Computing and Informatics, Vol. 22, 2003, 1001–1025, V 2003-Aug-26

CafeOBJ: LOGICAL FOUNDATIONS AND METHODOLOGIES

Răzvan Diaconescu

Institute of Mathematics "Simion Stoilow"
PO Box 1-764, Bucharest 014700, Romania
e-mail: Razvan.Diaconescu@imar.ro

Kokichi Futatsugi

Graduate School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
e-mail: kokichi@jaist.ac.jp

Kazuhiro Ogata

NEC Software Hokuriku, Ltd. / JAIST
e-mail: ogatak@acm.org

Abstract. CafeOBJ is an executable industrial strength multi-logic algebraic specification language which is a modern successor of OBJ and incorporates several new algebraic specification paradigms.

In this paper we survey its logical foundations and present some of its methodologies.

Keywords: CafeOBJ, algebraic specification, institutions, abstract machines

1 INTRODUCTION

CafeOBJ is an *executable* industrial strength algebraic specification language which is a modern successor of OBJ and incorporates several new algebraic specification paradigms. Its definition is given in [10], a presentation of its logical foundations can be found in [12], and a presentation of some methodologies developed around CafeOBJ can be found in [13, 14]. CafeOBJ is intended to be mainly used for system specification, formal verification of specifications, rapid prototyping, or even programming.

In the first part of this paper we survey the mathematical foundations of CafeOBJ, while in the second part we survey some of its methodologies, including some recent ones.

Let us briefly overview some of CafeOBJ most important features.

Equational Specification and Programming.

Equational specification and programming is inherited from OBJ [23, 16] 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.

Behavioural Specification.

Behavioural specification [33, 34, 19, 20, 11, 24] provides a generalisation of ordinary 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 [20] 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 generalises process algebra and transition systems (see [20]). CafeOBJ behavioural specification paradigm is based on *coherent hidden algebra* (abbreviated 'CHA') of [11], which is both a simplification and extension of classical hidden algebra of [20] in several directions, most notably by allowing operations with multiple hidden sorts in the arity. Coherent hidden algebra comes very close to the "observational logic" of Bidoit and Hennicker [24].

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

- hidden sorts (for states of systems),

[1] Although this paradigm may be used as programming, from the applications point of view, this aspect is secondary to its specification side.

- behavioural operations (for direct “actions” and “observations” on states of systems),
- behavioural coherence declarations for (non-behavioural) operations (which may be either derived (indirect) “observations” or “constructors” on states of systems), and
- behavioural axioms (stating behavioural satisfaction).

The main behavioural proof method is based on *coinduction*. In CafeOBJ, coinduction can be used either in the classical hidden algebra sense [20] for proving behavioural equivalence of states of objects, or for proving behavioural transitions (which appear when applying behavioural abstraction to rewriting logic).

Besides language constructs, CafeOBJ supports behavioural specification and verification by several methodologies. CafeOBJ currently highlights a methodology for concurrent object composition which features high reusability not only of specification code but also of verifications [10, 25]. Behavioural specification in CafeOBJ may also be effectively used as an object-oriented (state-oriented) alternative for classical data-oriented specifications. Experiments seem to indicate that an object-oriented style of specification even of basic data types (such as sets, lists, etc.) may lead to higher simplicity of code and drastic simplification of verification process [10].

Behavioural specification is reflected at the execution level by the concept of *behavioural rewriting* [10, 11] which refines ordinary rewriting with a condition ensuring the correctness of the use of behavioural equations in proving strict equalities.

Rewriting Logic Specification.

Rewriting logic specification in CafeOBJ is based on a simplified version of Meseguer's *rewriting logic* (abbreviated as ‘RWL’) [27] 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 [4], the current CafeOBJ design does not fully support *labelled* 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 with dynamic definition encoding into equational logic both the proof theory of RWL and the user defined transitions (rules). At the level of the semantics, this amounts to the fact that the CafeOBJ RWL models are preorders rather than categories. This avoids many of the semantical complications resulting from the labelled version of RWL.

¿¿From a methodological perspective, CafeOBJ develops the use of RWL transitions for specifying and verifying the properties of *declarative encoding of algorithms* (see [10]) as well as for specifying and verifying transition systems. The restriction of RWL to its unlabelled version is also motivated by the fact that RWL plays only a secondary importance role in CafeOBJ methodologies.

Module System.

The principles of the CafeOBJ module system are inherited from OBJ which builds on ideas first realized in the language Clear [2], most notably institutions [17, 15]. CafeOBJ module system features

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

However, the concrete design of the language revises the OBJ view on importation modes and parameters [10].

Type System and Partiality.

CafeOBJ has a type system that allows subtypes based on *order sorted algebra* (abbreviated 'OSA') [21, 18]. 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. 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') [28].

2 INSTITUTIONAL SEMANTICS

Today one of the fundamental principles of algebraic specification research and development is that each algebraic specification and programming languge or system has an underlying logic in which all language constructs can be rigorously defined as mathematical entities and such that the semantics of specifications or programs is given by the model theory of this underlying logic. All modern algebraic specification languges, including CafeOBJ, follow strictly this principle, other two important modern algebraic specification languages being CASL [31] and Maude [4].

On the other hand, there is a very big number of algebraic specification languages in use, some of them tailored to specific classes of applications, hence a large class of logics underlying algebraic specification languages. However much of the algebraic specification phenomena is independent of the actual language and its underlying logic. This potential to do algebraic specification at a general level is realized by the theory of institutions [17], which is a categorical abstract model theoretic meta-theory of logics originally intended for specification and programming, but also very suitable for model theory [9, 8, 36].

The use of the concept of institution in algebraic specification is manifold:

- It provides a rigorous concept of logic underlying algebraic specification languages, a logic thus being a mathematical entity.
- It provides a framework for developing basic algebraic specification concepts and results independently of the actual underlying logic. This leads to greater conceptual clarity, apropriate uniformity and unity, with the benefit of a simpler and more efficient top-down approach on algebraic specification theory which contrasts the conventional bottom-up approach.
- It provides a framework for rigorous translations, encodings, and representations between algebraic specification systems via various morphism concepts between institutions.

2.1 Institutions

Institution theory assumes some familiarity with category theory. We generally use the same notations and terminology as Mac Lane [26], except that composition is denoted by ";" and written in the diagrammatic order. The application of functions (functors) to arguments may be written either normally using parentheses, or else in diagrammatic order without parentheses, or, more rarely, by using sub-scripts or super-scripts. The category of sets is denoted as $\mathbb{S}et$, and the category of categories[2] as $\mathbb{C}at$. The opposite of a category $\mathbb{C}$ is denoted by $\mathbb{C}^{\mathrm{op}}$. The class of objects of a category $\mathbb{C}$ is denoted by $|\mathbb{C}|$; also the set of arrows in $\mathbb{C}$ having the object a as source and the object b as target is denoted as $\mathbb{C}(a, b)$.

