Proof Scores: A Survey

ADRIÁN RIESCO, Universidad Complutense de Madrid, Spain
KAZUHIRO OGATA, Japan Advanced Institute of Science and Technology, Japan
MASAKI NAKAMURA, Toyama Prefectural University, Japan
DANIEL GĂINĂ, Kyushu University, Japan
DUONG DINH TRAN, Japan Advanced Institute of Science and Technology, Japan
KOKICHI FUTATSUGI, Japan Advanced Institute of Science and Technology, Japan

Proof scores can be regarded as outlines of the formal verification of system properties. They have been historically used by the OBJ family of specification languages. The main advantage of proof scores is that they follow the same syntax as the specification language they are used in, so specifiers can easily adopt them and use as many features as the particular language provides. In this way, proof scores have been successfully used to prove properties of a large number of systems and protocols. However, proof scores also present a number of disadvantages that prevented a large audience from adopting them as proving mechanism.

In this paper we present the theoretical foundations of proof scores; the different systems where they have been adopted and their latest developments; the classes of systems successfully verified using proof scores, including the main techniques used for it; the main reasons why they have not been widely adopted; and finally we discuss some directions of future work that might solve the problems discussed previously.

CCS Concepts: • **Theory of computation** → **Proof theory**; • **Security and privacy** → *Logic and verification*; • **Software and its engineering** → **Formal methods**; **Software verification**.

Additional Key Words and Phrases: CafeOBJ, Theorem proving, System specification, rewriting

ACM Reference Format:
Adrián Riesco, Kazuhiro Ogata, Masaki Nakamura, Daniel Găină, Duong Dinh Tran, and Kokichi Futatsugi. 2025. Proof Scores: A Survey. 1, 1 (May 2025), 35 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Proof scores [51] were originally proposed in the 90's as a promising technique for proving properties of systems using term rewriting. The development of algebraic specification languages executable by rewriting, and in particular of the OBJ family of programming languages, allowed specifiers to put this methodology into practice, revealing proof scores as a powerful alternative to standard theorem proving approaches.

The key feature of proof scores is its complete integration into the specification process: proof scores use the same syntax and evaluation mechanism as the specification language, hence easing the verification process. This smooth

Authors' addresses: Adrián Riesco, Universidad Complutense de Madrid, Madrid, Spain, ariesco@fdi.ucm.es; Kazuhiro Ogata, Japan Advanced Institute of Science and Technology, Nomi, Japan, ogata@jaist.ac.jp; Masaki Nakamura, Toyama Prefectural University, Imizu, Japan, masaki-n@pu-toyama.ac.jp; Daniel Găină, Kyushu University, Fukuoka, Japan, daniel@imi.kyushu-u.ac.jp; Duong Dinh Tran, Japan Advanced Institute of Science and Technology, Nomi, Japan, duongtd@jaist.ac.jp; Kokichi Futatsugi, Japan Advanced Institute of Science and Technology, nomi, Japan, futatsugi@jaist.ac.jp.

integration of the specification methodology into the verification process has allowed practitioners to analyze several systems and protocols over the last decades. However, there are some weak points since the efforts were directed to the development of new features for the specification languages executable by rewriting assuming that proof scores would benefit from these new features (as it happens), but leaving the verification burden such as checking that it follows "sound rules," to the users.

Taking these ideas into account, and despite their good properties, the use of proof scores has been mostly limited to academic environments, without reaching a wide audience, in particular software development companies. Although this lack of industrial applications is not limited to proof scores, it is worth analyzing the reasons why this happens and how it could be solved.

In this paper we discuss the past, present, and future of proof scores. In particular, for the past we describe their history, as well as most of the systems and protocols that have been specified and verified thus far, which can be used as a bibliography by topic. For the present, we give an updated reference guide to work with them, discuss their theoretical basis, and explain in detail a protocol that illustrates some directions to consider in the future. In turn, we analyze past and present to understand the problems and challenges when using proof scores and suggesting the most promising lines of future work.

The rest of the paper is organized as follows: Section 2 briefly introduces the history of proof scores. Section 3 presents the theoretical ideas of proof scores, while Section 4 introduces structured specifications and their application to proof scores. Section 5 lists the tools supporting proof scores, discussing their different features. Section 6 presents proof scores in practice, illustrating how to use proof scores using a running example. Moreover, it also describes complementary verification techniques that complement proof scores and ease its use. Section 7 presents the related work and its relation with proof scores. Finally, Section 8 concludes and details lines of ongoing and future work. Supplemental material for this paper discussing the open issues and challenges proof scores face in the future and possible ways to deal with them, together with a summary of protocols successfully analyzed using proof scores, giving particular examples of each category, is described in [118]. The examples shown in the survey are available at https://github.com/ariesco/proof-scores-survey

2 HISTORY IN A NUTSHELL

The terminology "Proof Score" was coined by Joseph A. Goguen [62], one of the first researchers to apply Category Theory (CT) to Computer Science (CS) and one of the pioneers in the field of Algebraic Specification. His PhD work [60] is on categories of Fuzzy sets called Goguen Categories [136]. As another application of CT to CS, Goguen, together with Burstall, invented the Theory of Institutions [66], which was born as the theoretical foundation of parameterized programming [61] and then came to be used as a framework to formalize logics. Goguen and Burstall designed Clear [20], a specification language, based on the Theory of Institutions. Although Clear was not implemented, Goguen, together with Futatsugi, Jouannaud, and Meseguer, designed and implemented OBJ2 [52], an algebraic specification language, which was followed by OBJ3 [70]. OBJ3 is one of the earliest computer languages that made it possible to do parametrized programming. One key concept of Clear and OBJ3 to support parametrized programming is pushout-based parametrized modules, which take modules as their parameters. Clear and OBJ3 have influenced module systems in some programming languages, such as Ada, Standard ML, and C++ [64]. Standard ML modules [97] were based on prototype designs of modules for Hope [21], a functional programming language, which were influenced by Clear. Among the other earliest computer languages that supported parameterized programming are ACT ONE [42] and

HISP [53]. OBJ3 was succeeded by CafeOBJ[1] [35] and Maude[2] [26]. Thus, CafeOBJ and Maude are sibling languages. OBJ3 was implemented in Common Lisp and so was CafeOBJ, while Maude was implemented in C++. In contrast to OBJ3 and Maude, which have one implementation, CafeOBJ has two implementations. The second implementation [117] of CafeOBJ, called CafeInMaude, was conducted in Maude, which can be used as logical framework as well.

Goguen [62] describes proof scores as follows:

> The basic idea is to transform theorem proving problems into **proof scores**, which consist of declarations and rewritings such that if everything evaluates as desired, then the problem is solved. This approach is neither fully automatic nor fully manual, but rather calls for machines to do the most routine tasks, such as substitution, simplification and reduction, and for humans to do the most interesting tasks, such as deciding proof strategies. Moreover, partially successful proofs often return information that suggests what to try next, ...

Goguen and his colleagues initially used proof scores to tackle theorem proving problems about data structures, such as natural numbers. There was criticism about algebraic specification techniques, which are based on elegant theories but do not have any practical applications. Eric G. Wagner mentions such criticism in his article [135] about algebraic specifications. Wagner was a member of the ADJ group [63, 135] and so was Goguen. The criticism has been addressed by introducing hidden algebra and observational specifications [36, 68] and the work of Futatsugi, Ogata, and others, who have demonstrated that proof scores could be used to tackle theorem proving problems about a wide range of systems, among which are authentication protocols and electronic commerce protocols (see Section [118] for details).

There was criticism specific to proof scores as well. This is because humans are supposed to do the most interesting tasks, such as deciding proof strategies. Thus, proof scores are subject to human errors. This criticism has been addressed by developing proof assistants, such as the Inductive Theorem Prover (ITP) [28] and the Constructor-based Inductive Theorem Prover (or CITP) [54, 56]. Following the development of CITP, a proof assistant was developed inside CafeOBJ and the CafeInMade Proof Assistant (CiMPA) [115] was developed for CafeInMaude. ITP, CITP, the CafeOBJ proof assistant, and CiMPA can prevent human errors but may dilute the merits of proof scores, although those proof assistants generate proof scores inside. To address the issue, Riesco and Ogata developed an automatic proof script generator for CiMPA from manually written proof scripts in CafeOBJ, called the CafeInMaude Proof Generator (CiMPG) [115].

3 THEORETICAL FOUNDATIONS

In this section we discuss the fundamental theoretical concepts required to understand how the proof score approach works. The discussion is limited to basic order-sorted specifications even though both Maude and CafeOBJ (see Section 5) are based on more advanced logical semantics which support, for example, rewriting or behavioural specifications.

Signatures. We introduce *order-sorted signatures*, which describe the relations between *sorts* and how terms are built. The main ideas underlying these signatures are *inheritance* and *polymorphism*.

Definition 3.1 (Order-sorted signatures [69]). A *many-sorted signature* is a pair (S, F) where S is a set of sorts and $F = \{\sigma : w \longrightarrow s \mid w \in S^* \text{ and } s \in S\}$ is a set of function symbols. For each function symbol $\sigma : w \longrightarrow s$ in F, we say that σ is its name, w is its arity, and s is its co-arity. Notice that F can be regarded as an $(S^* \times S)$-sorted set $\{F_{w,s}\}_{(w,s) \in S^* \times S}$, where $F_{w,s} = \{\sigma \mid \sigma : w \longrightarrow s \in F\}$. An order-sorted signature is a triple $(S, \leq, F)$ such that (S, F) is a many-sorted signature, and $(S, \leq)$ is a partially ordered set.

[1] https://cafeobj.org/
[2] http://maude.cs.illinois.edu/w/index.php/The_Maude_System

A function symbol $\sigma \in F_{\lambda,s}$ with the empty arity λ is called a *constant*. The set $S/_{\equiv_\leq}$ of connected components of $(S, \leq)$ is the quotient of S under the equivalence relation $\equiv_\leq$ generated by $\leq$. The equivalence relation $\equiv_\leq$ is canonically extended to strings over S. The set of ground terms $\mathcal{T}_\Sigma$ over a signature $\Sigma = (S, \leq, F)$ is defined as follows:

$$1)\ \frac{}{c \in \mathcal{T}_{\Sigma,s}} \text{ if } c : \longrightarrow s \in F \quad 2)\ \frac{t_1 \in \mathcal{T}_{\Sigma,s_1}\ \ldots\ t_n \in \mathcal{T}_{\Sigma,s_n}}{\sigma(t_1,\ldots,t_n) \in \mathcal{T}_{\Sigma,s}} \text{ if } \sigma : s_1 \ldots s_n \longrightarrow s \in F \quad 3)\ \frac{t \in \mathcal{T}_{\Sigma,s_0}}{t \in \mathcal{T}_{\Sigma,s}} \text{ if } s_0 \leq s$$

We denote by $\mathcal{T}_\Sigma(X)$ the set of terms with variables from X.

Definition 3.2 (Regularity). A signature $\Sigma = (S, \leq, F)$ is *regular* iff for any function symbol $\sigma : w \longrightarrow s \in F$ and any string of sorts $w_0 \in S^*$ such that $w_0 \leq w$, the set $\{(w', s') \in S^* \times S \mid w_0 \leq w' \text{ and } \sigma : w' \longrightarrow s' \in F\}$ has a least element.

By regularity each term has a least sort. In practice, we work only with regular signatures.

Example 3.3 (Natural numbers). Let Σ_{NAT} be the following signature of natural numbers:

(1) The set of sorts is $\{Nat, NzNat\}$ such that $NzNat \leq Nat$.
(2) The set of function symbols is $F = \{0 : \longrightarrow Nat, s\_ : Nat \longrightarrow NzNat, \_+\_ : Nat\ Nat \longrightarrow Nat\}$.

Clearly, the signature of Example 3.3 is regular.

Models. The models consist of order-sorted algebras which give the denotational semantics of specifications.

Definition 3.4 (Order-sorted algebras [69]). Let $\Sigma = (S, \leq, F)$ be an order-sorted signature.
An order-sorted algebra $\mathcal{A}$ over Σ consists of an (S, F)-algebra, that is,

- an S-sorted set $\{\mathcal{A}_s\}_{s \in S}$, and
- a function $\mathcal{A}_\sigma : \mathcal{A}_w \longrightarrow \mathcal{A}_s$ for each function symbol $\sigma : w \longrightarrow s \in F$, where $\mathcal{A}_w = \mathcal{A}_{s_1} \times \cdots \times \mathcal{A}_{s_n}$ whenever $w = s_1 \ldots s_n$ and $\mathcal{A}_w$ is a singleton whenever w is the empty string,

satisfying the following properties:

- $\mathcal{A}_s \subseteq \mathcal{A}_{s'}$ whenever $s \leq s'$, and
- for all $\sigma : w_1 \longrightarrow s_1 \in F$ and $\sigma : w_2 \longrightarrow s_2 \in F$ such that $w_1 \equiv_\leq w_2$, the functions $\mathcal{A}_\sigma : \mathcal{A}_{w_1} \longrightarrow \mathcal{A}_{s_1}$ and $\mathcal{A}_\sigma : \mathcal{A}_{w_2} \longrightarrow \mathcal{A}_{s_2}$ return the same value for the same arguments (in $\mathcal{A}_{w_1} \cap \mathcal{A}_{w_2}$).

An order-sorted homomorphism $h : \mathcal{A} \longrightarrow \mathcal{B}$ is a many-sorted homomorphism such that $h_s(a) = h_{s'}(a)$ for all sorts $s, s' \in S$ with $s \equiv_\leq s'$ and all elements $a \in \mathcal{A}_s \cap \mathcal{A}_{s'}$. We denote by $\mathtt{Mod}(\Sigma)$ the category whose objects are order-sorted Σ-algebras and arrows are order-sorted Σ-hommorphisms.

