Submitted to **World Congress of Formal Methods'99**

Component-based Algebraic Specification and Verification in CafeOBJ

Răzvan Diaconescu[†] and Kokichi Futatsugi and Shusaku Iida

Japan Advanced Institute of Science and Technology

Draft of 6 March 1999

We present a formal method for component-based system specification and verification which is based on the new algebraic specification language CafeOBJ, which is a modern successor of OBJ incorporating several new developments in algebraic specification theory and practice.
We first give an overview of the origins and of the main features of CafeOBJ, including its logical foundations, and then we focus on the behavioural specification paradigm in CafeOBJ, surveying the object-oriented CafeOBJ specification and verification methodology based on behavioural abstraction.
The last part of this paper further focuses on a component-based behavioural specification and verification methodology which features high reusability of both specification code and verification proof scores. This methodology constitutes the basis for an industrial strength formal method around CafeOBJ.

1. Introduction

In this introduction we will give a brief overview of CafeOBJ, including its origins, its main features, and its logical foundations.

1.1. *Origins of CafeOBJ*

CafeOBJ (whose definition is given by (DF98b))is a modern successor of the OBJ language incorporating several new major developments in algebraic specification theory and practice. It is aimed to be an industrial strength language, suitable both for researchers and for practitioners.

The origins of OBJ can be traced back to Goguen's gradual realization, around 1970, that Lawvere's characterization of the natural numbers as a certain initial algebra (Law64) could be extended to other data structures of interest for Computing Science. The influence of Saunders Mac Lane was also important during that period, leading to the beginning of the famous ADJ group (Gog89) led by Goguen. During the ADJ group period, a mathematical theory of abstract data types as initial algebras was developed. Together with considering term rewriting as the computational side of abstract data types, this constitutes the pillar of the OBJ basic specifications level. It is important to mention that from the very beginning the design of OBJ had been

[†] On leave from the Institute of Mathematics of the Romanian Academy, PO Box 1-764, Bucharest 70700, ROMANIA.

emerging directly from clean and elegant mathematical theories, this process being (rather subtle) reflected as one of the main strength of the language.

Another major step in the development of OBJ was the relativization of algebraic specification over any logic due to Goguen and Burstall's *institutions* (GB92) (categorical abstract model theory for specification and programming). This pushed the theory of algebraic specification into a modern age. At the beginning institutions provided support for developing advanced structuring specification techniques (i.e., module composition systems) independently of the actual logical formalism, as emerging from the research on Clear (BG80). However, today, after nearly two decades, their significance has been widely expanded. For example, institutions support in an essential way the design of multi-paradigm (declarative) systems.

It could be said that initial algebra semantics, rewriting, and institutions, are the conceptual pillars of the OBJ world. Their development and refinement can be easily noticed if one takes a close look at the chain of successive versions of OBJ, culminating with OBJ3 (GWM).

Following the vitally important idea of module composition of Clear, several attempts of implementing modularized algebraic specification languages were done including the early pioneering design and implementation of HISP language (FO80). After these experiences, the stabilization of the OBJ design (and its most prominent implementation at SRI) started after the design and prototype implementation of OBJ2 at SRI in 1984 (FGJM85). It coincides with several attempts to extend OBJ towards other paradigms, most notably constraint logic programming (GM86, Dia94), object-oriented programming (GM87). Although, due to the indisputable strength of algebraic specification, all these attempts were successful, the interest of the OBJ community has been recently shifting towards a new language generation focusing more on the recent internal developments in algebraic specification rather than in integrating powerful paradigms from the outside world. Two such examples are CafeOBJ (DF98b) and Maude (CELM96). With respect to CafeOBJ, although some experimental design and implementation were done in the past (FS92), (DF98b) is the first definitive definition of the language.

1.2. *CafeOBJ main features*

1.2.1. *Equational Specification and Programming.* This is inherited from OBJ (GWM , FGJM85) and constitutes the basis of the language, the other features being somehow built on top of it. As with OBJ, CafeOBJ is *executable* (by term rewriting), which gives an elegant declarative way of functional programming, often referred as *algebraic programming*.[1] As with OBJ, CafeOBJ also permits equational specification modulo several equational theories such as associativity, commutativity, identity, idempotence, and combinations between all these. This feature is reflected at the execution level by term rewriting *modulo* such equational theories.

1.2.2. *Behavioural Specification.* Behavioural specification (GD94b, GM97, Dia98a) provides another novel generalization of ordinary algebraic specification but in a different direction. Behavioural specification characterizes 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

[1] Please notice that although this paradigm may be used as programming, this aspect is still secondary to its specification side.

software systems since it naturally embeds other useful paradigms such as concurrency, object-orientation, constraints, nondeterminism, etc. (see (GM97) 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, which also generalizes process algebra and transition systems (see (GM97)).

CafeOBJ directly supports behavioural specification and its proof theory through special language constructs, such as

— hidden sorts (for states of systems),
— behavioural operations (for direct "actions" and "observations" on states of systems),
— behavioural coherence declarations for (non-behavioural) operations (which might be either derived (indirect) "observations" or "constructors" on states of systems), and
— behavioural axioms (stating behavioural satisfaction).

The advanced coinduction proof method receives support in CafeOBJ via a default (candidate) coinduction relation (denoted =*=). In CafeOBJ, coinduction can be used either in the classical HSA sense (GM97) for proving behavioural equivalence of states of objects, or for proving behavioural transitions (which appear when applying behavioural abstraction to RWL).[2]

Besides language constructs, CafeOBJ supports behavioural specification and verification by several methodologies.[3] CafeOBJ currently highlights a methodology for concurrent object composition which features high reusability not only of specification code but also of verifications (DF98b, IMD 98). Behavioural specification in CafeOBJ might also be effectively used as an object-oriented (state-oriented) alternative for traditional ADT specifications. Several cases seem to indicate that an object-oriented style of specification even of basic data types (such as sets, lists, etc.) might lead to higher simplicity of code and drastic simplification of verification process (DFI98).

Behavioural specification is reflected at the execution level by the concept of *behavioural rewriting* (DF98b, Dia98a) which refines ordinary rewriting with a condition ensuring the correctness of the use of behavioural equations in proving strict equalities.

1.2.3. *Rewriting Logic Specification.* Rewriting logic specification in CafeOBJ is based on a simplified version of Meseguer's *rewriting logic* (Mes92) specification framework for concurrent systems which gives a non-trivial extension of traditional algebraic specification towards concurrency. RWL incorporates many different models of concurrency in a natural, simple, and elegant way, thus giving CafeOBJ a wide range of applications. Unlike Maude (CELM96), the current CafeOBJ design does not fully support *labeled* RWL which permits full reasoning about multiple transitions between states (or system configurations), but provides proof support for reasoning about the *existence* of transitions between states (or configurations) of concurrent systems via a built-in predicate (denoted ==>) with dynamic definition encoding both the proof theory of RWL and the user defined transitions (rules) into equational logic.