Definition 1. An *institution* $(\mathbb{S}ign, Sen, \mathrm{MOD}, \models)$ consists of

1. a category $\mathbb{S}ign$, whose objects are called *signatures*,
2. a functor $Sen\colon \mathbb{S}ign \to \mathbb{S}et$, giving for each signature a set whose elements are called *sentences* over that signature,
3. a functor $\mathrm{MOD}\colon \mathbb{S}ign^{\mathrm{op}} \to \mathbb{C}at$ giving for each signature Σ a category whose objects are called Σ-*models*, and whose arrows are called Σ-(*model*) *homomorphisms*, and
4. a relation $\models_\Sigma \subseteq |\mathrm{MOD}(\Sigma)| \times Sen(\Sigma)$ for each $\Sigma \in |\mathbb{S}ign|$, called Σ-*satisfaction*,

such that for each morphism $\varphi\colon \Sigma \to \Sigma'$ in $\mathbb{S}ign$, the *satisfaction condition*

[2] We steer clear of any foundational problem related to the "category of all categories"; several solutions can be found in the literature, see, for example [26].

$$M' \models_{\Sigma'} Sen(\varphi)(e) \text{ iff } \mathrm{MOD}(\varphi)(M') \models_{\Sigma} e$$

holds for each $M' \in |\mathrm{MOD}(\Sigma')|$ and $e \in Sen(\Sigma)$. We may denote the reduct functor $\mathrm{MOD}(\varphi)$ by $\_\restriction_\varphi$ and the sentence translation $Sen(\varphi)$ simply by $\varphi(\_)$. When $M = M'\restriction_\varphi$ we say that M' is an *expansion of M along φ*.

The signatures of the institution provide the syntactic entities for the specification language, the models provide possible implementations, the sentences are formal statements encoding the properties of the implementations, and the satisfaction relation tells when a certain implementation satisfies a certain property.

The institution underlying CafeOBJ is defined in [12].

2.2 Specifications

The concept of CafeOBJ specification is a special case of structured specification in an arbitary institution instantiated to the CafeOBJ institution. Our institution-independent structured specifications follows [35], however CafeOBJ specifications can be constructed by employing only a subset of the specification building operations defined in [35].[3]

Definition 2. Given an institution $(\mathbb{S}ign, Sen, \mathrm{MOD}, \models)$, its *structured specifications* (or just *specifications* for short) are defined from the finite presentations by iteration of the specification building operators presented below. The semantics of each specification SP is given by its *signature* $Sig[\mathrm{SP}]$ and its category of *models* $Mod[\mathrm{SP}]$, where $Mod[\mathrm{SP}]$ is a full subcategory of $\mathrm{MOD}(Sig[\mathrm{SP}])$.

PRES. Each finite *presentation* (Σ, E) (i.e. Σ is a signature and E is a finite set of Σ-sentences) is a specification such that

- $Sig[(\Sigma, E)] = \Sigma$, and
- $Mod[(\Sigma, E)] = \mathrm{MOD}(\Sigma, E)$.[4]

UNION. For any specifications SP_1 and SP_2 such that $Sig[\mathrm{SP}_1] = Sig[\mathrm{SP}_2]$ we can take their *union* $\mathrm{SP}_1 \cup \mathrm{SP}_2$ with

- $Sig[\mathrm{SP}_1 \cup \mathrm{SP}_2] = Sig[\mathrm{SP}_1] = Sig[\mathrm{SP}_2]$, and
- $Mod[\mathrm{SP}_1 \cup \mathrm{SP}_2] = Mod[\mathrm{SP}_1] \cap Mod[\mathrm{SP}_2]$.

TRANS. For any specification SP and and signature morphism $\varphi\colon Sig(\mathrm{SP}) \to \Sigma'$ we can take its *translation along φ* denoted by $\mathrm{SP} \star \varphi$ and such that

- $Sig[\mathrm{SP} \star \varphi] = \Sigma'$, and
- $Mod[\mathrm{SP} \star \varphi] = \{M' \in \mathrm{MOD}(\Sigma') \mid M'\restriction_\varphi \in Mod[\mathrm{SP}]\}$.

[3] CafeOBJ specifications do not involve the "derivation" building operation.

[4] $\mathrm{MOD}(\Sigma, E)$ is the subcategory of all Σ-models satisfying all sentences in E.

FREE. For any specification SP' and signature morphism $\varphi\colon \Sigma \to Sig[\mathrm{SP}']$ we can take the *persistently free specification of* SP' *along* φ denoted SP'^{φ} and such that

- $Sig[\mathrm{SP}'^{\varphi}] = Sig[\mathrm{SP}']$, and
- $Mod[\mathrm{SP}'^{\varphi}] =$
  $\{M' \in Mod[\mathrm{SP}'] \mid M'$ strongly persistently $\beta_{\mathrm{SP}'};\mathrm{MOD}(\varphi)$-free $\}$, where $\beta_{\mathrm{SP}'}$ is the subcategory inclusion $Mod[\mathrm{SP}'] \to \mathrm{MOD}(Sig[\mathrm{SP}'])$.

The strongly persistent freeness property says that for each $N' \in Mod[\mathrm{SP}']$ and for each model homomorphism $h\colon M'\restriction_{\varphi} \to N'\restriction_{\varphi}$ there exists an unique model homomorphism $h'\colon M' \to N'$ such that $h'\restriction_{\varphi} = h$.

Definition 3. A *specification morphism* $\varphi\colon \mathrm{SP}_1 \to \mathrm{SP}_2$ between specifications SP_1 and SP_2 is a signature morphism $\varphi\colon Sig[\mathrm{SP}_1] \to Sig[\mathrm{SP}_2]$ such that $M\restriction_{\varphi} \in Mod[\mathrm{SP}_1]$ for each $M \in Mod[\mathrm{SP}_2]$.

Specifications and their morphisms form a category $\mathbb{S}pec$.

With the exception of 'including' or 'using' imports (see [10]), any CafeOBJ specification construct can be reduced to the kernel specification building language of Definition 2. In the case of initial denotations, 'including' and 'using' imports can be included by adding corresponding variants of the building operation FREE.

For example, CafeOBJ imports correspond to specification inclusions (a simple import can be obtained as union (UNION) between a structured specification and a presentation (PRES)), module parameters to specification injections, 'views' to arbitrary specification morphisms, parameter instantiations to specification pushouts (obtained by translations (TRANS) and union (UNION)), modules with initial denotation are obtained as free specifications (FREE), etc.

3 THE CAFEOBJ INSTITUTION

3.1 Grothendieck institutions

Institution morphisms

CafeOBJ is a multi-logic language. This means that different features of CafeOBJ require different underlying institutions. For example, behavioural specification has coherent hidden algebra [11] as underlying institution, while rewriting logic specification has rewriting logic as underlying institution. Both institutions are in fact extensions of the more conventional equational logic institution. On the other hand, they can be combined to 'coherent hidden rewriting logic' which extends both of them. Other features of CafeOBJ require other institutions. Therefore, consequently to its multi-logic aspect, CafeOBJ involves a system of institutions and extension relationships between them rather than a single institution.

The solution to the multi-logic aspect of CafeOBJ is given by the concept of *Grothendieck institution* which flattens the underlying system of institutions to a single institution in which the flattened components still retain their identity. Grothendieck institutions have been invented in [7], but their spirit already appeared in [5], and although initially motivated by CafeOBJ semantics, they provide the solution for the semantics of any multi-logic language. For example, CASL, when used together with its extensions, has also adopted Grothendieck institutions as its semantics [29].

