

Behavioral Verification of Distributed Concurrent Systems with BOBJ

Joseph Goguen

Dept. Computer Science & Engineering
University of California at San Diego
jgoguen@ucsd.edu

Kai Lin

San Diego Supercomputer Center
klin@sdsc.edu

Abstract

Following condensed introductions to classical and behavioral algebraic specification, this paper discusses the verification of behavioral properties using BOBJ, especially its implementation of conditional circular coinductive rewriting with case analysis. This formal method is then applied to proving correctness of the alternating bit protocol, in one of its less trivial versions. We have tried to minimize mathematics in the exposition, in part by giving concrete illustrations using the BOBJ system.

1. Introduction

Faced with increasingly complex software and hardware systems, including distributed concurrent systems, where the interactions among components can be very subtle, developers are turning more and more to formal methods. Such methods use specifications written in mathematical logic, and sound proof rules that support refinements, yielding rigorous mathematical proofs of significant properties. The goal is not only to increase quality and decrease cost, but also to allow assertions about reliability that can be checked in a precise way. Formal methods can be applied throughout a development cycle, or at selected steps, in which case refinement proofs can insure the correctness of key decisions at those steps.

Formal methods involve specification and verification. Formal specifications describe a system and its desired properties in a formal language, using notation derived from the underlying logic. Formal verification uses formal logic to prove that a specification satisfies certain desirable properties. A proof can be viewed as a test of a specification, which can help understand requirements, improve specifications, and detect design errors. A specification without proofs may contain inconsistencies or inappropriate assumptions. In our opinion, code level verification is a difficult task that is often not worth the trouble (since code level errors are a small percentage of the total errors in pro-

grams [4]), whereas design level verification is easier and more likely to uncover subtle bugs, because it does not require dealing with the arbitrary complexities of programming language semantics.

These points are illustrated by our proof of the alternating bit protocol in Section 4. There are actually many different ways to specify the alternating bit protocol, some of which are rather trivial to verify, but our specification with fair lossy channels is not one of them. The proof shows that this specification is a behavioral refinement of another behavioral specification having perfect channels, and that the latter is behaviorally equivalent to perfect transmission (we thank Prof. Dorel Lucanu for this interpretation).

Many important contemporary computer systems are distributed and concurrent, and are designed within the object paradigm. It is a difficult challenge for formal methods to handle all the features involved within a uniform framework. Hidden algebra, which was introduced in [12] and elaborated in [21, 22, 16, 32, 34], is a systematic approach to such problems. Hidden algebra allows models that only satisfy their specifications behaviorally, in that they appear to exhibit the required behavior under all relevant experiments; this is important because many clever implementations used in practice only satisfy their specifications in this sense. Hidden algebra extends many sorted algebra by distinguishing between “visible” sorts used to model data, and “hidden” sorts used to model states. This framework provides natural ways to handle the most troubling features of large systems, including concurrency, distribution, nondeterminism, as well as the usual features of the object paradigm, including classes, subclasses (inheritance), and local states with attributes and methods, in addition to abstract data types, generic modules, and more generally, the powerful module system of parameterized programming [11]. Hidden algebra generalizes the approaches of process algebra and transition system to include non-monadic parameterized methods and attributes; this extra power can sometimes dramatically simplify verification.

Behavioral equations were introduced by Reichel [31] in 1981, and have since been used by many researchers. Be-

behavioral logic is a diverse research area, including not just hidden algebra, but also the coherent hidden algebra of Diaconescu [8, 7], and the observational logic of Bidoit and Hennicker [25, 3]. These approaches fall into two broad categories, depending on whether or not a fixed data algebra is assumed for all models. A new generalization of hidden algebra treats these variants in a uniform way [16, 34]. Coalgebra is another related area, in that it also supports behavioral specification, and uses coinduction; e.g., see the survey paper [26].

Of course, the proof rules in these logics are sound, but they are also incomplete [6], so there cannot be any algorithm for proving all true statements. Context induction [24, 2] and general coinduction [18, 19] are two popular proof techniques for verifying behavioral properties, but both need intensive human intervention. Circular coinduction, introduced in [34], is a powerful method, effectively implemented by the circular coinductive rewriting algorithm, which has automatically proved many behavioral properties [16, 17]. Behavioral equivalence generalizes the notion of bisimilarity used in process algebra, where there is a very large literature, including proof methods that seem to be special cases of coinduction. However, this lies outside the scope of this paper, so we just mention Milner’s very influential process algebra CCS [29], and [30], where the notion of bisimilarity seems to have originated.

BOBJ [16, 17, 34, 32] is an executable algebraic specification language developed in the Meaning and Computation Lab at the University of California, San Diego, for supporting behavioral specification and verification, based on recent developments in hidden algebra. In addition to rewriting for order sorted equational logic¹, BOBJ also implements order sorted behavioral rewriting and conditional circular coinductive rewriting with case analysis (the latter abbreviated C4RW). This paper illustrates this method by verifying the Alternating Bit Protocol (abbreviated ABP), in one of its less trivial versions; BOBJ seems to be the first system to support automatic coinduction proofs of anything like this complexity. Such proofs have only recently become possible, due to the implementation of C4RW in BOBJ, and a new *event mark stream* approach to fairness, as discussed in Section 4.2 below. CafeOBJ [7] and Spike [2] also support behavioral specification and verification, but C4RW is only implemented in BOBJ.

Section 2 explains some basics of classical and hidden algebraic specification, Section 3 discusses the C4RW algorithm, while Section 4 discusses conclusions and some future work. We try to keep theory to a minimum, and we also describe only features of BOBJ that are necessary for our examples; much more information about BOBJ can be found in the thesis of Kai Lin [27].

¹ I.e., many sorted with subsorts [20, 23].

We offer many thanks to Grigore Roşu for his work on circular coinductive rewriting, and to Kokichi Futatsugi for his ongoing support and encouragement. We also thank Monica Marcus for finding a bug, and Dorel Lucanu for his helpful remarks.

2. Algebraic Specification and BOBJ

This section begins with a review of classical algebraic specification, including both loose and initial specification, and then moves on to behavioral algebraic specification, particularly its foundation in hidden algebra, and its implementation in BOBJ.

2.1. Loose and Data Specification

This subsection introduces the basic concepts, notation and terminology that we need from classical algebraic specification; readers may wish to consult this material on an as-needed basis. Given a set S , an S -sorted set A is a family of sets A_s , one for each $s \in S$. The elements of S are called *sorts* and the notation $\{A_s \mid s \in S\}$ is used. A *signature* Σ is an $(S^* \times S)$ -sorted set $\{\Sigma_{w,s} \mid \langle w, s \rangle \in S\}$. The elements of $\Sigma_{w,s}$ are called operation (or function) symbols of *arity* w , *sort* s , and *type* $\langle w, s \rangle$; in particular, $\sigma \in \Sigma_{[],s}$ is a constant symbol ($[]$ denotes the empty string). If σ has the type $\langle w, s \rangle$, we write $\sigma : w \rightarrow s$, and constants are written $c : \rightarrow s$ when $c \in \Sigma_{[],s}$.

Signatures are given in BOBJ by giving sorts after the keywords `sort` or `sorts`, and operations after the keywords `op` or `ops`. The *form* of an operation follows the `op` keyword, then a colon followed by a list of the sorts for arguments to that operation, followed by an arrow, followed by the value sort of the operation. Underbar characters may serve as place holders within the form, to indicate where the arguments should go; the number of underbars and argument sorts should be the same, as in the `in` operation below. If there are no underbars but the argument sort list is non-empty, as with the `insert` operation below, the operation is assumed to have syntax that requires opening and closing parentheses, with commas between arguments, as in `insert(2, S)`. The following is a simple signature for a theory of sets in BOBJ notation:

```
sorts Elt Set .
op empty : -> Set .
op _in_ : Elt Set -> Bool .
op insert : Elt Set -> Set .
```

Note that overloading is possible (and is helpful for readability) in this framework, since the same form can have more than one type. For example, the form `_in_` could also be an operation on lists, with type $\langle \text{List List}, \text{Bool} \rangle$, where `Bool` is the sort of Booleans, from the builtin module `BOOL`, which is imported by default into every other module.

Σ is a *ground signature* iff $\Sigma_{[],s} \cap \Sigma_{[],s'} = \emptyset$ whenever $s \neq s'$ and $\Sigma_{w,s} = \emptyset$ unless $w = []$. *Union* is defined componentwise, by $(\Sigma \cup \Sigma')_{w,s} = \Sigma_{w,s} \cup \Sigma'_{w,s}$. A common case is union with a ground signature X , where we use the notation $\Sigma(X)$ for $\Sigma \cup X$.

A Σ -*algebra* A consists of an S -sorted set also denoted A , plus an *interpretation* of Σ in A , which is a family of arrows $i_{s_1 \dots s_n, s} : \Sigma_{s_1 \dots s_n, s} \rightarrow [A_{s_1} \times \dots \times A_{s_n} \rightarrow A_s]$ for each type $\langle s_1 \dots s_n, s \rangle \in S^* \times S$, which interpret the operation symbols in Σ as actual operations on A . For constant symbols, the interpretation is given by $i_{[],s} : \Sigma_{[],s} \rightarrow A_s$. Usually we write just σ for $i_{w,s}(\sigma)$, but if we need to make the dependence on A explicit, we may write σ_A . A_s is called the *carrier* of A of sort s . Given Σ -algebras A and B , a Σ -*homomorphism* $h : A \rightarrow B$ is an S -sorted arrow $h : A \rightarrow B$ such that $h_s(\sigma_A(m_1, \dots, m_n)) = \sigma_B(h_{s_1}(m_1), \dots, h_{s_n}(m_n))$ for each $\sigma \in \Sigma_{s_1 \dots s_n, s}$ and all $m_i \in A_{s_i}$ for $i = 1, \dots, n$, and such that $h_s(c_A) = c_B$ for each constant symbol $c \in \Sigma_{[],s}$.

A Σ -*congruence relation* \sim on a Σ -algebra A is a S -indexed equivalence relation such that if $\sigma : s_1 \dots s_n \rightarrow s$ and $a_i, b_i \in A_{s_i}$ with $a_i \sim b_i$ for $1 \leq i \leq n$, then $\sigma(a_1, \dots, a_n) \sim \sigma(b_1, \dots, b_n)$. Given a Σ -congruence relation \sim on A , the *quotient Σ -algebra* A/\sim is a Σ -algebra A/\sim such that $(A/\sim)_s$ is A_s/\sim_s for any sort s and $\sigma([a_1], \dots, [a_n]) = [\sigma(a_1, \dots, a_n)]$ for any $a_i \in A_{s_i}$ and $1 \leq i \leq n$ and $\sigma : s_1 \dots s_n \rightarrow s \in \Sigma$.

Given an S -sorted signature Σ , the S -sorted set T_Σ of Σ -*terms* is the smallest S -sorted set that such that $\Sigma_{[],s} \subseteq T_{\Sigma,s}$ and given $\sigma \in \Sigma_{s_1 \dots s_n, s}$ and $t_i \in T_{\Sigma, s_i}$ then $\sigma(t_1 \dots t_n) \in T_{\Sigma, s}$. Notice that T_Σ is a Σ -algebra by interpreting $\sigma \in \Sigma_{[],s}$ as just σ , and $\sigma \in \Sigma_{s_1 \dots s_n, s}$ as the operation sending t_1, \dots, t_n to the list $\sigma(t_1 \dots t_n)$. Thus, T_Σ is called the Σ -*term algebra*. Note that because of overloading, terms do not always have a unique parse. The following is the key property of this algebra:

Theorem 1 Given a signature Σ with no overloaded constants and a Σ -algebra A , there is a unique Σ -homomorphism $T_\Sigma \rightarrow A$. \square

The proofs of this and other theorems in this section are omitted; they can be found in [28, 9], among other places.