From a methodological perspective, CafeOBJ develops the use of RWL transitions for speci-

[2] However, until the time this paper was written, the latter has not been yet explored sufficiently, especially practically.
[3] This is still an open research topic, the current methodologies might be developed further and new methodologies might be added in the future.

fying and verifying the properties of *declarative encoding of algorithms* (see (DFI98)) as well as for specifying and verifying transition systems.

1.2.4. *Module System.* The principles of the CafeOBJ module system are inherited from OBJ which builds on ideas first realized in the language Clear (BG80). CafeOBJ module system features

— several kinds of imports,
— sharing for multiple imports,
— parameterized programming allowing
  – multiple parameters,
  – views for parameter instantiation,
  – integration of CafeOBJ specifications with executable code in a lower level language
— module expressions.

However, the theory supporting the CafeOBJ module system represents an updating of the original Clear/OBJ concepts to the more sophisticated situation of multi-paradigm systems involving theory morphisms across institution embeddings (Dia98b), and the concrete design of the language revise the OBJ view on importation modes and parameters (DF98b).

1.2.5. *Type System and Partiality.* CafeOBJ has a type system that allows subtypes based on *order sorted algebra* (abbreviated **OSA**) (GM92, GD94a). This provides a mathematically rigorous form of runtime type checking and error handling, giving CafeOBJ a syntactic flexibility comparable to that of untyped languages, while preserving all the advantages of strong typing.

Since at this moment there are many order sortedness formalisms, many of them very little different from others, and each of them having its own technical advantages and disadvantages and being most appropriate for a certain class of applications, we decided to keep the concrete order sortedness formalism open at least at the level of the language definition. Instead we formulate some basic simple conditions which any concrete CafeOBJ order sorted formalism should obey. These conditions come close to Meseguer's OSA^R (Mes98) which is a revised version of other versions of order sortedness existing in the literature, most notably Goguen's OSA (GD94a).

CafeOBJ does not directly do partial operations but rather handles them by using error sorts and a sort membership predicate in the style of *membership equational logic* (abbreviated **MEL**) (Mes98). The semantics of specifications with partial operations is given by MEL.

1.3. *The CafeOBJ specification and verification environment*

Although this is rather a feature of the current system rather than of the language, due to its importance for the effective use of the current CafeOBJ system, we briefly survey it here.

The CafeOBJ system includes an environment supporting specification documents with formal contents over networks and enabling formal verifications of specifications. The CafeOBJ environment takes advantage of current InterNet technologies and can be thought as consisting of four parts:

— The **interpreter** in isolation acts very much like the OBJ3 interpreter by checking syntax and

evaluating (reducing) terms. In addition, the CafeOBJ interpreter incorporates an abstract TRS machine and a compiler.

— The **proof assistant** extends the theorem proving capabilities of the interpreter with more powerful, dedicated provers. A proof assistant taking into account the particulars of CafeOBJ is considered; this incorporates two kinds of inductive theorem provers, one based on completion procedures, and the other on explicit structural induction.

— The **document manager** takes care of processing of specification documents over networks by analyzing specification documents for showing contents to the user (via browsers, editors, etc.) by extracting instructions of evaluations and proofs, by searching for suitable documents in libraries, and by managing documents over networks, retrieving, storing, caching them as requested.

— **Specification libraries** focus on several specific problem domains, such as object-oriented programming, database management, interactive systems, etc.

1.4. *CafeOBJ Logical Foundations*

CafeOBJ is a declarative language with firm mathematical and logical foundations in the same way as other OBJ-family languages (OBJ, Eqlog (GM86, Dia94), FOOPS (GM87), Maude (Mes92)) are. The reference paper for the CafeOBJ mathematical foundations is (DF98a), while the book (DF98b) gives a somehow less mathematical easy-to-read (including many examples) presentation of the semantics of CafeOBJ. In this section we give a very brief overview of the CafeOBJ logical and mathematical foundations, for a full understanding of this aspect of CafeOBJ the reader is referred to (DF98a) and (DF98b).

The mathematical semantics of CafeOBJ is based on state-of-the-art algebraic specification concepts and results, and is strongly based on category theory and the theory of institutions (GB92, Dia98b, DGS93). The following are the principles governing the logical and mathematical foundations of CafeOBJ:

P1. there is an underlying logic[4] in which all basic constructs and features of the language can be rigorously explained.

P2. provide an integrated, cohesive, and unitary approach to the semantics of specification in-the-small and in-the-large.

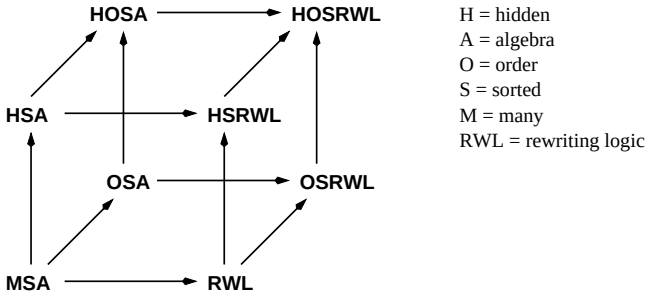
P3. develop all ingredients (concepts, results, etc.) at the highest appropriate level of abstraction.

The CafeOBJ cube. CafeOBJ is a multi-paradigm language. Each of the main paradigms implemented in CafeOBJ is rigorously based on some underlying logic; the paradigms resulting from various combinations are based on the combination of logics. The following table shows the correspondence between specification/programming paradigms and logics as they appear in the actual version of CafeOBJ, also pointing to some basic references.

[4] Here "logic" should be understood in the modern relativistic sense of "institution" which provides a mathematical definition for a logic (see (GB92)) rather than in the more classical sense.

ABBREVIATION	LOGIC	SPEC/PGM PARADIGM	BASIC REF.
MSA	many sorted algebra	algebraic specification	(Gogar)
OSA	order sorted algebra	algebraic specification with subtypes	(Gogar, GM92, GD94a)
HSA	hidden sorted algebra	behavioural concurrent specification	(Dia98a, GM97, GD94b)
HOSA	hidden order sorted algebra	behavioural specification with subtypes	(GD94b, BD94)
RWL	rewriting logic	rewriting logic specification	(Mes92)
OSRWL	order sorted rewriting logic	rewriting logic specification with subtypes	
HSRWL	hidden sorted rewriting logic	behavioural rewriting logic specification	(Dia96b)
HOSRWL	hidden order sorted rewriting logic	behavioural rewriting logic specification with subtypes	(DF98a)

There are some embedding relations between these logics, which correspond to institution embeddings (i.e., a strong form of institution morphisms of (GB92, DGS93)) and which are shown by the following **CafeOBJ cube** (the orientation of arrows correspond to embedding "less complex" into "more complex" logics).