Institution morphism [17] provide the necessary concept for relating different institutions:

Definition 4. An *institution morphism* $(\Phi, \alpha, \beta)\colon (\mathbb{S}ign', Sen', \mathrm{MOD}', \models') \to (\mathbb{S}ign, Sen, \mathrm{MOD}, \models)$ consists of

1. a functor $\Phi\colon \mathbb{S}ign' \to \mathbb{S}ign$,
2. a natural transformation $\alpha\colon \Phi; Sen \Rightarrow Sen'$, and
3. a natural transformation $\beta\colon \mathrm{MOD}' \Rightarrow \Phi^{\mathrm{op}}; \mathrm{MOD}$

such that for any signature Σ' the following *satisfaction condition* holds

$$M' \models'_{\Sigma'} \alpha_{\Sigma'}(e) \ \text{ iff } \ \beta_{\Sigma'}(M') \models_{\Sigma'\Phi} e$$

for any model $M' \in \mathrm{MOD}'(\Sigma')$ and any sentence $e \in Sen(\Sigma'\Phi)$.

An *adjoint* institution morphism is an institution morphism such that the functor $\Phi\colon \mathbb{S}ign' \to \mathbb{S}ign$ has a left adjoint.[5]

The institutions and their morphisms with the obvious composition form a category denoted $\mathbb{I}ns$.

This type of structure preserving institution mapping, introduced already in the seminal paper [17], has a forgetful flavour in that it maps from a "richer" institution to a"poorer" institution. The dual concept of institution mapping, called *comorphism* [22] in which the mapping between the categories of signatures is reversed, can be interpreted in the actual examples as embedding a "poorer" institution into a "richer" one. Any adjunction between the categories of signatures determines an 'duality' pair of institution morphism and institution comorphism; this has been observed first time in [38] and [1]. Below we may notice that all institution morphisms involved in the semantics of CafeOBJ are adjoint.

Indexed institutions

Let us now recall the concept of *indexed category* [32]. A good reference for indexed categories also discussing applications to algebraic specification theory is [37]. An *indexed category* [37] is a functor $B\colon I^{\mathrm{op}} \to \mathbb{C}at$; sometimes we denote $B(i)$ as B_i (or B^i) for an index $i \in |I|$ and $B(u)$ as B^u for an index morphism $u \in I$. The following 'flattening' construction providing the canonical fibration associated to an

[5] Adjoint institution morphisms have been previously called 'embedding' institution morphisms in [5] and [7].

indexed category is known under the name of the *Grothendieck construction*, and plays an important role in mathematics. Given an indexed category $B\colon I^{\mathrm{op}} \to \mathbb{C}at$, let $B^\sharp$ be the *Grothendieck category* having $\langle i, \Sigma\rangle$, with $i \in |I|$ and $\Sigma \in |B_i|$, as objects and $\langle u, \varphi\rangle\colon \langle i, \Sigma\rangle \to \langle i', \Sigma'\rangle$, with $u \in I(i, i')$ and $\varphi\colon \Sigma \to \Sigma' B^u$, as arrows. The composition of arrows in $B^\sharp$ is defined by $\langle u, \varphi\rangle; \langle u', \varphi'\rangle = \langle u;u', \varphi;(\varphi' B^u)\rangle$.

Indexed institutions [7] extend indexed categories to institutions.

Definition 5. Given a category I of indices, an *indexed institution* $\mathcal{J}$ is a functor $\mathcal{J}\colon I^{\mathrm{op}} \to \mathbb{I}ns$.

For each index $i \in |I|$ let us denote the institution $\mathcal{J}^i$ by $(\mathbb{S}ign^i, \mathrm{MOD}^i, Sen^i, \models^i)$ and for each index morphism $u \in I$ let us denote the institution morphism $\mathcal{J}^u$ by $(\Phi^u, \alpha^u, \beta^u)$.

Grothendieck institutions

Grothendieck institutions [7] extend the flattening Grothendieck construction from indexed categories to indexed institutions.

Definition 6. The *Grothendieck institution* $\mathcal{J}^\sharp$ of an indexed institution $\mathcal{J}\colon I^{\mathrm{op}} \to \mathbb{I}ns$ is defined as follows:

1. its category of signatures $\mathbb{S}ign^\sharp$ is the Grothendieck category of the *indexed* category of signatures $\mathbb{S}ign\colon I^{\mathrm{op}} \to \mathbb{C}at$ of the indexed institution $\mathcal{J}$,
2. its model functor $\mathrm{MOD}^\sharp\colon (\mathbb{S}ign^\sharp)^{\mathrm{op}} \to \mathbb{C}at$ is given by
   - $\mathrm{MOD}^\sharp(\langle i, \Sigma\rangle) = \mathrm{MOD}^i(\Sigma)$ for each index $i \in |I|$ and signature $\Sigma \in |\mathbb{S}ign^i|$, and
   - $\mathrm{MOD}^\sharp(\langle u, \varphi\rangle) = \beta^u_{\Sigma'}; \mathrm{MOD}^i(\varphi)$ for each $\langle u, \varphi\rangle\colon \langle i, \Sigma\rangle \to \langle i', \Sigma'\rangle$,
3. its sentence functor $Sen^\sharp\colon \mathbb{S}ign^\sharp \to \mathbb{S}et$ is given by
   - $Sen^\sharp(\langle i, \Sigma\rangle) = Sen^i(\Sigma)$ for each index $i \in |I|$ and signature $\Sigma \in |\mathbb{S}ign^i|$, and
   - $Sen^\sharp(\langle u, \varphi\rangle) = Sen^i(\varphi); \alpha^u_{\Sigma'}$ for each $\langle u, \varphi\rangle\colon \langle i, \Sigma\rangle \to \langle i', \Sigma'\rangle$,
4. $M \models^\sharp_{\langle i,\Sigma\rangle} e$ iff $M \models^i_\Sigma e$ for each index $i \in |I|$, signature $\Sigma \in |\mathbb{S}ign^i|$, model $M \in |\mathrm{MOD}^\sharp(\langle i, \Sigma\rangle)|$, and sentence $e \in Sen^\sharp(\langle i, \Sigma\rangle)$.

By the satisfaction condition of the institution $\mathcal{J}^i$ for each index $i \in |I|$ and the satisfaction condition of the institution morphism $\mathcal{J}^u$ for each index morphism $u \in I$:

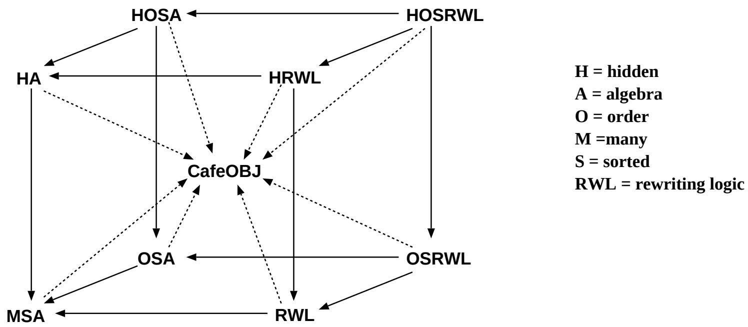
Proposition 1. $\mathcal{J}^\sharp$ is an institution and for each index $i \in |I|$, there exists a canonical institution morphism $(\Phi^i, \alpha^i, \beta^i)\colon \mathcal{J}^i \to \mathcal{J}^\sharp$ mapping any signature $\Sigma \in |\mathbb{S}ign^i|$ to $\langle i, \Sigma\rangle \in |\mathbb{S}ign^\sharp|$ and such that the components of α^i and β^i are identities.

By [5, 7], under adequate conditions, the important properties of institutions (including theory colimits, free construction, called liberality, model amalgamation, called exactness, inclusion systems) can be 'globalized' from the components of the indexed institution to the Grothendieck institution.