Given a signature Σ and a ground signature X disjoint from Σ , we can form the $\Sigma(X)$ -algebra $T_{\Sigma(X)}$ and then view it as a Σ -algebra by forgetting the names of the new constants in X ; let us denote this Σ -algebra by $T_{\Sigma(X)}$. It has the following universal *freeness* property:

Theorem 2 Given a Σ -algebra A and $a : X \rightarrow A$, there is a unique Σ -homomorphism $a : T_{\Sigma(X)} \rightarrow A$ extending a , in the sense that $\bar{a}_s(x) = a_s(x)$ for each $x \in X_s$ and $s \in S$; sometimes we will write just a instead of \bar{a} . \square

A Σ -*equation* consists of a ground signature X of variable symbols (disjoint from Σ) plus two $\Sigma(X)$ -terms of

the same sort $s \in S$; we may write such an equation abstractly in the form $(\forall X) t = t'$ and concretely in the form $(\forall x, y, z) t = t'$ when $|X| = \{x, y, z\}$ and the sorts of x, y, z can be inferred from their uses in t and in t' . Similarly, a Σ -conditional equation consists of a ground signature X of variable symbols plus a set of pairs of $\Sigma(X)$ -terms u_i, u'_i and t, t' , each pair of the same sort and $1 \leq i \leq n$, written in the form $(\forall X) t = t'$ if $u_1 = u'_1, \dots, u_n = u'_n$. Hereafter we use the word “equation” for both the conditional and unconditional cases. A *specification* or *theory* P is a pair (Σ, E) , consisting of a signature Σ and a set E of Σ -equations.

Suppose Σ -equation e is $(\forall X) t = t'$ if $u_1 = u'_1, \dots, u_n = u'_n$ and A is a Σ -algebra, we say A *satisfies* this equation, written $A \models_\Sigma e$, iff for any map $\theta : X \rightarrow A$, if $\theta(u_i) = \theta(u'_i)$ for $1 \leq i \leq n$, then $\theta(t) = \theta(t')$. Given a specification $P = (\Sigma, E)$, $A \models P$ iff $A \models_\Sigma e$ for every $e \in E$.

Given a set of Σ -equations E , we define the *provability relation* \vdash_Σ for Σ -equations is defined by the following rules:

1. $E \vdash_\Sigma (\forall X) t = t$
2. If $E \vdash_\Sigma (\forall X) t = t'$, then $E \vdash_\Sigma (\forall X) t' = t$
3. If $E \vdash_\Sigma (\forall X) t = t'$ and $E \vdash_\Sigma (\forall X) t' = t''$, then $E \vdash_\Sigma (\forall X) t = t''$
4. If $(\forall Y) t = t'$ if $u_1 = u'_1, \dots, u_n = u'_n \in E$ and $\theta : Y \rightarrow T_{\Sigma(X)}$ and $E \vdash_\Sigma (\forall X) \theta(u_i) = \theta(u'_i)$ for $1 \leq i \leq n$, then $E \vdash_\Sigma (\forall X) \theta(t) = \theta(t')$.
5. If $E \vdash_\Sigma (\forall Y) t_i = t'_i$ and $t_i \in T_{\Sigma(X), s_i}$ for $1 \leq i \leq n$ and $\sigma : s_1 \dots s_n \rightarrow s$, then $E \vdash_\Sigma (\forall X) \sigma(t_1, \dots, t_n) = \sigma(t'_1, \dots, t'_n)$.

Theorem 3 Completeness: If e is a Σ -equation, $E \models_\Sigma e$ iff $E \vdash_\Sigma e$. \square

The *loose semantics* of a specification (Σ, E) is the class of all Σ -algebras that satisfy the equations in E .

Example 1 A Simple Loose Theory In BOBJ, loose specifications are given in modules delimited by the keywords `th` and `end`, with sorts and operations as before, with variables following the keywords `var` or `vars`, and equations following the keyword `eq`. The following is a loose theory for *monoids*, which are sets with a binary operation (here indicated by juxtaposition) that is associative and has an identity (which is here denoted `e`):

```
th MONOID is sort Elt .
  op e : -> Elt .
  op _ _ : Elt Elt -> Elt .
  vars E E' E'' : Elt .
  eq e E = E .
  eq E e = E .
  eq E (E' E'') = (E E') E'' .
end
```

Instead of writing out the associative and identity laws explicitly, we can also give them as what are called *attributes* in BOBJ, in the following manner:

```

th MONOID is sort Elt .
  op e : -> Elt .
  op _ _ : Elt Elt -> Elt [assoc id: e].
end

```

The `assoc` attribute actually does more than the associative equation: it enables parsing and pattern matching modulo associativity; similarly, the attribute `id`: enables pattern matching modulo identity (see Section 2.2). \square

Given a specification (Σ, E) , a natural congruence relation \equiv_E can be defined directly from \vdash_Σ by $t \equiv_E t'$ iff $E \vdash_\Sigma (\forall \emptyset) t = t'$, and we have the following important result:

Theorem 4 Given a specification $S = (\Sigma, E)$, for any Σ -algebra A with $A \models S$, there exists a unique Σ -homomorphism from T_Σ / \equiv_E to A . \square

This property of T_Σ / \equiv_E is called *initiality*. A useful characterization of initiality is the following:

Theorem 5 Given a set of E of Σ -equations, a Σ -algebra A is initial iff it has no junk (the Σ -homomorphism $T_\Sigma \rightarrow A$ is surjective) and no confusion (it satisfies *only* the equations that can be deduced from E). \square

The *initial semantics* of a specification (Σ, E) is the class of its initial algebras. It can be shown that all the initial algebras of a specification are Σ -isomorphic. By Theorem 4, T_Σ / \equiv_E is an initial algebra of (Σ, E) . Because any element in T_Σ / \equiv_E can be generated by operations, induction is valid for proving properties of initial algebras. Generally, more than one induction scheme is valid for a given specification.

Example 2 The Peano Numbers Below is a simple initial theory of natural numbers in the style of Peano, with addition. Initial theories in BOBJ are delimited by the keywords `dth` (the “d” is for “data”) and `end`.

```

dth PEANO is sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  vars M N : Nat .
  eq M + 0 = M .
  eq M + s N = s(M + N) .
end

```

The first two operations are constructors, and one can do induction over them to prove properties of addition, in the usual way. (These numbers differ from those provided in the builtin module `NAT`, which use Java numbers and provide many operations beyond addition.) \square

Given signatures Σ, Σ' with sorts S, S' , then a *signature morphism* $\Sigma \rightarrow \Sigma'$ is a pair (f, g) where $f: S \rightarrow S'$, and g is an $(S^* \times S)$ -indexed function $g_{w,s}: \Sigma_{w,s} \rightarrow \Sigma'_{f(w),f(s)}$. A *view*, or *theory morphism*, from a theory $T = (\Sigma, E)$ to a theory $T' = (\Sigma', E')$ is a signature morphism $v: \Sigma \rightarrow$

Σ' such that if $(\forall X) t = t'$ is an equation in E , then $E' \vdash (\forall \overline{X}) \overline{v}(t) = \overline{v}(t')$ where $\overline{X}_{f(s)} = X_s$ for any sort $s \in \Sigma$ and $\overline{v}: T_{\Sigma(X)} \rightarrow T_{\Sigma'(\overline{X})}$ is the Σ -homomorphism induced by v ; we may write $v: T \rightarrow T'$. Note that BOBJ does not check semantic correctness of views, but only their syntax; therefore users must check the semantics.

Example 3 A Simple View

```

view V from MONOID to PEANO is
  sort Elt to Nat .
  op ( _ _ ) to ( _+_ ) .
end

```

The BOBJ syntax for views is straightforward, except that when items are omitted, BOBJ attempts to figure out those missing items; the resulting views are called *default views*; see [23] for details. \square

A *parameterized specification* or *parameterized theory* is a pair (T_1, T_2) of specifications such that T_1 is included in T_2 ; we call T_1 the *parameter* or *interface theory* and T_2 the *body*. In Example 4 below, T_1 is `ELT` and T_2 is `SET`. Instantiation of (T_1, T_2) with an actual parameter P requires a view $T_1 \rightarrow P$ to describe the binding of actual to formal parameters; often a default view can be used. Following ideas developed for the Clear specification language [5, 13], the instantiation is given by a colimit construction. Although not needed in this paper, by exploiting the power of colimits, the “parameterized programming” module system used in BOBJ (and other algebraic specification languages) goes well beyond that of standard programming languages, and in fact, the parameterized modules of Clear and earlier versions of OBJ strongly influenced the module systems of Ada, ML, and C++; see [11, 23] for details.

Example 4 A Parameterized Initial Theory of Sets The initial theory `SET` allows us to form sets of elements from any collection that has an equality relation defined on it satisfying the law of identity, given in its interface theory `ELT`. Parameterization of a module M by an interface I is indicated with the notation $M[X :: I]$, where X is the formal parameter of the parameterized module.

```

th ELT is sort Elt .
  op eq : Elt Elt -> Bool .
  var E : Elt .
  eq eq(E, E) = true .
end

dth SET[X :: ELT] is sort Set .
  op empty : -> Set .
  op _in_ : Elt Set -> Bool .
  op insert : Elt Set -> Set .
  vars E1 E2 : Elt . var S : Set .
  eq E1 in empty = false .
  eq E1 in insert(E2, S) =
    eq(E1, E2) or E1 in S .
  eq insert(E1, S) = S if E1 in S .
end

```

We can tell BOBJ to instantiate SET with the builtin module INT of integers, and call the result INTSET using a default view, with the following:

```
dth INTSET is
  pr SET[INT] .
end
```

Note that this uses a default view from ELT to NAT, and the pr (for “protecting”) indicates a module importation. \square

Two additional features from parameterized programming that will be used in our main example are renaming and sums of modules. The first allows selected sorts and operations to be renamed within a module; this can be very helpful when reusing modules in new contexts. The sum just combines two or more modules, taking proper account of any shared submodules that may have arisen through importation. The syntax of these features is illustrated in the following:

```
dth PEANO+INT is
  pr PEANO *(sort Nat to Peano, op 0 to zero)
  + NAT .
end
```

Here the sort and constant of PEANO are renamed to avoid conflict with those of NAT, and are then combined. Now the BOBJ parser will be able to determine whether the sort of any given term is Peano or Nat, even though the operation `+_` is still overloaded.

2.2. Term Rewriting

Given a signature Σ and ground signatures X, Y of *variable symbols* (disjoint from Σ), a *substitution* θ is a S -sorted set $\{\theta_s: X_s \rightarrow T_{\Sigma,s}(Y)\}$. By Theorem 2, every such θ extends uniquely to a Σ -homomorphism $\bar{\theta}: T_{\Sigma}(X) \rightarrow T_{\Sigma}(Y)$. For any term $t \in T_{\Sigma,s}(X)$, let $\theta(t) = \bar{\theta}_s(t)$. Given a term $p \in T_{\Sigma,s}(X)$ and a term $t \in T_{\Sigma,s}(Y)$, we say p *matches* t if there exists a substitution θ such that $\theta(p)$ is syntactically the same as t .

Given a signature Σ and a ground signature X of variable symbols (disjoint from Σ), a Σ -*rewrite rule* is a pair of terms, written $l \rightarrow r$, such that l and r have the same sort and all variables in r also appear in l . A Σ -*rewrite system*, or *term rewriting system*, abbreviated *TRS*, R is a set of Σ -rewrite rules. A term t *rewrites to* a term t' using R , written $t \rightarrow_R t'$ or just $t \rightarrow t'$, iff there exists a rewrite rule $l \rightarrow r$ in R and a substitution θ such that t has a subterm $\theta(l)$ and t' can be obtained from t by replacing $\theta(l)$ with $\theta(r)$; the term $\theta(l)$ is called the *redex* of the rewrite. Let \rightarrow_R^* be the reflexive and transitive closure of \rightarrow_R . R is *confluent* iff $t \rightarrow_R^* t_1$ and $t \rightarrow_R^* t_2$, then there exists a term t' such that $t_1 \rightarrow_R^* t'$ and $t_2 \rightarrow_R^* t'$. R is *terminating* iff there is no infinite rewriting $t_1 \rightarrow_R^* t_2 \rightarrow_R^* \dots$. A *normal form* of t under R is a term t' such that t' cannot be written and

$t \rightarrow_R^* t'$; we may write $[[t]]_R$ for the normal form of t under R . A TRS is *canonical* iff it is confluent and terminating. It can be shown that in a canonical TRS, every Σ -term has a unique normal form, called its *canonical form*. A survey of basic term rewriting for the one sorted case is given in [1].

