COMP*

Language Definition
Syntax, Semantics, and Methodologies

Răzvan Diaconescu and Ionuț Țuțu

Simion Stoilow Institute of Mathematics
of the Romanian Academy, Romania
razvan.diaconescu@imar.ro, ionut.tutu@imar.ro

Summary

The structure of this document is as follows:

1. A brief presentation of the mathematical theory that underlies the COMP system. The full mathematical foundations of COMP can be studied in [2]; here, we limit the mathematics to what is needed to understand the definition of the language and the basic specification methodologies of COMP.

2. The syntax of COMP, which provides the grammar for building and parsing COMP specifications.

3. The semantics of COMP, which provides the mathematical meaning for specifications written in COMP and constitutes the basis of the COMP interpreter.

4. Specification and verification methodologies for COMP.

1 Mathematical foundations

COMP has a model-theoretic semantics based on *hidden algebra* (hereafter abbreviated HA), which is a refined form of *many-sorted algebra* (MSA). In the following, we will

*This work was supported by a grant of the Romanian Ministry of Education and Research, CCCDI – UEFISCDI, project number PN-III-P2-2.1-PED-2019-0955, within PNCDI III.

present these logical systems very briefly, the only purpose being to support the understanding of the definition of COMP. For a deeper understanding, the reader should look into the rather rich literature on the subject (e.g., [11, 12]).

1.1 Many-sorted algebra

This is the traditional framework for algebraic specification and constitutes the core framework for all algebraic-specification formalisms. In the following, we introduce the main concepts of many-sorted algebra needed for understanding COMP.

Definition 1.1 (Many-sorted signatures)**.** *We let S^* denote the set of all finite sequences of elements from a set S, with ε indicating the empty sequence. A(n S-sorted)* signature *(S, F) is an $(S^* \times S)$-indexed set $F = \{F_{w\to s} \mid w \in S^*, s \in S\}$ of sets of* operation symbols*, indexed by* arities *$w \in S^*$ and* sorts *$s \in S$.*

Note that this definition admits *overloading*, meaning that the sets $F_{w\to s}$, for $w \in S^*$ and $s \in S$, need *not* be disjoint. We call $\sigma \in F_{\varepsilon\to s}$ a *constant symbol* of sort s.

Definition 1.2 (Terms)**.** *An (S, F)-*term *t of sort $s \in S$ is a syntactic structure of the form $\sigma(t_1, \ldots, t_n)$, where $\sigma \in F_{w\to s}$ is an operation symbol of arity $w = s_1 \ldots s_n \in S^*$ ant sort $s \in S$, and $t_1, \ldots, t_n$ are (S, F)-terms of sorts $s_1, \ldots s_n$.*

Definition 1.3 (Many-sorted sets)**.** *Given a* sort set *S, an S-*indexed *(or* sorted*)* set *A is a family $\{A_s\}_{s\in S}$ of sets indexed by the elements of S; in this context, we typically write $a \in A$ to indicate that $a \in A_s$ for some sort $s \in S$.*

Given an S-indexed set A and $w = s_1 \cdots s_n \in S^*$, we let A_w denote the Cartesian product $A_{s_1} \times \cdots \times A_{s_n}$; in particular, we let $A_\varepsilon = \{\star\}$, some one-point set.

Definition 1.4 (Many-sorted algebras)**.** *An (S, F)-*algebra *A consists of*

- *an S-indexed set A (where A_s is called the* carrier *of A of sort s), and*
- *a function $A_\sigma \colon A_w \to A_s$ for each operation symbol $\sigma \in F_{w\to s}$.*

Note that, if $\sigma \in F_{\varepsilon\to s}$ is a constant symbol, A_σ is an element in A_s, i.e. $A_\sigma \in A_s$.

Definition 1.5 (Term evaluation)**.** *Any (S, F)-*term *$t = \sigma(t_1, \ldots, t_n)$, where $\sigma \in F_{w\to s}$ is an operation symbol and $t_1, \ldots, t_n$ are (S, F)-(sub)terms corresponding to the arity w, gets* interpreted as an element *$A_t \in A_s$ in an (S, F)-algebra A by $A_t = A_\sigma(A_{t_1}, \ldots, A_{t_n})$.*

Definition 1.6 (Congruences)**.** *An (S, F)-*congruence *on an (S, F)-algebra A is an S-sorted family of equivalence relations $\equiv = \{\equiv_s \subseteq A_s \times A_s\}$ that also satisfies the following* congruence property*: given any $\sigma \in F_{w\to s}$ and any $a \in A_w$, we have $A_\sigma(a) \equiv_s A_\sigma(a')$ whenever $a \equiv_w a'$.*[1]

[1] Meaning $a_i \equiv_{s_i} a'_i$ for all $1 \le i \le n$, where $w = s_1 \cdots s_n$ and $a = (a_1, \ldots, a_n)$.

Definition 1.7 (Quantifier-free equations)**.** *A* quantifier-free (S, F)-equation *is an equality $t = t'$ between (S, F)-terms t and t' of the same sort.*

Definition 1.8 (Quantifier-free sentences)**.** *The set of* quantifier-free (S, F)-sentences *is the least set that*

- *contains all (S, F)-equations,*
- *is closed under the usual binary connectives $\wedge$ (which stands for* conjunction*), $\vee$ (*disjunction*), $\Rightarrow$ (*implication*) and under the unary connective $\neg$ (*negation*).*

A quantifier-free conditional (S, F)-equation *is an implication sentence of the form $e_1 \wedge \cdots \wedge e_n \Rightarrow e$, where $e, e_1, ..., e_n$ are all quantifier-free equations.*

Definition 1.9 (Universal sentences)**.** *A set X of variables for a signature (S, F) is a set of new constants. Then $(S, F \cup X)$ denotes the signature obtained from (S, F) by adjoining X to F. A* universal(ly quantified) (S, F)-sentence *is a sentence of the form $\forall X \cdot \rho$, where ρ is a quantifier-free $(S, F \cup X)$-sentence.*

The *satisfaction relation* between algebras and sentences is the usual Tarskian satisfaction defined inductively on the structure of sentences.

Definition 1.10 (Satisfaction)**.** *Given an arbitrary but fixed signature (S, F) and an (S, F)-algebra A, we define:*

- *$A \models t = t'$ if $A_t = A_{t'}$ for quantifier-free equations;*
- *$A \models \rho_1 \wedge \rho_2$ if $A \models \rho_1$ and $A \models \rho_2$;*
- *$A \models \rho_1 \vee \rho_2$ if $A \models \rho_1$ or $A \models \rho_2$;*
- *$A \models \rho_1 \Rightarrow \rho_2$ if $A \models \rho_2$ when $A \models \rho_1$;*
- *$A \models \neg\rho$ if $A \not\models \rho$;*
- *$A \models \forall X \cdot \rho$ if $A' \models \rho$ for all $(S, F \cup X)$-algebras A' such that $A'_\varsigma = A_\varsigma$ for each sort or operation symbol ς in (S, F).*

1.2 Hidden algebra

This is the mathematical framework that underlies the so-called 'behavioural specification' paradigm [9, 10, 6, 7, 3, 8, 11], which is a generalisation of ordinary (many-sorted) algebraic specification. Behavioural specification characterises how objects (and systems) *behave*, not how they are implemented. This new form of abstraction can be very powerful in the specification and verification of software systems since it naturally embeds other useful paradigms such as concurrency, object-orientation, constraints, nondeterminism, etc. (see [7] for details). Behavioural abstraction is

achieved by using specification with hidden sorts and a behavioural concept of satisfaction based on the idea of indistinguishability of states that are observationally the same. Our brief presentation of the main concepts of hidden algebra given below follows the so-called 'coherent-hidden-algebra approach' [3, 4].