If one replaces institution morphisms to institution comorphisms, one can define the so-called ‘comorphism-based’ Grothendieck institutions [30]. When the institution morphisms of the indexed institution are adjoint, the Grothendieck institution and the corresponding comorphism-based Grothendieck institution are isomorphic. It will be easy to notice that this actually happens in the case of CafeOBJ.

3.2 The CafeOBJ cube

Now we are ready to define the actual CafeOBJ institution as the Grothendieck institution of the indexed institution below, called the CafeOBJ cube. (The actual CafeOBJ cube consists of the full arrows, the dotted arrows denote the morphisms from components of the indexed institution to the Grothendieck institution.)



The details of the institutions of the CafeOBJ cube can be found in [12]. Below we present them briefly.

The institution MSA of many-sorted algebra has the so-called ‘algebraic signatures’ (consisting of sets of sort symbols and sorted function symbols) as signatures, algebras interpreting the sort symbols as sets and the function symbols as functions, and (possibly conditional) universally quantified equations as sentences. The satisfaction between algebras and equations is the standard Tarskian satisfaction.

As in other algebraic specification languages, conditions of equations are encoded as Boolean-values terms, hence in reality MSA should be thought as a *constraint equational logic* in the sense of [6]. Alternatively, one may adopt membership equational logic [28] as the base equational logic institution.

OSA extends the MSA institution with order sortedness such that the set of sorts of a signature is a partially ordered set rather than a discrete set, and algebras interpret the subsort relationship as set inclusion. The forgetful institution morphism from OSA to MSA just forgets the order sortedness.

The institution RWL has the same signatures as MSA but the models interpret the sort symbols as preorders and the function symbols as preorder functors (i.e. functors between preorders). Besides equations, RWL has other sentences too, the so-called ‘transitions’ which can be regarded as of one-directional equations not obeying the symmetry rule. Their satisfaction by the models is determined by the preorder relation between the interpretation of the terms of the transition. The forgetful institution morphism from RWL to MSA essentially forgets the preorder relationship between the elements of models.

Signatures of the institution HA of 'coherent hidden algebra' [11] are special MSA signatures in which the set of sort symbols is divided into 'visible' sorts for the data and 'hidden sorts' for states (of abstract machines) and we mark a subset of the function symbols as 'behavioural'. Behavioural function symbols must have exactly one hidden sort in the arity. Besides ordinary strict equations, HA has also 'behavioural equations' and 'coherence declarations'. An algebra satisfies a behavioural equation when all strings of applications of behavioural functions to the interpretation of the sides of the equation give the same results in the visible sorts. Coherence declarations can be regarded as just abbreviations for a special type of conditional behavioural equations (see [11, 12]) for details. There is a forgetful institution morphism from HA to MSA forgetting the distinction between visibile and hidden.

These extensions of MSA towards three different paradigms can be all combined into HOSRWL (see [12] for details). All institutions of the CafeOBJ cube can be seen as sub-institution of HOSRWL by the adjoint comorphisms corresponding to the forgetful institution morphisms.

Various extensions of CafeOBJ can be further considered by transforming the CafeOBJ cube into a 'hyper-cube' and by flattening it to a Grothendieck institution.

Some comparison between CafeOBJ on one hand, and CASL and Maude on the other hand, two efforts very close to CafeOBJ, can be made at the level of their underlying institutions. Maude is based on labelled rewriting logic, built on top of membership equational logic, while CASL is based on order sorted partial first order logic.

Proof calculus

The proof calculus of CafeOBJ is obtained by flattening the 'indexed' proof calculus corresponding to the CafeOBJ cube, consisting of well known proof calculi for each of the CafeOBJ institutions. Recall that while the equational and rewriting logic proof calculi are sound and complete, the behavioural proof calculus is only sound [3]. It is rather easy to remark that by falttening, the soundness and the completeness of the proof calculus can be 'globalized' from the components of the indexed logic to the Grothendieck logic.

4 DESCRIPTION AND VERIFICATION OF DATA TYPES

Data types are described in terms of initial algebra [42]. In this section, three data types are used to explain how to describe data types and how to verify lemmas on the data types. Each of the data types is declared in one module. The data types are process IDs, labels and queues. The three data types are used in the next section, where we discuss a mutual exclusion program using a queue.

Suppose that we need an arbitrary number of process IDs instead of a fixed number of specific process IDs. Therefore, process IDs are declared as follows:

```
mod* PID {
   [Pid]
   op _=_ : Pid Pid -Bool {comm}
   var I : Pid
   eq (I = I) = true .
}
```

The module denotes an arbitrary set of process IDs, which is designated by keyword `mod*`. The keyword indicates that the module is a loose semantics declaration, meaning an arbitrary model (implementation) that respects all requirements written in the module.

Labels are the names given to atomic commands in the program. We need three specific labels, which are l1, l2 and cs. Therefore, labels are declared as follows:

```
mod! LABEL {
   [Label]
   ops l1 l2 cs : -Label
   op _=_ : Label Label -Bool {comm}
   var L : Label
   eq (L = L) = true .
   eq (l1 = l2) = false .
   eq (l1 = cs) = false .
   eq (l2 = cs) = false .
}
```

The module denotes the exact three labels, which is designated by keyword `mod!`. The keyword indicates that the module is a tight (initial) semantics declaration, meaning the smallest model (implementation) that respect all requirements written in the module.

Queues of arbitrary data types are defined, instead of those of a specific data type. Such queues are declared in a parameterized module. Therefore, we need to declare a module that is used as a formal parameter of the parameterized module. The module is declared as follows:

```
mod* EQTRIV {
   [Elt]
   op _=_ : Elt Elt -Bool {comm}
}
```

The formal parameter indicates the constraint of actual parameters with which the parameterized module is instantiated, but does not indicate a specific module. Therefore, module `EQTRIV` should be interpreted by loose semantics.

Then, the parameterized module is declared as follows:

```
mod! QUEUE (D :: EQTRIV) {
   [Queue]
   op empty  :              -Queue
   op __     : Elt.D Queue -Queue
   op put    : Queue Elt.D -Queue
   op get    : Queue        -Queue
   op top    : Queue        -Elt.D
   op empty? : Queue        -Bool
   op _\in_  : Elt.D Queue -Bool
   var Q : Queue
   vars X Y : Elt.D
   eq put(empty,X)  = X empty .
   eq put((Y Q),X)  = Y put(Q,X) .
   eq get(empty)    = empty .
   eq get((X Q))    = Q .
   eq top((X Q))    = X .
   eq empty?(empty) = true .
   eq empty?((X Q)) = false .
   eq X \in empty   = false .
   eq X \in (X Q)   = true .
   ceq X \in (Y Q)  = X \in Q if not(X = Y) .
}
```

Constant `empty` and juxtaposition operator `__` are the constructors of queues. We want queues to be made from these two constructors only. Therefore, module `QUEUE` should be interpreted by tight semantics.

Besides, in the next section, we need lemmas on queues, which are as follows:

Lemma 1 (of queues)**.** For any q : `Queue` and any x, y : `Elt.D`,

$$(\texttt{not empty?}(q) \texttt{ and } \texttt{top}(q) = x) \texttt{ implies } (\texttt{top}(\texttt{put}(q, y)) = x)\,, \tag{1}$$

$$\texttt{not empty?}(\texttt{put}(q, x))\,, \tag{2}$$

$$\texttt{empty?}(q) \texttt{ implies } (\texttt{not } x \texttt{ \\in } q)\,, \tag{3}$$

$$x \texttt{ \\in put}(q, x)\,, \tag{4}$$

$$(x \texttt{ \\in } q) \texttt{ implies } (x \texttt{ \\in put}(q, y))\,. \tag{5}$$

The first three are proved by case analyses only, and the remaining by structural induction on queues. All proofs are done by writing proof scores in CafeOBJ. Proof scores of (1) and (5) are shown in this paper.