The order-sorted algebra of natural numbers $\mathcal{N}$ over NAT interprets *Nat* as the set of all natural numbers, *NzNat* as the set of all positive natural numbers, and 0 as zero, $s\_$ as the successor function, and $\_+\_$ as the addition.

Clearly, $\mathcal{T}_\Sigma$ can be organized as an order-sorted algebra, interpreting each function symbol $\sigma : s_1 \ldots s_n \longrightarrow s \in F$ as a function $\mathcal{T}_\sigma : \mathcal{T}_{s_1} \times \mathcal{T}_{s_n} \longrightarrow \mathcal{T}_s$ defined by $\mathcal{T}_\sigma(t_1, \ldots, t_n) = \sigma(t_1, \ldots, t_n)$ for all $i \in \{1, \ldots, n\}$ and $t_i \in \mathcal{T}_{s_i}$. Similarly, for any set of variables X for Σ, the set of terms with variables $\mathcal{T}_\Sigma(X)$ can be organized as a Σ-algebra.

Sentences. The sentences consist of universally quantified conditional equations which make the specifications executable by rewriting.

Definition 3.5 (Conditional equations). The sentences over a signature $\Sigma = (S, \leq, F)$ are (conditional) equations of the form $\forall X \cdot t_0 = t_0'$ if $t_1 = t_1' \wedge \ldots \wedge t_n = t_n'$, where $n \in \mathbb{N}$, X is a finite set of variables for Σ, and t_i and t_i' are terms with

variables from X such that the sorts of t_i and t'_i are in the same connected component. We denote by $\mathsf{Sen}(\Sigma)$ the set of Σ-sentences.

If $n = 0$ then the sentence is an equation $\forall t_0 = t'_0$.

Example 3.6 (Addition). Some examples of sentences over the signature Σ_{NAT} are the equations which inductively define the addition of natural numbers, $E_{\mathsf{NAT}} := \{\forall x : Nat \cdot 0 + x = x,\ \forall x, y : Nat \cdot s\ y + x = s(x + y)\}$.

Satisfaction relation. The satisfaction of a sentence by an order-sorted algebra is the usual Tarskian satisfaction based on the interpretation of terms. More concretely, given a signature Σ, an order-sorted Σ-algebra $\mathcal{A}$ satisfies a Σ-sentence $\forall X \cdot t_0 = t'_0$ if $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n$, in symbols, $\mathcal{A} \models_\Sigma \forall X \cdot t_0 = t'_0$ if $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n$, iff for all evaluations $f : X \to \mathcal{A}$ we have $f^\#(t_0) = f^\#(t'_0)$ whenever $f^\#(t_i) = f^\#(t'_i)$ for all $i \in \{1, \ldots, n\}$, where $f^\# : \mathcal{T}_\Sigma(X) \to \mathcal{A}$ is the unique homomorphism extending the valuation $f : X \to \mathcal{A}$. The satisfaction relation is straightforwardly generalized to sets of sentences: $\mathcal{A} \models_\Sigma E$ iff $\mathcal{A} \models_\Sigma e$ for all $e \in E$. We say that a set of sentences E *satisfies* a set of sentences E', in symbols, $E \models_\Sigma E'$, iff $\mathcal{A} \models_\Sigma E$ implies $\mathcal{A} \models_\Sigma E'$, for all Σ-algebras $\mathcal{A}$. Notice that the algebra of natural numbers $\mathcal{N}$ with addition defined above satisfies both equations from Example 3.6, since $0 + n = n$ for all $n \in \mathbb{N}$ and $\mathcal{N}_s\ m + n = \mathcal{N}_s(m + n)$ for all natural numbers $m, n \in \mathbb{N}$.

Signature morphisms. A very important topic in algebraic specifications, in particular, and in computer science, in general, is modularization. One crucial concept for modularization is the notion of signature morphism.

Definition 3.7. A signature morphism $\chi\colon (S, \leq, F) \to (S', \leq', F')$ is a many-sorted algebraic signature morphisms $\varphi\colon (S, F) \to (S', F')$ such that the function $\chi\colon (S, \leq) \to (S', \leq')$ is monotonic and χ preserves the subsort polymorphism, that is, χ maps each function symbols with the same name $\sigma : w_1 \to s_1$ and $\sigma : w_2 \to s_2$ and such that $w_1 \equiv_\leq w_2$ to some function symbols $\sigma' : w'_1 \to s'_1$ and $\sigma' : w'_2 \to s'_2$ that have the same name.

In Definition 3.7, since $\chi\colon (S, \leq) \to (S', \leq')$ is monotonic, $w_1 \equiv_\leq w_2$ implies $\chi(w_1) \equiv_{\leq'} \chi(w_2)$. In practice, most of signature morphisms used are inclusions, which obviously preserves the subsort polymorphism.

Example 3.8. Let Σ_{TRIV} be the signature which consists of one sort *Elt*, and $\chi : \Sigma_{\mathsf{TRIV}} \to \Sigma_{\mathsf{NAT}}$ be the signature morphism which renames *Elt* to *Nat*.

Example 3.9. Let Σ_{NATL} be the signature of lists of natural numbers which extends the signature of natural numbers Σ_{NAT} with the sorts $\{NeList, List\}$ such that $Nat < NeList < List$, and the following function symbols.

- $empty : \to List$
- $\_;\_ : List\ List \to List$
- $\_;\_ : NeList\ List \to NeList$
- $\_;\_ : List\ NeList \to NeList$
- $tail : NeList \to List$
- $head : NeList \to Nat$

The inclusion $\iota : \Sigma_{\mathsf{NAT}} \hookrightarrow \Sigma_{\mathsf{NATL}}$ is another example of signature morphism.

Any signature morphism $\chi : \Sigma \to \Sigma'$ can be extended to a function $\chi : \mathsf{Sen}(\Sigma) \to \mathsf{Sen}(\Sigma')$ which translates the sentences over Σ along χ in a symbolwise manner. On the other hand, for any signature morphism $\chi : \Sigma \to \Sigma'$, there exists a reduct functor $\restriction_\chi\ \_ : \mathsf{Mod}(\Sigma') \to \mathsf{Mod}(\Sigma)$ defined by:

- For all $\mathcal{A}' \in |\mathsf{Mod}(\Sigma')|$, $(\mathcal{A}' \restriction_\chi)_x = \mathcal{A}' \restriction_{\chi(x)}$, where x is any sort or function symbol from Σ.
- For all $h' : \mathcal{A}' \to \mathcal{B}' \in \mathsf{Mod}(\Sigma')$, $(h' \restriction_\chi)_s = h'_{\chi(s)}$, where s is any sort from Σ.

If χ is an inclusion then we may write $\mathcal{A}' \restriction_\Sigma$ instead of $\mathcal{A}' \restriction_\chi$. For example, if $\chi : \Sigma_{\mathsf{TRIV}} \to \Sigma_{\mathsf{NAT}}$ is the renaming described in Example 3.8 and $\mathcal{N}$ is the model of natural numbers with addition then $\mathcal{N} \restriction_\chi$ is the set of natural numbers.

Example 3.10. Let $\mathcal{L}$ be the algebra over Σ_{NATL} defined by:

- $\mathcal{L}$ interprets all symbols in Σ_{NAT} as $\mathcal{N}$, the model of natural numbers with addition,
- $\mathcal{L}_{List}$ consists of all lists of natural numbers, while $\mathcal{L}_{NeList}$ consists of all non-empty lists of natural numbers,
- $\mathcal{L}_{empty}$ is the empty list, $\mathcal{L}_{;}$ is the concatenation of lists, $\mathcal{L}_{head}$ extracts the top of a non-empty list, and $\mathcal{L}_{tail}$ deletes the first element of a non-empty list.

Notice that the reduct of $\mathcal{L}$ to the signature Σ_{NAT} is $\mathcal{N}$, in symbols, $\mathcal{L} \upharpoonright_{\Sigma_{\mathsf{NAT}}} = \mathcal{N}$. An important result in algebraic specifications, which is at the core of many important developments concerning modularity is the satisfaction condition.

Proposition 3.11 (Satisfaction condition). *For all signature morphisms $\chi : \Sigma \to \Sigma'$, all Σ-sentences γ and all order-sorted Σ'-algebras $\mathcal{A}'$, we have $\mathcal{A}' \models \chi(\gamma)$ iff $\mathcal{A}' \upharpoonright_\chi \models \gamma$, where $\chi(\gamma)$ denotes the translation of γ along χ.*

Pairs (Σ, E) consisting of a signature Σ and a set of sentences E over Σ, are called presentations in algebraic specification literature. The notion of signature morphism extends straightforwardly to presentations.

Definition 3.12. A presentation morphism $\chi : (\Sigma, E) \to (\Sigma', E')$ is a signature morphism $\chi : \Sigma \to \Sigma'$ s.t. $E' \models \chi(E)$.

An example of presentation morphism is the inclusion $\iota : (\Sigma_{\mathsf{NAT}}, E_{\mathsf{NAT}}) \hookrightarrow (\Sigma_{\mathsf{NATL}}, E_{\mathsf{NATL}})$, where $E_{\mathsf{NAT}} = \{\forall x : Nat \cdot 0 + x = x,\ \forall x, y : Nat \cdot s\ y + x = s(x + y)\}$ defined in Example 3.6 and $E_{\mathsf{NATL}} = E_{\mathsf{NAT}} \cup \{\forall e : Elt, l : List \cdot head(e; l) = e, \forall e : Elt, l : List \cdot tail(e; l) = l\}$.

Proof calculus. The satisfaction relation $E \models E'$ between sets of sentences is the semantic way to establish truth. The syntactic approach to truth consists of defining *entailment relations* between sets of sentences involving only syntactic entities. The correctness of entailment relations can be established only in the presence of satisfaction relation.

Definition 3.13. An entailment system consists of a family of entailment relations $\{\_ \vdash_\Sigma \_ \mid \Sigma \text{ is an order-sorted signature}\}$ between sets of sentences with the following properties:

$$(\textit{Monotonicity})\ \frac{}{E \vdash_\Sigma E_1}\ [E_1 \subseteq E] \qquad (\textit{Union})\ \frac{E \vdash_\Sigma E_1 \quad E \vdash_\Sigma E_2}{E \vdash_\Sigma E_1 \cup E_2}$$

$$(\textit{Translation})\ \frac{E \vdash_\Sigma E_1}{\chi(E) \vdash_{\Sigma'} \chi(E_1)}\ [\chi : \Sigma \to \Sigma'] \qquad (\textit{Cut})\ \frac{E \vdash_\Sigma E_1 \quad E \cup E_1 \vdash_\Sigma E_2}{E \vdash_\Sigma E_2}$$

Notice that we have placed the side conditions of the entailment properties in square brackets. For example, in case of (*Monotonicity*), $E_1 \subseteq E$ implies $E \vdash_\Sigma E_1$. Entailment relations are inductively defined by proof rules.

Definition 3.14 (Order-sorted entailment system). The entailment system of order-sorted algebra is the least entailment system generated by the following proof rules:

$$(\textit{Reflexivity})\ \frac{}{E \vdash_\Sigma \forall X \cdot t = t} \qquad (\textit{Symmetry})\ \frac{E \vdash_\Sigma \forall X \cdot t_1 = t_2}{E \vdash_\Sigma \forall X \cdot t_2 = t_1} \qquad (\textit{Transitivity})\ \frac{E \vdash_\Sigma \forall X \cdot t_1 = t_2 \quad E \vdash_\Sigma \forall X \cdot t_2 = t_3}{E \vdash_\Sigma \forall X \cdot t_1 = t_3}$$

$$(\textit{Congruence})\ \frac{E \vdash_\Sigma \forall X \cdot t_1 = t_1' \ \ldots\ E \vdash_\Sigma \forall X \cdot t_n = t_n'}{E \vdash_\Sigma \forall X \cdot \sigma(t_1, \ldots, t_n) = \sigma(t_1', \ldots, t_n')} \qquad (\textit{Substitutivity})\ \frac{E \vdash_\Sigma \forall X \cdot e}{E \vdash_\Sigma \forall Y \cdot \theta(e)}\ [\theta : X \to T_\Sigma(Y)]$$

$$(\textit{Implication}_1)\ \frac{E \vdash_\Sigma t = t' \text{ if } \bigwedge_{i=1}^n t_i = t_i'}{E \cup \{t_1 = t_1', \ldots, t_n = t_n'\} \vdash_\Sigma t = t'} \qquad (\textit{Implication}_2)\ \frac{E \cup \{t_1 = t_1', \ldots, t_n = t_n'\} \vdash_\Sigma t = t'}{E \vdash_\Sigma t = t' \text{ if } \bigwedge_{i=1}^n t_i = t_i'}$$

$$(\textit{Quantification}_1)\ \frac{E \vdash_{\Sigma(X)} e}{E \vdash_\Sigma \forall X \cdot e} \qquad (\textit{Quantification}_2)\ \frac{E \vdash_\Sigma \forall X \cdot e}{E \vdash_{\Sigma(X)} e}$$

Theorem 3.15 (Completeness). *The order-sorted equational logic calculus is sound (i.e., $\vdash \subseteq \models$) and complete (i.e. $\models \subseteq \vdash$).*

Completeness of order-sorted equation deduction is one of the fundamental results in algebraic specification, which has a long tradition starting with Birkhoff [16], continuing with completeness of many-sorted equational deduction [59] and order-sorted equational deduction [69], and ending with completeness of institutions with Horn clauses [55].