BOBJ’s term rewriting capability provides an operational semantics for modules, by viewing equations as rewrite rules, i.e., by applying equations in the forward direction. Term rewriting for initial and loose theories is invoked with the command `red`, followed by a term (and a period). For example,

```
select INTSET .
red 3 in insert(1,insert(2,insert(3,empty))).
```

constructs the set $\{1, 2, 3\}$ and then tests whether 3 is in it, in the context of the module INTSET, which is made the module currently in focus by using the `select` command. Here is the output produced by the above (slightly reformatted to fit within the two column format of this paper):

```
reduce in INTSET : 3 in insert(1, insert(2,
  insert(3, empty)))
result Bool: true
rewrite time: 165ms      parse time: 4ms
```

If some operations have attributes for associativity, commutativity, or identity, then rewriting is done module those equations; we do not go into the details here, because this feature is not needed in our ABP example, but the details can be found in [23, 1] and many other places.

The builtin BOBJ module TRUTH, which is included in BOOL and is therefore by default imported into every other module, provides a polymorphic binary operation denoted `==` which compares the normal forms of its two arguments. For example,

```
red insert(3,insert(3,empty))
== insert(3,empty) .
```

returns `true`, since the two canonical forms are identical; otherwise it returns `false`. If the TRS is canonical, then `true` is returned iff the two terms are provably equal, but if the TRS is non-terminating, reduction may go into an infinite loop, and if the TRS is not confluent, reduction could return `false` when the terms are nonetheless provably equal. The negation of `==`, denoted `=/=`, is also a BOBJ builtin polymorphic operation.

Example 5 Substitution It may be interesting to see an example of substitution in BOBJ, using the `open` and `close` feature, which allows material to be temporarily added to the module currently in focus. Note that a period is required after `open`, but is forbidden after `close`.

```
open .
ops x y z : -> Nat .
eq x = 1 .
eq y = 2 .
eq z = 3 .
```

```

red 3 in insert(x,insert(y,insert(z,empty))) .
close

```

Here the `ops` keyword allows a number of operations with the same type to be introduced together; here they serve as variables for the term to be reduced, while the three equations define a substitution. \square

A *conditional rewrite rule* is a rewrite rule $l \rightarrow r$ with a *condition* $t_1 = t'_1 \wedge \dots \wedge t_n = t'_n$ such that all variables in the condition also appear in l , and it can be denoted as $l \rightarrow r$ *if* $t_1 = t'_1, \dots, t_n = t'_n$. A *conditional term rewriting system* R is a set of conditional or unconditional rewrite rules. A meaningful conditional TRS should contain at least one unconditional rewrite rule. The rewrite relation \rightarrow_R induced by R is defined as follows: a term t *rewrites to* a term t' using R iff there exists a rewrite rule $l \rightarrow r$ *if* $t_1 = t'_1, \dots, t_n = t'_n$ in R and a substitution θ such that t has a subterm $\theta(l)$, and t' can be obtained from t by replacing $\theta(l)$ with $\theta(r)$, and $[[t_i]]_R = [[t'_i]]_R$ for $1 \leq i \leq n$. Since this is a recursive definition, we take \rightarrow_R to be the least binary relation satisfying the definition.

2.3. Hidden Logic

Behavioral specifications characterize how systems behave in response to relevant experiments, rather than how they are implemented. It distinguishes visible from hidden sorts, with equality being strict on visible sorts and behavioral on hidden sorts, in the sense of indistinguishability under experiments. More concretely, a hidden sort is treated as a black box, whose status can only be observed and updated by the operations defined on it. Behavioral specification uses observations to specify the behavior of operations without looking inside the black box. Behavioral specification provides a more natural and flexible abstraction layer for system design. Behavioral satisfaction is more general than strict satisfaction, so that behavioral specifications impose fewer constraints on the semantics of modules. As a result, not all the inference rules of the ordinary equational reasoning are valid for behavioral modules, and special care must be taken in proving behavioral properties; however, a small modification of equational deduction works for behavioral specifications.

A *hidden signature* Σ is a signature with its sorts partitioned into visible sorts V and hidden sorts H . Operations in Σ with one hidden argument and a visible result are called *attributes*, and those with one hidden argument and a hidden result are called *methods*. Constants of hidden sort are called *hidden constants*. A *hidden Σ -algebra* is just a Σ -algebra. The elements of visible sort in a hidden Σ -algebra represent *data*, and those of hidden sorts represent *states*; the subalgebra of visible sorts and operations of visible type is called the *data algebra*. A *behavioral specification* or *theory* is a triple (Σ, Γ, E) where Σ is a hid-

den signature, and Γ is a hidden subsignature of Σ , and E is a (finite) set of Σ -equations. The operations in Γ are called *behavioral*.

The definition hidden algebra given above allows a loose interpretation for the data algebra, following the general approach of [16, 34]. However, this is not appropriate for our ABP example, since if `true` and `false` become identified in the Boolean subalgebra, the protocol cannot work correctly. This may be remedied by requiring every hidden algebra over a given signature to have a *fixed* data algebra, as in the original version of hidden algebra, or alternatively, by allowing so called data constraints, in the sense of [13], as additional sentences. Note that general results proved for the loose data approach will also apply to fixed data algebras, so there is no loss of generality in proceeding in this way.

Given a hidden signature Γ , a Γ -*context* denoted $C[\square]$ for sort s is a Γ -term in $T_\Gamma(\{\square\} \cup Z)$ having exactly one special variable \square of the sort s , where Z is an infinite set of special variables different from \square . If $C[\square]$ is a Γ -context of sort s and $t \in \Sigma_s$, let $C[t]$ denote the result of substituting t for \square . A Γ -context $C[\square]$ for hidden sort s is called Γ -*experiment* if its sort is visible.

If Γ is a subsignature of a hidden signature Σ and A is Σ -algebra and \sim is an equivalence on A , then an operation σ in $\Sigma_{s_1 \dots s_n, s}$ is *congruent* for \sim iff $\sigma_A(a_1, \dots, a_n) \sim \sigma_A(a'_1, \dots, a'_n)$ whenever $a_i \sim a'_i$ for $1 \leq i \leq n$. A *hidden Γ -congruence* on A is an equivalence relation on A that is congruent for each operation in Γ and is the identity on visible sorts. The Γ -congruence \equiv_Σ^Γ , called *behavioral equivalence*, on A is defined as follows: two data elements are equivalent iff they are equal, and two states are equivalent iff they cannot be distinguished by Γ -experiments, i.e., iff any experiment produces the same value when applied to them. The following is a basic result of hidden algebra:

Theorem 6 Given a hidden subsignature Γ of Σ and a Σ -algebra A , \equiv_Σ^Γ is the largest hidden Γ -congruence on A . \square

An operation σ is Σ -*behaviorally congruent* for A (or simply *congruent*) iff it is congruent for \equiv_Σ^Γ . A hidden Σ -algebra A Γ -*behaviorally satisfies* a Σ -equation $e = (\forall X) t = t'$ *if* $u_1 = u'_1, \dots, u_n = u'_n$, written $A \models_\Sigma^\Gamma e$, iff for any mapping $\theta: X \rightarrow A$, if $\theta(u_i) \equiv_\Sigma^\Gamma \theta(u'_i)$ for $1 \leq i \leq n$, then $\theta(t) \equiv_\Sigma^\Gamma \theta(t')$. If E is a set of Σ -equations, then $A \models_\Sigma^\Gamma E$ iff $A \models_\Sigma^\Gamma e$ for any $e \in E$. We say A *behaviorally satisfies* a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$ iff $A \models_\Sigma^\Gamma E$; we write $A \models_\Sigma^\Gamma \mathcal{B}$. Define $E \models_\Sigma^\Gamma e$ iff $A \models_\Sigma^\Gamma E$ implies $A \models_\Sigma^\Gamma e$ for every algebra A . Define $\mathcal{B} \models_\Sigma^\Gamma e$ iff $E \models_\Sigma^\Gamma e$.

Given a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$, the *provability relation* \Vdash_Σ^Γ for Σ -equations is defined by the following rules:

1. Reflexivity: $E \Vdash_\Sigma^\Gamma (\forall X) t = t$.

2. Symmetry: If $E \Vdash_{\Sigma}^{\Gamma} (\forall X) t_1 = t_2$, then $E \Vdash_{\Sigma}^{\Gamma} (\forall X) t_2 = t_1$.
3. Transitivity: If $E \Vdash_{\Sigma}^{\Gamma} (\forall X) t_1 = t_2$ and $E \Vdash_{\Sigma}^{\Gamma} (\forall X) t_2 = t_3$, then $E \Vdash_{\Sigma}^{\Gamma} (\forall X) t_1 = t_3$.
4. Substitution: If $(\forall Y) t = t'$ i.f. $u_1 = u'_1, \dots, u_n = u'_n$ in E and $\theta: Y \rightarrow T_{\Sigma}(X)$ and $E \Vdash_{\Sigma}^{\Gamma} (\forall X) \theta(u_i) = \theta(u'_i)$ for $1 \leq i \leq n$, then $E \Vdash_{\Sigma}^{\Gamma} (\forall X) \theta(t) = \theta(t')$.
5. Congruence:
 - (a) If $E \Vdash_{\Sigma}^{\Gamma} (\forall X) t = t'$ where $t, t' \in T_{\Sigma \cup X, v}$ and $v \in V$, and $t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n \in T_{\Sigma \cup X}$, then $E \Vdash_{\Sigma}^{\Gamma} (\forall X) \sigma(t_1, \dots, t_{i-1}, t, t_{i+1}, \dots, t_n) = \sigma(t_1, \dots, t_{i-1}, t', t_{i+1}, \dots, t_n)$.
 - (b) If $E \Vdash_{\Sigma}^{\Gamma} (\forall X) t_i = t'_i$ for $1 \leq i \leq n$ and σ is congruent operation in Σ , then $E \Vdash_{\Sigma}^{\Gamma} (\forall X) \sigma(t_1, \dots, t_n) = \sigma(t'_1, \dots, t'_n)$.

Define $\mathcal{B} \Vdash (\forall X) t = t'$ iff $E \Vdash_{\Sigma}^{\Gamma} (\forall X) t = t'$. These rules specialize those of ordinary equational deduction by considering all sorts visible. Note that (5b) only applies to congruent operations. If all operations are congruent, then ordinary equational deduction is sound for behavioral satisfaction. The following expresses soundness with respect to both equational and behavioral satisfaction, generalizing a result in [6,7] that equational deduction is sound when all operations are congruent.

Theorem 7 If $\mathcal{B} \Vdash (\forall X) t = t'$, then $\mathcal{B} \equiv_{\Sigma}^{\Gamma} (\forall X) t = t'$ and also $E \models (\forall X) t = t'$. \square

General coinduction [18, 19, 32] can be used to prove that a Σ -equation $(\forall X) t = t'$ is behaviorally satisfied by a behavioral specification \mathcal{B} by the following steps:

- Define a binary relation R on terms (R is called the *candidate relation*).
- Show that R is a hidden Σ -congruence.
- Prove that $t R t'$.

The soundness of general coinduction follows directly from Theorem 6. The major problem with this coinduction is that it requires human intervention to define an appropriate candidate relation.

Context induction [24] can also be used to prove behavioral properties, using well-founded induction on the context structure to show that it is valid for all experiments. But in many real examples, context induction is not trivial and requires extensive human input, for example, in the form of inductive lemmas that can be difficult to discover and difficult to prove [10].

It often happens that some experiments are unnecessary in a context induction, because the equations imply that some experiments are equivalent to others. A similar but dual situation occurs in abstract data type theory when all the elements can be generated from a subset of operations,

called the constructors, generators, or basis (when induction is involved). A general definition of cobasis is introduced in [33], and a simplified version can be given as follows: a *cobasis* Δ is a subset of operations in Γ that generates enough experiments, in the sense that no other experiment can distinguish any two states that cannot be distinguished by these experiments.