In a module, say `LEMMA`, operators `lemma1` and `lemma5` are declared as follows:

```
op lemma1 : Queue Elt.D Elt.D -Bool
op lemma5 : Queue Elt.D Elt.D -Bool
```

Equations that let `lemma1` and `lemma5` denote (1) and (5) are declared as follows:

```
eq lemma1(Q,X,Y) = (not empty?(Q) and top(Q) = X implies top(put(Q,Y)) =
X) .
eq lemma5(Q,X,Y) = (X \in Q implies X \in put(Q,Y)) .
```

where `Q`, `X` and `Y` are CafeOBJ variables for `Queue`, `Elt.D` and `Elt.D`, respectively.

In module `LEMMA`, three constants are also declared as follows:

```
op q : -Queue
ops x y : -Elt.D
```

Constant `q` is used to denote an arbitrary `Queue`, and constants `x` and `y` to denote an arbitrary `Elt.D` in proof scores. The case can be split into multiple ones by declaring equations on these constants in proof scores. Let us consider that the case is split into two: one where `q` is empty, and the other where `q` is not. The former can be denoted by declaring the following equation:

```
eq q = empty .
```

The latter can be denoted by declaring the following equation:

```
eq (q = empty) = false .
```

There is another way of denoting the latter. To do this, two more constants that also denote arbitrary objects should be declared as follows:

```
op z : -Elt.D
op qq : -Queue
```

Then, the latter can be denoted by declaring the following equation:

```
eq q = z qq .
```

Moreover, the latter can be split into two more: one where `z` equals `x`, and the other where `z` does not. This case split can be done by declaring the following equations, each equation for each case:

```
eq z = x .
```

```
eq (z = x) = false .
```

It is time that you could read the proof scores of (1) and (5). The proof score of (1) is as follows:

```
open LEMMA -- Case 1
   eq q = empty .
   red lemma1(q,x,y) .
close
```

```
open LEMMA -- Case 2
   op z : -Elt.D .
   op qq : -Queue .
   eq q = z qq .
   red lemma1(q,x,y) .
close
```

Command `red` reduces a given term by regarding declared equations as left-to-right rewrite rules. In either case, we expect the given term to be reduced to `true`, and it is actually done.

The proof score of (5) is as follows:

```
open LEMMA -- Base case
   eq q = empty .
   red lemma5(q,x,y) .
close
```

```
open LEMMA -- Inductive case 1
   op z : -Elt.D .
   op qq : -Queue .
   eq q = z qq .
   eq z = x .
   red lemma5(q,x,y) .
close
```

```
open LEMMA -- Inductive case 2
   op z : -Elt.D .
   op qq : -Queue .
   eq q = z qq .
   eq (z = x) = false .
   red lemma5(qq,x,y) implies lemma5(q,x,y) .
close
```

Term `lemma5(qq,x,y)` denotes the inductive hypothesis.

Proof scores of (2), (3) and (4) can be written in a similar way. Once we prove these lemmas, we can declare them as equations in module `QUEUE`. The equations are as follows:

```
ceq (top(put(Q,Y)) = X) = true    if not empty?(Q) and top(Q) = X .
eq  empty?(put(Q,X))    = false .
ceq X \in Q             = false   if empty?(Q) .
eq  X \in put(Q,X)      = true .
ceq X \in put(Q,Y)      = true    if X \in Q .
```

which correspond to (1), (2), (3), (4) and (5), respectively.

Notes on equality between terms CafeOBJ provides built-in equality operator `_==_`. Given two terms s and t which sorts are the same, s `==` t is reduced to true if the results of reducing the two terms are the same, and to false otherwise, even if the two terms might denote the same data. If you are confident that your specification is confluent, it might be safe to use `_==_`. Besides, if `_==_` is used, more case split should be done. Suppose that two constants are used in a proof score, which are declared as follows:

```
ops x y : -Elt.D
```

In the proof score, `x` is treated as different from `y` if `_==_` is used, provided that no equations on `x` and `y` such as (`eq x = y .`) are declared. Therefore, we should consider another case where `x` equals `y`. If user-defined operator `_=_`, instead of `_==_`, is used as in this section, `x` and `y` are treated as a completely arbitrary `Elt.D`, namely that `x` may be the same as `y` or different from `y`, provided that no equations on `x` and `y` are declared.

5 DESCRIPTION AND VERIFICATION OF ABSTRACT MACHINES

Abstract machines are described in terms of coherent hidden algebra [11]. In this section, a system in which multiple processes execute a parallel program is used to explain how to describe abstract machines and how to verify that they have properties. The program supposedly solves the mutual exclusion problem, namely that it allows at most one process to enter the critical section, where resources such as I/O devises that have to be accessed by at most one process at any given time are used. The parallel program for process i is as follows:

l1: *put*(*queue*,i)
l2: **repeat until** *top*(*queue*) $= i$
Critical Section
cs: *get*(*queue*)

Process i repeatedly executes this program, namely that if the process at label cs executes *get*(*queue*), it moves to label l1. *queue* is a queue of process IDs, which is shared by all processes. Process ID i is put into the queue at the end by *put*(*queue*,i), and the top of the queue is obtained and deleted by *top*(*queue*) and *get*(*queue*), respectively. These operations are supposed to be done atomically. Besides, each iteration of the loop at label l2 is also supposed to be atomic.

5.1 Observational Transition Systems

The system under consideration is modeled in terms of a restricted type of coherent hidden algebra, which is called observational transition systems, or OTSs. OTSs are affected by UNITY [40].

We assume that there exists a universal state space called Υ. We also suppose that each data type used has been defined beforehand, including the equivalence between two data values v_1, v_2 denoted by $v_1 = v_2$. An OTS $\mathcal{S} = \langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ consists of:

- $\mathcal{O}$: A set of observable values. Each $o \in \mathcal{O}$ is a function $o : \Upsilon \to D$, where D is a data type and may be different for each observable value. Given an OTS $\mathcal{S}$ and two states $v_1, v_2 \in \Upsilon$, the equivalence between two states, denoted by $v_1 =_{\mathcal{S}} v_2$, w.r.t. $\mathcal{S}$ is defined as $v_1 =_{\mathcal{S}} v_2 \stackrel{\text{def}}{=} \forall o \in \mathcal{O}. o(v_1) = o(v_2)$.
- $\mathcal{I}$: The set of initial states such that $\mathcal{I} \subset \Upsilon$.
- $\mathcal{T}$: A set of conditional transition rules. Each $\tau \in \mathcal{T}$ is a function $\tau : \Upsilon/{=_{\mathcal{S}}} \to \Upsilon/{=_{\mathcal{S}}}$ on equivalence classes of Υ w.r.t. $=_{\mathcal{S}}$. Let $\tau(v)$ be the representative element of $\tau([v])$ for each $v \in \Upsilon$ and it is called *the successor state* of v w.r.t. τ. The condition c_τ for a transition rule $\tau \in \mathcal{T}$, which is a predicate on states, is called *the effective condition*. The effective condition is supposed to satisfy the following requirement: given a state $v \in \Upsilon$, if c_τ is false in v, namely τ is not *effective* in v, then $v =_{\mathcal{S}} \tau(v)$.