Definition 1.11 (Hidden algebra signatures)**.** *A* hidden-algebra signature *is a tuple* (H, V, F, BF) *that consists of:*

- *disjoint sets* H *of* hidden sorts *and* V *of (ordinary)* visible sorts*;*
- *an indexed family* F *of* $(H \cup V)$*-sorted operation symbols such that* $(H \cup V, F)$ *is an ordinary many-sorted signature; and*
- *a distinguished subset* $BF_{w \to s} \subseteq F_{w \to s}$ *of* behavioural operations *for each arity* $w \in (H \cup V)^*$ *and sort* $s \in H \cup V$ *such that* w *contains at least one hidden sort.*

Definition 1.12 (Hidden congruence)**.** *Given an* HA*-signature* (H, V, F, BF) *and an* $(H \cup V, F)$*-algebra* A*, a* hidden (H, V, F, BF)-congruence $\sim$ *on* A *is just an* $(H \cup V, BF)$*-congruence whose components on visible sorts are all identities.*

Definition 1.13 (Behavioural equivalence)**.** *The largest hidden* (H, V, F, BF)*-congruence* $\sim_A$ *on an* $(H \cup V, F)$*-algebra* A *– which is guaranteed to exist by a crucial result found, e.g., in [11] – is called the* behavioural equivalence on A*.*

Definition 1.14 (Hidden algebras)**.** *Given an* HA*-signature* (H, V, F, BF)*, an* (H, V, F, BF)-algebra *is a many-sorted* $(H \cup V, F)$*-algebra* A *such that each operation (interpretation of a symbol in* F*) preserves the behavioural equivalence relation* $\sim_A$*.*

Note that $\sim_A$ is always automatically preserved by behavioural operations and by data operations (i.e., those operations that are free of hidden sorts). Hence only the interpretation of other operation symbols in F may narrow that class of many-sorted $(H \cup V, F)$-algebras to the class of hidden (H, V, F, BF)-algebras.

Definition 1.15 (Behavioural sentences)**.** *Given a hidden algebraic signature* (H, V, F, BF)*, a* quantifier-free behavioural (H, V, F, BF)-equation $t \sim t'$ *consists of a pair of* $(H \cup V, F)$*-terms of the same sort. Universal* (H, V, F, BF)*-sentences are defined in a similar manner to Definition 1.7 by considering both strict equations* $t = t'$ *and behavioural equations* $t \sim t'$ *as atoms.*

Definition 1.16 (Behavioural satisfaction)**.** *An* (H, V, F, BF)*-algebra* A *satisfies a behavioural equation* $t \sim t'$*, which we denote by* $A \models t \sim t'$*, when* $A_t \sim_A A_{t'}$*. This is extended to universal* (H, V, F, BF)*-sentences similarly to Definition 1.10.*

2 Syntax

We define the syntax of COMP using the following extended BNF notation:

$\langle a \rangle$	non-terminals are written in italics and surrounded by angle brackets;
`a`	terminals (literal text) are written in a typewriter font;
(a)	"(" and ")" are metaparentheses used to define a as syntactical *unit*;
$[a]$	"[" and "]" indicate optional syntax, i.e. "", the empty string, or "a";
$a \mid b$	"|" is used to separate alternatives, i.e. either a or b;
a^*	"*" indicates zero or more repetitions op the preceding unit;
$a+$	"+" indicates one or more repetitions op the preceding unit.

A COMP specification consists of a sequence of interrelated modules, which are either data-type modules or behavioural-object modules.

⟨*specification*⟩ ::= (⟨*data-module*⟩ | ⟨*bobj-module*⟩)*

2.1 Data types

Data-type modules (⟨*data-module*⟩) are specified using a syntax similar to that of Maude's functional modules, except that the Maude keywords `fmod … endfm` are replaced with `data … enddata`. The non-terminal ⟨*module-name*⟩ is special; it matches *simple identifiers*, which are sequences of ASCII characters that do not contain (unescaped) spaces, commas, or parentheses.

⟨*data-module*⟩ ::= `data` ⟨*module-name*⟩ `is`
[(⟨*data-declaration*⟩ `.`)*]
`enddata`

A data declaration is either an import or an MSA declaration.

⟨*data-declaration*⟩ ::= ⟨*data-import*⟩ | ⟨*msa-declaration*⟩

COMP admits three importation modes, just like Maude: `protecting`, the most restrictive, which abides by the *no junk* and *no confusion* policy; `extending`, which allows *junk*, but not *confusion*; and `including`, the most permissive, which allows both *junk* and *confusion* – for a detailed discussion, see [1] and also [5].

⟨*data-import*⟩ ::= `protecting` ⟨*module-name*⟩
| `extending` ⟨*module-name*⟩
| `including` ⟨*module-name*⟩

Base MSA declarations are of either sorts, operations, variables, or sentences (which are implicitly universally quantified, as per Definition 1.9). Binary operations may be declared with equational attributes such as associativity (`assoc`), commutativity (`comm`), and identity (`id: e`, with `e` denoting the identity element).

Similarly to ⟨*module-name*⟩, the non-terminals ⟨*sort*⟩, ⟨*symbol*⟩, ⟨*term*⟩, and ⟨*sentence*⟩ are special. The first two match simple identifiers, while the latter correspond to sequences of identifiers written according to the grammars for terms and sentences of the MSA-signature under consideration.

⟨*msa-declaration*⟩ ::= `sort` ⟨*sort*⟩
| `sorts` ⟨*cs-sort-list*⟩
| `op` ⟨*symbol*⟩ `:` ⟨*arity*⟩ `->` ⟨*sort*⟩ [`[` ⟨*attribute*⟩+ `]`]
| `ops` ⟨*cs-symbol-list*⟩ `:` ⟨*arity*⟩ `->` ⟨*sort*⟩ [`[` ⟨*attribute*⟩+ `]`]
| `var` ⟨*symbol*⟩ `:` ⟨*sort*⟩
| `vars` ⟨*cs-symb-list*⟩ `:` ⟨*sort*⟩
| `ax` ⟨*sentence*⟩

⟨*cs-sort-list*⟩ ::= ⟨*sort*⟩ (`,` ⟨*sort*⟩)*

⟨*cs-symbol-list*⟩ ::= ⟨*symbol*⟩ (`,` ⟨*symbol*⟩)*

⟨*arity*⟩ ::= `()` | ⟨*sort*⟩+

⟨*attribute*⟩ ::= `assoc` | `comm` | `id: (` ⟨*term*⟩ `)`

As an example, the following is a simple specification of natural numbers with addition. It consists of two data-type modules: `NAT`, defining natural numbers, and `NAT/ADDITION`, introducing addition, which is defined in the usual inductive manner.

```
data NAT is
  sort Nat .
  op 0 : () -> Nat .
  op s_ : Nat -> Nat .
enddata

data NAT/ADDITION is
  protecting NAT .
  op _+_ : Nat Nat -> Nat [assoc comm] .
  vars M, N : Nat .
  ax 0 + N = N .
  ax (s M) + N = s (M + N) .
enddata
```

2.2 Objects

Object declarations are characteristic to COMP. Their syntax is as follows:

⟨*bobj-module*⟩ ::= `bobj` ⟨*bobj-name*⟩ [`with states` ⟨*sort*⟩] `is`
[⟨*composition*⟩ `.`]
[(⟨*bobj-declaration*⟩ `.`)*]
`endbo`

The non-terminal ⟨*bobj-name*⟩ is special; it matches simple identifiers, just like ⟨*module-name*⟩ – as it is used in the syntax of data-type modules – and ⟨*sort*⟩.