The mathematical structure represented by this cube is that of a *lattice of institution embeddings* (Dia98b, DF98a). By employing other logical-based paradigms the CafeOBJ cube may be thought as a hyper-cube (see (DF98a, DF98b) for details). It is important to understand that th CafeOBJ logical foundations are based on the CafeOBJ cube rather than on its flattening represented by HOSRWL.[5]

[5] The reason for this is explained in (DF98a, DF98b).

2. Behavioural Specification in CafeOBJ

Behavioural specification might be the most distinctive feature of CafeOBJ within the broad family of algebraic specification languages. As mentioned above, behavioural specification paradigm is incorporated into the design of the language in a rather direct way. Also, this paradigm constitutes the core of the current CafeOBJ object-oriented specification and verification methodologies. We devote this section to a methodological presentation of the behavioural specification paradigm in CafeOBJ, trying also to explain the main concepts behind this paradigm.

2.1. *Basic behavioural specification*

Basic behavioural specification is the simplest level of behavioural specification in which the operations are either *actions* or *observations* on the states of the objects. Let us consider an object-oriented (or "state-oriented") CafeOBJ specification for lists:

```
mod! TRIV+ (X :: TRIV) {
  op err :  -> ?Elt
}
mod* LIST  {
  protecting(TRIV+)
  *[ List ]*
  op nil : -> List
  bop cons : Elt List-> List  -- action
  bop car : List -> ?Elt      -- observation
  bop cdr : List -> List      -- action
  vars E E' : Elt
  var L : List
  eq car(nil) = err .
  eq car(cons(E, L)) = E .
  beq cdr(nil) = nil .
  beq cdr(cons(E, L)) = L .
}
```

This is quite different from the usual data-oriented specification of lists. In our behavioural specification, lists are treated as *objects* with states (the sort of states is the *hidden* sort List), and the usual list operations (*cons* and *cdr*) *act* on the states of the list object or (*car*) *observe* the states. Actions and observations are specified as *behavioural* operations. In general, a behavioural operation is called *action* iff its sort is hidden (i.e., state type), and is called *observation* iff its sort is visible (i.e., data type). Behavioural operations are restricted to have *exactly* one hidden sort in their arity, this monadicity property being characteristic to behavioural operations (either actions or observations). Behavioural operations define the *behavioural equivalence* relation between the states of the object, denoted as $\sim$:

$$s \sim s' \text{ iff } c(s) = c(s')$$

for all *visible behavioural contexts c*. A behavioural context c is any string of behavioural operations (this makes sense because of the monadicity property on hidden sorts of the behavioural operations). c is visible iff its sort is visible; this is the same as saying that c has an observation at the top. It is important to notice that behavioural equivalence is a semantic notion; this means

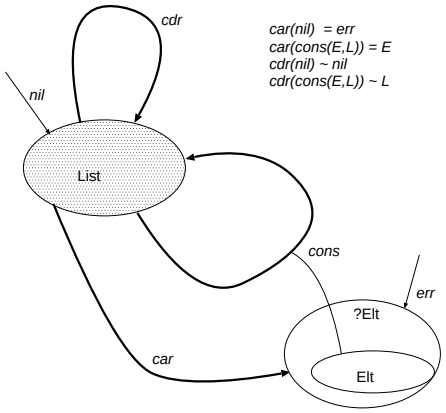
that whenever we consider a behavioural equivalence relation we need to consider a model (i.e., an implementation) for the specification[6].

CafeOBJ methodologies introduce a graphical notation extending the classical ADJ-diagram notation for data types for behavioural specification in which

G1. *Sorts are represented by elipsoidal disks with visible (data) sorts represented in white and hidden (state) sorts represented in grey, and with subsort inclusion represented by disk inclusion, and*

G2. *Operations are represented by multi-source arrows with the monadic part from the hidden sort thickenned in case of behavioural operations.*

The list specification can be therefore vizualised as follows:



Several other aspects of this specifications need special attention. The first one concerns the data of this specification and the error handling aspect of this methodology. LIST specifies a list object over any set of elements. “Any set of elements” is specified by the built-in module TRIV which specifies one sort (Elt) with loose denotation (hence its denotation is given by all sets); this is used as a parameter of the specification LIST and can be instantiated to any concrete data type. The error handling aspect arises because of the partiality of *car*. TRIV+ just introduces a new error element (*err*). The error supersort ?Elt is built-in[7] and *err* is the only new element belonging to [the denotation of] ?Elt; this is ensured by the free extension of [the loose denotation of] TRIV which is specified by giving TRIV+ initial denotation (**mod!**). Notice that this style of error handling contrasts the more complex data-oriented approach which uses a subsort for the non-empty lists and overloads the list operations on this subsort. This methodological simplification is mainly possible because of the loose denotation of behavioural specification (with the adequate “loose” behavioural equality) which avoids the strictness of the initial denotation of the data-oriented approach.

Another aspect is given by the use of behavioural equations in the specification LIST. Behavioural equations represent behvioural equivalence relations between states rather than strict equalities. Therefore each model (implementation) of LIST does not need to interpret *cdr(cons(e,l))*

[6] Which needs not to be a concrete one.

[7] It is provided by the system.

as l, where e is an element and l is a list[8], but rather as a state behavioural equivalent to l. For example, if one implements the list object as an array with pointer, in this model (implementation) this equality does not hold strictly, but it holds behaviourally. Generally speaking, behavioural equality is the meaningful equality on hidden sorts, while the strict equality is the meaningful equality for the visible (data) sorts. However, there are situations when the strict equality on hidden sorts is also necessary. Behavioural abstraction also provides a nice way of error handling for hidden sorts, as shown by the other behavioural equation. Thus instead of introducing a (hidden) error for *cdr(nil)*, we rather shift the error handling to the data type by saying this is behaviourally equivalent to *nil*.[9] A finer analysis of the behavioural equivalence on the list object (see the section below) tells us that the behavioural equality between *cdr(nil)* and *nil* is exactly the same with saying that $car(cdr^n(nil)) = err$ for all natural numbers n, which is the natural minimal condition for the behaviour of *nil*.

2.2. *Behavioural specification with hidden constructors*

Behavioural specification with hidden constructors is a more advanced level of behavioural specification which relies on the important novel concept of *behavioural coherence* first defined and studied in (DF98b, Dia98a) and which was first realized by the CafeOBJ language (DF98b).

At the general level, a *hidden constructor* is an operation on hidden sorts[10] whose sort is also hidden and which is not declared behavioural. This means that such operation does *not* take part in the definition of the behavioural equivalence relation. Also (and related to the above), a hidden constructor need not be monadic on the hidden sorts, thus it may admit several hidden sorts in the arity.

