat #9:iNatList)) = (true)
  )
.)
(auto .)
```

The following is the output of the ITP:

```
==================================
label-sel: po8#0@0
==================================
A{#5:iNat ; #6:iNat ; #7:BitPacketQueue ; #8:BitQueue ; #9:iNatList}
 gen-list(#5:iNat) = #5:iNat #9:iNatList
 & all-packets(#7:BitPacketQueue,off,#5:iNat) = true
 & all-bits(#8:BitQueue,off) = true & #5:iNat = #6:iNat
 ==> gen-list(#5:iNat)~(#6:iNat #9:iNatList) = true

++++++++++++++++++++++++++++++++

rewrites: 10751 in 173ms cpu (181ms real) (61990 rewrites/second)
Eliminated current goal.

q.e.d

++++++++++++++++++++++++++++++++

rewrites: 9172 in 51ms cpu (51ms real) (177962 rewrites/second)

==================================
label-sel: po46#1@0
==================================
```

```
A{#5:iNat ; #6:iNat ; #7:BitPacketQueue ; #8:BitQueue ; #9:iNatList}
 gen-list(#5:iNat) = #5:iNat #9:iNatList
 & all-packets(#7:BitPacketQueue,on,#5:iNat) = true
 & all-bits(#8:BitQueue,on) = true & #5:iNat = #6:iNat
 ==> gen-list(#5:iNat)~(#6:iNat #9:iNatList) = true

++++++++++++++++++++++++++++++++

rewrites: 10751 in 179ms cpu (182ms real) (59745 rewrites/second)
Eliminated current goal.

q.e.d

++++++++++++++++++++++++++++++++
```

This completes the proof of:

$\texttt{ABP} \Vdash \texttt{good-queues} \wedge \texttt{inv} \Rightarrow \bigcirc\texttt{inv}$.

For the proof of $\texttt{init} \Rightarrow \Box\texttt{good-queues}$ the InvA tool gives the following output:

```
rewrites: 10072 in 32ms cpu (35ms real) (314730 rewrites/second)
Checking ABP-PREDS ||- init(S:Sys) => good-queues(S:Sys) ...
Proof obligations generated:  2
Proof obligations discharged: 2
Success!