Every object has a designated state sort, which from a hidden-algebra perspective is regarded as a hidden sort. That sort can be declared explicitly, using the optional syntax "`with states` ⟨*sort*⟩", or it can be left implicit, by omitting the optional "`with states …`" syntax, in which case it is declared as `State`.

Objects that are not composed – i.e., for which "⟨*composition*⟩ ." is opted out – are called *base objects*. The consist solely of a list of behavioural-object declarations, which are either data-type imports or HA declarations.

⟨*bobj-declaration*⟩ ::= ⟨*data-import*⟩ | ⟨*ha-declaration*⟩

Hidden-algebra declarations extend MSA declarations with *actions* and *observations*, both of which are particular kinds of behavioural operations.

⟨*ha-declaration*⟩ ::= ⟨*msa-declaration*⟩
| `act` ⟨*symbol*⟩ `:` ⟨*arity*⟩ `->` ⟨*sort*⟩
| `obs` ⟨*symbol*⟩ `:` ⟨*arity*⟩ `->` ⟨*sort*⟩

Actions and observations are *monadic* behavioural operations, meaning that their arities contain a single hidden (state) sort. Moreover, for each action $a\colon w \to s$, s is the unique hidden sort in w; and for each observation $o\colon w \to s$, s is a visible sort.

The following is an example of a base object. It captures a bank-account with one observation, `balance` (the amount of savings held in the account), and two actions: `deposit` (which increases the balance), and `withdraw` (which decreases the balance).

```
bobj ACCOUNT with states Account is
  protecting NAT/OPS .
  act deposit : Account Nat -> Account .
  act withdraw : Account Nat -> Account .
  obs balance : Account -> Nat .
  ...
endbo
```

The effect of the actions `deposit` and `withdraw` on the observation `balance` is specified (in place of the ellipsis in the previous listing) through the following conditional equations over variables `A : Account` and `N : Nat`.

```
ax balance(deposit(A, N)) = balance(A) + N .
ax balance(withdraw(A, N)) = balance(A) - N if N <= balance(A) = true .
ax balance(withdraw(A, N)) = balance(A) if N <= balance(A) = false .
```

Composed objects are obtained through three types of compositions: parallel, synchronized, and indexed. While the former type is a degenerated case of the second, the latter is a refined case of the second.

⟨*composition*⟩ ::= ⟨*parallel-comp*⟩ | ⟨*sync-comp*⟩ | ⟨*indexed-comp*⟩

2.3 Parallel compositions

This is the simplest form of composition. Its syntax is:

⟨*parallel-comp*⟩ ::= `composing` ⟨*bobj-ref-list*⟩

⟨*bobj-ref-list*⟩ ::= ⟨*bobj-ref*⟩ (`and` ⟨*bobj-ref*⟩)+

⟨*bobj-ref*⟩ ::= ⟨*bobj-name*⟩ | `(` ⟨*bobj-name*⟩ `as` ⟨*bobj-name*⟩ `)`

That is, when composing objects, we can reference to them either by their original names or by new names introduced using the `(… as …)` renaming syntax. The final names of the objects involved in a composition need to be distinct.

As an example, the following specification describes an bank-account system with two kinds of accounts, called `A` and `B`, which operate independently, in parallel. The system is obtained through the parallel composition of two copies of `ACCOUNT`:

```
bobj ACCOUNT-SYS with states AccountSys is
  composing (ACCOUNT as A) and (ACCOUNT as B) .
endbo
```

2.4 Synchronized compositions

The syntax for synchronized compositions is as follows:

⟨*sync-comp*⟩ ::= `syncing` (⟨*bobj-name*⟩ | ⟨*bobj-ref-list*⟩)

Here we have two options:

- add synchronization to an existing object, using the syntax `syncing` ⟨*bobj-name*⟩;
- create a new (synchronized) compound object, through `syncing` ⟨*bobj-ref-list*⟩.

As an example, we may define a system of two bank accounts with a transfer operation between them by adding synchronization to `ACCOUNT-SYS`:

```
bobj ACCOUNT-SYS-TRANSFER is
  syncing ACCOUNT-SYS .
  var AS : AccountSys . var N : Nat .

  act transfer : AccountSys Nat -> AccountSys .
  ax A/Account(transfer(AS, N)) = withdraw(A/Account(AS), N) .
  ax B/Account(transfer(AS, N)) = deposit(B/Account(AS), N)
     if N <= A/balance(AS) = true .
  ax B/Account(transfer(AS, N)) = B/Account(AS)
     if N <= A/balance(AS) = false .
endbo
```

Alternatively, we may compose the two accounts directly with synchronization:

```
bobj ACCOUNT-SYS-ALT-TRANSFER with states AccountSys is
  syncing (ACCOUNT as A) and (ACCOUNT as B) .
  var AS : AccountSys . var N : Nat .

  act transfer : AccountSys Nat -> AccountSys .
  ax A/Account(transfer(AS, N)) = withdraw(A/Account(AS), N) .
  ax B/Account(transfer(AS, N)) = deposit(B/Account(AS), N)
     if N <= A/balance(AS) = true .
  ax B/Account(transfer(AS, N)) = B/Account(AS)
     if N <= A/balance(AS) = false .
endbo
```

For now, in both of the above examples, we do not pay attention to the code below the third line of the specification; that part is subject to a dedicated explanation in Section 3 of this document.

2.5 Indexed compositions

These are the most complex kinds of object compositions. Their syntax is as follows:

⟨*indexed-comp*⟩ ::= `indexing` ⟨*bobj-name*⟩ `on` ⟨*module-name*⟩ `by` ⟨*sort*⟩

The idea behind this syntax is that components are given as copies of ⟨*bobj-name*⟩ that are indexed by elements of (visible) sort ⟨*sort*⟩, which is declared in the indexing module ⟨*module-name*⟩. We distinguish two types of situations:

- ⟨*module-name*⟩ references a data-type module, in which case we say that the composition is *static*;
- ⟨*module-name*⟩ references a behavioural object, hence the presence of components is managed by an object, in which case the composition is *dynamic*.

The following is an example of the dynamic situation (we include for now only the part that is relevant for the `indexing … on … by …` syntax).

```
bobj ACCOUNT-SYS-DYN with states AccountSys is
  indexing ACCOUNT on USER-DB by UId .
  ...
endbo
```

In the listing above, the indexing specification `USER-DB` specifies the manager object. It is based on a predefined data-type specification `UID` of user identifiers.

```
bobj USER-DB with states UserDB is
  protecting UID .
  obs _in_ : UId UserDB -> Bool .
  act add : UId UserDB -> UserDB .
  act delete : UId UserDB -> UserDB .
  ...
endbo
```

3 Semantics

3.1 Behavioural objects

The following concept applies both to many-sorted algebra and to hidden algebra.

Definition 3.1 (Specification)**.** *A* specification *is a pair* (Σ, E)*, where* Σ *is a signature and* E *is a finite set of* Σ*-axioms/sentences.*

To each COMP-specification module we associate either an MSA-specification, in the case of ⟨*data-module*(*s*)⟩, or a particular type of HA-specification, called behavioural object, in the sense of the definition below, in the case of ⟨*bobj-module*(*s*)⟩. This constitutes the semantics of COMP specifications.