An algebraic specification language, in general, and CafeOBJ, in particular, should be understood on three interconnected semantic levels: (i) denotational semantics, (ii) proof-theoretical definition, and (ii) operational semantics. The denotational semantics consists of classes of models described by the specifications and the satisfaction relation determined by them. The proof-theoretical aspect is given by the entailment relations, which provides a practical way of reasoning formally about the properties of models described by specifications. The operational semantics consists of order-sorted term-rewriting [65], which has several benefits such as an increased level of automation for proofs.

The design of the specification and verification methodology is largely based on the denotational semantics. Therefore, this section focused on describing the underlying logic, order-sorted algebra, and the language constructs such as free semantics, parameterization and imports. The operational semantics of algebraic specification languages is important during formal proofs for evaluating functions by providing the structured, rule-based process to compute values from functions applied to specific arguments. This ensures that every step in the proof is grounded in the formal definitions of the system, leading to correct and an increased automation level in comparison to theorem proving methods which are not powered by term rewriting.

4 STRUCTURED SPECIFICATIONS

The foundational work on module algebra began with the seminal contributions of Professor Jan Bergstra and his collaborators [14]. To the best of our knowledge, this marked the start of the concept of module algebra and the formal study of rules for constructing module expressions—referred to here as structured specifications—which are used to describe software systems.[3] Rather than reiterating the well-established importance of modularizing software, we emphasize that the mathematical foundations supporting this modularization are grounded in the specification-building operators that define structured specifications. These operators provide the formal mechanisms for composing and manipulating modular software components. We refer the reader to [30, 33, 34, 125] for the latest developments in this area. This section is devoted to a brief presentation of fundamentals of structured specifications and their application to the specification methodology.

4.1 Reachability

In practice, only order-sorted algebras that are *reachable* by some constructor operations are of interest. Consider, for example, the signature of natural numbers with addition NAT described in Example 3.3. One model of interest is $\mathcal{N}$, the model of natural numbers. Other model of interest can be $\mathcal{Z}_n$, the model of integers modulo n, where $n > 1$, which interprets both sorts *Nat* and *NzNat* as $\{\widehat{0}, \widehat{1}, \ldots, \widehat{n-1}\}$ and the function symbols in the usual way. Both $\mathcal{N}$ and $\mathcal{Z}_n$ are reachable by the operations $0 : \longrightarrow Nat$ and $s\_ : Nat \longrightarrow NzNat$, that is, the unique homomorphisms $\mathcal{T}_{\mathsf{NAT}} \longrightarrow \mathcal{N}$ and $\mathcal{T}_{\mathsf{NAT}} \longrightarrow \mathcal{Z}_n$ are surjective. Next, we present constrained and loose sorts, and then we generalize the concept of reachability to any signature $\Sigma = (S, \leq, F)$ for which we distinguish a subset $F^c \subseteq F$ of constructors.

Definition 4.1 (Constrained and loose sorts). A sort $s \in S$ is called *constrained* if s has a constructor $\sigma : w \longrightarrow s \in F^c$, otherwise s is called loose.

[3] To be more precise, in this context a *module* is a specification with a name.

Definition 4.2 (Reachable algebras). An order-sorted algebra $\mathcal{A}$ over a signature $\Sigma = (S, \leq, F)$ is reachable by some constructors $F^c \subseteq F$ if there exists a set of X of variables of loose sort and a valuation $f : X \to \mathcal{A}$ such that the unique extension $f^\# : \mathcal{T}_{\Sigma^c}(X) \to \mathcal{A}\ \restriction_{\Sigma^c}$ of $f : X \to \mathcal{A}$ to a Σ^c-homomorphism is surjective, where $\Sigma^c = (S, \leq, F^c)$.

We give some examples of reachable algebras with loose sorts.

Example 4.3. Let Σ_{BAG} be the signature defined as follows:

- The set of sorts is $\{Elt, Bag\}$ such that $Elt \leq Bag$.
- The set of function symbols is $\{empty : \to Bag, \_ \circ \_ : Bag\ Bag \to Bag, take : Bag\ Elt \to Bag\}$.

Let $F^c_{\mathsf{BAG}} = \{empty : \to Bag, \_ \circ \_ : Bag\ Bag \to Bag\}$ be a set of constructors for the signature Σ_{BAG}. The sort *Elt* is loose while the sort *Bag* is constrained. We give some examples of reachable algebras by the constructors in F^c_{BAG}:

(1) the model of sets of natural numbers, denoted $\mathcal{B}_1$, which interprets *empty* as the empty set, $\circ$ as the union of sets, and *take* as the extraction of an element from a set.
(2) the model of strings with elements from $\{a, b\}$, denoted $\mathcal{B}_2$, which interprets *empty* as the empty string $\circ$ as the concatenation of strings and *take* as the extraction of the first occurrence of an element in a string.
(3) the model of integers, denoted $\mathcal{B}_3$, which interprets both sorts *Elt* and *Bag* as the set of integers, *empty* as zero, $\circ$ as addition and *take* as subtraction.

For the first case, let $X = \{x_i \mid i \in \mathbb{N}\}$ be a countably infinite set of variables, and $f : X \to \mathcal{B}_1$ a valuation defined by $f(x_i) = n$ for all $n \in \mathbb{N}$. It is easy to check that the extension $f^\# : T_{\Sigma^c_{\mathsf{BAG}}}(X) \to \mathcal{B}_1\ \restriction_{\Sigma^c_{\mathsf{BAG}}}$ of $f : X \to \mathcal{B}_1$ to a Σ^c_{BAG}-homomorphism is surjective. Similar arguments can be used to show that $\mathcal{B}_2$ and $\mathcal{B}_3$ are reachable by F^c_{BAG}.

4.2 Congruences

Given a signature $\Sigma = (S, \leq, F)$, a *congruence* $\equiv$ on an algebra $\mathcal{A}$ is an S-sorted equivalence on $\mathcal{A}$ (that is, an equivalence $\equiv_s$ on $\mathcal{A}_s$ for all sorts $s \in S$) compatible with

- the functions, that is, if $a_i \equiv_{s_i} b_i$ for all $i \in \{1, \ldots, n\}$ then $\mathcal{A}_\sigma(a_1, \ldots, a_n) \equiv_s \mathcal{A}_\sigma(b_1 \ldots, b_n)$, for all function symbols $\sigma : s_1 \ldots s_n \to s \in F$, and
- the sort order, that is, $a \equiv_{s_1} b$ iff $a \equiv_{s_2} b$ for all sorts $s_1, s_2 \in S$ such that $s_1 \equiv_\leq s_2$ and all elements $a, b \in \mathcal{A}_{s_1} \cap \mathcal{A}_{s_2}$.

Any set of conditional equations E generates a congruence on the initial model of terms $\mathcal{T}_\Sigma$ as follows:

$$\equiv^E := \{(t, t') \mid E \models t = t'\}$$

where $E \models t = t'$ means that $\mathcal{A} \models E$ implies $\mathcal{A} \models t = t'$ for all Σ-algebras $\mathcal{A}$. The quotient algebra $\mathcal{T}_\Sigma/_{\equiv^E}$ denoted $\mathcal{T}_{(\Sigma,E)}$ has the following initial property [69]: for each algebra A satisfying E, there exists a unique homomorphism $\mathcal{T}_{(\Sigma,E)} \to \mathcal{A}$. The initial algebra $\mathcal{T}_{(\Sigma,E)}$ of E is unique up to isomorphism and it gives the *tight denotation* of the equational specification (Σ, E).

Recall the signature of natural numbers with addition defined in Example 3.3 and the set of sentences E_{NAT} of Example 3.6. Notice that $\mathcal{N}$, the model of natural numbers that interprets all function symbols in the usual way, is isomorphic to $T_{(\Sigma_{\mathsf{NAT}}, E_{\mathsf{NAT}})}$ the initial algebra of E_{NAT}.

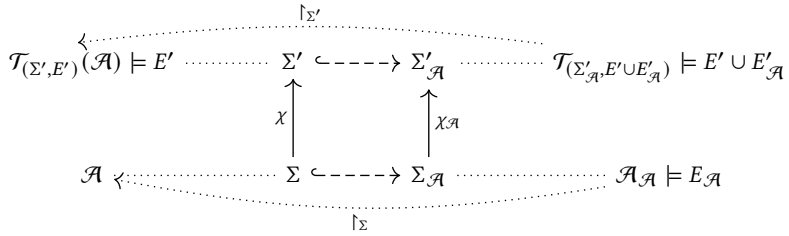
4.3 Free models

In algebraic specification, the notion of free algebra is one of the basic concepts, which is a generalization of the initial algebra. In initial semantics, the idea is to define a minimal model of the specification where the behavior of the data

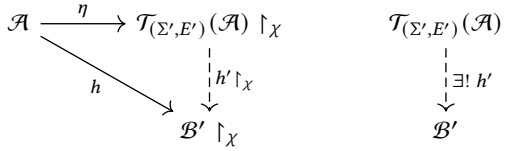
type and its operations are determined uniquely. The initial algebra is important because it guarantees that if two terms are equal under the specified operations (according to the equations), then they are equal in the free algebra. This forms the basis of equational reasoning in algebraic specifications. When using initial semantics, the free algebra acts as the default or "best" interpretation of the abstract data type, ensuring that no unintended properties or relationships are introduced. In addition, free models cannot be fully specified by first-order theories because the set of semantic consequences of a first-order theory is recursively enumerable, whereas the properties of free models—when formalized as sentences—are not, due to Gödel's incompleteness theorem. This theorem implies that there are true statements about free models that cannot be derived from any first-order theory, meaning such models may possess properties that lie beyond the expressive power of first-order logic to capture in a complete and consistent manner.

An example of free algebra is given by the NATL-model of lists with elements from $\mathcal{Z}_2$, the algebra of integers modulo 2 which interprets the symbols from Σ_{NAT} in the usual way. In the following, we will describe briefly its construction.

The notion of free algebra is best explained using signature morphisms. Let $\chi : \Sigma \to \Sigma'$ be a signature morphism and $\mathcal{A}$ a Σ-algebra. Without loss of generality, we assume that the elements of $\mathcal{A}$ are different from the constants of both Σ and Σ'. In this case, each element $a \in \mathcal{A}_s$ can be regarded as a new constant $a : \to s$ for Σ, where s is any sort of Σ. Let $\Sigma_{\mathcal{A}}$ be the signature obtained from Σ by adding the elements of $\mathcal{A}$ as new constants. Similarly, we let $\Sigma'_{\mathcal{A}}$ denote the signature obtained from Σ' by adding elements $a \in \mathcal{A}_s$ as new constants $a : \to \chi(s)$, where s is any sort of Σ. The result is a new signature morphism $\chi_{\mathcal{A}} : \Sigma_{\mathcal{A}} \to \Sigma'_{\mathcal{A}}$ which extends χ by mapping each constant $a : \to s$ to $a : \to \chi(s)$, for all sorts s of Σ and all elements $a \in \mathcal{A}_s$.



We let $\mathcal{A}_{\mathcal{A}}$ denote the expansion of $\mathcal{A}$ to $\Sigma_{\mathcal{A}}$, which interprets each constant $a : \to s$ as a, for all sorts s of Σ and all elements $a \in \mathcal{A}_s$. We denote by $E_{\mathcal{A}}$ the set of all ground $\Sigma_{\mathcal{A}}$-equations satisfied by $\mathcal{A}_{\mathcal{A}}$, and by $E'_{\mathcal{A}}$ the translation of $E_{\mathcal{A}}$ along $\chi_{\mathcal{A}}$. Notice that $\mathcal{A}_{\mathcal{A}}$ is the initial algebra of $E_{\mathcal{A}}$. For any set of conditional Σ'-equations E', the *free* (Σ', E')-*algebra over* $\mathcal{A}$ *via* χ, denoted $\mathcal{T}_{(\Sigma',E')}(\mathcal{A})$, is $\mathcal{T}_{(\Sigma'_{\mathcal{A}},E' \cup E'_{\mathcal{A}})} \restriction_{\Sigma'}$, where $\mathcal{T}_{(\Sigma'_{\mathcal{A}},E' \cup E'_{\mathcal{A}})}$ is the initial algebra of $E' \cup E'_{\mathcal{A}}$. Since $\mathcal{A}_{\mathcal{A}}$ is the initial algebra of $E_{\mathcal{A}}$ and $\mathcal{T}_{(\Sigma'_{\mathcal{A}},E' \cup E'_{\mathcal{A}})} \models E'_{\mathcal{A}}$, there exists a unique arrow $\eta_{\mathcal{A}} : \mathcal{A}_{\mathcal{A}} \to \mathcal{T}_{(\Sigma'_{\mathcal{A}},E' \cup E'_{\mathcal{A}})} \restriction_{\chi_{\mathcal{A}}}$. The reduct $\eta : \mathcal{A} \to \mathcal{T}_{(\Sigma',E')}(\mathcal{A}) \restriction_{\chi}$ of $\eta_{\mathcal{A}} : \mathcal{A}_{\mathcal{A}} \to \mathcal{T}_{(\Sigma'_{\mathcal{A}},E' \cup E'_{\mathcal{A}})} \restriction_{\chi_{\mathcal{A}}}$ to the signature Σ is a universal arrow to the functor $\restriction_{\chi} \_ : \mathsf{Mod}(\Sigma', E') \to \mathsf{Mod}(\Sigma)$, that is, for each Σ'-algebra $\mathcal{B}'$ that satisfies E' and any homomorphism $h : \mathcal{A} \to \mathcal{B}' \restriction_{\chi}$ there exists a unique Σ'-homomorphism $h' : \mathcal{T}_{(\Sigma',E')}(\mathcal{A}) \to \mathcal{B}'$ such that $\eta; h' \restriction_{\chi} = h$.[4]



Consider the signature inclusion $\Sigma_{\mathsf{NAT}} \hookrightarrow \Sigma_{\mathsf{NATL}}$, defined in Example 3.9, and $\mathcal{Z}_2$, the Σ_{NAT}-algebra of integers modulo 2. Let $E_{\mathsf{NATL}} := E_{\mathsf{NAT}} \cup \{\forall e : Elt, l : List \cdot head(e; l) = e, \forall e : Elt, l : List \cdot tail(e; l) = l\}$, where E_{NAT} is the set of sentences

[4] $\mathsf{Mod}(\Sigma', E')$ is the full subcategory of $\mathsf{Mod}(\Sigma')$ of all Σ'-algebras satisfying E'.

defined in Example 3.6. The model of lists with elements from $\mathcal{Z}_2$ is formally defined as the free $(\Sigma_{\mathsf{NATL}}, E_{\mathsf{NATL}})$-algebra over $\mathcal{Z}_2$, that is, $\mathcal{T}_{(\Sigma_{\mathsf{NATL}},E_{\mathsf{NATL}})}(\mathcal{Z}_2)$. In this case, the universal arrow $\eta : \mathcal{Z}_2 \to \mathcal{T}_{(\Sigma_{\mathsf{NATL}},E_{\mathsf{NATL}})}(\mathcal{Z}_2) \restriction_{\Sigma_{\mathsf{NAT}}}$ is the identity.