An OTS is described in CafeOBJ. Observable values are denoted by CafeOBJ observations, and transition rules by CafeOBJ actions.

An execution of $\mathcal{S}$ is an infinite sequence $v_0, v_1, \ldots$ of states satisfying:

- *Initiation*: $v_0 \in \mathcal{I}$.
- *Consecution*: For each $i \in \{0, 1, \ldots\}$, $v_{i+1} =_{\mathcal{S}} \tau(v_i)$ for some $\tau \in \mathcal{T}$.

A state is called *reachable* w.r.t. $\mathcal{S}$ iff it appears in an execution of $\mathcal{S}$. Let $\mathcal{R}_{\mathcal{S}}$ be the set of all the reachable states w.r.t. $\mathcal{S}$.

All properties considered in this section are invariants, which are defined as follows:

$$\textsf{invariant}\ p \stackrel{\text{def}}{=} (\forall v \in \mathcal{I}.\, p(v)) \land (\forall v \in \mathcal{R}_{\mathcal{S}}. \forall \tau \in \mathcal{T}. (p(v) \Rightarrow p(\tau(v)))) \,,$$

which means that predicate p is true in any reachable state of $\mathcal{S}$. Let $\boldsymbol{x}$ be all free variables except for one for states in p. We suppose that $\textsf{invariant}\, p$ is interpreted as $\forall \boldsymbol{x}.(\textsf{invariant}\, p)$ in this paper.

5.2 Description of the System

Two kinds of observable values and three kinds of transition rules are used to model the system under consideration, which are as follows:

- Observable values.
  - *queue* denotes the queue shared by all processes, which is initially empty.
  - pc_i ($i \in Pid$) denotes the label of a command that process i will execute next, which is initially l1.
- Transition rules.
  - $want_i$ ($i \in Pid$) denotes the command corresponding to label l1.
  - try_i ($i \in Pid$) denotes the command corresponding to label l2.
  - $exit_i$ ($i \in Pid$) denotes the command corresponding to label cs.

Pid is a set of process IDs.

The OTS modeling the system is described in module `QLOCK`, which imports modules `LABEL`, `PID` and `QUEUE(PID)`. `QUEUE(PID)` is the module that is `QUEUE` instantiated with `PID`. The signature of `QLOCK` is as follows:

```
*[Sys]*
-- any initial state
op init : -Sys
-- observations
bop pc    : Sys Pid -Label
bop queue : Sys -Queue
-- actions
bop want : Sys Pid -Sys
bop try  : Sys Pid -Sys
bop exit : Sys Pid -Sys
```

The state space Υ is represented by hidden sort `Sys`, observable values *queue* and pc_i by observations `queue` and `pc`, respectively, and transition rules $want_i$, try_i and $exit_i$ by actions `want`, `try` and `exit`, respectively. Constant `init` denotes any initial state of the OTS.

Equations defining the three actions are show, where `S` is a CafeOBJ variable for `Sys`, and `I` and `J` for `Pid`. Action `want` is defined with equations as follows:

```
op c-want : Sys Pid -Bool
eq c-want(S,I) = (pc(S,I) = l1) .
--
ceq pc(want(S,I),J)  = (if I = J then l2 else pc(S,J) fi) if c-want(S,I) .
ceq queue(want(S,I)) = put(queue(S),I)                    if c-want(S,I) .
ceq want(S,I)        = S                                  if not
c-want(S,I) .
```

Operator `c-want` denotes the effective condition of transition rule $want_i$.

Action `try` is defined with equations as follows:

```
op c-try : Sys Pid -Bool
eq c-try(S,I) = (pc(S,I) = l2 and top(queue(S)) = I) .
--
ceq pc(try(S,I),J)  = (if I = J then cs else pc(S,J) fi) if c-try(S,I) .
eq  queue(try(S,I)) = queue(S) .
ceq try(S,I)        = S                                  if not c-try(S,I) .
```

Operator `c-try` denotes the effective condition of transition rule try_i.

Action `exit` is defined with equations as follows:

```
op c-exit : Sys Pid -Bool
eq c-exit(S,I) = (pc(S,I) = cs) .
--
ceq pc(exit(S,I),J)  = (if I = J then l1 else pc(S,J) fi) if c-exit(S,I) .
ceq queue(exit(S,I)) = get(queue(S))                       if c-exit(S,I) .
ceq exit(S,I)        = S                                   if not
c-exit(S,I) .
```

Operator `c-exit` denotes the effective condition of transition rule $exit_i$.

5.3 Verification of the System

We verify that the system under consideration has the following invariant:

Claim 1 (Mutual Exclusion).

$$\textsf{invariant}\ (\texttt{pc}(s,i) = \texttt{cs and pc}(s,j) = \texttt{cs implies } i = j)\,. \tag{1}$$

This invariant means that at most one process can executes the critical section at any given time. To prove the invariant, we need three more invariants, which are as follows:

Claim 2.

$$\textsf{invariant}\ (\texttt{pc}(s,i) = \texttt{cs implies top}(\texttt{queue}(s)) = i)\,, \tag{2}$$
$$\textsf{invariant}\ (\texttt{pc}(s,i) = \texttt{l2 or pc}(s,i) = \texttt{cs implies not empty?}(\texttt{queue}(s)))\,, \tag{3}$$
$$\textsf{invariant}\ (\texttt{pc}(s,i) = \texttt{l2 implies } i \texttt{ \\in queue}(s))\,. \tag{4}$$

How to Construct Proof Scores

We briefly describe how to construct proof scores of invariants [49]. Suppose that all predicates and action operators takes only states as their arguments for simplicity. Invariants are often proved by induction on the number of transition rules applied. Suppose that we prove that the system has "$\textsf{invariant}\, p_1(s)$" by induction on the number of transition rules applied, where s is a free variable for states.

It is often impossible to prove $\textsf{invariant}\, p_1(s)$ alone. Suppose that it is possible to prove $\textsf{invariant}\, p_1(s)$ together with $n-1$ other predicates. Let the $n-1$ other predicates be $p_2(s), \ldots, p_n(s)$. That is, we prove $\textsf{invariant}\, p_1(s) \wedge \ldots \wedge p_n(s)$. Let $p(s)$ be $p_1(s) \wedge \ldots \wedge p_n(s)$.

Let us consider an inductive case in which it is shown that any transition rule denoted by CafeOBJ action operator a preserves $p(s)$. To this end, it is sufficient to show $p(s) \Rightarrow p(a(s))$. This formula can be proved compositionally. The proof of the formula is equivalent to the proofs of the n formulas:

$$\begin{aligned} p(s) &\Rightarrow p_1(\mathrm{a}(s)),\\ &\vdots\\ p(s) &\Rightarrow p_n(\mathrm{a}(s))\,. \end{aligned}$$

Moreover, it suffices to prove the following n formulas, if possible, instead of the previous n formulas:

$$\begin{aligned} p_1(s) &\Rightarrow p_1(a(s)), \\ &\vdots \\ p_n(s) &\Rightarrow p_n(a(s))\,. \end{aligned}$$

But, some of them may not be proved because their inductive hypotheses are too weak. Let $p_i(s) \Rightarrow p_i(a(s))$, where $0 \leq i \leq n$, be one of such formulas. Let SIH_i be a formula that is sufficient to strengthen the inductive hypothsis $p_i(s)$. SIH_i can be $p_{i_1}(s) \wedge \ldots \wedge p_{i_k}$, where $1 \leq i_1, \ldots, i_k \leq n$. Then, all we have to do is to prove $(SIH_i \wedge p_i(s)) \Rightarrow p_i(a(s))$.