Definition 3.2 (Behavioural object)**.** *A* behavioural object B *is a pair* (SP_B, h_B) *consisting of a behavioural (HA) specification* SP_B *whose behavioural operations are all monadic*[2] *and a distinguished hidden sort* h_B *of* SP_B*.*

[2]Recall that a behavioural operation is monadic when its arity contains exactly one hidden sort.

Definition 3.3 (Behavioural-object algebra)**.** *Let $B = (SP_B, h_B)$ be a behavioural object. Then a* B-algebra *is just an ordinary algebra of the specification SP_B.*

In the remaining part of this section we show how the semantics of COMP specifications is defined by recursion on the hierarchical structure of their components.

3.2 The semantics of component-free specifications

For base COMP-specification modules (i.e., for modules without components), the corresponding behavioural object is defined as follows:

- $H_B = \{h_B\}$, where h_B is the `states` sort.
- V_B consists of all data-type sorts.
- F_B consists of all operations declared in the module.
- BF_B consists of all actions and observations declared in the module.

These constitute the hidden-algebra signature and designated hidden sort of the module. With regard to the axioms of the module, we let:

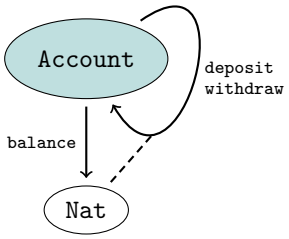
- E_B be the set of all sentences (universally quantified, potentially conditional) of the COMP module; this includes all sentences of imported data types.

The bank-account object

As an example, for the `ACCOUNT` object specification defined in Section 2.2, we have:

- $H_{\texttt{ACCOUNT}} = \{h_{\texttt{ACCOUNT}}\} = \{\texttt{Account}\}$;
- $V_{\texttt{ACCOUNT}}$ consists of all sorts in the predefined module `NAT` (`Nat` is such a sort);
- $F_{\texttt{ACCOUNT}}$ consists of all data operations declared in `NAT`, including addition (`_+_`) and subtraction (`_-_`), plus the two specified actions (`deposit` and `withdraw`) and one observation (`balance`);
- $BF_{\texttt{ACCOUNT}}$ consists precisely of those two actions and of the observation `balance`; for instance, $(BF_{\texttt{ACCOUNT}})_{\texttt{AccountNat}\to\texttt{Account}} = \{\texttt{deposit}, \texttt{withdraw}\}$.

A partial representation of the signature of `ACCOUNT`, including all actions and observations and which uses an ADJ diagram is as follows:



The set $E_{\texttt{ACCOUNT}}$ of axioms consists of all sentences declared in `NAT` plus the three conditional equations from the body of `ACCOUNT`. In standard mathematical notation, there are as follows:

- $\forall \{A, N\} \cdot \texttt{balance}(\texttt{deposit}(A, N)) = \texttt{balance}(A) + N$;
- $\forall \{A, N\} \cdot N \leq \texttt{balance}(A) \Rightarrow \texttt{balance}(\texttt{withdraw}(A, N)) = \texttt{balance}(A) - N$;
- $\forall \{A, N\} \cdot N \not\leq \texttt{balance}(A) \Rightarrow \texttt{balance}(\texttt{withdraw}(A, N)) = \texttt{balance}(A)$.

3.3 The semantics of parallel compositions

The semantics of an object `B` obtained through a simple parallel composition of two component objects `B1` and `B2`, and specified in COMP as follows:

```
bobj B with states St is
  composing B1 and B2 .
  ...
endbo
```

is given by:

- $H_{\texttt{B}} = H_{\texttt{B1}} \uplus H_{\texttt{B1}} \uplus \{\texttt{St}\}$; i.e., the *disjoint* union of $H_{\texttt{B1}}$ and $H_{\texttt{B2}}$, plus `St`.
- $h_{\texttt{B}} = \texttt{St}$.
- $V_{\texttt{B}} = V_{\texttt{B1}} \cup V_{\texttt{B2}}$, which allows data sharing between `B1` and `B2`.
- $F_{\texttt{B}}$ gathers together the operations in $F_{\texttt{B1}}$ and $F_{\texttt{B2}}$, and adds a set of new behavioural operations, as follows:
  - for each component `Bi`, a 'projection' $\pi_i \colon h_{\texttt{B}} \to h_{\texttt{Bi}}$;
  - for each action $\sigma \in (BF_{\texttt{Bi}})_{h_{\texttt{Bi}} w \to h_{\texttt{Bi}}}$, a `B`-action $\sigma_i \colon h_{\texttt{B}} w \to h_{\texttt{B}}$;[3]
  - for each observation $\zeta \in (BF_{\texttt{Bi}})_{h_{\texttt{Bi}} w \to s}$, a `B`-observation $\zeta_i \colon h_{\texttt{B}} w \to s$.
- $BF_{\texttt{B}}$ extends $BF_{\texttt{B1}}$ and $BF_{\texttt{B2}}$ with the projections (one for each component), `B`-actions, and `B`-observations described above.
- $E_{\texttt{B}}$ adds the following quantified equations to the axioms of `B1` and `B2`:
  - $\forall \{X, W\} \cdot \pi_i(\sigma_i(X, W)) = \sigma(\pi_i(X), W)$, for $i \in \{1, 2\}$;
  - $\forall \{X, W\} \cdot \pi_j(\sigma_i(X, W)) = \pi_j(X)$, for $\{i, j\} = \{1, 2\}$;
  - $\forall \{X, W\} \cdot \zeta_i(X, W) = \zeta(\pi_i(X), W)$, for $i \in \{1, 2\}$.

[3] To simplify the notation, we write the arities of actions and observations with the hidden sort in the head position, as in $h_{\texttt{B}} w$, regardless of its actual position in the arity.

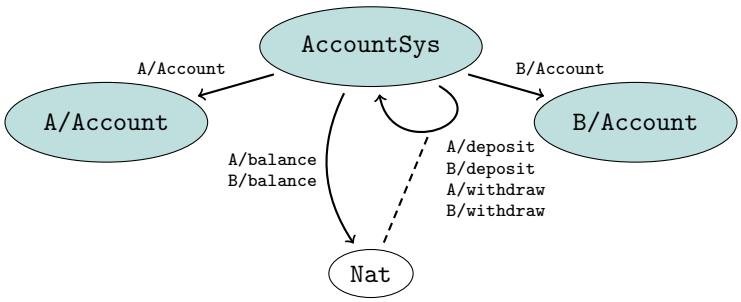
This means that the actions originating from one component do not change the state of the other component, hence each component can operate independently. This axiomatization supports the terminology *parallel composition*.

When writing COMP specifications in ASCII:

- the projections π_i are encoded as `Bi/`$h_{\texttt{Bi}}$; for example, `B/Account`;
- the actions σ_i are encoded as `Bi/`σ; for example, `B/deposit`;
- the observations ζ_i are encoded as `Bi/`ζ; for example, `B/balance`.