4.4 Specification building operators

Structured specifications are constructed from some basic specifications by iteration of several specification building operators. In algebraic specification languages such as CafeOBJ or Maude (see Section 5), the definition of all language constructs is based on specification building operators, which can be regarded as primitive operators. In software engineering, structuring constructs give the possibility of systematic reuse of already defined *modules*, which are in fact labeled specifications. The semantics of a specification SP consists of a signature Σ_{SP} and a class of models $\mathcal{M}_{\mathsf{SP}}$. For any structured specification SP, we can also define its set of sentences E_{SP}.

BASIC A basic specification is a presentation (Σ, E), where Σ is a signature and E is a set of sentences over Σ.
$\Sigma_{(\Sigma,E)} = \Sigma$, $E_{(\Sigma,E)} = E$, and $\mathcal{M}_{(\Sigma,E)} = \{\mathcal{A} \in |\mathsf{Mod}(\Sigma)| \mid \mathcal{A} \models E\}$.

CONS For any specification SP, the restriction of SP to some constructor operators F^c from Σ_{SP}, denoted $\mathsf{SP}|_{F^c}$, is defined by $\Sigma_{(\mathsf{SP}|_{F^c})} = \Sigma_{\mathsf{SP}}$, $E_{(\mathsf{SP}|_{F^c})} = E_{\mathsf{SP}}$, and $\mathcal{M}_{(\mathsf{SP}|_{F^c})} = \{\mathcal{A} \in \mathcal{M}_{\mathsf{SP}} \mid \mathcal{A} \text{ is reachable by } F^c\}$.

UNION For any specifications SP_1 and SP_2 with the same signature Σ, the union $\mathsf{SP}_1 \cup \mathsf{SP}_2$ is defined by
$\Sigma_{(\mathsf{SP}_1 \cup \mathsf{SP}_2)} = \Sigma$, $E_{(\mathsf{SP}_1 \cup \mathsf{SP}_2)} = E_{\mathsf{SP}_1} \cup E_{\mathsf{SP}_2}$, and $\mathcal{M}_{(\mathsf{SP}_1 \cup \mathsf{SP}_2)} = \mathcal{M}_{\mathsf{SP}_1} \cap \mathcal{M}_{\mathsf{SP}_2}$.

TRANS The translation of SP along a signature morphism $\chi : \mathsf{Sig(SP)} \to \Sigma$ denoted by $\mathsf{SP} \star \chi$ is defined by
$\Sigma_{(\mathsf{SP}\star\chi)} = \Sigma$, $E_{(\mathsf{SP}\star\chi)} = \chi(E_{\mathsf{SP}})$, and $\mathcal{M}_{(\mathsf{SP}\star\chi)} = \{\mathcal{A} \in |\mathsf{Mod}(\Sigma)| \mid \mathcal{A} \restriction_{\chi} \in \mathcal{M}_{\mathsf{SP}}\}$.

$\mathcal{H}$-FREE Given a class of homomorphisms $\mathcal{H}$, for all specifications SP, all basic specifications (Σ, E), and all presentation morphisms $\chi : (\Sigma_{\mathsf{SP}}, E_{\mathsf{SP}}) \to (\Sigma, E)$, the $\mathcal{H}$-free (Σ, E)-specification over SP via χ, denoted $(\Sigma, E)!_{\mathcal{H}}\mathsf{SP}$, is defined by
$\Sigma_{(\Sigma,E)!_{\mathcal{H}}\mathsf{SP}} = \Sigma$, $E_{(\Sigma,E)!_{\mathcal{H}}\mathsf{SP}} = E$, and
$\mathcal{M}_{(\Sigma,E)!_{\mathcal{H}}\mathsf{SP}} = \{T_{(\Sigma,E)}(\mathcal{A}) \mid \mathcal{A} \in \mathcal{M}_{\mathsf{SP}} \text{ with the universal arrow } \eta : \mathcal{A} \to T_{(\Sigma,E)}(\mathcal{A}) \restriction_{\chi} \text{ in } \mathcal{H}\}$.

$\mathcal{H}$-FREE is the standard free semantics operator. This operator is parameterized by a class of homomorphisms $\mathcal{H}$, which is one of the following ones: (a) ID, the class of identities, (b) EX, the class of inclusions, or (c) US, the class of all homomorphisms. Identities yield *protecting* imports, which do neither collapse elements nor add new elements to the models of the imported module. In the algebraic specification literature, these conditions are known as "no junk and no confusion" conditions, respectively. Inclusions yield importations that allow "junk" but forbid "confusion." If $\mathcal{H}$ consists of all homomorphisms then both "junk" and "confusion" are allowed.

There are other specification building operators defined in the algebraic specification literature, which are useful in practice and cannot be derived from the ones defined in this survey. Some examples are the derivation across signature morphisms used for hiding information [124] or the extension operator used for capturing non-protecting importation modes [33]. A basic property of structured specifications is given in the following lemma.

Lemma 4.4. *For each structured specification* SP, $\mathcal{A} \in \mathcal{M}_{\mathsf{SP}}$ *implies* $\mathcal{A} \models E_{\mathsf{SP}}$.

The predefined module BOOL is, perhaps, the most used (labelled) specification in the OBJ languages. BOOL is not a specification of Boolean algebras. It defines the truth values used in the verification methodology.

Example 4.5 (Booleans). The following is the specification of truth values *true* and *false* with the usual operations defined on them.

```
mod! BOOL{
```

```
[Bool]                              eq true and A = A .
op true : -> Bool                   eq false and A = false .
op false : -> Bool                  eq A and A = A .
op _and_ : Bool Bool -> Bool        eq false xor A = A .
op _or_ : Bool Bool -> Bool         eq A xor A = false .
op _xor_ : Bool Bool -> Bool        eq A and (B xor C) = A and B xor A and C .
op not_ : Bool -> Bool              eq not A = A xor true .
op _implies_ : Bool Bool -> Bool    eq A or B = A and B xor A xor B .
vars A B C : Bool .                 eq A implies B = not (A xor A and B) . }
```

BOOL is defined with initial semantics, which means that the denotational semantics of BOOL consists of (the class of all algebras isomorphic to) the initial algebra of $(\Sigma_{\mathsf{BOOL}}, E_{\mathsf{BOOL}})$, that is, the algebra with the carrier set $\{true, false\}$ that interprets the Boolean operations in the usual way. The free semantics has many implications in both specification and verification methodologies, which will be explained gradually in the present contribution.

Example 4.6 (Labels). The following specification defines some labels denoting the state of programs/processes trying to access a common resource.

```
mod* LABEL{
 pr(BOOL)                          eq [e1]: (L = L) = true .
 sort Label                        ceq[e2]: L1 = L2 if (L1 = L2) = true .
 ops rs ws cs : -> Label {constr}  eq [l1]: (rs = cs) = false .
 op _=_ : Label Label -> Bool      eq [l2]: (ws = cs) = false .
 vars L L1 L2 : Label              eq [l3]: (cs = rs) = false . }
```

The specification BOOL is imported in protecting mode. Therefore, Σ_{LABEL} is obtained from Σ_{BOOL} by adding the symbols described on the left column of Example 4.6. The constants *rs*, *ws*, and *cs* denote the reminder section, waiting section, and critical section, respectively. The operation `_=_ : Label Label` $\longrightarrow$ `Bool` denotes the equality. By the first two equations, $e1$ and $e2$, the object level equality denotes the real equality among labels, that is, two labels a, b are equal in a model $\mathcal{A} \in \mathcal{M}_{\mathsf{LABEL}}$ iff $\mathcal{A}_{=}(a, b) = \mathcal{A}_{true}$. Since BOOL is imported in protecting mode, the equations $l1$ — $l3$ say that no "confusion" is allowed between the labels *rs*, *ws*, and *cs*. Because the constants *rs*, *ws*, and *cs* are also constructors, no "junk" is allowed to the sort `Label`. It follows that the semantics of LABEL consists of the initial model.

Note that the importation of BOOL in protecting mode and the equations $e1, e2$ allow one to write sentences semantically equivalent to inequalities. For example, $l1$, $l2$, and $l3$ are semantically equivalent to $rs \neq cs$, $ws \neq cs$, and $cs \neq rs$, respectively. Unsurprisingly, there are structure specifications which are inconsistent, i.e., with no models.

Since LABEL imports BOOL in protecting mode, there exists a signature inclusion $\iota : \Sigma_{\mathsf{BOOL}} \hookrightarrow \Sigma_{\mathsf{LABEL}}$. If we ignore the constructor declarations, the left column of Example 4.6 corresponds to $\mathsf{BOOL} \star \iota$, while the right column to $(\Sigma_{\mathsf{LABEL}}, \{e1, e2, l1, l2, l3\})$. Let F^{c}_{LABEL} denote the set of constructors $\{rs :\longrightarrow \mathtt{Label}, ws :\longrightarrow \mathtt{Label}, cs :\longrightarrow \mathtt{Label}\}$ and notice that $\mathsf{LABEL} = (\mathsf{BOOL} \star \iota \cup (\Sigma_{\mathsf{LABEL}}, E_{\mathsf{BOOL}} \cup \{e1, e2, l1, l2, l3\}))|_{F^{c}_{\mathsf{LABEL}}}$.

Example 4.7. The following is a specification of the usual ordering on natural numbers, which is then extended to an ordering on $\omega + 1$.

```
mod! PNAT{
 protecting(BOOL)
 [Nat]
 op 0 : -> Nat                              mod! OMEGA{
```

```
op s_ : Nat -> Nat
op _<=_ : Nat Nat -> Bool
vars X Y : Nat
eq [o1]: X <= X = true .
cq [o2]: X <= s Y = true if X <= Y .}
```

```
extending(PNAT)
op omega : -> Nat
vars X Y : Nat
eq [o3]: s omega = omega .
eq [o4]: X <= omega = true .}
```

Since BOOL is imported in protecting mode, Σ_{PNAT} is obtained from Σ_{BOOL} by adding the sort Nat and the function symbols described on the left column of Example 4.7. We have $\mathsf{PNAT} = (\Sigma_{\mathsf{PNAT}}, E_{\mathsf{BOOL}} \cup \{o1, o2\})!_{\mathsf{ID}}\mathsf{BOOL}$. Similarly, Σ_{OMEGA} is obtained from Σ_{PNAT} by adding the constant $omega :\longrightarrow Nat$. It follows that $\mathsf{OMEGA} = (\Sigma_{\mathsf{OMEGA}}, E_{\mathsf{PNAT}} \cup \{o3, o4\})!_{\mathsf{EX}}\mathsf{PNAT}$.

4.5 Parameterized specifications

Another important concept for the modularization of system specifications is that of parameterized specification, which plays a crucial role in scaling up to the complexity of software. The basic mechanism for the instantiation of parameters relies on pushouts of signature morphisms. The concept of parameterized specification and of pushout-style instantiation of parameters originates from the work on Clear [19] and constitutes the basis of parameterized specifications in OBJ family of languages.

Presentation morphisms are extended naturally to specifications. A *specification morphism* $\chi : \mathsf{SP} \longrightarrow \mathsf{SP}'$ is a signature morphism $\chi : \Sigma_{\mathsf{SP}} \longrightarrow \Sigma_{\mathsf{SP}'}$ such that for all $\mathcal{A}' \in \mathcal{M}_{\mathsf{SP}'}$ we have $\mathcal{A} \restriction_{\chi} \in \mathcal{M}_{\mathsf{SP}}$. If both SP and SP′ are structured specifications then we require also that $E_{\mathsf{SP}'} \models \chi(E_{\mathsf{SP}})$.

Definition 4.8 (Parameterized specification). A parameterized specification SP(P) is a specification morphism $\mathsf{P} \longrightarrow \mathsf{SP}$ such that the underlying signature morphism is an inclusion $\Sigma_{\mathsf{P}} \subseteq \Sigma_{\mathsf{SP}}$ which

(1) does not allow new subsorts for P, that is, for all $s \in S_{\mathsf{SP}}$ and all $s_0 \in S_{\mathsf{P}}$ such that $s \leq_{\mathsf{SP}} s_0$, we have $s \leq_{\mathsf{P}} s_0$, and
(2) preserves the subsort polymorphic families of operations, that is, for all $\sigma : w \longrightarrow s \in F_{\mathsf{P}}$ and all $\sigma : w' \longrightarrow s' \in F_{\mathsf{SP}}$ such that $w \equiv_{\leq_{\mathsf{SP}}} w'$, we have $\sigma : w' \longrightarrow s' \in F_{\mathsf{P}}$.