Besides, we may have to split the case into muptiple subcases in order to prove $(SIH_i \wedge p_i(s)) \Rightarrow p_i(a(s))$. Suppose that the case is split into l subcases. The l subcases are denoted by l formulars $case^i_1, \ldots, case^i_l$, which should satisfy $(case^i_1 \vee \ldots \vee case^i_l) = \text{true}$. Then, the proof can be replaced with the l formulas:

$$\begin{aligned} &(case^i_1 \wedge SIH_i \wedge p_i(s)) \Rightarrow p_i(a(s)), \\ &\qquad\qquad\vdots \\ &(case^i_l \wedge SIH_i \wedge p_i(s)) \Rightarrow p_i(a(s)), \end{aligned}$$

SIH_i may not be needed for some subcases.

Proof scores of invariants are based what has been discussed. Let us consider that we write proof scores of the n invariants discussed. We first write a module, say `INV`, where $p_i(s)\,(i = 1, \ldots, n)$ is expressed as a CafeOBJ term as follows:

```
op inv_1 : H -Bool
...
op inv_n : H -Bool
eq inv_1(S) = p_1(S) .
...
eq inv_n(S) = p_n(S) .
```

where H is a hidden sort and S is a CafeOBJ variable for H. Term $p_i(S)\,(i = 1, \ldots, n)$ denotes $p_i(s)$.

We are going to mainly describe the proof of the ith invariant. Let *init* denote any initial state of the system. To show that $p_i(s)$ holds in any initial state, the following proof score is written:

```
open INV
  red inv_i(init) .
close
```

We next write a module, say `ISTEP`, where two constants s, s' are declared, denoting any state and the successor state after applying a transition rule in the state, and the predicates to prove in each inductive case are expressed as a CafeOBJ term as follows:

op $istep_1$: -Bool
...
op $istep_n$: -Bool
eq $istep_1$ = $inv_1(s)$ implies $inv_1(s')$.
...
eq $istep_n$ = $inv_n(s)$ implies $inv_n(s')$.

In each inductive case, the case is usually split into multiple subcases. Suppose that we prove that any transition rule denoted by CafeOBJ action operator a preserves $p_i(s)$. As described, the case is supposed to be split into the l subcases $case^i_1, \ldots, case^i_l$. Then, the CafeOBJ code showing that the transition rule preserves $p_i(s)$ for $case^i_j$ ($j = 1, \ldots, l$) looks like this:

open ISTEP
  Declare constants denoting arbitrary objects.
  Declare equations denoting $case^i_j$.
  Declare equations denoting facts if necessary.
  eq s' = $a(s)$.
  red $istep_i$.
close

Constants may be declared for denoting arbitrary objects. Equations are used to express $case^i_j$. If necessary, equations denoting facts about data structures used, etc. may be declared as well. The equation with s' as its left-hand side specifies that s' denotes the successor state after applying the transition rule denoted by a in the state denoted by s.

If $istep_i$ is reduced to `true`, it is shown that the transition rule preserves $p_i(s)$ in this case. Otherwise, we may have to strengthen the inductive hypothesis in the way described. Let $\mathtt{SIH}_i$ be the term denoting SIH_i. Then, instead of $istep_i$, we reduce the term ($\mathtt{SIH}_i$ and $inv_i(s)$) implies $inv_i(s')$, or $\mathtt{SIH}_i$ implies $istep_i$.

The way to construct proof scores can be also applied to proofs of data types.

Proof Scores

We partly show the proof of invariant (1). In module `INV`, the following operator is declared and defined:

```
op inv1 : Sys Pid Pid -Bool
eq inv1(S,I,J) = (pc(S,I) = cs and pc(S,J) = cs implies I = J) .
```

In the module, constants `i` and `j` for `Pid` are declared.

In module `ISTEP`, the following operator denoting the predicate to prove in each inductive case is declared and defined:

```
op istep1 : Pid Pid -Bool
eq istep1(I,J) = inv1(s,I,J) implies inv1(s',I,J) .
```

Let us consider the inductive case where we show that any transition rule denoted by action `try` preserves the predicate of invariant (1). We use constant `k` for `Pid`, which is used as the second argument of `try`. In this inductive case, the state space is split into five sub-spaces, which is shown as follows:

1	`c-try(s,k)`	`i = k`	`j = k`
2			`not(j = k)`
3		`not(i = k)`	`j = k`
4			`not(j = k)`
5	`not c-try(s,k)`		

Each case is denoted by the predicate obtained by connecting ones appearing in the row with conjunction.

In this paper, the proof score of case 2 is shown, which is as follows:

```
open ISTEP
-- arbitrary objects
   op k : -Pid .
-- assumptions
   -- eq c-try(s,k) = true .
   eq pc(s,k) = l2 .
   eq top(queue(s)) = k .
   --
   eq i = k .
   eq (j = k) = false .
-- successor state
   eq s' = try(s,k) .
-- check if the predicate is true.
   red inv2(s,j) implies istep1(i,j) .
close
```

In this proof score, invariant (2) is used to strengthen the inductive hypothesis denoted by `inv1(s,i,j)`.

Remarks

Although the invariants discussed in this section do not seem worth verifying just because they seem trivial, the same method with which they are verified can be applied to non-trivial problems such that distributed systems and security protocols have more interesting properties [44, 45, 46, 47]. It is worth stating that we found that 2KP and 3KP electronic payment protocols [39] do not have a desired property [48] while we were trying to verify it with the method described in this section.

OTSs can be extended to deal with time constraints [41, 43].

REFERENCES

[1] M. Arrais and J.L. Fiadeiro. Unifying theories in different institutions. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science, pages 81–101. Springer, 1996. Proceedings of 11th Workshop on Specification of Abstract Data Types. Oslo, Norway, September 1995.

[2] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer, 1980. Lecture Notes in Computer

Science, Volume 86; based on unpublished notes handed out at the Symposium on Algebra and Applications, Stefan Banach Center, Warsaw, Poland, 1978.

[3] Samuel Buss and Grigore Roşu. Incompleteness of behavioural logics. In Horst Reichel, editor, *Coalgebraic Methods in Computer Science*, volume 33 of *Electronic Notes in Theoretical Computer Science*, pages 61–79. Elsevier Science, 2000.

[4] Manuel Clavel, Steve Eker, Patrick Lincoln, and Jose Meseguer. Principles of Maude. *Electronic Notes in Theoretical Computer Science*, 4, 1996. Proceedings, First International Workshop on Rewriting Logic and its Applications. Asilomar, California, September 1996.

[5] frm-eRăzvan Diaconescu. Extra theory morphisms for institutions: logical semantics for multi-paradigm languages. *Applied Categorical Structures*, 6(4):427–453, 1998. A preliminary version appeared as JAIST Technical Report IS-RR-97-0032F in 1997.

[6] frm-eRăzvan Diaconescu. Category-based constraint logic. *Mathematical Structures in Computer Science*, 10(3):373–407, 2000.

[7] frm-eRăzvan Diaconescu. Grothendieck institutions. *Applied Categorical Structures*, 10(4):383–402, 2002. Preliminary version appeared as IMAR Preprint 2-2000, ISSN 250-3638, February 2000.