The parallel composition of two account objects

The `ACCOUNT-SYS` object defined in Section 2.3 as `composing (ACCOUNT as A) and (ACCOUNT as B)` has four actions (two for each component) and two observations (one for each component) declared at the compound level – in addition to the projections. Its signature can be visualised in the following ADJ diagram:



All operations are generated automatically by the COMP compiler. Moreover, the compiler relates (via projections) the actions at the level of the compound object to the actions of the component objects through the following sets of universally quantified equations (over variables `AS : AccountSys` and `N : Nat`):

```
ax A/Account(A/deposit(AS, N)) = deposit(A/Account(AS), N) .
ax B/Account(B/deposit(AS, N)) = deposit(B/Account(AS), N) .
ax A/Account(A/withdraw(AS, N)) = withdraw(A/Account(AS), N) .
ax B/Account(B/withdraw(AS, N)) = withdraw(B/Account(AS), N) .

ax A/Account(B/deposit(AS, N)) = A/Account(AS) .
ax B/Account(A/deposit(AS, N)) = B/Account(AS) .
ax A/Account(B/withdraw(AS, N)) = A/Account(AS) .
ax B/Account(A/withdraw(AS, N)) = B/Account(AS) .
```

The observations on the compound object are just abbreviations of the observations of the components. These are also build automatically by the COMP compiler:

```
ax A/balance(AS) = balance(A/Account(AS)) .
ax B/balance(AS) = balance(B/Account(AS)) .
```

3.4 The semantics of synchronisation

Composition with synchronisation refines parallel composition by bringing in synchronicity between components. The composition syntax `syncing …` signals that we are going to add synchronisation to an existing compound object. When that compound object is a parallel composition we may perform the two steps in only one step by using the second variant of `syncing`.

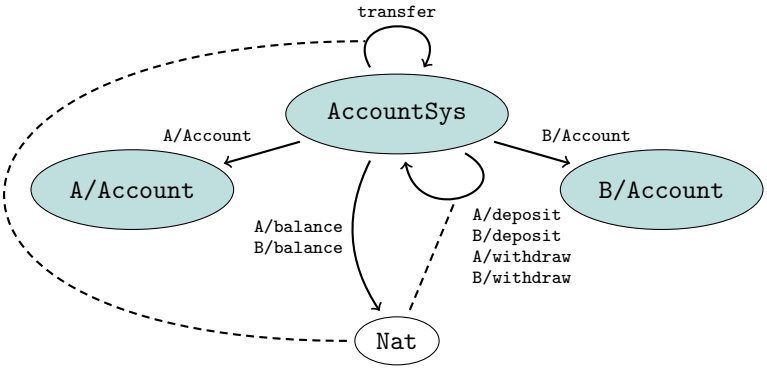
Synchronising a compound object `B` is achieved by adding new actions at the topmost level of `B`. Let `B'` denote the newly obtained synchronised object. Then $H_{\texttt{B'}} = H_{\texttt{B}}$, $h_{\texttt{B'}} = h_{\texttt{B}}$, and the hidden algebra specification of `B'` just extends the specification of `B` with data sorts, the operations, and the axioms declared in the body of the COMP specification of `B'`.

Synchronising the parallel composition of two accounts

As an example, we add a `transfer` action to the bank account system `ACCOUNT-SYS`, which we have previously defined as parallel composition of two accounts. `Transfer` is specified as an action at the level of the compound object and models a transfer of financial resources from the first account to the second one. The COMP specification of this object has been introduced in Section 2.4. Let us recall it:

```
bobj ACCOUNT-SYS-TRANSFER is
  syncing ACCOUNT-SYS .
  var AS : AccountSys . var N : Nat .

  act transfer : AccountSys Nat -> AccountSys .
  ax A/Account(transfer(AS, N)) = withdraw(A/Account(AS), N) .
  ax B/Account(transfer(AS, N)) = deposit(B/Account(AS), N)
     if N <= A/balance(AS) = true .
  ax B/Account(transfer(AS, N)) = B/Account(AS)
     if N <= A/balance(AS) = false .
endbo
```

The ADJ-diagram representation of its signature is as follows:



Although simple, this example showcases two important synchronisation situations:

broadcasting appears because `transfer` changes the states of both components, and

client-server computing appears because `transfer` is related to a deposit of the account of type `A` by using information of an account of type `B`.

3.5 The semantics of indexed compositions

Static indexing

Suppose we would like to extend the parallel composition of two bank accounts (`ACCOUNT-SYS`) to an undefined number of bank accounts. More generally, this means a parallel composition of multiple copies of one object `B`, *any* number of copies. In COMP this can be achieved by writing:

```
bobj B' with states St is
  indexing B on DATA by Idx .
  ...
endbo
```

where `Idx` is a sort in the data type `DATA`. Then

- $H_{\mathtt{B'}} = H_{\mathtt{B}} \uplus \{\mathtt{St}\}$.
- $h_{\mathtt{B'}} = \mathtt{St}$.
- $V_{\mathtt{B'}} = V_{\mathtt{B}} \cup V_{\mathtt{DATA}}$ (where $V_{\mathtt{DATA}}$ is the set of sorts of the specification `DATA`).
- $F_{\mathtt{B'}}$ adds the operations of `DATA` to $F_{\mathtt{B}}$, together with:
  - a parameterized behavioural 'projection' $\pi\colon \mathtt{Idx}\, h_{\mathtt{B'}} \to h_{\mathtt{B}}$;
  - for each action $\sigma \in (BF_{\mathtt{B}})_{h_{\mathtt{B}} w \to h_{\mathtt{B}}}$, a `B'`-action $\sigma'\colon \mathtt{Idx}\, h_{\mathtt{B'}}, w \to h_{\mathtt{B'}}$;

– for each observation $\zeta \in (BF_{\mathtt{B}})_{h_{\mathtt{B}} w \to s}$, a `B'`-observation $\zeta' : \mathtt{Idx}\, h_{\mathtt{B}'} w \to s$.

- $BF_{\mathtt{B}'}$ adds the new behavioural operations of $F_{\mathtt{B}'}$ to $BF_{\mathtt{B}}$.
- $E_{\mathtt{B}'}$ adds to $E_{\mathtt{B}}$ the axioms of `DATA` plus, for each `B`-action σ, the sentences:
  – $\forall \{i, X, W\} \cdot \pi(i, \sigma'(i, X, W)) = \sigma(\pi(i, X), W)$;
  – $\forall \{i, j, X, W\} \cdot i \neq j \Rightarrow \pi(j, \sigma'(i, X, W)) = \pi(j, X)$;

  and for each `B`-observation ζ the equation:
  – $\forall \{i, X, W\} \cdot \zeta'(i, X, W) = \zeta(\pi(i, X), W)$.

Dynamic indexing

In this case, the set of the components may differ across different states of the compound object. For instance, in a system of bank accounts, individual accounts may be deleted or added to the system. The dynamics of the set of components is managed by a special component. A specification like

```
bobj B' with states St is
  indexing B on BOBJ by Idx .
  ...
endbo
```

should be understood and a synchronised composition of `BOBJ` with multiple copies of `B`, where the synchronicity refers to the management of the indexing. Its semantics refines the semantics of parallel indexed compositions as follows:

- $H_{\mathtt{B}'} = H_{\mathtt{B}} \uplus H_{\mathtt{BOBJ}} \uplus \{\mathtt{St}\}$.
- $h_{\mathtt{B}'} = \mathtt{St}$.
- $V_{\mathtt{B}'} = V_{\mathtt{B}} \cup V_{\mathtt{BOBJ}}$.
- $F_{\mathtt{B}'}$ gathers together the operations of $F_{\mathtt{B}}$ and of $F_{\mathtt{BOBJ}}$, and then adds
  – a parameterized behavioural 'projection' $\pi_{\mathtt{B}} : \mathtt{Idx}\ h_{\mathtt{B}'} \to h_{\mathtt{B}}$;
  – a 'projection' $\pi_{\mathtt{BOBJ}} : h_{\mathtt{B}'} \to h_{\mathtt{BOBJ}}$;
  – for each action $\sigma \in (BF_{\mathtt{B}})_{h_{\mathtt{B}} w \to h_{\mathtt{B}}}$, a `B'`-action $\sigma' : \mathtt{Idx}\ h_{\mathtt{B}'} w \to h_{\mathtt{B}'}$;
  – for each action $\sigma \in (BF_{\mathtt{BOBJ}})_{h_{\mathtt{BOBJ}} w \to h_{\mathtt{BOBJ}}}$, a `B'`-action $\sigma' : h_{\mathtt{B}'} w \to h_{\mathtt{B}'}$;
  – for each observation $\zeta \in (BF_{\mathtt{B}})_{h_{\mathtt{B}} w \to s}$, a `B'`-observation $\zeta' : \mathtt{Idx}\ h_{\mathtt{B}'} w \to s$;
  – for each observation $\zeta \in (BF_{\mathtt{BOBJ}})_{h_{\mathtt{BOBJ}} w \to s}$, a `B'`-observation $\zeta' : h_{\mathtt{B}'} w \to s$.
- $BF_{\mathtt{B}'}$ consists of the above behavioural operations added to $BF_{\mathtt{B}}$ and $BF_{\mathtt{BOBJ}}$.

- $E_{\mathtt{B}'}$ gathers together $E_{\mathtt{B}}$ and $E_{\mathtt{BOBJ}}$, and also adds the following sentences:
  - $\forall\{X, W\} \cdot \pi_{\mathtt{BOBJ}}(\sigma'(X, W)) = \sigma(\pi_{\mathtt{BOBJ}}(X), W)$, for each `BOBJ`-action σ;
  - $\forall\{X, W\} \cdot \pi_{\mathtt{BOBJ}}(\sigma'(X, W)) = \pi_{\mathtt{BOBJ}}(X)$, for each `B`-action σ;
  - $\forall\{X, W\} \cdot \zeta'(X, W) = \zeta(\pi_{\mathtt{BOBJ}}(X), W)$, for each `BOBJ`-observation ζ;
  - $\forall\{i, X, W\} \cdot \zeta'(i, X, W) = \zeta(\pi_{\mathtt{B}}(i, X), W)$, for each `B`-observation ζ.

When writing indexed COMP specifications in ASCII:

- the projections π and $\pi_{\mathtt{B}}$ are encoded as $\mathtt{B}/h_{\mathtt{B}}$;
- the projection $\pi_{\mathtt{BOBJ}}$ is encoded as $\mathtt{BOBJ}/h_{\mathtt{BOBJ}}$;
- the actions $\sigma' : \mathtt{Idx}\ h_{\mathtt{B}'} w \to h_{\mathtt{B}'}$ are encoded as $\mathtt{B}/\sigma$;
- the actions $\sigma' : h_{\mathtt{B}'} w \to h_{\mathtt{B}'}$ are encoded as $\mathtt{BOBJ}/\sigma$;
- the observations $\zeta' : \mathtt{Idx}\ h_{\mathtt{B}'} w \to s$ are encoded as $\mathtt{B}/\zeta$;
- the observations $\zeta' : h_{\mathtt{B}'} w \to s$ are encoded as $\mathtt{BOBJ}/\zeta$.

4 Specification and verification methodologies

4.1 The methodology of specifying component-free objects

There is a freedom in the way component-free objects are specified that contrasts with the rather rigid specification methodologies for composing objects. However, the following methodology is often used when specifying component-free objects, say `B`. It follows the principle that after applying any `B`-action we are able for all possible cases to evaluate all observations on the result `B`-state. Mathematically, this means that $E_{\mathtt{B}}$ contains only the following groups of (possibly conditional) equations:

- For any `B`-action $\sigma : h_{\mathtt{B}} w \to h_{\mathtt{B}}$ and `B`-observation $\zeta : h_{\mathtt{B}} w' \to v$, $E_{\mathtt{B}}$ includes

  $$\{\forall\{X, W, W'\} \cdot C^k_{\sigma,\zeta} \Rightarrow \zeta(\sigma(X, W)) = \tau^k_{\sigma,\zeta} \mid 1 \leq k \leq n_{\sigma,\zeta}\}$$

  where $\tau^k_{\sigma,\zeta}$ is a term and $C^k_{\sigma,\zeta}$ is a condition, neither of them containing any `B`-action, and such that for each `B`-algebra A we have:

  $$A \models \forall\{X, W, W'\} \cdot \bigvee\nolimits_k C^k_{\sigma,\zeta} \qquad \text{(completeness)}$$
  $$A \models \forall\{X, W, W'\} \cdot \neg(C^k_{\sigma,\zeta} \wedge C^{k'}_{\sigma,\zeta}) \qquad \text{(disjointness)}$$

- For any `B`-constant c and each `B`-observation ζ, a similar set of axioms like above for `B`-actions, but now the τ's are data terms.

This methodology allows us to reduce the (rather difficult, in general) problem of verifying whether two states are behaviourally equivalent to a much simpler check of whether all observations yield the same outcome when applied to the two states.

Revisiting the bank-account specification

To illustrate the principle, let us consider once more the `ACCOUNT` object defined in Section 3.2. The simplest `ACCOUNT`-algebra, which we denote by M, implements only a minimal information, namely the current balance:

- $M_{\texttt{Account}} = \omega = \{0, 1, 2, \dots\}$, the set of the natural numbers;
- M interprets the entities of the imported module `NAT` as the common sets of numbers with their common operations (addition, subtraction, etc.);
- $M_{\texttt{deposit}}(a, n) = a + n$;
- $M_{\texttt{withdraw}}(a, n) = \begin{cases} a - n, & \text{when } n \leq a, \\ a, & \text{otherwise;} \end{cases}$
- $M_{\texttt{balance}}(a) = a$.

A more complex `ACCOUNT`-algebra M' may implement more information: for instance, the history of all actions on the account. This corresponds to the following situation:

- the imported data (`NAT`) is interpreted in the standard way, like M above does;
- $M'_{\texttt{Account}}$ consists of all lists a of integers such that $S(a) \geq 0$, where $S(a)$ represents the sum of all elements of a;
- $M'_{\texttt{deposit}}(a, n) = (n\ a)$;
- $M'_{\texttt{withdraw}}(a, n) = \begin{cases} (-n\ a), & \text{when } S(a) \leq n, \\ a, & \text{otherwise;} \end{cases}$
- $M'_{\texttt{balance}}(a) = S(a)$.

Many other `ACCOUNT`-algebras are possible. For instance a model 'in-between' M and M' may store only the account balance and the number of actions performed. Regardless of which algebra we choose, the COMP methodology provides a simple and intuitive characterization of behavioural equivalences.

Since the specification of `ACCOUNT` conforms to the above methodology for component-free objects, it follows that two accounts (in any `ACCOUNT`-algebra) are behaviourally equivalent if and only if the have the same balance. In particular, we have:

In the algebra M: $a \sim_M a'$ if and only if $a = a'$.

In the algebra M': $a \sim_{M'} a'$ if and only if $S(a) = S(a')$.

The following section sets the basis for more complex methodologies meant to facilitate a lifting of this characterization from base to compound objects.