Behavioral rewriting [7] is to behavioral deduction what standard rewriting is to standard equational deduction, a simple but useful proof method. Circular coinductive rewriting proves behavioral equalities by combining behavioral rewriting with circular coinduction [16]; it also strengthens the duality with induction by allowing coinductive hypotheses to be used in proofs.

Based on the notion of cobasis, a more powerful proof method called *circular coinduction* is introduced in [34]. A enriched behavioral deduction system can be got by adding the following rule: Suppose Δ is a cobasis of a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$ and $<$ is a well founded partial order on Γ -contexts which is preserved by the operations in Γ . For any terms t_1 and t_2 in $T_{\Sigma}(X)$, if for any $\delta \in \Delta$ and for appropriate variables W , $\mathcal{B} \Vdash_{\Sigma}^{\Gamma} (\forall X)(\forall W) \delta(t_1, W) = c[\theta(t_1)]$ and $\mathcal{B} \Vdash_{\Sigma}^{\Gamma} (\forall X)(\forall W) \delta(t_2, W) = c[\theta(t_2)]$ and $c < \delta$, or else $\mathcal{B} \Vdash_{\Sigma}^{\Gamma} (\forall X)(\forall W) \delta(t_1, W) = u$ and $\mathcal{B} \Vdash_{\Sigma}^{\Gamma} (\forall X)(\forall W) \delta(t_2, W) = u$ for some Γ -term u , then $\mathcal{B} \Vdash_{\Sigma}^{\Gamma} (\forall X) t_1 = t_2$.

2.4. Behavioral Specification

This section describes behavioral modules in BOBJ. Their denotational semantics is the class of all algebras (i.e., implementations) that behaviorally satisfy specifications, and their operational semantics is given by behavioral rewriting. Behavioral modules in BOBJ are defined between the keywords `bth` and `end`. Sorts in behavioral modules are considered hidden unless declared with the keyword `dsort`, for visible sorts in behavioral modules. Similarly, operations in behavioral modules are considered congruent unless given the attribute `ncong`.

Example 6 Behavioral Theory of Sets The following is a behavioral specification for sets:

```

bth BSET[X :: ELT] is sort Set .
  op empty : -> Set .
  op _in_ : Elt Set -> Bool .
  op insert : Elt Set -> Set .
  vars E1 E2 : Elt . var S : Set .
  eq E1 in empty = false .
  eq E1 in insert(E2, S) =
    eq(E1, E2) or E1 in S .
end

```

The first equation describes the observational result on `empty` via `_in_`, and the next two equations give the observation result on `insert` via `_in_`.

Comparing this behavioral theory with the initial theory for sets given in Example 4, the most important difference is that this theory does not have the equation $\text{insert}(E1, S) = S$ if $E1 \text{ in } S$. We will later show that this equation can be proved as a behavioral property of this specification (see Example 15). Although the other equations look the same, they are methodologically different. Data theories are usually designed with respect to some constructors, but behavioral theories are designed with respect to observations. For example, `empty` and `insert` are “constructors” of the data theory `SET`, i.e., all ground sets can be created with them; and then all other operations can be defined based on the terms generated by these constructors. In fact, the operation `_in_` is defined in this style. \square

2.4.1. Methodology To design a behavioral theory, some operations are selected as a *cobasis* to generate the behavioral equivalence relation, and other operations are defined with respect to these basic observers. For example, in the behavioral theory `BSET`, the operation `_in_` is selected as a unique observer in a cobasis, which means that, two sets are behaviorally equivalent iff they always return the same visible results under the observation of `_in_`, i.e., iff they have the same elements.

2.5. Behavioral Rewriting

`BOBJ` provides an operational semantics for behavioral modules, again by applying equations as rewrite rules. But because of non-congruent operations, ordinary rewriting is not in general sound, as illustrated by the following example of a behavioral theory with a non-congruent operation.

Example 7 Nondeterministic Stacks This behavioral theory illustrates one way that nondeterminism can arise in hidden algebra specifications, on a variant of stacks:

```

both NDSTACK is sort Stack .
  protecting NAT .
  op push _ : Stack -> Stack [ncong] .
  op top _ : Stack -> Nat .
  op pop _ : Stack -> Stack .
  var S : Stack .
  eq pop push S = S .
end

```

The operation `push` places a nondeterministically chosen natural number on the top of a stack. Notice that even for behaviorally equivalent stacks `S1` and `S2`, `push(S1)` and `push(S2)` may insert different natural numbers onto `S1` and `S2`; therefore `push(S1)` and `push(S2)` may be distinguishable by the attribute `top`, so that `push` should be declared non-congruent. The only equation in this specification says that a stack is not behaviorally changed by pushing a new element and then popping it.

Notice that `push(pop(push(S))) == push(S)` is not behaviorally satisfied, although `pop(push(S))`

and `S` are behaviorally equivalent. However, with ordinary rewriting, `push(pop(push(S)))` is reduced to `push(S)`. \square

Behavioral rewriting is invoked with the command `red`, which handles non-congruent operations properly. A term $C[\theta(l)]$ behaviorally rewrites to $C[\theta(r)]$, where $C[\square]$ is a context and $l \rightarrow r$ is a rewrite rule, iff one of the following is satisfied:

1. The redex does not have a non-trivial context.
2. All operations from the top of C down to \square are congruent.
3. The context of the redex has a subcontext D such that all the operations from the top of D to \square are congruent and D has a visible sort.

For example, `push(pop(push(S)))` cannot be reduced to `push(S)`, because the context `push(\square)` doesn't satisfy the conditions above.

Example 8 Behavioral Theory of Streams The behavioral specifications `STREAM` and `ZIP` below are much used in this section. The first declares infinite streams parameterized by the “trivial” interface theory `TRIV`, which only requires that some sort be designated.

```

th TRIV is sort Elt . end

both STREAM[X :: TRIV] is sort Stream .
  op head_ : Stream -> Elt .
  op tail_ : Stream -> Stream .
  op _&_ : Elt Stream -> Stream .
  var E : Elt . var S : Stream .
  eq head(E & S) = E .
  eq tail(E & S) = S .
end

```

The operation `_&_` inserts an element into the head of a stream, and `head` and `tail` respectively return the first element, and the stream after removing its first element.

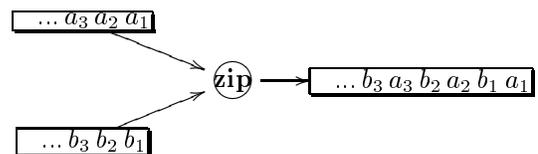
The next specification adds an operation which “zips” two streams together by taking elements from them alternately:

```

both ZIP[X :: TRIV] is pr STREAM[X] .
  op zip : Stream Stream -> Stream .
  vars S S' : Stream .
  eq head zip(S,S') = head S .
  eq tail zip(S,S') = zip(S', tail S) .
end

```

The picture below shows the application of `zip` to two input streams:



The command `red` does behavioral rewriting when in the context of a behavioral theory. For example,

```

open ZIP[NAT] .
  ops ones twos : -> Stream .
  vars S S' : Stream .
  vars N M : Nat .
  eq head ones = 1 .
  eq tail ones = ones .
  eq head twos = 2 .
  eq tail twos = twos .
  red head tail tail zip(ones, twos).
close

```

□

2.6. Behavioral Views

Behavioral parameterized theories can use any kind of theory as their interfaces, but the interfaces of non-behavioral theories must not be behavioral theories, i.e., behavioral theories are only allowed as interfaces for other (parameterized) behavioral theories.

For behavioral theories, we might require a view to send visible sorts to visible sorts, and to satisfy the following condition: For any equation $(\forall X) t = t'$ in E_1 , $(\forall \bar{X}) \bar{v}(t) = \bar{v}(t')$ is behaviorally satisfied by the target module. Unfortunately this definition does not work in general, as shown by the following:

Example 9 Views of Behavioral Theories A behavioral theory for lists of natural numbers is below:

```

bth LIST is sort List .
  pr NAT .
  op nil : -> List .
  op cons : Nat List -> List .
  op _in_ : Nat List -> Bool .
  op head_ : List -> Nat .
  op tail_ : List -> List .
  vars N M : Nat . vars L L1 L2 : List .
  eq N in nil = false .
  eq N in cons(M, L) = N == M or N in L .
  eq head cons(N,L) = N .
  eq tail cons(N,L) = L .
end

```

Now we define a view from BSET[NAT] to LIST, with BSET from Example 6, as follows:

```

view BSET-TO-LIST from BSET[NAT] to LIST is
  sort Set to List .
  op empty to nil .
  op (_in_) to (_in_) .
  op insert to cons .
end

```

Then the translations of equations in the module BSET[NAT] by the view above are all behaviorally satisfied by LIST, and it is also straightforward to prove the following behavioral property in BSET:

$$(\forall N : \text{Nat})(\forall S : \text{Set}) \text{insert}(N, \text{insert}(N, S)) = \text{insert}(N, S)$$

This property and similar properties might have been used in designing other parameterized behavioral theories, such

as APP[X : BSET[LIST]]. However, the corresponding property

$$(\forall N : \text{Nat})(\forall S : \text{List}) \text{cons}(N, \text{cons}(N, S)) = \text{cons}(N, S)$$

is not behaviorally satisfied by LIST, because the experiment $\text{head}(\text{tail}(\square))$ can usually distinguish $\text{cons}(N, \text{cons}(N, S))$ and $\text{cons}(N, S)$. This means the view BSET-TO-LIST should not be used to instantiate APP above, and shows the inadequacy of the definition for behavioral theory morphism suggested above. □

Given behavioral theories $\mathcal{B}_i = (\Sigma_i, \Gamma_i, E_i)$ for $i = 1, 2$, let the set of visible sorts and the set of hidden sorts in \mathcal{B}_i be V_i and H_i , respectively. Then a *behavioral view* from \mathcal{B}_1 to \mathcal{B}_2 is a signature morphism $v : \Sigma_1 \rightarrow \Sigma_2$ such that: (1) $v(s) \in V_2$ for any sort $s \in V_1$; and (2) for any equation $(\forall X) t = t'$, if $\mathcal{B}_1 \models (\forall X) t = t'$, then $\mathcal{B}_2 \models (\forall \bar{X}) \bar{v}(t) = \bar{v}(t')$ where $\bar{X}_{v(s)} = X_s$ for any sort $s \in \Sigma_1$ and $\bar{v} : T_{\Sigma_1(X)} \rightarrow T_{\Sigma_2(\bar{X})}$ is the homomorphism induced by v .

Notice that this definition of behavior views requires verifying all behavioral properties of the source module, which is impossible in practice. It is sufficient to define a signature morphism v from \mathcal{B}_1 to \mathcal{B}_2 such that (1) all translated equations of \mathcal{B}_1 are behaviorally satisfied by \mathcal{B}_2 ; and (2) the image of a cobasis of \mathcal{B}_1 under v is a cobasis of \mathcal{B}_2 . This is because it then follows that any behavioral property of \mathcal{B}_1 is also behaviorally satisfied by \mathcal{B}_2 . In practice, the condition (2) above can be satisfied by making some operations non-congruent.

Example 10 Views between Behavioral Theories (Cont.)

A different behavioral theory for lists of natural numbers is the following:

```

bth LISTNC is sort List .
  pr NAT .
  op nil : -> List .
  op cons : Nat List -> List .
  op _in_ : Nat List -> Bool .
  op head_ : List -> Nat [ncong] .
  op tail_ : List -> List [ncong] .
  vars N M : Nat . vars L L1 L2 : List .
  eq N in nil = false .
  eq N in cons(M, L) = N == M or N in L .
  eq head cons(N,L) = N .
  eq tail cons(N,L) = L .
end

```

The only difference between LISTNC and LIST is that head and tail are declared non-congruent. Now we define the following view:

```

view BSET-TO-LISTNC from BSET[NAT] to LISTNC is
  sort Set to List .
  op empty to nil .
  op (_in_) to (_in_) .
  op insert to cons .
end

```

Because the cobasis $\{_in_ \}$ of $BSET[NAT]$ is mapped to the cobasis $\{_in_ \}$ of $LISTNC$ and all the equations translated from the equations in $BSET[NAT]$ are behaviorally satisfied by $LISTNC$, we see that $BSET\text{-}TO\text{-}LISTNC$ is indeed a view. \square

The definition of behavioral views allows the source to be a behavioral, initial, or loose module; hidden sorts may map to visible sorts, but it does not allow views from initial or loose modules to behavioral modules, since these must map visible sorts to hidden sorts; for example, the instantiation $STREAM[LIST[NAT]]$ is not correct, because the interface $TRIV$ of $STREAM$ has a unique visible sort $TRIV$ whereas the principal sort $List$ of $LIST[NAT]$ is hidden, and can not be used to replace a visible sort.