In the data-oriented specification of lists there is a difference in nature between *cons* and *cdr*, in that *cons* is a "constructor" and *cdr* is a "destructor". This different nature of *cons* and *cdr* reflects in the behavioural specification too and is formally supported by the fact that one may prove (from the specification LIST) that for all lists l and l',

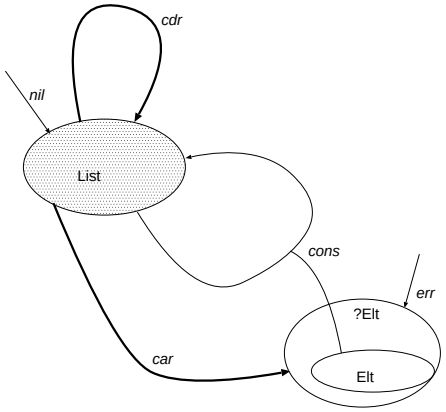
$$l \sim l' \text{ iff } car(cdr^n(l)) = car(cdr^n(l'))$$

for all natural numbers n. Technically this means that for the purpose of defining the appropriate behavioural equivalence for lists, *cons* does not play any rôle, therefore it may be specified as an ordinary operation, hence *cons* is a hidden constructor. Consequently, the only *real* behavioural operations are the observation *car* and the action *cdr*. This new specification for lists can be visualized by the following CafeOBJ diagram:

[8] Better said, a state of the list object.

[9] Recall that in LISP *cdr(nil)* is also equal to *nil* but under a LISP concept of equality; it may be worthwhile trying to think LISP equality in behavioural abstraction terms.

[10] Which may also have visible sorts in the arity.



This "neutrality" of *cons* with respect to the behavioural equivalence may be understood by the fact that *cons preserves* the behavioural equivalence defined by *cdr* and *car* only. This basic property of hidden constructors is called *coherence* (DF98b, Dia98a), which in general means the preservation of the behavioural equivalence relation by the hidden constructors. In CafeOBJ the coherence property is user specified as an operation attribute:

```
op cons : Elt List -> List    {coherent}
```

The semantic meaning of a coherence declaration is that the corresponding specification admits only models for which the operation is coherent (i.e., it preserves the behavioural equivalence). For methodological reasons CafeOBJ admits potentially non-coherent operations (in the absence of the coherence declaration), however in the final version of the specification all hidden constructors should be declare coherent both for semantical and operational reasons.

2.3. *Behavioural coherence methodologies*

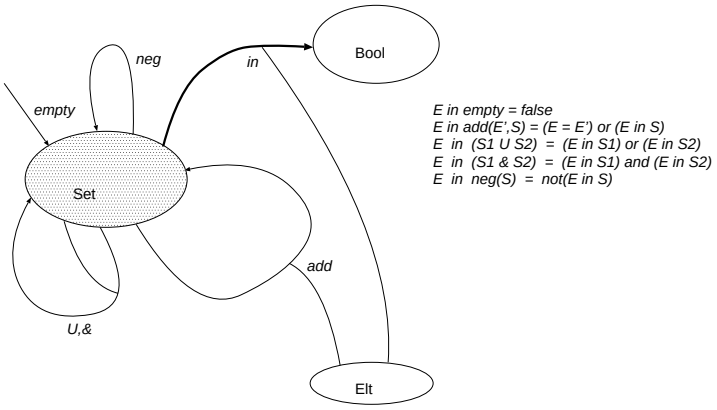
In the above list example the coherence of *cons* can be proved as a formal property of the specification [11]. This means that in any model of this specification the interpretation of *cons* automatically preserves the behavioural equivalence, so the class of models (implementations) of the specification with *cons* not specified as coherent coincides with its subclass of models for the case when *cons* is specified as coherent. Such constructors, which occurs frequently and which are practically desirable are called the *conservative*.

The opposite case is represented by the *non-conservative* constructors, which corresponds to the situation when the class of models for the case when the operation is specified as coherent is a strict subclass of the class of models when the operation is not specified as coherent. Proof-theoretically, this means the coherence property of the operation cannot be formally proved as a consequence property of the [rest of the] specification. Because of its semantical aspect, the methodology of non-conservative constructors is more advanced and sophisticated than the conservative one. However it might be very useful in dealing with non-terminating computations[12].

[11] The CafeOBJ proof score for this is rather simple; we leave it as exercise for the reader.

[12] This is very similar to the use of commutativity attribute for operations in classical algebraic specification, a good example of non-conservative methodology is given in (Dia98a).

Proving behavioural coherence. We now concentrate to an example illustrating the behavioural coherence methodology of conservative constructors. Consider the following behavioural specification of sets:



This specification has only one behavioural operation, namely the observation *_in_*. The hidden constructors *add*, *_U_*, *_&_*, and *neg* can be proved coherent by the following CafeOBJ proof score:

```
open .
 ops s1 s2 s1' s2' : -> Set . -- arbitrary sets as temporary constants
 ops e e' : -> Elt . -- arbitrary elements as temporary constants
 ceq S1 =*= S2 = true if (e in S1) == (e in S2) . -- definition of behavioural equivalence
 beq s1 = s1' . -- hypothesis
 beq s2 = s2' . -- hypothesis
red add(e, s1)  =*= add(e, s1') . -- beh coherence of add(_) for variable clash at Elt
red add(e', s1)  =*= add(e', s1') . -- beh coherence of add(_) for no variable clash at Elt
red (s1 U s2) =*= (s1' U s2') . -- beh coherence of _U_
red (s1 & s2) =*= (s1' & s2') . -- beh coherence of _&_
red neg(s1) =*= neg(s1') . -- beh coherence of neg_
close
```

Notice the simplicity of this proof score which uses the built-in default coinduction relation =*= which in practice is oftenly the behavioural equivalence. Once the coherence of the hidden constructors is formally proved, their coherence declarations are added to the specification, thus obtaining the final version of the specification under the methodology of conservative hidden constructors.

2.4. *Behavioural Verification*

One of the great advantages of behavioural specification lies in the simplicity of the verification stage which sometimes contrasts sharply with the complexity of corresponding data type verifications. Sets are one of the examples showing clearly the greater simplicity of behavioural verifications. While the verification of set-theoretic properties in the data approach gets into a very complex induction process, behavioural properties of sets can be proved almost immediately. The following is the very simple CafeOBJ proof score for one of De Morgan laws:

```
open .
```

```
  op e : -> Elt .
  ops s1 s2 s3 : -> Set .
  ceq S1:Set =*= S2:Set = true if (e in S1) == (e in S2) . -- definition of behavioural equivalence
red neg(s1 U s2) =*= (neg(s1) & neg(s2)) . -- proof of de Morgan law
close
```