4.2 The compositionality of the algebras

Consider the simple parallel composition of two objects, `B1` and `B2`, and let `B` denote the compound object. Then the algebras of `B` and the algebras of the component objects `B1` and `B2` are related by the following important properties:

1. Given any `B`-algebra A, if we consider A_1 and A_2 the reducts of A that interpret only the syntactic entities of `B1` and `B2`, respectively, then A_1 is a `B1`-algebra and A_2 is a `B2`-algebra.

2. Conversely, given a `B1`-algebra A_1 and a `B2`-algebra A_2 that share the same interpretations of data types and data operations that are common to `B1` and `B2`, there exists a 'standard' `B`-algebra A that expands both A_1 and A_2. The full mathematical details of this result can be found in [2]. Here, we merely recall the most important bits of the definition of A:
   - $A_{h_{\mathtt{B}}} = A_{h_{\mathtt{B1}}} \times A_{h_{\mathtt{B2}}}$, i.e., the Cartesian product of the two state spaces;
   - for each $i \in \{1, 2\}$, $A_{\pi_i}(a_1, a_2) = a_i$, thus justifying the term 'projection';
   - for each `B1`-action σ, $A_{\sigma_1}((a_1, a_2), W) = ((A_1)_\sigma(a_1, W), a_2)$;
   - for each `B2`-action σ, $A_{\sigma_2}((a_1, a_2), W) = (a_1, (A_2)_\sigma(a_2, W))$.

3. Within the context of the previous properties, the behavioural equivalence on A is the conjunction of the behavioural equivalences of A_1 and on A_2. More precisely, for any elements $a, a' \in A_{h_{\mathtt{B}}}$, we have:

$$a \sim a' \quad \text{if and only if} \quad \pi_1(a) \sim \pi_1(a') \text{ and } \pi_2(a) \sim \pi_2(a').$$

These three properties have been developed mathematically in [2]. While the former two properties express a semantical correctness of simple parallel compositions, the latter one represents the foundation for the COMP verification method.

4.3 Enhancing simple parallel compositions

After a `composing` declaration, the COMP grammar allows for declarations of other actions, observations, or just operations, and also of other sentences than those presented above. The COMP specification of parallel composition considers two categories of such declarations motivated by the following situations:

- The necessity to be able to use observations on the components directly at the compound level.
- The necessity to initialise the compound object through the initialisation of its components.

Such enhancements have to preserve the three semantic properties discussed above. This is achieved by observing the following set of rules:

- No additional `B`-actions (other than the predefined ones) can be declared.
- When specifying a `B`-observation ζ we write *one* equation of the form

  $$\forall\{X, W\} \cdot \zeta(X, W) = c_\zeta[\pi_i(X)]$$

  where $c_\zeta[z]$ is a `Bi`-context (which means that the sort of the variable z is $h_{\texttt{Bi}}$).
- For each constant c of sort $h_{\texttt{B}}$ and *each* $i \in \{1, 2\}$, we should write an equation of the form $\pi_i(c) = c_i$, where c_i is a constant of sort $h_{\texttt{Bi}}$.
- Besides the above equations, no other sentences should be introduced.

4.4 Equivalence, associativity, and commutativity of parallel compositions

The three semantic properties mentioned in Section 4.2 are common to all types of behavioural-object compositions available in COMP. However, parallel compositions enjoy some distinctive additional properties that are generally characteristic to the phenomenon of parallelism in computation. The mathematical details of these properties can be found in [2]; here, we just present them rather informally.

The first such property is a preliminary one, and concerns only behavioural objects, not necessarily having correspondents in other concurrency formalisms. Let us say that two behavioural objects B and B' are *equivalent* when there is a one-one correspondence between B-algebras and B'-algebras that preserves the interpretations of the states (designated by the sorts h_B and $h_{B'}$) as well as the behavioural equivalence relation on states. In fact, by this equivalence we abstract behavioural objects to their essence, their states at the compound level, and therefore ignore their components. If we do that, we find that:

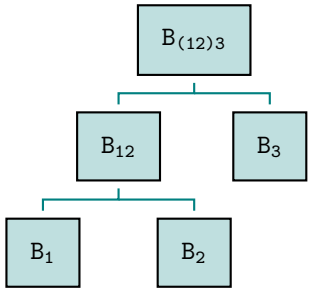
> Any two COMP specifications of a parallel composition of two objects determine equivalent compound objects.

The second property is a *commutativity* property:

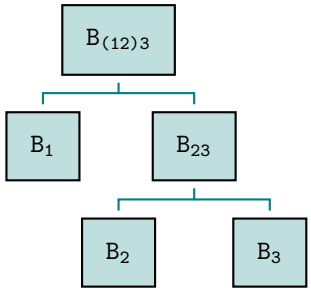
> In a parallel composition, the order of the components is immaterial. That is, by changing their order we get equivalent compound objects.

The third property is an *associativity* property. Assume we have three objects: B_1, B_2, and B_3. We have three ways to compose them:

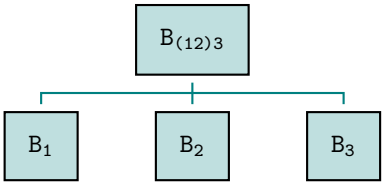
1. First compose B_1 and B_2 to get an object B_{12}, then compose B_{12} with B_3.



2. First compose B_2 with B_3 to get an object B_{23}, then compose B_1 with B_{23}.



3. Compose B_1, B_2, and B_3 simultaneously.



Then all three parallel compositions above yield equivalent compound objects.

4.5 The methodology of synchronisation

Consider the following synchronization of an existing compound object `B`:

```
bobj B' is
  syncing B .
  ...
endbo
```

According to this methodology, the specification of `B'` has to conform to a set of rules that guarantee the three properties of Section 4.2. The meaning of these rules is that each synchronisation action may affect all components, and its effect on each component can be described as an effect of a local action. Moreover, these effects may depend on cases, which do not overlap and which cover all possible situations. The set of rules for synchronisation are as follows:

- Except for data declarations (sorts and operations), the signature declarations in `B'` may add only actions on the (hidden) sort $h_{\mathtt{B'}} = h_{\mathtt{B}}$.
- For each `B'`-action σ and each component $\mathtt{B}_i$ of `B` there is a set of quantified, potentially conditional, equations of the form:

  $$\{\forall\{X, W\} \cdot C^k_{\sigma,i} \Rightarrow (\pi_i(\sigma(X, W)) = \tau^k_{\sigma,i}[\pi_i(X)]) \mid 1 \leq k \leq n_{\sigma,i}\}$$

  where:
  - each $\tau^k_{\sigma,i}$ is a $\mathtt{B}_i$-context or a $\mathtt{B}_i$-constant, and
  - $C^k_{\sigma,i}$ is a quantifier-free sentence formed from strict equations whose members are either data terms or terms of the form $c[\pi_j(X)]$, where $c[z]$ is a behavioural context

  such that for each `B`-algebra A we have:

  $A \models \forall\{X, W\} \cdot \bigvee_k C^k_{\sigma,i}$ (completeness)

  $A \models \forall\{X, W\} \cdot \neg(C^k_{\sigma,i} \wedge C^k_{\sigma,i})$ (disjointness)
- The COMP specification of `B'` does not contain any other sentences.

Revisiting the synchronized composition of two accounts

Now let us see in detail how the methodological rules for specifying synchronicity in COMP are observed in the specification of `ACCOUNT-SYS-TRANSFER`:

- `B` is `ACCOUNT-SYS` while `B'` is `ACCOUNT-SYS-TRANSFER`.
- `B'` declares only one (synchronisation) action, namely `transfer`.
- There is only one axiom for `transfer` corresponding to the first account. This is unconditional, which means that $C^1_{\mathtt{transfer},1}$ is just *true*.
- The term $\tau^1_{\mathtt{transfer},1}$ is $\mathtt{withdraw}(z, N)$.
- There are two equations for `transfer` corresponding to the second account.
  - The term $\tau^1_{\mathtt{transfer},2}$ is just z, a variable.