2.7. Concurrent Connection

The concurrent connection operation is important for constructing distributed concurrent systems from components. $BOBJ$ provides a binary associative infix operator $||$ that takes two (or more) modules as arguments, creating a new hidden sort that is the tupling of the principal sorts of the component modules. More concretely, a concurrent connection yields a behavioral specification whose states are tuples of the states of its components, adding a new sort called $Tuple$, a tupling operation $\langle _, _, \dots, _ \rangle : S_1 S_2 \dots S_n \rightarrow Tuple$ and projection operations $i^* : Tuple \rightarrow S_i$ where i ranges from 1 to the number of modules connected and S_i is the principal sort of the i -th component, plus the following “tupling equation”

$$eq \langle 1^*(T), 2^*(T), \dots, n^*(T) \rangle = T$$

which says that all states are tuples of component states. This construction has been shown to be behaviorally equivalent to concurrent connection defined in a more abstract (category theoretic) way which intuitively captures concurrency [14]. In particular, equations which say that methods in component i commute with or “interleave with”, to use a term from concurrency theory, methods in component j , for $i \neq j$, can be proved to hold behaviorally.

The code below is equivalent to what $BOBJ$ provides for the case of $n = 2$ modules. The sort $Tuple$ is always hidden, even when the module is instantiated with component where all sorts are visible.

```
bth 2 || [1 2 :: TRIV] is sort Tuple .
  op <_,_> : Elt.1 Elt.2 -> Tuple .
  op 1*_ : Tuple -> Elt.1 [ncong].
  op 2*_ : Tuple -> Elt.2 [ncong].
  var E1 : Elt.1 . var E2 : Elt.2 .
  var T : Tuple .
  eq 1* <E1, E2> = E1 .
  eq 2* <E1, E2> = E2 .
  eq <1* T, 2* T> = T .
end
```

The connection operation $\langle _, _ \rangle$ is a congruent operation with two hidden arguments, but the untupling operations are *non-congruent*, because two pairs may be behaviorally equivalent, while their corresponding components are not behaviorally equivalent. $BOBJ$ also has builtin polymorphic tupling for visible sorts, but we do not discuss this here because it is not needed for our ABP example.

3. Verification of Behavioral Properties

Behavioral rewriting can prove simple behavioral properties, but more powerful methods are needed to verify more difficult behavioral properties. Unlike general coinduction [19] and context induction [2], conditional circular coinductive rewriting with case analysis provides a powerful way to prove behavioral properties, with intensive human intervention. This section describes and illustrates the use of circular coinductive rewriting in $BOBJ$.

3.1. Cobasis Discovery and Declaration

Cobases are important for behavioral specification; they are not only used in designing behavioral specifications and in defining views, but also in the verification of behavioral properties. $BOBJ$ has an algorithm that computes default cobases for behavioral specifications based on the congruence criteria of [34]. This algorithm first takes all operations in Γ as a cobasis, and then removes operations that it finds to be redundant. The current version uses behavioral rewriting to check behavioral equivalence, which is stricter than the congruence criterion in [34], which requires behavioral equivalence. The computed default cobasis can be displayed by the command “show cobasis <module-name>”.

Example 11 Default Cobasis for $BSET$ In the context of the behavioral theory $BSET$ in Example 6, the command “show cobasis Set” produces the output:

```
The cobasis for Set is:
  op _ in _ : Elt Set -> Bool [prec 41]
```

The algorithm starts with the cobasis containing the operations $empty$, $insert$ and $_in_$. Because of the equations E in $empty = false$ and $E1$ in $insert(E2, S) = E1 == E2$ or $E1$ in S , the operations $empty$ and $insert$ are removed. \square

Example 12 Default Cobases of $STREAM$ and ZIP Let $STREAM$ and ZIP be the behavioral theories defined in Example 8. $BOBJ$ ’s cobasis algorithm discovers that $\&$ and zip are not needed for $STREAM$ and ZIP . Thus the computed default cobasis for sort $Stream$ is:

```
op head : Stream -> Elt
op tail : Stream -> Stream
```

\square

Given a behavioral theory, its default cobasis might not be the best cobasis for a particular design, or verification problem. BOBJ allows setting cobases for behavioral theories manually with the following syntax:

```
cobasis <cobasis-name> from <module-name> is
  ( <operation-declaration> . )+
end
```

This declares a cobasis for the behavioral theory with the given name, where the body of the declaration gives a list of congruent operations in the behavioral theory. For example, if LIST is the behavioral theory defined in Example 9, then we can declare a cobasis of LIST with:

```
cobasis SIMPLE-COBASIS-OF-LIST from LIST is
  op head : List -> Nat .
  op tail : List -> List .
end
```

This is correct because two lists are behaviorally equivalent iff the results of observing them by head and tail are behaviorally equivalent. Note that this cobasis is smaller than the default cobasis of LIST, since it doesn't contain the operation `_in_`. It is straightforward to see that if two lists cannot be distinguished by any experiment with head and tail, then they also can not be distinguished by any experiment with head, tail and nil.

Another way to declare a cobasis for a module is to use the command

```
set cobasis of <module-name> .
```

This sets the cobasis of the current module to the default cobasis of the indicated module. For example, if we first load the module LISTNC in Example 10, and then load the module LIST in Example 9, the following sets the cobasis of LIST to the default cobasis of LISTNC.

```
set cobasis of LISTNC .
```

BOBJ does not verify the correctness of cobasis declarations, so users must do that themselves.

3.2. The C4RW Algorithm

Circular coinductive rewriting proves behavioral equalities by integrating behavioral rewriting [7] with circular coinduction [16, 17]. This section presents this algorithm, and gives examples showing that it is quite powerful in practice. It takes as input a pair of terms, and returns true whenever it can prove the terms behaviorally equivalent, and otherwise returns false or else fails to terminate, much as with proving term equality by rewriting. Here we describe the BOBJ implementation; its correctness is shown in [34].

Case analyses are first class citizens, that can be named, reused, and combined with other case analyses. The following is (part of) the syntax, though it is probably easier to absorb through examples:

```
<cred> ::= cred with <case-name>
         <term> == <term>
<cases> ::= cases <case-name> for <mod-name> is
          ( <var> | <context> | <case> )+
          end
<context> ::= context <term> .
<case> ::= case ( <equation> . )+
```

The circular coinductive rewriting algorithm, denoted C4RW, is presented in Figure 1; its input is a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$ with a cobasis $\Delta \subseteq \Gamma$, and a pair of Σ -terms (t, t') . In Figure 1, \mathcal{G} denotes the goal equations, which must have visible sorted conditions; elements of the set \mathcal{C} are called *circularities*; $bnf_{\mathcal{B}, \mathcal{C}}(t)$ denotes the normal form of t under behavioral rewriting with \mathcal{B} and \mathcal{C} , where \mathcal{C} can only be applied at the top level; and for Σ a signature, a Σ -case definition is a pair (p, L) where p is a Σ -term called the *pattern*, and L is a list of sets of Σ -equations, where each such set is called a *case*. The variables in p get bound to terms when the case definition is used. Also, $E(d)$, where d is the condition of an equation, indicates the translation from a Boolean expression to a set of equations. Finally, note that the equality sign plays three different roles: for equations in \mathcal{G} and \mathcal{C} , it is a symbol that separates the left and right sides; for **let** (i.e., assignment) statements, it separates the variable from the term assigned to that variable; and otherwise, e.g., in **if** statements, it denotes the syntactic identity relation on terms.

Example 13 A Simple Case Analysis We first define a module APP as follows:

```
dth APP is pr NAT .
  ops odd even : Nat -> Bool .
  op sum : Nat -> Nat .
  var N : Nat .
  eq odd(0) = false .
  eq odd(s 0) = true .
  eq odd(s s N) = odd(N) .
  eq even(N) = not odd(N) .
  eq sum(0) = 0 .
  eq sum(s N) = sum(N) + s N .
end
```

where the operations odd(N) and even(N) test whether N is odd and even respectively, and sum(N) returns the sum of $1 + \dots + N$ for any natural number N. The following is a case analysis named ODD-EVEN for this module:

```
cases ODD-EVEN for APP is
  var N : Nat .
  context sum(N) .
  case eq odd(N) = true .
  case eq even(N) = true .
end
```

A case analysis must be associated with a previously defined module, and is valid only for that module. In the example above, ODD-EVEN is associated with APP and it can be used only for APP. For any case analysis, a unique context must be defined. A context is a term preceded by the keyword context, which specifies the target of the analysis. ODD-EVEN has the context sum(N), which tells BOBJ

to use case analysis on any subterms matched by this pattern. A typical case definition should contain several cases, and each may contain one or more equations that can be used in that case. ODD-EVEN defines two cases: one is for odd natural numbers, and the other is for even natural numbers. Showing correctness of case analysis declarations is the user's responsibility. \square

The following intuitive explanation may help. If case analysis is used with a `cred` command, whenever the left and right sides of a goal are expanded by a cobasis operation and they reduce to different normal forms under behavioral rewriting with circularities, BOBJ checks if the context of some case analysis matches a subterm in either normal form. If it does and the matching substitution is θ , then subgoals are created according to the cases for the matching context, with each case enriched by the substitution instances under θ of the equations in that case. This is repeated until all subgoals reduce to true, which means the proof task is proved; otherwise, a new subgoal is created, whose left and right sides are the two different normal forms. If no further case analysis applies to some subgoal, then the original goal fails, and a new circularity is added for it.

The following command invokes circular coinductive rewriting with a specific case analysis declaration on a specific equational goal:

```
cred with <case-name> <term> == <term> .
```

In executing this command, BOBJ automatically uses case analysis whenever the context of the indicated case definition is found in proof tasks.