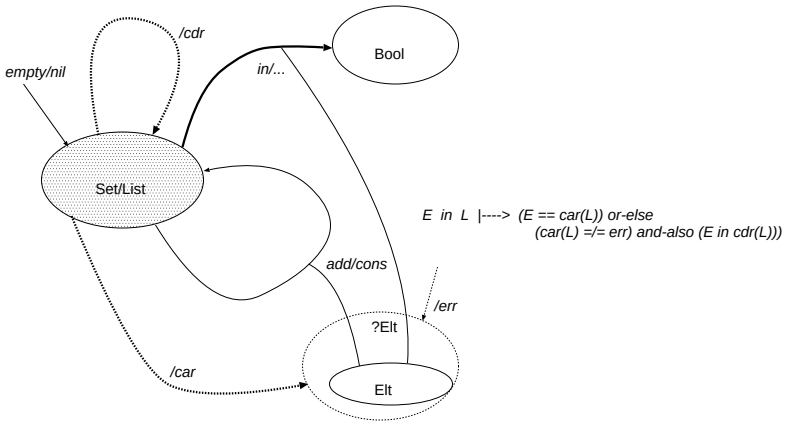
Behavioural rewriting. The execution of behavioural specifications is done by behavioural rewriting, which is a refinement of ordinary (term) rewriting that ensures the correctness of rewriting when using behavioural equations as rewrite rules. The basic condition of behavioural rewriting requires the existence of a path formed by behavioural or coherent operations on top of the redex. When inferring strict equalities, it is required in addition that the top of such path is of visible sort. For example, when proving the behavioural coherence of *add*,

```
red add(e, s1)  =*= add(e, s1') .
```

means a strict equality reduction. In this case the first behavioural equation of the corresponding proof score cannot be used as a first rewriting step since the condition of behavioural rewriting is not fulfilled. This triggers the use of the conditional equation instead as a first rewriting step, and only after this the use of behavioural equations of the proof score fall under the required condition.

2.5. *Behavioural refinement*

Object refinement in behavioural specification is a relax form of behavioural specification morphism (see (DF98b) for more details). As an example we show how behavioural lists refine behavioural sets, which corresponds to the basic intuition of sets implemented as lists:



For simplicity of presentation we considered here only the case of basic sets, without union, intersection, and negation[13]. The refinement of behavioural basic sets to lists was represented above by extending the graphical notation previously introduced with:

G3. *Refinement of sorts and operations is written by _/_ and sharing the same figure (disk or arrow) in the diagram.*

[13] Our example can be easily extended to union and intersection, but not so easily to negation.

G4.*Newly introduced sorts and operations are represented by dotted lines.*

In this refinement, the hidden sort Set is refined to the hidden sort List (this means that any state of the set object and be implemented by a state of the list object), *add* is refined to *cons*. The list object has the observation *car* and the action *cdr* as new behavioural operations and also adds the error handling. The set object observation *_in_* is refined to a derived observation (using some operational versions of the Boolean connectives). This refinement can be encoded in CafeOBJ by the following module import:

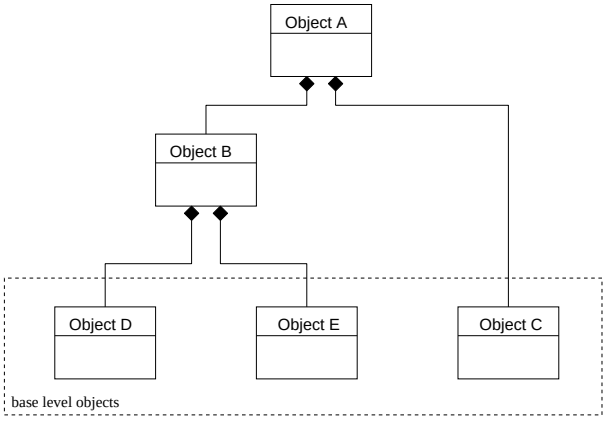
```
mod* LIST' {  protecting(LIST)
  op _in_ : Elt List -> Bool  {coherent} -- coherence provable from the rest of spec
  vars E E' : Elt
  var L : List
  eq E in L = (E == car(L)) or-else (car(L) =/= err  and-also  E in cdr(L)) . }
```

The following is the proof score for the fact that the mapping defined above is indeed a refinement, i.e., the property of *add* holds for *cons*:[14]

```
open LIST' .
  ops e e1 e2 : -> Elt . -- arbitrary elements as temporary constants
  op l : -> List . -- arbitrary list as temporary constant
  eq e1 in l = true . -- the basic case when the element does belong to the list
  eq e2 in l = false . -- the basic case when the element does not belong to the list
red e  in nil       == false . -- the nil case
red e1 in cons(e,l) == true  .
red e2 in cons(e,l) == false .
red e  in cons(e,l) == true  . -- the element clash case
close
```

3. Concurrent Object Composition in CafeOBJ

In this section we present the object composition method of CafeOBJ based on the behavioural specification paradigm. We present here a simplified method which does not use behavioural coherence. We use UML to represent object composition:



[14] This involves a small case analysis.

In the above UML figure, B is composed of D and E, A of B and C, and non-compound objects (i.e., objects with no components) are called *base level objects*. A composition in UML is represented by line tipped by a diamond, and if necessary, qualified by the numbers of components (**1** for one and ***** for many).

Projection operations from the hidden sort of the states of the compound object to the hidden sorts of the states of the component objects constitute the main technical concept underlying the CafeOBJ composition method; projection operations are related to the lines of UML figures. Projection operations are subject to some precise mathematical conditions (see (IMD 98, DF98b) for details), which can be informally summarized as follows:

— all actions of the compound object are related via the projection operations to actions in each of the component, and
— each observation of the compound object is related via the projection operations to an observation of some component.

In the compound objects we only define communication between the components; this means that the only equations at the level of the specification of the compound objects are the ones relating the actions and observations of the compound objects to those of the components as described above.

3.1. *Parallel connection*

The components of a composite object are connected (unsynchronized) in parallel if there is no synchronization between them. In order to define the concept of synchronization, we have to introduce the concept of *action group*. Two actions of a compound object are in the same action group when they change the state of the same component object via a projection operation. Synchronization appears when:

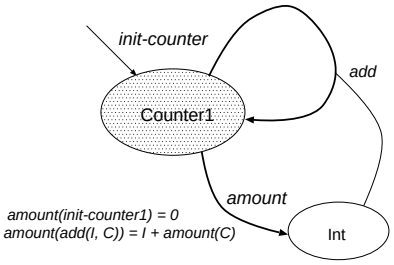
— there exists an overlapping between some action groups, or
— the projected state of the compound object (via a projection operation) depends on the state of a different (from the object corresponding to the projection operation) component.

The first case is sometimes called *broadcasting* and the second case is sometimes called *client-server computing*. In the unsynchronized case, we have full concurrency between all the components, which means that all the actions of the compound object can be applied concurrently, therefore the components can be implemented as distributed processes or concurrent processes with multi-thread which are based on asynchronous communications.