The specification P is the *parameter* and the specification SP is the *body* of the parameterized specification SP(P).

The conditions of Definition 4.8 will ensure that the instantiation of parameters is well-defined [72]. From a practical point of view, parameters are importations in protecting mode. We give an example of parameterized specification written in CafeOBJ notation.

Example 4.9 (Queue). Let TRIV be the specification that consists of only one sort *Elt*, that is, $\Sigma_{\mathsf{TRIV}} = (\{Elt\}, \emptyset, \emptyset)$ and $E_{\mathsf{TRIV}} = \emptyset$. The following is a specification of queues of arbitrary elements:

```
mod! QUEUE (X :: TRIV) {
 [EQueue NeQueue < Queue]
 op empty : -> EQueue {constr}
 op _;_ : Elt.X Queue -> NeQueue {constr}
 op head : NeQueue -> Elt.X
 op tail : NeQueue -> Queue
 op add : Queue Elt.X -> Queue
```

```
var Q : Queue
var E F : Elt.X
eq [q1]: top(E ; Q) = E .
eq [q2]: head(E ; Q)= Q .
eq [q3]: add(empty,F) = F ; empty .
eq [q4]: add(E ; Q, F)= E ; add(Q,F). }
```

The parameter of QUEUE is TRIV. The signature Σ_{QUEUE} is obtained from $\Sigma_{\mathsf{TRIV}} = (\{Elt\}, \emptyset, \emptyset)$ by adding the function symbols defined on the right column of Example 4.9, and $E_{\mathsf{QUEUE}} = \{q1, q2, q3, q4\}$. Obviously, the inclusion $\Sigma_{\mathsf{TRIV}} \hookrightarrow$

$\Sigma_{\mathtt{QUEUE}}$ satisfies the reflection and preservation conditions required by Definition 4.8. Since QUEUE is defined with free semantics and TRIV is imported in protecting mode, $\mathtt{QUEUE} = (\Sigma_{\mathtt{QUEUE}}, E_{\mathtt{QUEUE}})!_{\mathtt{ID}}\mathtt{TRIV}$.

Recall the specification of natural numbers of Example 4.7, and assume we want a specification of lists of natural numbers. Let $v : \mathtt{TRIV} \to \mathtt{PNAT}$ be the specification morphism which maps *Elt* to *Nat*. The instance of QUEUE(TRIV) by v, denoted QUEUE(TRIV $\Leftarrow v$), or simply, QUEUE(PNAT) if there is no danger of confusion, is defined as the pushout of $\mathtt{QUEUE} \overset{\chi}{\hookleftarrow} \mathtt{TRIV} \overset{v}{\to} \mathtt{PNAT}$:

(1) The signature of $\Sigma_{\mathtt{QUEUE(PNAT)}}$ is the vertex of the following pushout of signatures.

$$\begin{array}{ccc} \Sigma_{\mathtt{QUEUE(TRIV)}} & \overset{v'}{\dashrightarrow} & \Sigma_{\mathtt{QUEUE(PNAT)}} \\ {\scriptstyle\chi}\uparrow & & \uparrow{\scriptstyle\chi'} \\ \Sigma_{\mathtt{TRIV}} & \xrightarrow[v]{} & \Sigma_{\mathtt{PNAT}} \end{array}$$

The vertex $\Sigma_{\mathtt{QUEUE(PNAT)}}$ of the above pushout is obtained from $\Sigma_{\mathtt{QUEUE(TRIV)}}$ by substituting the sort Nat for Elt and by adding all the function symbols from $\Sigma_{\mathtt{PNAT}}$.

(2) Let $\mathcal{N}$ be the unique (up to isomorphism) algebra of PNAT and let $\mathbb{N} = \mathcal{N} \restriction_v$. We denote by $\mathcal{L}$ the free $(\Sigma_{\mathtt{QUEUE(TRIV)}}, E_{\mathtt{QUEUE(TRIV)}})$-algebra over $\mathbb{N}$ and notice that $\mathcal{L} \restriction_\chi = \mathbb{N}$. By [72, Theorem 23], there exists a unique algebra $\mathcal{L}'$ such that $\mathcal{L}' \restriction_{v'} = \mathcal{L}$ and $\mathcal{L}' \restriction_{\chi'} = \mathcal{N}$, which is exactly the free $(\Sigma_{\mathtt{QUEUE(PNAT)}}, E_{\mathtt{QUEUE(PNAT)}})$-algebra over $\mathcal{N}$. The class $\mathcal{M}(\mathtt{SP})$ consists of all algebras isomorphic to $\mathcal{L}'$.

(3) $E_{\mathtt{QUEUE(PNAT)}} = v'(E_{\mathtt{QUEUE(TRIV)}}) \cup E_{\mathtt{PNAT}}$.

It follows that $\mathtt{QUEUE(PNAT)} = (\Sigma_{\mathtt{QUEUE(PNAT)}}, E_{\mathtt{QUEUE(PNAT)}})!_{\mathtt{ID}}\mathtt{PNAT}$.

4.6 Specification proof calculus

In this section, we define an entailment system for reasoning formally about the properties of structured specifications. Its definition is based on the entailment system described in Definition 3.14 for inferring new sentences for an initial set of axioms.

Definition 4.10. An entailment system for reasoning formally about the logical consequences of structured specifications consists of a family of unary relations on the set of sentences $\{\mathtt{SP} \vdash \_ \mid \mathtt{SP} \text{ is a structured specification}\}$ satisfying the following properties:

$$(\textit{Monotonicity})\ \frac{}{\mathtt{SP} \vdash E}\ [E \subseteq E_{\mathtt{SP}}] \qquad (\textit{Union})\ \frac{\mathtt{SP} \vdash E_1 \quad \mathtt{SP} \vdash E_2}{\mathtt{SP} \vdash E_1 \cup E_2}$$

$$(\textit{Translation})\ \frac{\mathtt{SP} \vdash E}{\mathtt{SP} \star \chi \vdash \chi(E)}\ [\chi : \Sigma_{\mathtt{SP}} \to \Sigma] \qquad (\textit{Cut})\ \frac{\mathtt{SP} \vdash E_1 \quad \mathtt{SP} \cup (\Sigma_{\mathtt{SP}}, E_1) \vdash E_2}{\mathtt{SP} \vdash E_2}$$

The operational semantics of a specification SP is obtained by orienting the equations in $E_{\mathtt{SP}}$ from left to right. An unconditional term rewriting step is defined by $C[l\theta] \Rightarrow_{\mathtt{SP}} C[r\theta]$ for all equations $\forall X \cdot l = r \in E_{\mathtt{SP}}$, all substitutions $\theta : X \to T_{(\Sigma_{\mathtt{SP}})}(Y)$ and all contexts C, that is, terms with exactly one occurrence of a special variable $\square$.[5] The term rewriting relation $\overset{*}{\Rightarrow}_{\mathtt{SP}}$ is the reflexive-transitive closure of $\Rightarrow_{\mathtt{SP}}$. For details about the definition of term rewriting relation, we recommend [131]. Two terms t_1 and t_2 are *joinable*, in symbols, $\mathtt{SP} \vdash t_1 \downarrow t_2$ if there exists another term t such that $t_1 \overset{*}{\Rightarrow}_{\mathtt{SP}} t$ and $t_2 \overset{*}{\Rightarrow}_{\mathtt{SP}} t$. To understand the proof score methodology, it is not necessary to know the details of term rewriting relation, and its implementation can be regarded as a black-box.

[5] Notice that $C[l\theta]$ denotes the term obtained from C by substituting the term $l\theta$ for the variable $\square$.

A specification SP satisfies a set of sentences E over Σ_{SP}, in symbols, $\mathsf{SP} \models E$ iff $\mathcal{A} \models E$ for all order-sorted algebras $\mathcal{A} \in \mathcal{M}_{\mathsf{SP}}$. An example of structured entailment system is the semantic structured entailment system $\{\mathsf{SP} \models \_ \mid \mathsf{SP} \text{ is a structured specification}\}$. To check that $\mathsf{SP} \models E$ one needs to consider all order-sorted algebras $\mathcal{A} \in \mathcal{M}_{\mathsf{SP}}$, which is not feasible, since $\mathcal{M}_{\mathsf{SP}}$ is a class (not even a set). Therefore, an entailment system generated by a finite set of proof rules is proposed.

Definition 4.11 (Order-sorted specification calculus). The order-sorted specification calculus consists of the least entailment system for structured specifications closed under the following proof rules:

$$(\mathit{Implication}_1)\ \frac{\mathsf{SP} \vdash t_0 = t_0' \text{ if } \bigwedge_{i=1}^{n} t_i = t_i'}{\mathsf{SP} \cup (\Sigma_{\mathsf{SP}}, \{t_1 = t_1, \ldots, t_n = t_n'\}) \vdash t_0 = t_0'} \qquad (\mathit{Implication}_2)\ \frac{\mathsf{SP} \cup (\Sigma_{\mathsf{SP}}, \{t_1 = t_1', \ldots, t_n = t_n'\}) \vdash t_0 = t_0'}{\mathsf{SP} \vdash t_0 = t_0' \text{ if } \bigwedge_{i=1}^{n} t_i = t_i'}$$

$$(\mathit{Quantification}_1)\ \frac{\mathsf{SP} \vdash \forall X \cdot e}{\mathsf{SP} \star \chi \vdash e}\ [\ \chi : \Sigma_{\mathsf{SP}} \hookrightarrow \Sigma_{\mathsf{SP}}(X)\ ] \qquad (\mathit{Quantification}_2)\ \frac{\mathsf{SP} \star \chi \vdash e}{\mathsf{SP} \vdash \forall X \cdot e}\ [\ \chi : \Sigma_{\mathsf{SP}} \hookrightarrow \Sigma_{\mathsf{SP}}(X)\ ]$$

$$(\mathit{Rewriting})\ \frac{(\Sigma_{\mathsf{SP}}, E_{\mathsf{SP}}) \vdash t_1 \downarrow t_2}{\mathsf{SP} \vdash \forall X \cdot t_1 = t_2} \qquad (\mathit{Ind})\ \frac{\mathsf{SP} \star \chi^{\sigma} \cup (\Sigma_{\mathsf{SP}}(X^{\sigma}), \mathtt{IH}^{\sigma}) \vdash e^{\sigma} \text{ for all } \sigma : w \longrightarrow s \in F_{\mathsf{SP}}^{c} \text{ with } s \leq_{\mathsf{SP}} s_x}{\mathsf{SP} \vdash \forall x \cdot e}$$

where s_x is the sort of x, F_{SP}^{c} consists of all the constructors of SP, $X^{\sigma} = \{x_1 \ldots x_n\}$ is a set of new variables, one for each sort in the arity $w = s_1 \ldots s_n$, $\chi^{\sigma} : \Sigma_{\mathsf{SP}} \hookrightarrow \Sigma_{\mathsf{SP}}(X^{\sigma})$ is a signature inclusion, $\mathtt{IH}^{\sigma}$ is the induction hypothesis $\{e\theta \mid \theta : \{x\} \longrightarrow X^{\sigma} \text{ is a sort decreasing mapping}\}$, and e^{σ} is obtained from e by substituting $\sigma(x_1, \cdots, x_n)$ for x.

A few remarks are in order. (*Reflexivity*), (*Symmetry*), (*Transitivity*), (*Congruence*) and (*Substitutivity*) are replaced by (*Rewriting*). According to [65], we have $E_{\mathsf{SP}} \vdash \forall X \cdot t_1 = t_2$ iff $(\Sigma_{\mathsf{SP}}, E_{\mathsf{SP}}) \vdash t_1 \downarrow t_2$ provided that the term rewriting system $(\Sigma_{\mathsf{SP}}, E_{\mathsf{SP}})$ is *confluent* and *terminating*. Structural induction (*Ind*) is sound for all reachable models, which are defined by the specification building operator **CONS** in the presence of some constructor operators.

5 THE TOOLS

We present in this section the different implementations where proof scores can be executed. We discuss their unique features and the main differences between them. Although a detailed comparison of these programming languages is beyond the scope of this paper, interested readers can find a simple mutual exclusion protocol for two processes implemented for all of them at https://github.com/ariesco/proof-scores-survey.

5.1 OBJ3

Although it is currently discontinued, OBJ3 [67] was the first programming supporting verification by proof scores. Because CafeOBJ and Maude are sister languages extending OBJ3, it is worth discussing its main features.

In OBJ3 modules are called objects, which can be defined with both tight and loose semantics. Sorts, operators, equational axioms, and equations can be stated in objects. The predefined equality between terms checks syntactic equality, which might be inconvenient when a term cannot be completely reduced (it would return `false` prematurely), as it will probably happen in the intermediate steps of our proof. Hence, it is required in general to define a new equality predicate and define it by means of equations. Finally, structured specifications are possible, supporting *protecting* (no "junk" and no "confusion" are allowed, as explained in the previous section), *extending* (no "confusion" is allowed), and *using* (both "junk" and "confusion" are allowed).

Proof scores are defined in open-close environments. These environments allow users to import and enrich other modules and execute reduction commands without creating a new module. For example, a single proof score is usually enough for discharging the base case because no case splittings (and hence new equations) are required, but the rest of constructors require several open-close environment to distinguish different cases.