Example 14 Behavioral Equivalence over Streams We define two operations `map` and `iter` over streams in the following modules:

```
th FUN is sort E1t .
  op f_ : E1t -> E1t .
end

bth MAP[X :: FUN] is pr ZIP[X] .
  op map_ : Stream -> Stream .
  op iter_ : E1t -> Stream .
  var E : E1t . var S : Stream .
  eq head map S = f head S .
  eq tail map S = map tail S .
  eq head iter E = E .
  eq tail iter E = iter f E .
end
```

For any stream $s = e_1 e_2 e_3 \dots$, `map(s)` returns $f(e_1) f(e_2) f(e_3) \dots$. For any element e of `E1t`, `iter(e)` gives $e f(e) f^2(e) \dots$. We show that `map iter E = iter f E` with the following code below, in which tracing is first enabled:

```
set cred trace on .
cred map iter E == iter f E .
```

Here is the BOBJ output:

Input: (1) a behavioral theory $\mathcal{B} = (\Sigma, \Gamma, E)$
 (2) a cobasis Δ of \mathcal{B}
 (3) a set \mathcal{G} of conditional Σ -equations
 (4) a Σ -case definition (p, L)

Output: **true** if a proof of $\mathcal{B} \models \mathcal{G}$ is found,
 otherwise **false** (or non-terminating)

Procedure:

1. **let** $C = \emptyset$
2. **for each** $(\forall X) s = s' \text{ if } d \text{ in } \mathcal{G}$
3. move $(\forall X) s = s' \text{ if } d$ from \mathcal{G} to C
4. **let** θ be a substitution on X assigning new constants to the variables in d
5. **let** $B' = \mathcal{B} \cup E(\theta(d))$
6. **for each** $\delta \in \Delta$
7. **let** $u = \text{bnf}_{B',C}(\delta[\theta(s), W])$ and
 $u' = \text{bnf}_{B',C}(\delta[\theta(s'), W])$
8. **if** $u \neq u'$
9. **then if** $\text{CASE-ANAL}(u, u', d, B, C, \theta)$ is **true**
10. **then continue**
11. **else add** $(\forall X) (\forall W) \text{bnf}_{B,C}(\delta[s, W]) = \text{bnf}_{B,C}(\delta[s', W])$ **if** d to \mathcal{G}
12. **else continue**

Procedure $\text{CASE-ANAL}(u, u', d, B', C, \theta)$

1. **if** p matches a subterm of u or u' with substitution η
2. **then let** η' be η with a new constant substituted for each variable
3. **for each case** C in L
4. **if** $\text{bnf}_{B' \cup \eta'(C), \emptyset}(\theta(d)) = \text{false}$
5. **then continue**
6. **else let** $B'' = B' \cup \eta'(C) \cup E(\theta(d))$
7. **let** $t = \text{bnf}_{B'',C}(\eta'(u))$ and
 $t' = \text{bnf}_{B'',C}(\eta'(u'))$
8. **if** $t = t'$ or
 $\text{CASE-ANAL}(t, t', d, B' \cup \eta'(C), C)$
 is **true**
9. **then continue**
10. **else return false**
11. **else return true**

Figure 1. The C4RW Algorithm

```
c-reduce in MAP : map (iter E) == iter (f E)
using cobasis for MAP:
  op head _ : Stream -> E1t [prec 15]
  op tail _ : Stream -> Stream [prec 15]
-----
reduced to: map (iter E) == iter (f E)
-----
add rule (C1) : map (iter E) = iter (f E)
-----
target is: map (iter E) == iter (f E)
expand by: op head _ : Stream -> E1t [prec 15]
reduced to: true
          nf: f E
-----
target is: map (iter E) == iter (f E)
expand by: op tail _ : Stream -> Stream [prec 15]
deduced using (C1) : true
          nf: iter (f (f E))
-----
result: true
c-rewrite time: 81ms      parse time: 3ms
```

Similarly, we can do the following

```

open .
  vars S1 S2 : Stream .
  cred map zip(S1, S2) ==
    zip(map(S1), map(S2)) .
close

```

for which BOBJ returns true. \square

Example 15 A Proof Using C4RW Example 6 claimed that $\text{insert}(E, S) = S$ if E in S is a behavioral property of BSET. The following is a proof using C4RW:

```

cases CASES for ELT is
  vars X Y : Elt .
  context eq(X, Y) .
  case eq X = Y .
  case eq eq(X, Y) = false .
end

set cred trace on .
cred with CASES
  insert(E1, S) == S if E1 in S .

```

Two cases are tried when a subterm of the format $\text{eq}(t_1, t_2)$ is found in a proof goal, and one is enriched with the equation $t_1 = t_2$, and the other is enriched with $\text{eq}(t_1, t_2) = \text{false}$. Here is the BOBJ output (slightly reformatted to fit):

```

c-reduce in BSET : insert(E1, S) == S if E1 in S
use: CASES
using cobasis for BSET:
  op _ in _ : Elt Set -> Bool [prec 41]
-----
reduced to: insert(E1, S) == S
-----
add rule (C1) : insert(E1, S) = S if E1 in S
-----
target is: insert(E1, S) == S if E1 in S
expand by: op _ in _ : Elt Set -> Bool [prec 41]
reduced to: eq(~sysconst~Elt-0, e1) or
  (~sysconst~Elt-0 in s) == ~sysconst~Elt-0
  in s
-----
case analysis by CASES
-----
case 1 :
assume: ~sysconst~Elt-0 = e1
reduce: eq(~sysconst~Elt-0, e1) or
  (~sysconst~Elt-0 in s) == ~sysconst~Elt-0
  in s
  nf: true
-----
case 2 :
assume: eq(~sysconst~Elt-0, e1) = false
reduce: eq(~sysconst~Elt-0, e1) or
  (~sysconst~Elt-0 in s) == ~sysconst~Elt-0
  in s
  nf: true
-----
analyzed 2 cases, all cases succeeded
-----
result: true
c-rewrite time: 53ms      parse time: 3ms

```

\square

4. The Alternating Bit Protocol

The Alternating Bit Protocol is a well-established benchmark for proof technologies that address distributed con-

current non-deterministic systems; it is perhaps the simplest non-trivial example of such a system. The ABP is a communication protocol, i.e., a distributed algorithm for reliably transferring data from a source to a target using unreliable channels. There are actually many different ways to formalize the ABP, based on different assumptions about process structure, time, reliability of channels, and so on. This paper proves correctness for an ABP model of intermediate complexity, assuming synchronous discrete time, fair channels, and the ability to recognize transmission errors. The proof relies heavily on the specification and verification methods of hidden algebra, and their implementation in the BOBJ system, especially its C4RW algorithm. As far as we know, this paper presents the first complete algebraic proof for a system of this kind².

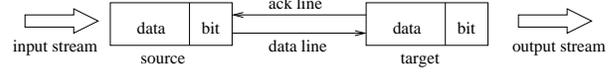


Figure 2. The Alternating Bit Protocol

The structure of the ABP is illustrated³ in Figure 2. It has: an input stream of data to be transmitted; a source and a target process, each having a data buffer and a one bit state; a data channel, for $(data, bit)$ pairs called *packets*; an acknowledgement channel, for packets consisting of a single bit; and an output data stream.

Here’s how the ABP works: the source process starts by repeatedly sending packets (D_1, B) into the data channel, where D_1 is the first element of the input stream, and B is 0 or 1. The target process starts by waiting until it receives the packet (D_1, B) , and then it repeatedly sends B over the acknowledgement channel. When the source process receives B , it begins repeatedly sending the packet $(D_2, 1 - B)$, 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 $(D_2, 1 - B)$, it begins sending packets containing $1 - B$ to the source process. And so on ... Note that we must assume that the two bits are distinct, or this process will fail, and that for convenience, our formalization of the ABP uses Booleans instead of bits; we also need that the natural numbers are distinct. So we take these to be a fixed data algebra for this problem.

It must be assumed that the channels are *fair* in the sense that if the sender persists, eventually a packet will get through; without this assumption, the algorithm is *not* correct, because data transmission might fail forever. This as-

- 2 Here we mean “algebraic” in the sense of algebraic specification theory, rather than, for example, process algebra.
- 3 This diagram and the subsequential informal discussion are intended to motivate the formal specification and proof; they should not be confused with the formalization itself, which is given later.

sumption also implies that the system is non-deterministic, since we do not know how long it will take before a packet gets through. Perhaps the single most challenging problem associated with algebraic correctness proofs for algorithms like ABP has been to formalize fairness. The formalization used here is novel, simple, and powerful; moreover, it makes good use of the capabilities of BOBJ, and is easily extended to other situations.

The formalization of correctness is a crucial part of the proof process. For the ABP, this is straightforward: the output stream should equal the input stream, except that the initial content of the target buffer and all erroneous transmissions should be disregarded.

Since streams are infinite “lazy” structures, *coinductive* methods are required, as opposed to the *inductive* methods that are appropriate for structures such as lists and natural numbers defined with initial semantics.

4.1. The Specification and Goal

Although our ABP specification has just four modules, each of which is small, there are some tricky points.

```

bth STREAM is sort DataStream . pr NAT .
  op hd_ : DataStream -> Nat .
  op tl_ : DataStream -> DataStream .
  op _&_ : Nat DataStream -> DataStream .
  var N : Nat . var Is : DataStream .
  eq hd(N & Is) = N .
  eq tl(N & Is) = Is .
  eq hd Is & tl Is = Is . *** lemma
end

```

```

bth FAIR-STREAM is
  pr STREAM * (sort DataStream to FairStream).
  dsort Mark .
  ops ok err : -> Mark .
  op eq : Mark Mark -> Bool [comm] .
  var M : Mark .
  eq eq(M, M) = true .
  eq eq(err, ok) = false .
  op fhd_ : FairStream -> Mark .
  op ftl_ : FairStream -> FairStream .
  var N : Nat . var F : FairStream .
  eq fhd(0 & F) = ok .
  eq ftl(0 & F) = F .
  eq fhd(N & F) = err if N > 0 .
  eq ftl(N & F) = p N & F if N > 0 .
end

```

```

bth 2CHAN-STATE is
  pr (FAIR-STREAM || FAIR-STREAM) *
    (sort Tuple to 2ChState).
  ops (data-ok_) (ack-ok_) : 2ChState -> Bool .
  ops (fhd1_) (fhd2_) : 2ChState -> Mark .
  op ftl_ : 2ChState -> 2ChState .
  vars F F' : FairStream . var Cs : 2ChState .
  eq fhd1 <F, F'> = fhd F .
  eq fhd2 <F, F'> = fhd F' .
  eq ftl <F, F'> = <ftl F, ftl F'> .
  eq data-ok Cs = eq(fhd1 Cs, ok) .
  eq ack-ok Cs = eq(fhd2 Cs, ok) .
  ops (data-ok-after_) (ack-ok-after_) :
    2ChState -> 2ChState .

```

```

  eq data-ok-after Cs = data-ok-after ftl Cs
    if not data-ok Cs .
  eq data-ok-after Cs = Cs if data-ok Cs .
  eq ack-ok-after Cs = ack-ok-after ftl Cs
    if not ack-ok Cs .
  eq ack-ok-after Cs = Cs if ack-ok Cs .
end

bth ABP is dsort 2PrState .
  pr 2CHAN-STATE + STREAM .
  op <_,_|_,_> : Bool Nat Bool Nat -> 2PrState .
  op [_,_,_] : 2PrState DataStream 2ChState ->
    DataStream .
  vars B B1 B2 : Bool . vars M N : Nat .
  var Is : DataStream . var Cs : 2ChState .
  eq [<B,M | B,N>, Is, Cs] =
    [<B,M | B,N>, Is, ftl Cs]
    if not ack-ok Cs .
  eq [<B1,M | B2,N>, Is, Cs] =
    [<B1,M | B2,N>, Is, ftl Cs]
    if B1 /= B2 and not data-ok Cs .
  eq [<B,M | B,N>, Is, Cs] =
    [<not B, hd Is | B,N>, tl Is, ftl Cs]
    if ack-ok Cs .
  eq tl [<B1,M | B2,N>, Is, Cs] =
    [<B1,M | B1,M>, Is, ftl Cs]
    if B1 /= B2 and data-ok Cs .
  eq hd [<B1,M | B2,N>, Is, Cs] = N
    if B1 /= B2 and data-ok Cs .
end

```

The first module specifies data streams, where the data items are natural numbers. It is the usual specification for streams, but note that it cannot be specified using initial or loose semantics; hidden algebra, or coalgebra, or a similar institution supporting coinduction is necessary. These streams are used for both the input and output data streams of the ABP. The last equation is actually an easily proved lemma, that is included here because it is needed in the correctness proof.

The second module, FAIR-STREAM, defines the “marks” that we use to describe channel behavior. Fair streams of these marks tell whether transmissions succeed or fail. The source and target processes will transfer packets when the streams indicate success, and will wait when they indicate failure. The constant `ok` indicates successful transmission, while `err` indicates failure; this module has initial semantics, so only these two values can appear in its models. This trick of indirectly representing channel behavior simplifies the specification and proof.

There is also a tricky point about the equality relation `eq` in this module. Our correctness proof introduces new values of sort `Mark` through quantifier elimination, and in fact, the proof is conducted in the larger term algebras generated by these constants, whereas the protocol itself lives in smaller algebras without them. In the larger term algebras, the operation `eq` is undefined in some cases, and this is intentional. For example, if `m` is a new constant of sort `Mark`, then both `eq(m, ok)` and `eq(m, err)` are undefined, although `eq(m, m)` is `true`. This contrasts with the

behavior of the builtin equality `==`, which always returns `true` or `false` (or possibly fails to terminate). This incompleteness of `eq` can be important when expressions like `not eq(X, ok)` occur in the condition of an equation and `X` is (for example) `m`, since then the equation cannot be applied, whereas it could be applied if `!=` (the negation of `==`) were used instead; of course, the equation can be applied when `X` is `err`. If `==` and `!=` were used instead of `eq` and `not eq`, the proof in this paper would fail. (By the way, the attribute “[`comm`]” of `eq` makes it a symmetric relation.)