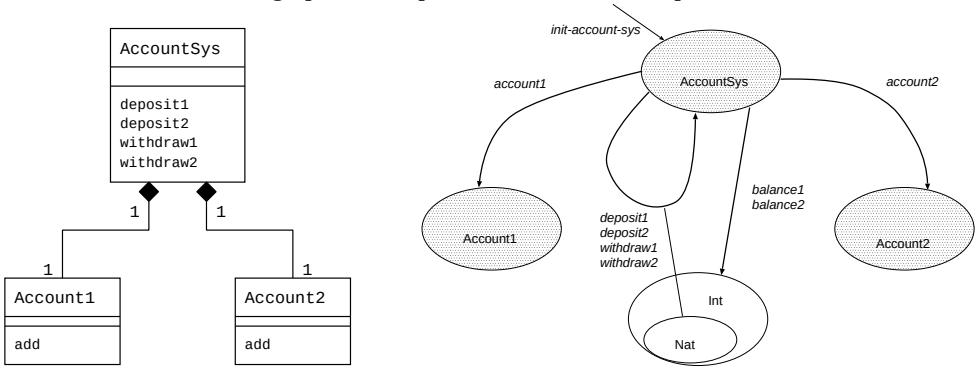
For unsynchronized parallel connection, we consider a bank account system example. Firstly, we consider a very simple bank account system which consists of a fixed numbers of individual accounts, lets actually consider the case of just two account. The specification of an account can be obtained just by renaming the specification COUNTER1 of a counter object with integers as follows

```
mod* ACCOUNT1 { protecting(COUNTER1 *{ hsort Counter -> Account1,
                                       op init-counter -> init-account1 })}
mod* ACCOUNT2 { protecting(COUNTER1 *{ hsort Counter -> Account2,
                                       op init-counter -> init-account2 })}
```

where COUNTER1 is represented in CafeOBJ graphical notation as follows:



We then compose these two account objects as in the following double figure containing both the UML and the CafeOBJ graphical[15] representation of this composition:



where *deposit1* and *withdraw1* are the actions for the first account, *balance1* is the observation for the first account, *account1* is the projection operation for the first account, and *deposit2*, *withdraw2*, *balance2*, and *account2* are the corresponding actions, observation, and projection operation for the second account. The equations for this parallel connection (composition) are as follows:

eq *balance1(AS) = amount(account1(AS))* .
eq *balance2(AS) = amount(account2(AS))* .
eq *account1(init-account-sys) = init-account1* .
eq *account1(deposit1(N, AS)) = add(N, account1(AS))* .
eq *account1(deposit2(N, AS)) = account1(AS)* .
eq *account1(withdraw1(N, AS)) = add(-(N), account1(AS))* .
eq *account1(withdraw2(N, AS)) = account1(AS)* .
eq *account2(init-account-sys) = init-account2* .
eq *account2(deposit1(N, AS)) = account2(AS)* .
eq *account2(deposit2(N, AS)) = add(N, account2(AS))* .
eq *account2(withdraw1(N, AS)) = account2(AS)* .
eq *account2(withdraw2(N, AS)) = add(-(N), account2(AS))* .

Notice that besides the first two equations relating the observations on the compound object to those on the components, the other equations relate the actions of the account system to the actions of the components. Remark that the actions corresponding to one component do not

[15] The CafeOBJ graphical representation corresponds to the module defining this object composition rather than to the "flattened" specification, hence the operations of the components are not included in the figure.

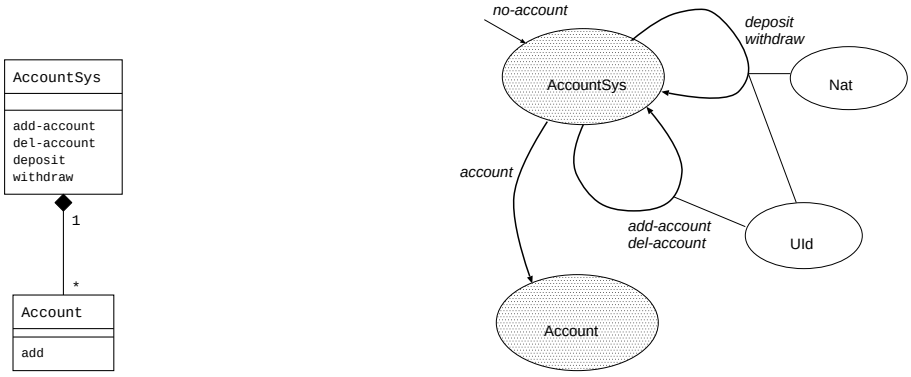
change the state of the second component (via the projection operation), hence this composition is unsynchronized. In fact these equations expressing the concurrency of composition need not be specified by the user, in their absence they may be generated internally by the system, thus reducing the specification of the composition to the essential information which should be provided by the user.

3.2. *Dynamic connection*

In this subsection, we extend the previous bank account system example to support an arbitrary number of accounts. The accounts are created or deleted dynamically, so we call such architecture pattern *dynamic connection* and we call the objects connected dynamically as *dynamic objects*. A dynamic object has an object identifier type as the arity of its initial state (which is quite a natural idea that in object-oriented programming languages, language systems automatically providing a pointer for each object when created). We therefore firstly extend the specification of the counter to a dynamic object

op *init-counter* : UId -> Counter

where UId is a sort for user identifiers. The structure of the new bank account system can be represented in UML and CafeOBJ graphical notation as follows:



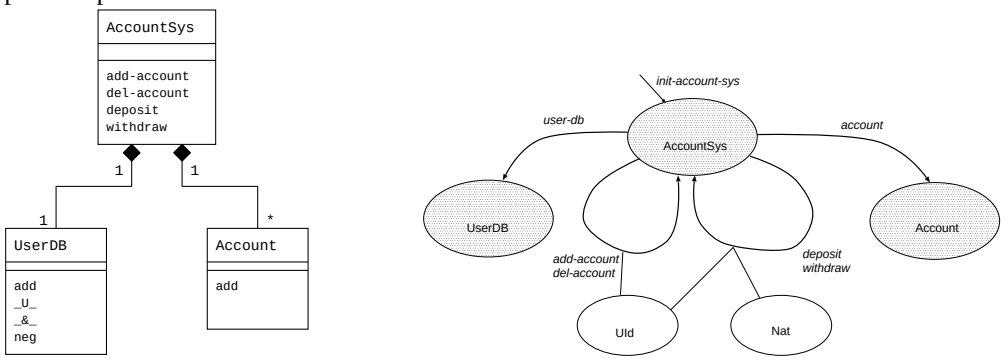
where the actions *add-account* and *del-account* maintain the user accounts. *add-account* creates accounts with some initial balance while *del-account* deletes the accounts; both of them are parameterized by the user identifiers UId. Each of *deposit* and *withdraw* is also parameterized by the user identifiers. Most notably, the projection operation for Account is also parameterized by UId. The initial state of AccountSys has no account, so it is mapped to the error state called *no-account*. Finally, the equations relate the actions of AccountSys to those of Account via the projection operation only when they correspond to the specified user account. Here is the essential part of the CafeOBJ code for the dynamic system of accounts specification:

eq *account(U, init-account-sys) = no-account* .
ceq *account(U, add-account(U', N, A)) = add(N, init-account(U))* if *U == U'* .
ceq *account(U, add-account(U', N, A)) = account(U, A)* if *U =/= U'* .
ceq *account(U, del-account(U', A)) = no-account* if *U == U'* .
ceq *account(U, del-account(U', A)) = account(U, A)* if *U =/= U'* .
ceq *account(U, deposit(U', N, A)) = add(N, account(U, A))* if *U == U'* .
ceq *account(U, deposit(U', N, A)) = account(U, A)* if *U =/= U'* .

ceq *account(U, withdraw(U', N, A)) = add(-(N), account(U, A))* if *U == U'* .
ceq *account(U, withdraw(U', N, A)) = account(U, A)* if *U =/= U'* .