[8] Răzvan Diaconescu. Institution-independent ultraproducts. *Fundamenta Informaticæ*, 2003. (to appear)

[9] Răzvan Diaconescu. An institution-independent proof of Craig Interpolation Theorem. *Studia Logica*, 76(1), 2004. (to appear)

[10] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[11] frm-eRăzvan Diaconescu and Kokichi Futatsugi. Behavioural coherence in object-oriented algebraic specification. *Universal Computer Science*, 6(1):74–96, 2000. First version appeared as JAIST Technical Report IS-RR-98-0017F, June 1998.

[12] frm-eRăzvan Diaconescu and Kokichi Futatsugi. Logical foundations of CafeOBJ. *Theoretical Computer Science*, 285:289–318, 2002.

[13] Răzvan Diaconescu, Kokichi Futatsugi, and Shusaku Iida. Component-based algebraic specification and verification in CafeOBJ. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 – Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1644–1663. Springer, 1999.

[14] Răzvan Diaconescu, Kokichi Futatsugi, and Shusaku Iida. CafeOBJ Jewels. In Kokichi Futatsugi, Ataru Nakagawa, and Tetsuo Tamai, editors, *Cafe: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000.

[15] Răzvan Diaconescu, Joseph Goguen, and Petros Stefaneas. Logical support for modularisation. In Gerard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 83–130. Cambridge, 1993. Proceedings of a Workshop held in Edinburgh, Scotland, May 1991.

[16] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and Jose Meseguer. Principles of OBJ2. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.

[17] frm-eJoseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.

[18] frm-eJoseph Goguen and Răzvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4(4):363–392, 1994.

[19] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Harmut Ehrig and Fernando Orejas, editors, *Recent Trends in Data Type Specification*, volume 785 of *Lecture Notes in Computer Science*, pages 1–34. Springer, 1994.

[20] Joseph Goguen and Grant Malcolm. A hidden agenda. Technical Report CS97-538, University of California at San Diego, 1997.

[21] frm-eJoseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992. Also, Programming Research Group Technical Monograph PRG–80, Oxford University, December 1989.

[22] frm-eJoseph Goguen and Grigore Roşu. Institution morphisms. *Formal Aspects of Computing*, 13:274–307, 2002.

[23] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*. Kluwer, 2000.

[24] Rolf Hennicker and Michel Bidoit. Observational logic. In A. M. Haeberer, editor, *Algebraic Methodology and Software Technology*, number 1584 in LNCS, pages 263–277. Springer, 1999. Proc. AMAST'99.

[25] frm-eShusaku Iida, Kokichi Futatsugi, and Răzvan Diaconescu. Component-based algebraic specification: - behavioural specification for component-based software engineering -. In *Behavioral specifications of businesses and systems*, pages 103–119. Kluwer, 1999.

[26] Saunders MacLane. *Categories for the Working Mathematician*. Springer, second edition, 1998.

[27] frm-eJosé Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[28] José Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Pressice, editor, *Proc. WADT'97*, number 1376 in Lecture Notes in Computer Science, pages 18–61. Springer, 1998.

[29] Till Mossakowski. Foundations of heterogeneous specification. In *WADT2002*.

[30] Till Mossakowski. Comorphism-based Grothendieck logics. In K. Diks and W. Rytter, editors, *Mathematical foundations of computer science*, volume 2420 of *LNCS*, pages 593–604. Springer, 2002.

[31] frm-eTill Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*, (286):367–475, 2002.

[32] R. Paré and D. Schumacher. *Indexed Categories and their Applications*, volume 661 of *Lecture Notes in Mathematics*, chapter Abstract Families and the Adjoint Functor Theorems, pages 1–125. Springer, 1978.

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

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

[35] frm-eDonald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Control*, 76:165–210, 1988. Earlier version in *Proceedings, International Symposium on the Semantics of Data Types*, Lecture Notes in Computer Science, Volume 173, Springer, 1985.

[36] frm-eAndrzej Tarlecki. Quasi-varieties in abstract algebraic institutions. *Journal of Computer and System Sciences*, 33(3):333–360, 1986. Original version, University of Edinburgh, Report CSR-173-84.

[37] frm-eAndrzej Tarlecki, Rod Burstall, and Joseph Goguen. Some fundamental algebraic tools for the semantics of computation, part 3: Indexed categories. *Theoretical Computer Science*, 91:239–264, 1991. Also, Monograph PRG–77, August 1989, Programming Research Group, Oxford University.

[38] Uwe Wolter. Institutional frames. In *Recent Trends in Data Type Specification. Proceedings*, volume 906 of *Lecture Notes in Computer Science*, pages 469–482. Springer Verlag, London, 1995.

[39] Bellare, M., Garay, J.A., Hauser, R., Herzberg, A., Krawczyk, H., Steiner, M., Tsudik, G., Van Herreweghen, E. and Waidner, M. Design, Implementation and Deployment of the *i*KP Secure Electronic Payment System. IEEE Journal of Selected Areas in Communications, Vol. 18, 2000, No. 4, pp. 611-627.

[40] frm-eChandy, K.M. and Misra, J. Parallel Program Design: A Foundation. Addison-Wesley, Reading, MA, 1988.

[41] frm-eFutatsugi, K. and Ogata, K. Real-time Systems. In: Rewriting, Proof, and Computation, Proceedings of the 1st International Symposium, RPC 2001, 2001, pp. 60–79.

[42] Goguen, J. Theorem Proving and Algebra. MIT Press, Cambridge, MA (to appear).

[43] frm-eOgata, K. and Futatsugi, K. Modeling and Verification of Distributed Real-Time Systems Based on CafeOBJ. In: Automated Software Engineering, Proceedings of the 16th International Conference, ASE 2001, IEEE CS Press, 2001, pp. 185–192.

[44] frm-eOgata, K. and Futatsugi, K. Formally Modeling and Verifying Ricart&Agrawala Distributed Mutual Exclusion Algorithm. In: Quality Software, Proceedings of the 2nd Asia-Pacific Conference, APAQS 2001, IEEE CS Press, 2001, pp. 357–366.

[45] frm-eOgata, K. and Futatsugi, K. Formal Analysis of Suzuki&Kasami Distributed Mutual Exclusion Algorithm. In: Formal Methods for Open Object-Based Distributed Systems, Proceedings of the IFIP TC6/WG6.1 Fifth International Conference, FMOODS 2002, Kluwer Academic Publishers, 2002, pp. 181–195.

[46] frm-eOgata, K. and Futatsugi, K. Formal Analysis of the *i*KP Electronic Payment Protocols. In: Software Security – Theories and Systems, Proceedings of the Mext-NSF-JSPS International Symposium, ISSS 2002, LNCS 2609, Springer, 2003.

[47] frm-eOgata, K. and Futatsugi, K. Formal Verification of the Horn-Preneel Micropayment Protocol. In: Verification, Model Checking and Abstract Interpretation, Pro-

ceedings of the 4th International Conference, VMCAI 2003, LNCS 2575, Springer, 2003, pp. 238–252.

[48] Ogata, K. and Futatsugi, K. Flaw and Modification of the *i*KP Electronic Payment Protocols. Information Processing Letters, Vol. 86, 2003, pp. 57–62.

[49] frm-eOgata, K. and Futatsugi, K.: Proof Scores in the OTS/CafeOBJ method. Submitted to FMOODS 2003.