– $C^1_{\texttt{transfer},2}$ is $N \leq \texttt{A/balance}(AS)$, which as equation is

$$(N \leq \texttt{A/balance}(AS)) = \texttt{true}$$

which in turn can be written as

$$(N \leq \texttt{balance}(\texttt{A/Account}(AS))) = \texttt{true}$$

The left-hand side of this equation is a term of the form $c[\pi_j(X)]$ while the right-hand side is a data term (the predefined constant `true`).

– $\tau^2_{\texttt{transfer},2}$ is $\texttt{deposit}(z, N)$.

– $C^2_{\texttt{transfer},2}$ is $N \not\leq \texttt{A/balance}(AS)$, which can be framed in terms of the methodological conditions like we did for $C^1_{\texttt{transfer},2}$.

Note that the completeness and the disjointness conditions are trivially fulfilled for both components. In the first case, that is because there is only one axiom whose condition of *true*; and in the second case, the natural-number relations $x \leq y$ and $x \not\leq y$ are, of course, both disjoint and complementary.

4.6 The methodology of indexed compositions

For indexed compositions, the three properties of Section 4.2 hold in a suitably adapted form. For instance, the compositionality of behavioural equivalences becomes:

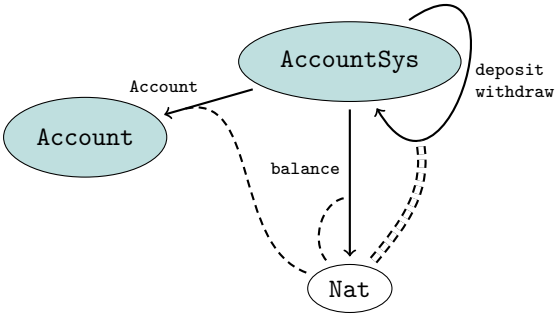
$$a \sim a' \quad \text{if and only if} \quad \pi(i, a) \sim \pi(i, a') \text{ for all indices } i.$$

A static system of multiple bank accounts

A COMP specification of a countable parallel composition of accounts is as follows:

```
bobj ACCOUNT-SYS-MULT with states AccountSys is
  indexing ACCOUNT on NAT by Nat .
endbo
```

This leads to the following ADJ-diagram for `ACCOUNT-SYS-MULT`:



A dynamic system of bank accounts

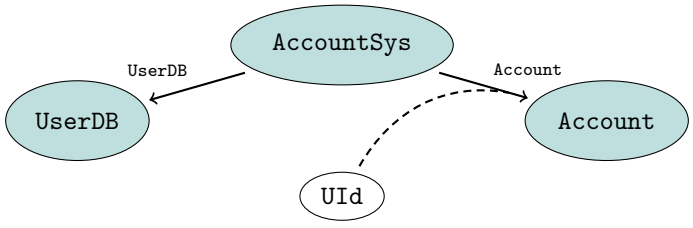
The static indexed parallel composition of bank accounts can be refined to a dynamic composition in which accounts can be added or deleted to the system by using the following user-database object for managing the indexing.

```
bobj USER-DB with states UserDB is
  protecting UID .
  vars U, U' : UId . var  DB : UserDB .

  op empty : () -> UserDB .
  obs _in_ : UId UserDB -> Bool .
  act add : UId UserDB -> UserDB .
  act delete : UId UserDB -> UserDB .

  ax U in empty = false .
  ax U in add(U', DB) = (U == U') or (U in DB) .
  ax U in delete(U', DB) = (U =/= U') and (U in DB) .
endbo
```

In this case, the `indexing` declaration

```
bobj ACCOUNT-SYS-DYN with states AccountSys is
  indexing ACCOUNT on USER-DB by UId .
endbo
```

determines the following projections:



The equations for the `USER-DB` actions are:

```
ax UserDB(add(U, AS)) = add(U, UserDB(AS)) .
ax Account(I, add(U, AS)) = init-account if I = U .
ax Account(I, add(U, AS)) = Account(I, AS) if not I = U .
ax UserDB(delete(U, AS)) = delete(U, UserDB(AS)) .
ax Account(I, delete(U, AS)) = no-account if I = U .
ax Account(I, delete(U, AS)) = Account(I, AS) if not I = U .
```

Note that, in this case, the object `ACCOUNT` is enhanced with two constants, `init-account` and `no-account`, whose observed values are as follows:

```
ax balance(init-account) = 0 .
ax balance(no-account) = 0 .
```

The equations for the `ACCOUNT` actions are as follows:

```
ax UserDB(deposit(AS, U, N)) = UserDB(AS) .
ax Account(I, deposit(AS, U, N)) = Account(I, AS)
   if not I = U or (U in UserDB(AS)) = false  .
ax Account(I, deposit(AS, U, N)) = deposit(Account(I, AS), N)
   if I = U and (U in UserDB(AS)) = true .
ax UserDB(withdraw(AS, U, N)) = UserDB(AS) .
ax Account(I, withdraw(AS, U, N)) = Account(I, AS)
   if not I = U or (U in UserDB(AS)) = false  .
ax Account(I, withdraw(AS, U, N)) = withdraw(Account(I, AS), N)
   if I = U and (U in UserDB(AS)) = true .
```

We may also add various synchronization actions to `ACCOUNT-SYS-DYN`, such as a transfer between any two accounts:

```
act transfer : AccountSys UId UId Nat -> AccountSys .
```

References

[1] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[2] Răzvan Diaconescu. Behavioural specification for hierarchical object composition. *Theor. Comput. Sci.*, 343(3):305–331, 2005.

[3] Răzvan Diaconescu and Kokichi Futatsugi. Behavioural coherence in object-oriented algebraic specification. *J. Univers. Comput. Sci.*, 6(1):74–96, 2000.

[4] Răzvan Diaconescu and Kokichi Futatsugi. Logical foundations of cafeobj. *Theor. Comput. Sci.*, 285(2):289–318, 2002.

[5] Razvan Diaconescu and Ionut Tutu. On the algebra of structured specifications. *Theor. Comput. Sci.*, 412(28):3145–3174, 2011.

[6] Joseph A. Goguen and Razvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Hartmut Ehrig, editor, *Recent Trends in Data Type Specification, 9th Workshop on Specification of Abstract Data Types Joint with the 4th COMPASS Workshop, Caldes de Malavella, Spain, October 26-30, 1992, Selected Papers*, volume 785 of *Lecture Notes in Computer Science*, pages 1–29. Springer, 1992.

[7] Joseph A. Goguen and Grant Malcolm. A hidden agenda. *Theor. Comput. Sci.*, 245(1):55–101, 2000.

[8] Rolf Hennicker and Michel Bidoit. Observational logic. In Armando Martin Haeberer, editor, *Algebraic Methodology and Software Technology, 7th International Conference, AMAST '98, Amazonia, Brasil, January 4-8, 1999, Proceedings*, volume 1548 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 1998.

[9] Horst Reichel. Behavioural equivalence – a unifying concept for initial and final specifications. In *Proceedings, Third Hungarian Computer Science Conference*. Akademiai Kiado, 1981.

[10] Horst Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras*. Clarendon, 1987.

[11] Grigore Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.

[12] Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2012.