rewrites: 57223 in 284ms cpu (283ms real) (201476 rewrites/second)
Checking
  ABP-PREDS+LEMMATA ||- good-queues(S:Sys) => O good-queues(S:Sys) ...
Proof obligations generated:  48
Proof obligations discharged: 48
Success!
```

Note that in the proof of inductive stability, module `ABP-PREDS+LEMMATA` is used instead of `ABP-PREDS`. The former module contains 10 lemmata about the auxiliary predicates used by state predicate `good-queues`. Without these lemmata, the InvA tool discharges automatically only 26 of the 48 proof obligations. See [23] for a complete explanation of these lemmata and their mechanical proof in the ITP. This concludes the proof of the inductive invariance of `good-queues` from `init` for `ABP`.

The main result about the correctness of the ABP is then established mechanically in the InvA with help of the ITP. Namely, the following inductive property holds:

$\texttt{ABP} \Vdash \texttt{init} \Rightarrow \Box\texttt{inv-main}$.

See [23] for mechanical proofs of the admissibility of modules `ABP`, `ABP-PREDS`, `ABP-PREDS+LEMMATA`, and also for the ITP proof scripts used as part of the main result in this section.

6 Related Work and Concluding Remarks

The Alternating Bit Protocol (ABP) is a well-established benchmark in the proof technologies that address concurrent, non-deterministic systems. As such, it has

been formally studied from different viewpoints using a wealth of formal techniques. They include process algebra [3,4], temporal Petri nets [27], the Calculus of Constructions [11], and timed rewriting logic [26], among many others.

In the framework of observational transition systems (OTS), ABP has been formally studied independently by K. Ogata and K. Futatsugi [20], and by K. Lin and J. Goguen [14]. In the former, the focus is on proving the same invariant property about reliable communication based on simultaneous induction. In the latter, the focus is on verifying liveness properties using conditional circular coinductive rewriting.

Figure 5 presents a comparison between the proof of the reliable communication property for ABP presented in [20], that uses proof scores, and the one presented here. This comparison is possible thanks to the authors of [20] who kindly shared the source code of their case study.

	Measure	[20]	This work
Model	LOC	286	208
Model + Predicates	LOC	286 + 63	208 + 200
State predicates	#	11	3
Lemmata	#	7	10
Proof scripts	LOC	5189	213
Proof scripts / # predicates	LOC	471.8	71

Fig. 5: Comparison of the ABP case study for the reliable communication property with a similar case study using proof scores in [20].

Note that the human proof effort in [20] is significantly higher than the one in proving the same property using the approach and tools of Section 3, as presented in this paper. However, this comparison needs to be taken with a grain of salt. In particular, the case study using proof scores in [20] does not benefit from automation techniques, not even for many proof obligations that are trivial base cases. In contrast, the combined power of InvA and ITP was of great help, not only because it automatically took care of many simple proof obligations, but also because of some of its equational inductive techniques such as cover-set induction [13].

This paper has presented a case study about the deductive analysis of inductive safety properties using the methodology, the proof system, and the Maude Invariant Analyzer tool (InvA) [23,24]. The subject of study is the Alterating Bit Protocol: a highly concurrent protocol for reliable data communication across a lossy channel. The invariant in this case study is about reliable communication, which is the main safety property of the ABP protocol. As a result of the case study, a fully mechanized proof for the correctness of the protocol is obtained with the InvA tool, and with help of Maude's ITP that was useful for discharging some equational proof obligations and auxiliary lemmata. The proof relies

heavily on the specification and verification methods developed in [23,24], and their implementation in the InvA tool.

Future work should focus on improving the management of proof obligations in the InvA tool, specially when analyzing large specifications. There is also a need for improving the proof heuristics used by the tool. As explained in Section 3, a series of heuristics are employed by the InvA for discharging proof obligations. However, it should be possible to improve some of them and implement some new ones. For example, the InvA tool implements some basic heuristic for checking unsatisfiability of numeric conditions modulo SMT. This could perhaps be combined with equational narrowing, which is already available in Maude. This should increase the number of proof obligations automatically discharged by the tool, and thus lessen the proof effort of the user. There is also the need for improving the techniques available to the user in tools such as the ITP. For instance, inductive techniques such as cover-set induction modulo AC should be investigated, implemented, and offered to the user. The current ITP version supports cover-set induction [13] but for the moment *not* modulo AC. Finally, the comparison in Figure 5 could be taken a step further by (i) extending the InvA tool with (semi)automatic lemma discovery by means of symbolic simulation based on narrowing [1] and rewriting modulo SMT [23], and (ii) by comparing InvA's degree of automation with the OTS/CafeOBJ method assisted with automatic and interactive theorem proving tools such as CrÈme [18] and the newly developed CITP [10].