5.2 CafeOBJ

CafeOBJ [50, 126] is a programming language that inherits OBJ3 features and philosophy and adds support for hidden algebras [68] and rewriting logic [100], which can be combined with order-sorted algebra. This combined approach has a formal mathematical foundation based in multiple institutions [50]. Besides these new theoretical features, CafeOBJ is implemented in Lisp and provides a better performance than OBJ3, new predicates, such as a `search` predicate to test reachability, and new tools, such as an inductive theorem prover [126]. Because CafeOBJ is currently the language with the best support for proof scores, we will use it for all the examples in this paper. We present below some details of the syntax, which has been introduced in the previous section; a complete example will be discussed in Section 6.1.

CafeOBJ supports modules with tight semantics, with syntax `mod! MOD-NAME {...}`, and loose semantics, with syntax `mod* MOD-NAME {...}`. Both types of modules can be parameterized and the same importation modes presented for OBJ3 can be used in CafeOBJ. Sorts are enclosed in square brackets, while the subsort relation is stated by means of `<`. Operators use the keyword `op` (`ops` if several operators are defined at the same time), followed by the operator name, `:`, the arity, `->`, the coarity, and a set of attributes enclosed in curly braces; among these attributes we have `ctor` for constructors, `assoc` for associativity, `comm` for commutativity, and `id:` for identity. The behavior of functions is defined by means of equations, with syntax `eq` (`ceq` for conditional equations). It is worth remarking that CafeOBJ provides a predefined `_=_` predicate for all sorts and a default equation indicating that it holds for terms syntactically equal. The user can enrich this definition by adding extra equations.

Proof scores, with syntax `open MOD-NAME . STMS close`, take the definitions in the module `MOD-NAME` and extend them with the statements in `STMS`. These statements include fresh constants, premises (possibly using the `:nonexec` attribute to prevent them from being used in reductions), and reduce commands, with syntax `red`.

5.3 Maude

Maude [26] is a programming language implementing rewriting logic [100], a logic of change that is parameterized by an equational logic. In Maude this logic is membership equational logic [17], an extension of order-sorted equational logic that allows specifiers to state terms as having a given sort. It supports parameterization, structured specifications, equational axioms, and reduction via rewriting. Maude has been widely used for specifying and verifying systems, mainly focusing on its built-in LTL model checker [26].

However, Maude does not support proof scores explicitly. When no extra constants or equations are needed, for example for base cases, we can just select the appropriate module and reduce the term. Otherwise, we need to create new modules, import the specification (Maude only allows theories to be imported in *including* mode, which allows junk and confusion to be added) and add constants and equations in the usual way. Then, reduce commands can be used outside the modules. This way of defining proof scores, although very similar to the ones we have already discussed, is more verbose. Because a major advantage of proof scores is their flexibility and its ease of use, this extra effort might make Maude inconvenient for large proofs. Moreover, as OBJ3, Maude does not include a predefined equality predicate and it must be explicitly defined. A detailed comparison between CafeOBJ and Maude can be found in [117].

5.4 CafeInMaude

CafeInMaude [117] is a CafeOBJ interpreter implemented in Maude [26]. It supports non-behavioral CafeOBJ specifications, open-close environments (and hence proof scores), and theorem proving. Because CafeInMaude supports CafeOBJ modules and open-close environments, it is possible to use exactly the same code presented in Section 5.2. However, CafeInMaude provides extra features, discussed in Section 6.2, when proof scores are identified with the `:id(...)` annotation.

5.5 Comparison between implementations

Table 1 summarizes the main aspects of the implementations above, where we have excluded OBJ3 because it is discontinued. We have considered, for each implementation, the following features:

- The implementation language, which in some cases might limit the performance.
- The support for open-close environments in the language.
- The support for hidden algebras [68], which are algebraic structures defined through their observable properties, focusing on external behavior without exposing their internal implementation.
- The existence of a search predicate, which has been used for exploring the state-space in specific kinds of proof scores [48].
- The existence of a search command, which has been used for checking invariant properties.
- The possibility of using model checking [25] for analyzing linear temporal logic (LTL) properties on the specifications. In this way, safety properties can be easily analyzed.
- The support of narrowing [45, 101], a generalization of term rewriting that allows logical variables in terms, replacing pattern matching by unification. This allows systems for symbolically evaluating these terms.
- The existence of a meta-level, where modules can be used as standard data, hence allowing developers to reason about them.

Although some of these features are not directly related to proof scores, they also illustrate the potential of these tools to improve their current approaches and their power as integrated ecosystems for specifying and analyzing the properties of systems.

CafeOBJ is the only language implementing hidden algebra, which makes this implementation the most appropriate for designers following this methodology; on the other hand, its Lisp implementation limits its performance and makes it difficult to extend. We also realize that Maude is the most developed system, where novel approaches for verification have been implemented lately (as discussed in [118], narrowing can be used to partially automate proofs). However, the lack of open-close environments prevents users from using proof scores at their full capability. CafeInMaude stands as an intermediate option: it supports open-close environments and the search predicate, making it appropriate for using proof scores. Moreover, it is implemented in Maude, which eases the integration of Maude features, such as narrowing, into proof scores.

6 PROOF SCORES IN PRACTICE

In this section, we first present the main approach to proof scores: verification of observational transition systems. We then discuss how new implementations and tools for CafeOBJ have improved the analysis of this kind of systems.

Table 1. Comparison between implementations

	Imp. lang.	Open-close environment	Hidden algebra	Search predicate	Search command	Model checking	Narrowing	Metalevel
CafeOBJ	Lisp	Yes	Yes	Yes	No	No	No	No
Maude	C++	No	No	No	Yes	Yes	Yes	Yes
CafeInMaude	Maude	Yes	No	Yes	No	Yes	No	Yes

6.1 Observational Transition Systems

Observational transition systems (OTS) [109, 110] are a general methodology for formally specifying distributed systems. The intuitive idea is to model system interactions by means of *transitions*, while the values of the relevant components of the system are obtained by means of *observations*.

The general notions of OTSs were first developed for behavioral specifications [37], which distinguished two different types of sorts: *visible sorts*, used to define the data structures, and *hidden sorts*, used to define the system. Because in these specifications the system is hidden it can only be analyzed via observations that show the value of particular components at each particular moment. This approach was based on *hidden algebras* [68], which thanks to its co-algebraic nature allowed specifiers to prove both invariant and liveness properties. The constructor-based approach (see Section 3 for details) followed here is limited to invariant properties.

The elements required for defining an OTS are:

- A sort `Sys` standing for the system.
- A set of sorts $\mathcal{S}$ standing for the auxiliary data structures used by `Sys`.
- The transitions available for the system. These transitions are defined as constructor functions that return terms of sort `Sys`. Following these ideas, a standard definition of `Sys` with an initial state and n transitions looks like:
  - `op init : -> Sys {constr}`
  - `op` $\mathtt{trans}_1$ `:` `Sys` $\mathtt{Sort}_1^1 \cdots \mathtt{Sort}_{k_1}^1$ `-> Sys {constr}`
  - $\cdots$
  - `op` $\mathtt{trans}_n$ `:` `Sys` $\mathtt{Sort}_1^n \cdots \mathtt{Sort}_{k_n}^n$ `-> Sys {constr}`

  With all sorts $\mathtt{Sort}_j^i \in \mathcal{S}$, with $i \in \{1, \ldots, n\}$, $j \in \{1, k_i\}$. In general, for each transition (except for `init`) to be triggered a condition $c_{trans_i}, i \in \{1, \ldots, n\}$, must be met. This condition is the so-called *effective condition*; when it is not met, we have $\mathtt{trans}_\mathtt{i}(s, x_1, \ldots, x_{k_i}) = s$, for `s` a variable of sort `Sys` and $x_1, \ldots, x_{k_i}$ variables of sort $\mathtt{Sort}_1^i \cdots \mathtt{Sort}_{k_i}^i$, respectively.
- Observation functions, which return the value of the data structures in the different states. These functions are of the form:
  - `op` $\mathtt{o}_1$ `:` `Sys` $\mathtt{Sort}_1^1 \cdots \mathtt{Sort}_{l_1}^1$ `->` $\mathtt{Sort}_1$
  - $\cdots$
  - `op` $\mathtt{o}_m$ `:` `Sys` $\mathtt{Sort}_1^m \cdots \mathtt{Sort}_{l_m}^m$ `->` $\mathtt{Sort}_m$

  With all sorts $\mathtt{Sort}_j^i \in \mathcal{S}$, $\mathtt{Sort}_i \in \mathcal{S}$, with $i \in \{1, \ldots, m\}$, $j \in \{1, l_i\}$, and $\mathtt{Sort}_i$ different to `Sys`.
- For each observation function o_i ($i \in \{1, \ldots, m\}$) we define equations for each transition, assuming the effective condition holds:
  - `eq` $\mathtt{o}_\mathtt{i}(\mathtt{init}, \mathtt{x}_1^\mathtt{i}, \cdots, \mathtt{x}_{l_i}^\mathtt{i}) = v$
  - `ceq` $\mathtt{o}_\mathtt{i}(\mathtt{trans}_1(\mathtt{s}, \mathtt{y}_1^1, \cdots, \mathtt{y}_{k_1}^1), \mathtt{x}_1^\mathtt{i}, \cdots\ \mathtt{x}_{1_i}^\mathtt{i}) = f(s, y_1^1, \cdots, y_{k_1}^1, x_1^i, \cdots\ x_{l_i}^i)$ if c_{trans_1}

– ...

– ceq $\mathtt{o_i}(\mathtt{trans_n}(\mathtt{s}, \mathtt{y}^{\mathtt{n}}_1, \cdots, \mathtt{y}^{\mathtt{n}}_{\mathtt{k_n}}), \mathtt{x}^{\mathtt{i}}_1, \cdots\ \mathtt{x}^{\mathtt{i}}_{\mathtt{l_i}}) = f(s, y^n_1, \cdots, y^n_{k_n}, x^i_1, \cdots\ x^i_{l_i})$ if c_{trans_n}
With x^i_j ($j \in \{1, \ldots, l_i\}$) and y^p_q ($p \in \{1, \ldots, n\}, q \in \{1, \ldots, k_p\}$) variables of sort $\mathtt{Sort}^{\mathtt{i}}_{\mathtt{j}}$ and $\mathtt{Sort}^{\mathtt{p}}_{\mathtt{q}}$, respectively, v a constant of sort $\mathtt{Sort}^{\mathtt{i}}$, and f an auxiliary function (possibly an observation function or a data structure) not involving any transition.

In the following we detail how to define OTSs and how to verify them using proof scores by means of the Qlock example. Qlock is a mutual exclusion protocol based on a queue with atomic operations and a binary semaphore. The basic idea is that processes can be in *remainder section*, *waiting section*, and *critical section*. When a process in the remainder section moves to the waiting section its process identifier is added to the queue; once it reaches the top of the queue it moves into the critical section. Processes exiting the critical section return to the remainder section and the top element is removed from the queue. Although this system is not very complex, proving mutual exclusion requires a proof score of approximately 350 lines of code and illustrates some interesting features, as we will see below.

To specify this system as an OTS we need to identify the transitions and the values we want to observe. It is clear in this case that transitions take place when the processes change the state, while we want to observe, for each process identifier, the state the process is in. Hence, we start our specification by defining the auxiliary data structures. The module `LABEL` for labels was already presented in Example 4.6. The module `PID` defines the process identifiers: we define the sort `Pid` for process identifiers, `ErrPid` for erroneous process identifiers, and `Pid&Err` as a supersort including both correct and erroneous identifiers. This module also indicates that correct and erroneous process identifiers are different:

```
mod* PID {    [ErrPid Pid < Pid&Err]
              op none : -> ErrPid
              var I : Pid
              var EI : ErrPid
              eq (I = EI) = false . }
```

We define next a module `TRIVerr` with loose semantics to define the elements of a generic queue, which will be later instantiated with the appropriate sorts. Because some functions on queues are partial, we define the existence of both correct elements (`Elt`) and erroneous elements (`ErrElt`), and a supersort enclosing them. The module also defines a particular element of sort `ErrElt`, `err`.

```
mod* TRIVerr {   [ErrElt Elt < Elt&Err]
                 op err : -> ErrElt}
```

The module `QUEUE` below is parameterized by `TRIVerr`. It defines the sort for the empty queue (`EQueue`), non-empty queues (`NeQueue`), and general queues (`Queue`) with the appropriate subsort relation between them. The constructors for queues are `empty`, which builds the empty queue, and `_|_`, which given an element and a general queue returns a non-empty queue. The functions for enqueueing (`enq`) and dequeueing (`deq`) are total and defined as usual; for the particular case of dequeueing an empty queue the function returns the same empty queue, as shown on the right. The function for obtaining the `top` element of a queue is also total when using the error sorts defined above: the top element for an empty queue is `err`:

```
mod! QUEUE(E :: TRIVerr) {
  [EQueue NeQueue < Queue]                          var Q : Queue
  op empty :             -> EQueue {constr}         vars X Y : Elt.E
  op _|_   : Elt.E Queue -> NeQueue {constr}        eq enq(empty,X) = X | empty .
```

```
  op enq : Queue Elt.E -> NeQueue                  eq enq(Y | Q,X) = Y | enq(Q,X) .
  op deq : Queue -> Queue                          eq deq(empty) = empty .
  op top : EQueue -> ErrElt.E                      eq deq(X | Q) = Q .
  op top : NeQueue -> Elt.E                        eq top(empty) = err .
  op top : Queue -> Elt&Err.E                      eq top(X | Q) = X .   }
```

We relate, by means of the view TRIVerr2PID, the elements in TRIVerr and the ones in PID in the expected way:

```
view TRIVerr2PID from TRIVerr to PID {
  sort Elt -> Pid,
  sort ErrElt -> ErrPid,
  sort Elt&Err -> Pid&Err,
  op err -> none   }
```