The second module also contains our novel formalization of fairness, based on the infinite streams of natural numbers of the first module. Here the number 0 represents an immediately successful transmission, while $n > 0$ represents n consecutive failures followed by a success. Thus, wherever you are in such a stream, there is always a success some finite distance in the future, which is the meaning of fairness. These streams define *when* a transmission succeeds, but not *what* is transmitted; let’s call them *event mark streams*⁴. Notice that this module introduces head and tail operations that are quite different from those of `STREAM`; `fhd` returns a mark telling whether or not transmission succeeds, and `ftl` returns the next event mark stream.

The third module, `2CHAN-STATE`, provides event mark streams for the two channels. The sort `2ChState` is constructed as the concurrent connection of two fair streams, using `BOBJ`’s builtin `||` operation; it has behavioral semantics. The operations `fhdl` and `fhd2` respectively extract the heads of the data and acknowledgement streams, while `ftl` extracts the pair of tails of the two streams. This much is straightforward, but some other operations are more tricky. The operations `data-ok` and `ack-ok` respectively check whether the data and the acknowledgement stream have succeeded; note that they both use the specially defined `eq` operation. The operations `data-ok-after` and `ack-ok-after` respectively extract the data and acknowledgement streams from the next success onward, and since their conditions use the operations `data-ok` and `ack-ok` respectively, they also indirectly use `eq` in their conditions; this will be important when new constants are introduced during the proof. It is worth noticing that definedness of the operations `data-ok-after` and `ack-ok-after` depends crucially on fairness; for example, if used as rewrite rules with an unfair event mark stream, they would fail to terminate⁵.

4 The streams in this specification are fair to successes, but are not fair to failures, since it is possible that failure never occurs, although success must always occur eventually. Variations of the same approach can be used to capture other kinds of fairness.

5 This could be accomplished within the formalism of this paper by introducing a new “infinite” constant `inf` of sort `Nat`, with the equations `s inf = inf` and `p inf = inf`, where `s` is successor and `p` is predecessor for natural numbers.

The final module, `ABP`, specifies the operation of the protocol, using all the previous modules. The operation `<_,_|_,_>` constructs states of sort `2PrState` for the two processes. The arguments of this constructor represent the states of a data buffer and a one bit register for each process, where the bits are represented by Boolean values. The keyword `dsort` indicates that it has initial semantics for the sort `2PrState`. The operation `[_,_,_]_` takes as its arguments a two process state, an input data stream, and a two channel state, returning the next output data stream state. Its four equations express the operational semantics of the ABP in a straightforward way, although presuming the tricks discussed above. When the two bits in the process state are the same, the source process is ready for an acknowledgement to arrive, and to then start sending another data item; and when they are different, the target process is ready to receive a new data packet and then to start sending acknowledgements. The first equation says that the source process waits when it is ready but the acknowledgement channel fails. The second equation says the target process waits when it is ready but the data channel fails. The third equation says the source process flips its bit when it is ready and receives an acknowledgement. The fourth equation says the target process flips its bit when it is ready and receives a data packet.

The correctness criterion for the ABP is very simply expressed by the equation

$$tl [St, Is, Cs] = Is,$$

where `St` is a system state where the source is ready, `Is` is an input data stream, and `Cs` is a state for the two channels; it just says that the entire input data stream is successfully transmitted (with the initial output ignored); note that this expresses *both* safety and liveness for the ABP. However, we do not prove this form, but instead we prove

$$[<B1, N | B2, M>, Is, Cs] = M \ \& \ Is \ \text{if} \ B1 = B2,$$

a hidden sorted conditional equation, from which the first version follows immediately.

We conclude this section by discussing the form of non-determinism involved in this specification, since we have found that some readers consider it obscure, or even controversial. The reason for this seems to be that each hidden model of the specification is itself deterministic, whereas most other computer science models of non-determinism directly include some form of choice. In the present example, there are an uncountable infinity of different fair event mark stream pairs, each of which determines a different behavior of the alternating bit protocol system. Although each model is deterministic, neither we nor the two processes “know” which events will occur; nevertheless, everything we prove about the specification holds for every one of these models. We could say that in hidden algebra, non-determinism consists of choice among models rather

than choice within models⁶, or we could say that the “real model” of a specification is the class of all hidden algebras that satisfy it. But no matter what view we take, this is a very convenient framework for specification and verification, which covers every possible behavior of the system.

4.2. The Correctness Proof

The proof begins by proving the following five lemmas for the ABP specification:

- (A) `fhd1 data-ok-after Cs = ok`
- (B) `[<B,M | B,N>, Is, Cs] =`
`[<B1,M | B2,N>, Is, ack-ok-after ftl Cs]`
`if not ack-ok Cs`
- (C) `[<B1,M | B2,N>, Is, Cs] =`
`[<B1,M | B2,N>, Is, data-ok-after ftl Cs]`
`if B1 /= B2 and not data-ok Cs`
- (D) `[<B,M | B,N>, Is, Cs] =`
`[<not B, hd Is | B,N>, tl Is,`
`data-ok-after ftl Cs]`
`if ack-ok Cs`
- (E) `tl [<B1,M | B2,N>, Is, Cs] =`
`[<B1,M | B1,M>, Is, ack-ok-after ftl Cs]`
`if B1 /= B2 and data-ok Cs`

The following is the proof score for this verification:

```
bth SETUP is pr ABP .
  vars-of ABP .
  op c : -> 2ChState .
  ops b1 b2 : -> Bool .
end

***> proof of Lemma A
open .
  op f : -> FairStream .
  op n : -> Nat .
  var Fs : FairStream .
  *** base case
  red fhd1 data-ok-after <0 & f, Fs> == ok .
  *** induction step
  eq fhd1 data-ok-after <n & f, Fs> = ok .
  red fhd1 data-ok-after <s n & f, Fs> == ok .
close

***> proof of Lemma B
open . *** base case
  eq fhd2 c = err .
  eq ack-ok-after ftl c = ftl c .
  red [<B,M | B,N>, Is, c] ==
    [<B1,M | B2,N>, Is, ack-ok-after ftl c] .
close

open . *** induction step
  eq fhd2 c = err .
  eq fhd2 ftl c = err .
  red [<B,M | B,N>, Is, ftl c] . *** LHS
  eq [<B,M | B,N>, Is, ftl ftl c] =
    [<B1,M | B2,N>, Is, ack-ok-after ftl ftl c] .
  red [<B,M | B,N>, Is, c] ==
```

```
  [<B,M | B,N>, Is, ack-ok-after ftl c] .
close

***> proof of Lemma C
open . *** base case
  eq fhd1 c = err .
  eq data-ok-after ftl c = ftl c .
  red [<b1,M | b2,N>, Is, c] ==
    [<B1,M | B2,N>, Is, data-ok-after ftl c] .
close

open . *** induction step
  eq fhd1 c = err .
  eq fhd1 ftl c = err .
  red [<B1,M | B2,N>, Is, ftl c] . *** LHS
  eq [<B1,M | B2,N>, Is, ftl ftl c] =
    [<B1,M | B2,N>, Is,
     data-ok-after ftl ftl c]
    if B1 /= B2 .
  red [<B1,M | B2,N>, Is, c] ==
    [<B1,M | B2,N>, Is, data-ok-after ftl c] .
close

set cobasis of STREAM .
set cred trace on .

***> proof of Lemma D
cases CASE-C for SETUP is
  vars B B1 B2 : Bool .
  vars M N P Q : Nat .
  vars Is Ds : DataStream .
  var Cs : 2ChState .
  context [<B,M | B,N>, Is, Cs] .
  case eq fhd1 ftl Cs = ok .
  eq fhd2 Cs = ok .
  case eq fhd1 ftl Cs = err .
  eq fhd2 Cs = ok .
  eq [<B1,P | B2, Q>, Ds, ftl ftl Cs] =
    [<B1,P | B2, Q>, Ds,
     data-ok-after ftl ftl Cs]
    if B1 /= B2 .
  *** red [<B1,P | B2, Q>, Ds, ftl Cs] . *** LHS
end

cred with CASE-C
[<B,M | B,N>, Is, Cs] ==
[<not B, hd Is | B,N>, tl Is,
 data-ok-after ftl Cs] .

***> proof of Lemma E
cases CASE-D for SETUP is
  vars B B1 B2 : Bool .
  vars M N P Q : Nat .
  vars Is Ds : DataStream .
  var Cs : 2ChState .
  context [<B1,M | B2,N>, Is, Cs] .
  case eq fhd2 ftl Cs = ok .
  eq fhd1 Cs = ok .
  case eq fhd2 ftl Cs = err .
  eq fhd1 Cs = ok .
  eq [<B,P | B,Q>, Ds, ftl ftl Cs] =
    [<B,P | B,Q>, Ds,
     ack-ok-after ftl ftl Cs] .
  *** red [<B,P | B,Q>, Ds, ftl Cs] . *** LHS
end

cred with CASE-D tl [<B1,M | B2,N>, Is, Cs] ==
[<B1,M | B1,M>, Is, ack-ok-after ftl Cs]
if B1 /= B2 .
```

⁶ This resembles the worldview of classical physics, where each universe is deterministic, but we don't know which one we are in.

Lemmas A, B and C are proved by induction, whereas Lemmas D and E are proved by case analysis. The proof of Lemma A is a straightforward induction on the natural number in the head of the event mark stream for the data channel. It says that the function `data-ok-after` always delivers an event mark stream indicating an immediate successful data transmission. The equation

```
fhd2 ack-ok-after Cs = ok
```

can also be proved as a lemma, but fortunately it is not needed, because adding it would cause some non-terminating reductions in the proof.

Each of the next four lemmas is a behavioral consequence of a corresponding axiom among the first four in the ABP specification. However, the assumptions that appear in the cases of their proofs are a bit tricky. For Lemma B, first notice that its condition, `not ack-ok Cs`, can only be satisfied if `eq(fhd2 Cs, ok)` is `false`, which according to the initial semantics of the module `MARK`, can only happen if `fhd2 Cs` is `err`. Therefore when setting up the base case for Lemma B, implication elimination allows us to assert `fhd2 c = err` before doing reduction to check the equation. The same reasoning allows us to assert the equation `fhd1 c = err` for the base case of Lemma C.

Lemmas B and C are proved by induction over the number of transmission failures. Since the condition of each equation says there must be at least one failure, the simplest case, which must be the base case, is that there is just one failure, and this implies that the tail of the appropriate channel indicates an immediate success. This and the definitions of the operations `ack-ok-after` and `data-ok-after` justify the second equation asserted for the base cases of Lemmas B and C.

For the induction steps of Lemmas B and C, we know there must be at least one more failure. Therefore tail of the appropriate channel must also indicate failure, which justifies the second assumed equation, the first being just as in the base cases. The third equation assumed in each case is the induction hypothesis. For Lemma B, strictly speaking this should be

```
[<B,M | B,N>, Is, ftl c] =
  [<B,M | B,N>, Is, ack-ok-after ftl ftl c] .
```

However, under the assumptions of this case, the leftside of this equation is not reduced, and in fact, the proof fails if it is attempted using this equation. The way to escape from this dilemma is to replace the leftside with its reduced form, which is

```
[<B,M | B,N>, Is, ftl ftl c] ,
```

and this is what actually appears in the proof score. The reduction before the induction hypothesis justifies this substitution, by calculating the reduced form of the lefthand side. Exactly the same situation arises for the induction step of

Lemma C, and it is handled in the same way, for the same reasons.

Lemma D is proved by case analysis on whether `fhd1 ftl Cs` is `ok`, or is `err`, and the first equation assumed for each case follows from one of these assumptions. Similar reasoning to that used for the base cases of Lemmas B and C, based on consequences of the condition of the lemma, justifies the second equation for each case. The second case of Lemma D also uses a special case of Lemma C, but under the assumptions of this case, its leftside is not reduced; so as before, we use its reduced form instead. (The reduction of the leftside is given as a comment here, since it cannot actually be run inside a case declaration.)