Acknowledgments. The authors would like to thank the anonymous referees for their comments that helped to improve the paper. This work was partially supported by NSF Grant CNS 13-19109.

References

1. K. Bae, S. Escobar, and J. Meseguer. Abstract logical model checking of infinite-state systems using narrowing. In F. van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*, volume 21 of *LIPIcs*, pages 81–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.
2. K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commununications of the ACM*, 12(5):260–261, May 1969.
3. J. Bergstra and J. Klop. Verification of an Alternating Bit Protocol by means of process algebra protocol. In W. Bibel and K. Jantke, editors, *Mathematical Methods of Specification and Synthesis of Software Systems '85*, volume 215 of *Lecture Notes in Computer Science*, pages 9–23. Springer Berlin / Heidelberg, 1986.
4. M. Bezem and J. F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *CONCUR*, volume 836 of *Lecture Notes in Computer Science*, pages 401–416. Springer, 1994.
5. R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.

6. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
7. M. Clavel and M. Egea. ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams. In M. Johnson and V. Vene, editors, *AMAST*, volume 4019 of *Lecture Notes in Computer Science*, pages 368–373. Springer, 2006.
8. F. Durán and J. Meseguer. A Church-Rosser checker tool for conditional order-sorted equational maude specifications. In P. C. Ölveczky, editor, *WRLA*, volume 6381 of *Lecture Notes in Computer Science*, pages 69–85. Springer, 2010.
9. K. Futatsugi, D. Gâinâ, and K. Ogata. Principles of proof scores in CafeOBJ. *Theoretical Computer Science*, 464:90–112, 2012.
10. D. Gâinâ, 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, Warsaw, Poland, September 3-6, 2013. Proceedings*, volume 8089 of *Lecture Notes in Computer Science*, pages 328–333. Springer, 2013.
11. 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*, volume 1158 of *Lecture Notes in Computer Science*, pages 135–152. Springer Berlin / Heidelberg, 1996.
12. R. Gutiérrez, J. Meseguer, and C. Rocha. Order-sorted equality enrichments modulo axioms. In F. Durán, editor, *Rewriting Logic and Its Applications*, volume 7571 of *Lecture Notes in Computer Science*, pages 162–181. Springer Berlin Heidelberg, 2012.
13. J. Hendrix. *Decision Procedures for Equationally Based Reasoning*. PhD thesis, University of Illinois at Urbana-Champaign, April 2008.
14. K. Lin and J. Goguen. A hidden proof of the Alternating Bit Protocol. Available at `http://cseweb.ucsd.edu/~goguen/pps/abp.ps`.
15. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *WADT*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.
16. J. Meseguer. Twenty years of rewriting logic. *JLAP*, 81(7-8):721–781, 2012.
17. J. Meseguer and J. A. Goguen. Initially, induction and computability. *Algebraic Methods in Semantics*, 1986.
18. M. Nakano, K. Ogata, M. Nakamura, and K. Futatsugi. Crème: an automatic invariant prover of behavioral specifications. *International Journal of Software Engineering and Knowledge Engineering*, 17(6):783–804, 2007.
19. K. Ogata and K. Futatsugi. Proof scores in the OTS/CafeOBJ Method. In E. Najm, U. Nestmann, and P. Stevens, editors, *FMOODS*, volume 2884 of *Lecture Notes in Computer Science*, pages 170–184. Springer, 2003.
20. K. Ogata and K. Futatsugi. Simulation-based verification for invariant properties in the OTS/CafeOBJ method. *Electronic Notes in Theorethical Computer Science*, 201:127–154, 2008.
21. K. Ogata and K. Futatsugi. Proof score approach to analysis of electronic commerce protocols. *International Journal of Software Engineering and Knowledge Engineering*, 20(2):253–287, 2010.
22. A. Pnueli. Deduction is forever. Invited talk at FM'99 avaliable online at `cs.nyu.edu/pnueli/fm99.ps`, 1999.
23. C. Rocha. *Symbolic Reachability Analysis for Rewrite Theories*. PhD thesis, University of Illinois at Urbana-Champaign, 2012. Available at `http://hdl.handle.net/2142/42200`.

24. C. Rocha and J. Meseguer. Proving safety properties of rewrite theories. In A. Corradini, B. Klin, and C. Cîrstea, editors, *CALCO*, volume 6859 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2011.
25. G. Roşu and A. Ştefănescu. Matching Logic: A New Program Verification Approach (NIER Track). In *ICSE'11: Proceedings of the 30th International Conference on Software Engineering*, pages 868–871. ACM, 2011.
26. L. Steggles and P. Kosiuczenko. A timed rewriting logic semantics for SDL: A case study of the Alternating Bit Protocol. *Electronic Notes in Theoretical Computer Science*, 15(0):83 – 104, 1998.
27. I. Suzuki. Formal analysis of the Alternating Bit Protocol by Temporal Petri Nets. *IEEE Transactions on Software Engineering*, 16(11):1273–1281, 1990.
28. P. Viry. Equational rules for rewriting logic. *TCS*, 285:487–517, 2002.