We use this view to instantiate the queue in the QLOCK module, which also includes the modules LABEL and PID:

```
mod* QLOCK {
  pr(LABEL + PID)
  pr(QUEUE(E <= TRIVerr2PID))
```

We define the sort Sys, standing for the system, and the corresponding transitions building it. In addition to init, which stands for the initial state, we define the transitions want (for a process moving from the remainder section to the waiting section), try (for processes in waiting section trying to enter the critical section), and exit (for processes in the critical section leaving it):

```
  [Sys]
  op init : -> Sys {constr}
  op want : Sys Pid -> Sys {constr}
  op try  : Sys Pid -> Sys {constr}
  op exit : Sys Pid -> Sys {constr}
```

In our system we are interested in the label corresponding to each process and the queue, so we define the appropriate observations to retrieve its value in a given system:

```
  op pc    : Sys Pid -> Label
  op queue : Sys -> Queue
```

We define some variables and define the observations for each state. In the case of the initial state all processes are in the reminder section and the queue is empty:

```
  var S : Sys      vars I J : Pid
  eq pc(init,I)  = rs .
  eq queue(init) = empty .
```

For the transition want we define the effective condition first. In this case to enter into the waiting section is only required that the process is in the remainder section:

```
  op c-want : Sys Pid -> Bool
  eq c-want(S,I) = (pc(S,I) = rs) .
```

When the condition holds the label corresponding to the process is ws, while the queue is modified by introducing the process identifier. When the condition does not hold the transition is not applied:

```
  ceq pc(want(S,I),J)  = (if I = J then ws else pc(S,J) fi) if c-want(S,I) .
```

```
  ceq queue(want(S,I)) = enq(queue(S),I)                    if c-want(S,I) .
  ceq want(S,I)        = S                                  if not c-want(S,I) .
```

We would proceed in a similar way with the rest of transitions. Finally, we define the properties we want to prove. The property `inv1` stands for mutual exclusion, while `inv2` states that the process in the critical section must be the one on the top of the queue:

```
  op inv1 : Sys Pid Pid -> Bool
  op inv2 : Sys Pid -> Bool
  eq inv1(S:Sys,I:Pid,J:Pid) = (((pc(S,I) = cs) and pc(S,J) = cs) implies I = J) .
  eq inv2(S:Sys,I:Pid) = (pc(S,I) = cs implies top(queue(S)) = I) . }
```

In the simplest case, we would just create an open-close environment with both properties and, if both are evaluated to `true`, then we would know they hold:

```
open QLOCK .
  op  s : -> Sys .
  ops i j : -> Pid .
  red inv1(S:Sys, I:Pid, J:Pid) .
  red inv2(S:Sys, I:Pid) .
close
```

As expected, they are not reduced to `true` and we need to inspect each possible transition (that is, applying induction) and each property separately. The proof for `init` is straightforward; it is enough to just reduce the term:

```
  open QLOCK .
     ops i j : -> Pid .
     red inv1(init,i,j) .
  close
```

```
open QLOCK .
   op i : -> Pid .
   red inv2(init,i) .
close
```

We focus next on how to prove `inv1` for the transition `want`, the rest of the proof follows the same ideas and is available in the repository above. The first approach for proving this property would be the open-close environment below:

```
open QLOCK .
   op s : -> Sys .
   ops i j k : -> Pid .
   eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
   eq [:nonexec] : inv2(s,I:Pid) = true .
   red inv1(want(s,k),i,j) .
close
```

Note that it contains the induction hypotheses as non-executable equations, so they are only used for informative purposes. When executed, we obtain the following result:

```
Result: true xor cs = pc(want(s,k),i) and cs = pc(want(s,k),j) xor i = j and
        cs = pc(want(s,k),i) and cs = pc(want(s,k),j) : Bool
```

We could try to use the inductive hypotheses to further reduce it, but it has no effect. Then, we notice that the terms `pc(want(s,k),i)` and `pc(want(s,k),j)` are not being reduced. This happens because there is no information

about the effective condition; in order to solve it we can indicate it holds, adding the corresponding equation to the environment:[6]

```
open QLOCK .
   op  s : -> Sys .
   ops i j k : -> Pid .
   eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
   eq [:nonexec] : inv2(s,I:Pid) = true .
   eq pc(s,k) = rs .
   red inv1(want(s,k),i,j) .
close
Result: true xor cs = if i = k then ws else pc(s,i) fi and cs = if j = k then ws else pc(s,j) fi xor i = j and
        cs = if i = k then ws else pc(s,i) fi and cs = if j = k then ws else pc(s,j) fi : Bool
```

In this case we need to state an equality between the fresh constants. For example, we can add the equation `i = k`:

```
open QLOCK .
   op  s : -> Sys .
   ops i j k : -> Pid .
   eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
   eq [:nonexec] : inv2(s,I:Pid) = true .
   eq pc(s,k) = rs .
   eq i = k .
   red inv1(want(s,k),i,j) .
close
Result: true : Bool
```

Although we have obtained `true`, the goal is not completely proven because it depends of some equations and we do not know what happens if they do not hold. For example, we would need to use `(i = k) = false` instead of `i = k`, which is not reduced to `true` and would require a reasoning similar to the one above.

This proof illustrates the strong points of proof scores: it is possible to prove the subgoals using the same syntax employed to specify the system. Although in this example the equations that we needed to add were simple, it is also possible to define more complex ones, possibly involving equational attributes such as associativity. Moreover, the result obtained when reducing the environment guides the proof, so the user must employ his/her expertise to choose the most promising ones.

On the other hand, this example also illustrates their weak point: the user is in charge of ensuring that all possible cases have been traversed; if one subgoal is not taken into account none of the tools will warn the user, which might be specially problematic with large proofs. This (partial) lack of formality is one of the reasons why proof scores have not been more widely applied, while theorem proving, which has a very strict syntax but also provides full confidence, is very popular. However, the equations added in open close-environments are not arbitrary: they are case splittings and follow some rules. For this reason, we should be able to find a general schema for a large class a proof scores, so we can automatically generate a proof in a "standard" theorem prover. In the next section we present a proof assistant providing several features implicitly used in proof scores and a proof generator that, given a proof score, tries to replicate the proof using the proof assistant.

[6]Notice that other options are also available, being the simplest one adding the equation `i = j`; the user is in charge of exploring the different alternatives and choosing the most appropriate ones.

6.2 The CafeInMaude Proof Assistant and Proof Generator

The CafeInMaude implementation of CafeOBJ eases the specification of systems and the definition of proof scores; in particular, it takes advantage of the underlying Maude implementation and its meta-level features to:

(1) Support extra features in CafeOBJ specifications. In particular, the current version of CafeInMaude supports the `owise` (standing for *otherwise*) attribute for equations, which indicates that the corresponding equation is only used if the rest of equations cannot be executed.
(2) Extend the syntax in an easy way. CafeInMaude is implemented in Maude, taking advantage of its powerful metalevel. In this way, the CafeOBJ grammar is defined as a Maude module used for parsing, while the translation into Maude is performed using equations. Because Maude's and CafeOBJ's syntax and semantics are very similar, CafeOBJ programmers can easily add new elements to the system to experiment with them. Once these elements have been thoroughly tested they might become part of the standard CafeInMaude distribution.
(3) Connect with other Maude tools. CafeInMaude translates CafeOBJ specifications into Maude specifications, so any tool dealing with Maude specifications can be connected with CafeInMaude. In particular, it is interesting to consider the Maude Formal Environment [27], which includes a termination checker, a confluence checker, a coherence checker, and a theorem prover, and the Maude integration with SMT solvers.
(4) Execute larger proofs than the Lisp implementation of CafeOBJ. As shown in [117], the performance of CafeInMaude is much better than the one provided by the CafeOBJ implementation described in Section 5.2, especially when the `search` predicate is used. This allows CafeInMaude to prove properties on some protocols that fail in CafeOBJ because the interpreter runs out of memory.

6.2.1 The CafeInMaude Proof Assistant. Beyond the features above, CafeInMaude provides a theorem prover, the CafeInMaude Proof Assistant (CiMPA) [115], supporting:

- Induction in one or more variables.
- Instantiating free variables with fresh constants (i.e., applying the theorem of constants). In this way, we can reason with the most appropriate values of the variables for the particular subgoal.
- Splitting the goal. When the current goal is composed of several equations, it is useful to discharge each of them separately.
- The use of the induction hypotheses as premises. We can use an implication with one of the induction hypotheses as premise to simplify the current subgoal.
- Case splitting by (i) true/false, (ii) constructors, and (iii) special combinations for associative sequences.
- Discharging the current subgoal by using reduction.

6.2.2 The CafeInMaude Proof Generator. While in CafeOBJ proof scores and proof scripts are unrelated entities (from the tool support point of view), CafeInMaude relates them by means of the CafeInMaudeProofGenerator (CiMPG), which makes proof scores a *formal* (rather than semi-formal) proof technique.

Each subgoal in a CiMPA proof extends the original CafeOBJ module with the assumptions (and the corresponding fresh constants needed to define them) generated due to induction and case splitting. In the same way, each open-close environment consists of a module extended with fresh constants and equations. Hence, it should be possible to relate, up to substitution, both of them and discharge the corresponding subgoal using reduction, possibly using first an implication with a substitution extracted from the open-close environment.

In addition to using induction at the beginning and, possibly, the theorem of constants, the CiMPG algorithm needs to build the sequence of case splittings. In order to generate this sequence it must compute (i) the kind of case splitting applied and (ii) the order in which the case splittings are applied. The details of the algorithm are complex and are described in detail in [115].

If CiMPG is successful, a formal proof for the theorems is obtained; otherwise, it points out those subgoals whose proof could not be generated, which helps users to find errors in proof scores. In this way the user can keep using proof scores, taking advantage of its flexible nature, while leaving the verification of the proof to CiMPG.

In [116] the CafeInMaude Proof Generator and Fixer-Upper (CiMPG+F), a tool for generating complete CiMPA proof scripts from incomplete proof scores, is presented. CiMPG+F follows a bounded depth-first strategy for applying cases splitting to subgoals until they are discharged or the bound is reached. The candidates for case splitting are, roughly speaking, chosen by reducing at the metalevel the subgoals and picking those terms that were not be reduced even though they were not built by using constructor functions.

6.3 Invariant Proof Score Generator

Using CafeInMaude, another tool called Invariant Proof Score Generator (IPSG) [133] has been implemented, which can automate the proof score writing process. Precisely, given a CafeOBJ formal specification, invariant properties formalizing the properties of interest together with an auxiliary lemma collection, IPSG can generate the proof scores verifying those properties.

Returning to the Qlock example introduced in Section 6.1, to prove the two properties `inv1` and `inv2`, we use the following script:

```
ipsgopen QLOCK .
  inv inv1(S:Sys, I:Pid, J:Pid) .
  inv inv2(S:Sys, I:Pid) .
  generate inv1(S:Sys, I:Pid, J:Pid) induction on S:Sys .
  generate inv2(S:Sys, I:Pid) induction on S:Sys .
close
set-output proof-scores.cafe .
:save-proof .
```

Feeding this script into IPSG, the tool will generate the proof scores and save them to file "`proof-scores.cafe`".

How proof scores are generated can be briefly summarized as follows. Starting from a collection of open-close fragments, where each of them does not contain any equation and the most typical induction hypothesis instance is used if it is an induction case, IPSG uses Maude metalevel functionalities to reduce the goal to its normal form. There are three possible cases:

- The obtained result is `true`. The proof is simply discharged.
- The obtained result is `false`. IPSG then tries to discharge the associated case by finding a suitable lemma instance based on the lemma collection provided by human users. The key idea is to enumerate all lemma instances constructed from the terms existing in the current open-close fragment to check if it can discharge the proof.
- The obtained result is x, which is neither `true` nor `false`. A sub-term of x, let's say x', will be then chosen by IPSG to split the current case associated with that open-close fragment into two sub-cases: one when x' holds and the other when it does not. The same procedure is applied for each sub-case produced until either `true` or `false` is returned for the reduction.

Which sub-term x' is chosen at each step determines the order of case splittings, which is a significant factor affecting the tool's efficiency. Some heuristics techniques were implemented for choosing case splitting order. For each induction case, the highest priority is given to those case splittings to reduce the effective condition of the associated transition to either true or false. If term x contains a sub-term **if** c **then** a **else** b **endif**, the sub-terms of the condition c are given higher priority than others. In summary, IPSG can automatically conduct case splitting such that true or false is returned for each sub-case. Human users, therefore, only need to concentrate on the most difficult task in interactive theorem proving, i.e., to conjecture lemmas.

IPSG does not support case splitting based on sort constructors as CiMPA does. To confirm the soundness of the proof scores produced by IPSG, CiMPG is first used to generate proof scripts from the proof scores, and the proof scripts are executed with CiMPA. The finding lemma process may take time if a single lemma instance cannot discharge a case but more than one lemma instance are needed.

Discussion. Among the challenges that proof scores face, two of them are due to their reliance in human involvement: missing open-close environments and making use of strategies unsupported by the theory (e.g. using `false` as a premise, which automatically discharges a goal). Using a theorem prover such as CiMPA (Section 6.2.1) solves these problems but takes us out of the proof scores approach. CiMPG (Section 6.2.2) relates proof scores and theorem provers (CiMPA in this case), so proof scores can be validated.

CiMPG+F (Section 6.2.2) and IPSG (Section 6.3) do not face these challenges, which are specific to proof scores, but more general challenges: those faced by human users when producing a proof. In particular, they help finding case splittings and when to use lemmas for discharging the current goal.