3.3. *Synchronized parallel connection*

In this subsection, we add a user database (UserDB) to the bank account system example for having a more sophisticated user management mechanism. This enables querying whether an user already has an account in the bank account system. The following is the UML and CafeOBJ graphical representation of this:



where the users data base is obtained just by reusing (renaming) the set object of Section 2.3. The new account system compound object contains both synchronization patterns: broadcasting and client-server computing. *add-account* is related to *add* of Account by the projection operation for Account and it is also related to *add* of UserDB by the projection operation for UserDB. So, there is an overlapping of action groups (broadcasting). Also, *add-account* is related to *add* of Account by the projection operation for Account using the information of UserDB (client-server computing). The same holds for del-account.

The following CafeOBJ code represents the equations for the projection operation for UserDB:

eq *user-db(init-account-sys) = empty* .
eq *user-db(add-account(U, AS)) = add(U, user-db(AS))* .
eq *user-db(del-account(U, AS)) = neg(add(U, empty)) & user-db(AS)* .
eq *user-db(deposit(U, N, AS)) = user-db(AS)* .
eq *user-db(withdraw(U, N, AS)) = user-db(AS)* .

The following is the CafeOBJ code for the equations for the projection operation for Account, we skip here the equations of *deposit* and *withdraw* which are the same as in the previous example, and we also skip the equation for *del-account* which is similar to that of *add-account*:

eq *account(U, init-account-sys) = no-account* .
ceq *account(U, add-account(U', N, AS)) = add(N, init-account(U))*
if *U == U' and not(U in user-db(AS))* .
ceq *account(U, add-account(U', N, AS)) = account(U, AS)* if *U =/= U' or U in user-db(AS)* .

For add-account, we check whether the user is already registered and if not map it to add. If the user is already registered in UserDB, then skip.

It is interesting to mention that the same test

red *balance*(*'u*:UId, *add-account*(*'u*, 100, *deposit*(*'u*, 30, *add-account*(*'u*, 100, *init-account-sys*))))

gets different results in the previous account system example and in the current synchronized example due to the finer user management in the synchronized case.

3.4. *Compositionality of verifications*

In object-oriented programming, reusability of the source code is important, but in object-oriented specification, reusability of the proofs is also very important because of the verification process. We call this *compositionality of verifications* of components. In the CafeOBJ object composition method this is achieved by the fact that the behavioural equivalence on the compound object is the conjunction of the behavioural equivalences of the component objects (this is a Theorem which can be found in (IMD 98)). Therefore, in the case of a hierarchic object composition, the behavioural equivalence for the whole system is just the conjunction of the behavioural equivalences of the base level objects, which are generally rather simple.

For example, the behavioural equivalence for the bank account system is a conjunction of the behavioural equivalence Account (indexed by the user identifiers) and UserDB, and these two are checked automatically by the CafeOBJ system. This means that behavioural proofs for the bank account system are almost automatic, without having to go through the usual coinduction process. Therefore, the behavioural equivalence *_R[_]_* of AccountSys can be defined by the following CafeOBJ code:

```
mod BEQ-ACCOUNT-SYSTEM {  protecting(ACCOUNT-SYSTEM)
  op _R[_]_ : AccountSys UId AccountSys -> Bool
  vars AS1 AS2 : AccountSys
  var U : UId
  eq AS1 R[U] AS2 = account(U, AS1) =*= account(U, AS2) and
                    user-db(AS1) =*= user-db(AS2) . }
```

Notice the use of the parameterized relation for handling the conjunction indexed by the user identifiers.

Now, we will prove the true concurrency of withdrawals of two different users, which can be considered as a safety property for this system of bank accounts and which is formulated as the following commutativity behavioural property:

$$withdraw(u1, n1, withdraw(u2, n2, as)) \sim withdraw(u2, n2, withdraw(u1, n1, as))$$

The following CafeOBJ code builds the proof tree containing all possible cases formed by orthogonal combinations of atomic cases for the users with respect to their membership to the user accounts data base. The basic proof term is *TERM*. The automatic generation of the proof tree (*RESULT*)is done by a meta-level encoding in CafeOBJ by using its rewrite engine for one-directional construction of the proof tree (this process uses the rewriting logic feature of CafeOBJ, hence the use of transitions (trans) rather than equations).

```
mod PROOF-TREE { protecting(BEQ-ACCOUNT-SYSTEM)
  ops n1 n2 : -> Nat  -- arbitrary amounts for withdrawal
  ops u u1 u1' u2 u2' : -> UId  -- arbitrary user identifiers
  op as : -> AccountSys  -- arbitrary state of the account system
  eq u1 in user-db(as) = true .  -- first user is in the data base
  eq u2 in user-db(as) = true .  -- second user is in the data base
  eq u1' in user-db(as) = false .  -- first user is not in the data base
```

```
  eq u2' in user-db(as) = false .  -- second user is not in the data base
  vars U U1 U2 : Uld
  op TERM : Uld Uld Uld -> Bool  -- basic proof term
  trans TERM(U, U1, U2) => withdraw(U1, n1, withdraw(U2, n2, as)) R[U]
                           withdraw(U2, n2, withdraw(U1, n1, as)) .
  op TERM1 : Uld Uld -> Bool
  trans TERM1(U, U1) => TERM(U, U1, u2) and TERM(U, U1, u2') .
  op TERM2 : Uld -> Bool
  trans TERM2(U) => TERM1(U, u1) and TERM1(U, u1') .
  op RESULT : -> Bool  -- final proof term
  trans RESULT => TERM2(u1) and TERM2(u1') and TERM2(u) . }
```

The execution of the proof term *RESULT* gives true after the system performs 233 rewrites.

4. Conclusions and Future Work

In this paper we presented the CafeOBJ object-oriented methodology for component-based specification and verification which is based on the CafeOBJ behavioural abstraction paradigm. We also presented the basic behavioural specification methodology in CafeOBJ and gave a brief overview of the CafeOBJ language, system and specification environment.