Similarly, Lemma E is proved by case analysis on whether `fhd2 ftl Cs` is `ok`, or is `err`, giving rise to the first equations in its cases, while the second equations follows from assuming its condition. Finally, the second case of Lemma E uses a special case of Lemma D; again, its left side is not reduced, and this is handled as before.

We use coinduction to prove Lemmas D and E, as indicated by the command `cred`, and therefore we need a cobasis. `BOBJ` can compute a default cobasis, but it is not the right one for this problem, because correctness of the ABP depends only on the input and output streams. For this reason, the correct cobasis is that of the module `STREAM`. Finally, we prove ABP correctness using these lemmas:

```
both ABP+ is dsort 2PrState .
pr 2CHAN-STATE + STREAM .
op <_,_> : Bool Nat Bool Nat -> 2PrState .
op [_,_,_] : 2PrState DataStream 2ChState ->
  DataStream .
vars B B1 B2 : Bool . vars M N : Nat .
var Is : DataStream . var Cs : 2ChState .
eq fhd1 data-ok-after Cs = ok . *** A
eq [<B,M | B,N>, Is, Cs] =
  [<B,M | B,N>, Is, ack-ok-after ftl Cs]
  if not ack-ok Cs . *** B
eq [<B1,M | B2,N>, Is, Cs] =
  [<B1,M | B2,N>, Is, data-ok-after ftl Cs]
  if B1 /= B2 and not data-ok Cs . *** C
eq [<B,M | B,N>, Is, Cs] =
  [<not B, hd Is | B,N>, tl Is,
   data-ok-after ftl Cs]
  if ack-ok Cs . *** D
eq tl [<B1,M | B2,N>, Is, Cs] =
  [<B1,M | B1,M>, Is, ack-ok-after ftl Cs]
  if B1 /= B2 and data-ok Cs . *** E
eq hd [<B1,M | B2,N>, Is, Cs] = N
  if B1 /= B2 and data-ok Cs .
end

cases CASE-OF-CHANNEL for ABP+ is
vars B1 B2 : Bool . vars N1 N2 : Nat .
var Is : DataStream . var Cs : 2ChState .
context [<B1,N1 | B2,N2>, Is, Cs] .
case eq fhd2 Cs = ok .
case eq fhd2 Cs = err .
eq fhd2 ack-ok-after ftl Cs = ok .
```

```

end

open .
  vars B1 B2 : Bool . vars N M : Nat .
  var Is : DataStream . var Cs : 2ChState .
  cred with CASE-OF-CHANNEL
    [<B1,N | B2,M>, Is, Cs] == M & Is
    if B1 == B2 .
close

```

The module ABP+ consists of the five lemmas, plus the fifth axiom of ABP. Since all these equations are behavioral consequences of the ABP specification⁷, it is sound to use them in proving the correctness criterion for ABP, and this is just what happens with the final `cred` command, which again uses the cobasis of `STREAM`. The BOBJ output, which can be found at www.cs.ucsd.edu/groups/tatami/bobj/abp.html, shows that this correctness proof uses the full power of C4RW; in particular, circularities are used in two subgoals of the final `cred`, because the word “deduced” appears there instead of the word “reduced.” It is inevitable that some form of case analysis is used in this kind of proof, because different values in fair event streams are handled by the system in quite different ways.

5. Conclusions and Further Research

The ABP proof in this paper provides nice illustrations for many BOBJ features, especially its C4RW algorithm. Some “tricks of the trade” used to make this proof succeed were already familiar long ago from proofs done in earlier versions of OBJ. These include the use of specially defined equality relations to avoid difficulties with negation in the conditions of rules, and replacing non-reduced leftsides of rules with their reduced forms. Other techniques are newer, such as user-defined cobases and case splits.

We have also found it useful to extend the old trick of comparing the reduced forms of two sides of an equational goal to discover new lemmas needed to complete the proof, by using `cred` with a case analysis having just one case. Used together, these provide considerable power for debugging specifications and proofs. However, the evidence for this is not seen in the proof, but rather in our experience in constructing the proof, which involved using these two tricks many times over.

Our favorite tricks are the formalization of event mark streams and fairness using natural number streams. We believe these are new ideas, and it seems clear that they generalize to other variants of fairness, such as a fair interleaving of two kinds of event, or more generally, n kinds of event,

⁷ More technically, we could say that ABP+ is a behavioral refinement of ABP, or equivalently, that ABP behaviorally simulates ABP+. Prof. Lucanu noted that our proof can be interpreted as showing that ABP with fair lossy channels simulates ABP with perfect channels, and that the latter is behaviorally equivalent to perfect transmission.

some of which are subject to errors and some of which are not. A builtin module providing all these options could be a useful addition to BOBJ.

We are now applying the techniques of this paper to other non-deterministic, distributed, concurrent algorithms; for example, we have proved the Petersen critical section algorithm, although the justifications for its proof score are not yet written down. A next step would be to consider real-time algorithms, building on recent work of Rutten, by using streams of pairs of event marks and positive real numbers as timed event mark streams.

It is interesting to consider what further support for proof automation might be added to BOBJ. Although we have found the current version of case analysis very useful, it can certainly be extended. For example, one might implement sums of case expressions, in addition to the products and exceptions which are already implemented. BOBJ’s combination of case analysis with coinduction is already a useful blend of theorem proving with model checking, but it would be interesting to see how much more could be done along these lines. For example, it might be possible to improve the automatic elimination of cases, and it would be especially good if efficiency could be improved when there are many cases; model checking technology might help with this.

One should also consider the higher level question of the right balance between mechanization and human input in theorem proving. When humans are in the loop, the readability of input and output becomes extremely important. The style of “*proof score*” used in this paper is an attempt to find such a balance, originating in the early days of OBJ. Proof scores interleave specification and proof commands with comments intended to clarify structure and intent. On the other hand, our Kumo⁸ system [15, 27] supports completely mechanical proofs, while still trying to make the best use of the respective strong points of humans and machines. However, this ambitious goal requires formalizing every form of inference used in proofs. The proofs of Lemmas B and C illustrate the difficulty that this might pose, since they are inductions over the structure of items with initial semantics inside of coinductive streams, and it seems that any inference rule that could justify such proofs would have to be rather specialized, so that some might consider it better to leave it informal, as in this paper. Alternatively, one might consider mechanical support for showing soundness of new rules, and then automatically adding them to the Kumo rule base for later use in proofs.

References

- [1] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge, 1998.
- [8] Kumo is a Japanese word for “spider,” chosen for this system because it weaves web sites for proofs.

- [2] Narjes Berregeb, Adel Bouhoula, and Michaël Rusinowitch. Observational proofs with critical contexts. In *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 38–53. Springer, 1998.
- [3] Michel Bidoit and Rolf Hennicker. Constructor-based observational logic. Technical Report LSV-03-9, Laboratoire Spécification et Vérification, CNRS de Cachan, March 2003.
- [4] Barry Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.
- [5] Rod Burstall and Joseph Goguen. An informal introduction to specifications using Clear. In Robert Boyer and J Moore, editors, *The Correctness Problem in Computer Science*, pages 185–213. Academic, 1981. Reprinted in *Software Specification Techniques*, Narain Gehani and Andrew McGettrick, editors, Addison-Wesley, 1985, pages 363–390.
- [6] Samuel Buss and Grigore Roşu. Incompleteness of behavioral logics. In Horst Reichel, editor, *Proceedings, Coalgebraic Methods in Computer Science (CMCS'00)*, volume 33 of *Electronic Notes in Theoretical Computer Science*, pages 61–79. Elsevier Science, March 2000.
- [7] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific, 1998. AMAST Series in Computing, Volume 6.
- [8] Răzvan Diaconescu and Kokichi Futatsugi. Behavioural coherence in object-oriented algebraic specification. *Journal of Universal Computer Science*, 6(1):74–96, 2000.
- [9] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, 1985. EATCS Monographs on Theoretical Computer Science, Vol. 6.
- [10] Marie-Claude Gaudel and Igor Privara. Context induction: an exercise. Technical Report 687, LRI, Université de Paris-Sud, 1991.
- [11] Joseph Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison Wesley, 1989.
- [12] Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991. Proceedings of a Conference held at Oxford, June 1989.
- [13] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.
- [14] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Hartmut Ehrig and Fernando Orejas, editors, *Proceedings, Tenth Workshop on Abstract Data Types*, pages 1–29. Springer, 1994. Lecture Notes in Computer Science, Volume 785.
- [15] Joseph Goguen and Kai Lin. Web-based support for cooperative software engineering. *Annals of Software Engineering*, 12:25–32, 2001. Special issue for papers from a conference in Taipei, December 2000.
- [16] Joseph Goguen, Kai Lin, and Grigore Roşu. Circular coinductive rewriting. In *Automated Software Engineering '00*, pages 123–131. IEEE, 2000. Proceedings of a workshop held in Grenoble, France.
- [17] Joseph Goguen, Kai Lin, and Grigore Roşu. Behavioral and coinductive rewriting. In *Proceedings, Rewriting Logic Workshop, 2000*. Elsevier, 2001. Electronic Notes on Theoretical Computer Science, Volume 36, at www.elsevier.nl/locate/entcs/volume36.html.
- [18] Joseph Goguen and Grant Malcolm. Hidden coinduction: Behavioral correctness proofs for objects. *Mathematical Structures in Computer Science*, 9(3):287–319, June 1999.
- [19] Joseph Goguen and Grant Malcolm. A hidden agenda. *Theoretical Computer Science*, 245(1):55–101, August 2000. Also UCSD Dept. Computer Science & Eng. Technical Report CS97-538, May 1997.
- [20] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992. Drafts exist from as early as 1985.
- [21] Joseph Goguen and Grigore Roşu. Hiding more of hidden algebra. In Jeannette Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 – Formal Methods*, pages 1704–1719. Springer, 1999. Lecture Notes in Computer Sciences, Volume 1709, Proceedings of World Congress on Formal Methods, Toulouse, France.
- [22] Joseph Goguen and Grigore Roşu. A protocol for distributed cooperative work. In Gheorghe Stefanescu, editor, *Proceedings, FCT'99, Workshop on Distributed Systems (Iasi, Romania)*, volume 28, pages 1–22. Elsevier, 1999. Electronic Lecture Notes in Theoretical Computer Science.
- [23] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000.
- [24] Rolf Hennicker. Context induction: a proof principle for behavioural abstractions. In A. Miola, editor, *Proceedings, International Symposium on the Design and Implementation of Symbolic Computation Systems*, volume 429 of *Lecture Notes in Computer Science*, pages 101–110. Springer, 1990.
- [25] Rolf Hennicker and Michel Bidoit. Observational logic. In *Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 1999.
- [26] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, June 1997.
- [27] Kai Lin. *Machine Support for Behavioral Algebraic Specification and Verification*. PhD thesis, University of California at San Diego, 2003.
- [28] José Meseguer and Joseph Goguen. Initiality, induction and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985.

- [29] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980. Lecture Notes in Computer Science, Volume 92.
- [30] David M.R. Park. *Concurrency and Automata on Infinite Sequences*. Springer, 1980. Lecture Notes in Computer Science, Volume 104.
- [31] Horst Reichel. Behavioural equivalence – a unifying concept for initial and final specifications. In *Proceedings, Third Hungarian Computer Science Conference*. Akademiai Kiado, 1981. Budapest.
- [32] Grigore Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.
- [33] Grigore Roşu and Joseph Goguen. Hidden congruent deduction. In Ricardo Caferra and Gernot Salzer, editors, *Proceedings, 1998 Workshop on First Order Theorem Proving*, pages 213–223. Technische Universität Wien, 1998. (Schloss Wilhelminenberg, Vienna, November 23-25, 1998).
- [34] Grigore Roşu and Joseph Goguen. Circular coinduction. In *Proceedings, Int. Joint Conf. Automated Deduction*. Springer, 2000. Sienna, June 2001.