Future work in this area will further explore and refine the current CafeOBJ methodologies exposed here with the aim of creating an industrial tool around these methodologies containing an industrial-oriented tutorial, a GUI interface probably based on the current CafeOBJ graphical notation, a graphical proof environment, etc.

References

BURSTALL, Rod ; DIACONESCU, Răzvan: Hiding and Behaviour: an Institutional Approach. **In:** ROSCOE, A. W. (Ed.): *A Classical Mind: Essays in Honour of C.A.R. Hoare*. Prentice-Hall, 1994. –

BURSTALL, Rod ; GOGUEN, Joseph: The Semantics of Clear, a Specification Language. **In:** BJORNER, Dines (Ed.): *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification.* Springer, 1980. – Lecture Notes in Computer Science, Volume 86.

CLAVEL, Manuel ; EKER, Steve ; LINCOLN, Patrick ; MESEGUER, Jose: Principles of Maude. **In:** *Electronic Notes in Theoretical Computer Science* 4 (1996). – Proceedings, First International Workshop on Rewriting Logic and its Applications. Asilomar, California, September 1996.

DIACONESCU, Răzvan ; FUTATSUGI, Kokichi: Logical Foundations of CafeOBJ. (1998). – Submitted to publication.

DIACONESCU, Răzvan ; FUTATSUGI, Kokichi: *AMAST Series in Computing*. Vol. 6 : CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. World Scientific, 1998

DIACONESCU, Răzvan ; FUTATSUGI, Kokichi ; IIDA, Shusaku: CafeOBJ Jewels. **In:** *Proceedings of the CafeOBJ Symposium'98* CafeOBJ Project, 1998

DIACONESCU, Răzvan ; GOGUEN, Joseph ; STEFANEAS, Petros: Logical Support for Modularisation. **In:** HUET, Gerard (Ed.) ; PLOTKIN, Gordon (Ed.): *Logical Environments*. Cambridge, 1993. – Proceedings of a Workshop held in Edinburgh, Scotland, May 1991, pp. 83–130

DIACONESCU, Răzvan: Category-based Semantics for Equational and Constraint Logic Programming. 1994. – DPhil thesis, University of Oxford

DIACONESCU, Răzvan: Foundations of Behavioural Specification in Rewriting Logic. **In:** *Electronic Notes in Theoretical Computer Science* 4 (1996). – Proceedings, First International Workshop on Rewriting Logic and its Applications. Asilomar, California, September 1996.

DIACONESCU, Răzvan: Behavioural Coherence in Object-Oriented Algebraic Specification / Japan Advanced Institute for Science and Technology. 1998 (IS-RR-98-0017F). Submitted to publication.

DIACONESCU, Răzvan: Extra Theory Morphisms for Institutions: logical semantics for multi-paradigm languages. **In:** *J. of Applied Categorical Structures* 6 (1998), Nr. 4, pp. 427–453. –

FUTATSUGI, Kokichi ; GOGUEN, Joseph ; JOUANNAUD, Jean-Pierre ; MESEGUER, Jose: Principles of OBJ2. **In:** *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, ACM, 1985, pp. 52–66

FUTATSUGI, Kokichi ; OKADA, Koji: Specification Writing as Construction of Hierarchically Structured Clusters of Operators. **In:** *Proceedings, 1980 IFIP Congress*. IFIP, 1980, pp. 287–292

FUTATSUGI, Kokichi ; SAWADA, Toshimi: A Preliminary View of CafeOBJ: a Multi-Paradigm Language for Cafe Environment. **In:** *Proc. of Urmqi International CASE Symposium*, 1992

GOGUEN, Joseph ; BURSTALL, Rod: Institutions: Abstract Model Theory for Specification and Programming. **In:** *Journal of the Association for Computing Machinery* 39 (1992), January, Nr. 1, pp. 95–146

GOGUEN, Joseph ; DIACONESCU, Răzvan: An Oxford Survey of Order Sorted Algebra. **In:** *Mathematical Structures in Computer Science* 4 (1994), Nr. 4, pp. 363–392

GOGUEN, Joseph ; DIACONESCU, Răzvan: Towards an Algebraic Semantics for the Object Paradigm. **In:** EHRIG, Harmut (Ed.) ; OREJAS, Fernando (Ed.): *Recent Trends in Data Type Specification* Vol. 785. Vol. 785, Springer, 1994, pp. 1–34

GOGUEN, Joseph ; MESEGUER, José: Eqlog: Equality, Types, and Generic Modules for Logic Programming. **In:** DEGROOT, Douglas (Ed.) ; LINDSTROM, Gary (Ed.): *Logic Programming: Functions, Relations and Equations*. Prentice-Hall, 1986. –

GOGUEN, Joseph ; MESEGUER, José: Unifying Functional, Object-Oriented and Relational Programming, with Logical Semantics. **In:** SHRIVER, Bruce (Ed.) ; WEGNER, Peter (Ed.): *Research Directions in Object-Oriented Programming*. MIT, 1987. –

GOGUEN, Joseph ; MESEGUER, José: Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. **In:** *Theoretical Computer Science* 105 (1992), Nr. 2, pp. 217–273. –

GOGUEN, Joseph ; MALCOLM, Grant: A hidden agenda / University of California at San Diego. 1997 (CS97-538).

GOGUEN, Joseph: Memories of ADJ. **In:** *Bulletin of the European Association for Theoretical Computer Science* 36 (1989), October, pp. 96–102. – Guest column in the 'Algebraic Specification Column.'

GOGUEN, Joseph: *Theorem Proving and Algebra*. MIT, To appear.

GOGUEN, Joseph ; WINKLER, Timothy ; MESEGUER, José ; FUTATSUGI, Kokichi ; JOUANNAUD, Jean-Pierre: Introducing OBJ. **In:** GOGUEN, Joseph (Ed.): *Algebraic Specification with OBJ: An Introduction with Case Studies*. Cambridge. – To appear.

IIDA, Shusaku ; MATSUMOTO, Michihiro ; DIACONESCU, Răzvan ; FUTATSUGI, Kokichi ; LUCANU, Dorel: Concurrent Object Composition in CafeOBJ / Japan Advanced Institute for Science and Technology. 1998 (IS-RR-98-0009S). Submitted to publication.

LAWVERE, F. W.: An Elementary Theory of the Category of Sets. **In:** *Proceedings, National Academy of Sciences, U.S.A.* 52 (1964), pp. 1506–1511

MESEGUER, José: Conditional rewriting logic as a unified model of concurrency. **In:** *Theoretical Computer Science* 96 (1992), Nr. 1, pp. 73–155

MESEGUER, José: Membership Algebra as a Logical Framework for Equational Specification. **In:** PARISI-PRESSICE, F. (Ed.): *Proc. WADT'97*, Springer, 1998, pp. 18–61