7 RELATED WORK

Several algebraic specification languages were developed from around-80's through 2000's, among which are as follows: ACT ONE [42], CafeOBJ [35], CASL [9], HISP [53], Larch [73], Maude [26], OBJ2 [52], and OBJ3 [70]. OBJ2, OBJ3, CafeOBJ, and Maude are members of the OBJ language family. Static data can be specified in OBJ2 and OBJ3, while dynamic systems can be specified as well in CafeOBJ and Maude.

Larch provides the Larch Prover (LP) [58]. Guttag et al. write the following in the Larch book [73] about proving:

> Proving is similar to programming: proofs are designed, coded, debugged, and (sometimes) documented.

This "proving as programming" concept is shared by proving by proof scores in OBJ languages. Although Larch and OBJ languages are based on algebras, there are some differences. OBJ languages are pure-algebraic specification languages, while Larch is not. OBJ languages adopt the complete set of rewrite rules proposed by Hsiang [82] for propositional calculus, while Larch does not. Larch uses the Knuth-Bendix completion procedure, while OBJ languages do not. Note that OBJ languages use a completion procedure to implement rewriting modulo associativity and/or commutativity, adding some equations [65].

There are several formal verification tools for Maude. Among them are the Inductive Theorem Prover (ITP) [28] (including its latest version, the nuITP [40]), the Constructor-based Inductive Theorem Prover (CITP) [56], the Maude LTL model checker [43], the LTLRuLF model checker [11], and the Maude Invariant Analyzer Tool (InvA) [119], where LTL stands for linear temporal logic and LTLRuLF stands for linear temporal logic of rewriting under localized fairness. Meseguer and Bae [11] proposed a tandem of logics that is the pair $(\mathcal{L}_S, \mathcal{L}_P)$ of logics, where $\mathcal{L}_S$ is the logic of systems and $\mathcal{L}_P$ is the logic of properties. The tandems of logics for these five tools are (OSEL, OSEL), (CbOSEL, OSEL),

(RWL, LTL), (RWL, LTLR), and (RWL, OSEL), respectively, where OSEL, CbOSEL, RWL, and LTLR stand for order-sorted equational logic, constructor-based order-sorted equational logic, rewriting logic, and linear temporal logic of rewriting, respectively. As indicated by the names, the first two tools are theorem provers, while the next two tools are model checkers. The fifth one is a theorem prover.

CASL has a tool support called the Heterogeneous Tool Set (Hets) [104]. Hets deals with multiple different logics seamlessly by treating translation from one logic to another (comorphisms) as a first-class citizen. Among the logics supported by Hets are CoCASL (a coalgebraic extension of CASL), ModalCASL (an extension of CASL with multi-modalities and term modalities) and Isabelle/HOL [108] (an interactive theorem prover for higher-order logic).

Although ITP, CITP, and InvA can automate theorem proving to some extent, human interaction with theorem proving is mandatory. Thus, the tools are interactive theorem provers or proof assistants. Among the other proof assistants are ACL2 [88], Coq [15], Isabelle/HOL [108], HOL4 [130], HOL Light [78], LEGO [96], NuPRL [85], Agda [18], Lean [10, 106], and PVS [113]. The proof assistants have notable applications in industry as well as academia. One distinguished case study conducted with ACL2 is a mechanically checked proof of the floating point division microcode program used on the AMD5$_{\mathrm{K}}$86 microprocessor [103]. The proof was constructed in three steps: (1) the divide microcode was translated into a formal intermediate language, (2) a manually created proof was transliterated into a series of formal assertions in ACL2, and (3) ACL2 certified the assertion that the quotient will always be correctly rounded to the target precision. There are several recent verification case studies using ACL2. Hardin [77] used it to formally verify his implementation of the Dancing Links optimization, an algorithm proposed by Knuth in his book *The Art of Computer Programming* [89, Volume 4B] to provide efficient element removal and restoration for a doubly linked list data structure. The data structure was implemented in the Restricted Algorithmic Rust (RAR), a subset of the Rust programming language crafted in his prior work [76]. He then used his RAR toolchain to *transpile* (i.e., perform a source-to-source translation) the RAR source into the Restricted Algorithmic C (RAC) [122]. The resulting RAC code was converted to ACL2 by leveraging the RAC-to-ACL2 translator [122] and the formal verification was finally conducted. Gamboa et al. [57] presented a tool that can suggest to ACL2 users additional hypotheses based on counterexamples generated when the theorem under verification does not hold as expected with the aim of making the theorem become true. Kumar et al. [93] formalized the GossipSub peer-to-peer network protocol popularly used in decentralized blockchain platform, such as Ethereum 2.0, and verified its security properties.

Coq was used in the CompCert project [95] in which it has been formally verified that an optimizing C compiler generates executable code from source programs such that the former behaves exactly as prescribed by the semantics of the latter. Coq was also used to formally prove the famous four-color theorem [71]. Making good use of Coq, the Iris separation logic framework [87] has emerged as an effective way to reason about concurrent programs in the context of recent advances in compilers and computer processors. Jung et al. [86] used Iris to verify some concurrent data structures where *safety memory reclamation* was taken into account, i.e., an unused memory block by a thread can be safely freed with a guarantee that no other thread accesses such a freed memory block. Mével et al. [102] verified the safety of a concurrent bounded queue with respect to a weak memory model, i.e., the compiler and/or processor can reorder the load and store operations of a single thread to improve the overall performance as long as the reordering does not affect the behavior of that thread. Iris was also applied in an industrial context where some data structures used in Meta were verified [22, 134]. Some other recent studies used Coq to reason about probabilistic programs [3, 140], smart contracts [7, 8, 107], and cryptographic protocols in the game-based model [80].

HOL4 and HOL Light are two members of the HOL theorem prover family. HOL4 is designed to be a high-performance, feature-rich environment, and capable of handling sophisticated proofs and formalization problems. In contrast, HOL

Light, which was mainly developed by John Harrison, focuses on simplicity and "minimalism." While HOL4 offers an extensive feature set and tools for interactive and automated theorem proving, HOL Light aims to provide a clean and straightforward system for higher-order logic proofs, focusing on ease of use and conceptual clarity. The HOL family provers and Isabelle/HOL have been used to verify the IEEE 754 floating-point standard [79] and the Kepler Conjecture [75].

Some proof assistants allow the extraction of executable programs from their formal specifications. Agda supports generating Haskell programs, while Coq supports generating OCaml and Scheme as well as Haskell programs. However, the generated programs are functional, not imperative or concurrent. A technique for translating CafeOBJ specifications describing OTS to Java concurrent programs has been proposed [74], but human users need to write the program manually. The technique only inserts some annotations into the specification to guide humans, who are assumed to not understand the CafeOBJ specifications, to write corresponding Java programs. For example, an annotation can let human user know that the associated code block should be run on a separate thread. Another study [128] has proposed a method and tool for generating Java sequential programs from OTS/CafeOBJ specifications. In summary, for concurrent and distributed systems, there is still a gap between formal specifications used for formal verification and the corresponding implementations. Verifying programs directly in their implementation languages like Java without translating or specifying them in another formal specification language is a possible way to mitigate this gap. KeY [5] makes it possible to do so, though only Java sequential programs are supported. The Java Modeling Language (JML) is used to specify desired properties with preconditions, postconditions, and invariants of the program under verification. Note that multithreading programs are not supported. KeYmaera X [47] is the hybrid version of KeY, designed for verifying hybrid systems. Although some case studies have demonstrated that it is possible to verify hybrid systems with the proof score approach [105, 112], the approach is not specifically designed for such systems.

Among formal methods along the line of one main stream are Z [137] and Event-B [1] in which stepwise refinement plays the central role. These formal methods are equipped with proof assistants or environments in which formal proofs related stepwise refinement are supported. For example, Z/EVES [123] is a proof assistant for Z, Rodin [2] is an open toolset for modeling and reasoning in Event-B in which a proof assistant is available. Event-B can be also verified with the ProB model checker [91], which is integrated into Rodin. Stepwise refinement or simulation-based verification can be conducted for OTSs in CafeOBJ [111, 132], but such formal verification is not supported by CiMPA and CiMPG. It is one possible direction to extend CiMPA and CiMPG so as to support stepwise refinement or simulation-based verification for OTSs.

SAT/SMT solvers and SAT/SMT-based formal verification techniques/tools have been intensively studied because they make it possible to automate formal verification experiments. Among SAT/SMT solvers are Z3 [31] and Yices [41]. The standard input format to SAT/SMT solvers is conjunctive normal form (CNF). Because CNF is not very user friendly, many other tools/environments have been developed in which SAT/SMT solvers are used internally. Among such tools/environments are Dafny [94], Symbolic Analysis Laboratory (SAL) [32] and Why? [46]. These tools/environments have high-level specification languages in which systems are specified and user interfaces for formal verification used by SAT/SMT solvers. Z3 is used by Dafny and Why?, while Yices is used by SAL and Why?. Why? also uses PVS, Isabelle/HOL, Coq, and many other automated theorem provers, such as Vampire [92] and E prover [127].

SAT/SMT solvers have been integrated into or combined with some existing formal methods tools/environments. ALC2 was extended with SMT solvers [114]. SMTCoq [44] is an open-source plugin for Coq that dispatches proof goals to external SAT/SMT. When such solvers successfully prove a goal, they are supposed to return a proof witness, or certificate, which is then used by SMTCoq to automatically reconstruct a proof of the goal within Coq. Barbosa et

al. [13] recently extended SMTCoq with a newly interactive tactic called abduce. When the SMT solver fails to prove a goal G valid under the given hypotheses H, instead of returning a counterexample, this new tactic exploits the abductive capabilities of the CVC5 SMT solver [12] to produce some additional assumptions (formulas) that are consistent with H and make G provable. Rocha et al. proposed rewriting modulo SMT [120] that is a combination of SMT solving, rewriting modulo theories, such as associativity and commutativity, and model checking. Rewriting modulo SMT is suited to model and analyze reachability properties of infinite-state open systems, such as cyber-physical systems.

Some recent work investigated the development of graphical user interfaces for the existing proof assistants, such as those for Coq [38, 39] and those for Lean [10, 106]. Korkut [90] developed a web-based graphical proof assistant - Proof Tree Builder, from which users can construct proofs for simple imperative programs based on Hoare logic [81]. Instead of writing textual proof commands, users of those tools can perform gestural actions like click and drag-and-drop on the graphical interface with terms representing, for example, the current goal, to construct the proof. The authors of those studies argued that this can provide an intuitive and quick way of proof construction.

The proof score approach to formal verification is unique in that (1) proofs (or proof scores) are written in the same language, CafeOBJ in this paper, as the one for systems and property specification and (2) proof scores can be written as programs and then are flexible as programs. However, proof scores are subject to human errors because proof scores basically need to be written by human users as programs. To overcome this weakness, CiMPA and some other proof assistants for CafeOBJ have been built, although these proof assistants dilute the merits of proof scores. To address this issue, CiMPG has been developed. CiMPG overcomes the weakness of proof scores and also keeps the merits of proof scores. One possible evolution of CiMPA and/or CiMPG is to integrate SAT/SMT solvers into them. This would be feasible because rewriting modulo SMT is available in Maude. It is, however, necessary to consider how to use SAT/SMT solvers in CiMPA and/or CiMPG so that the merits of proof scores can be enjoyed. Finally, the generate & check method [49] extend the proof score methodology by combining reductions and searches, which gives hints for future developments of the proof score technique.

Discussion: what features have contributed to the success of other tools? It is worth discussing why some tools have been more widely adopted than proof scores. We will focus on Event-B, Isabelle/HOL, Coq, and Lean.

As we sketched above, Event-B focuses on refinements to gradually add details and functionality to systems. The specifier starts with a very general system and introduces, step by step, improvements and components until the behavior reaches the desired level. Proof obligations are generated for the first specifications and for every enrichment step; automation is supported by tools like Rodin and ProB, introduced above. This process eases verification, as each step should be easier to verify than the complete system. It is also relevant the effort made by the community around the ABZ conference, which proposes case studies for specification and analysis in Event-B. The research in papers like [98, 99] is illustrative of the discussion above: the case studies, the adaptive exterior light system for cars and the model of a mechanical lung ventilator, were proposed in the ABZ conference (in the 2020 edition and 2024 edition, respectively). Both were modeled and progressively refined, had their proof obligations discharged by using Rodin and the model was validated by using the ProB model checker.

Coq and Lean are proof assistants based on the proposition-as-types paradigm of the calculus of inductive constructions. Historically, Coq has been used for verifying a wide range of systems thanks to its expressive logic, its high level of automation, and mature ecosystem, which includes plug-ins for Visual Studio, Emacs, and Vim. Recent studies include specifications and proofs for systems as different as the Ethereum Virtual Machine [6] and the semantics of quantum languages [129]. Lean, developed by Microsoft, inherits the philosophy of Coq. Although this theorem prover

is not as mature as Coq yet, it has a very active community. It is also integrated with Visual Code and most of the current efforts (e.g. [23, 24]) are focused on the mathlib library, an in-development unified library of mathematics.

Isabelle/HOL, possibly the most used higher-order logic theorem prover, has a number of advantages: (i) versatile user interface, including integration with Visual Studio and even an online coding platform, Isabelle/Cloud [138]; (ii) powerful automation by means of Sledgehammer [84], which tries to discharge automatically the current goal by invoking automatic theorem provers; (iii) human-readable proof scripts, written using the Isar (Intelligible semi-automated reasoning) language [83]; and (iv) a strong community and active development, being used nowadays for analyzing, for example, memory access violations [4] and cyber-physical systems [139].

Why using proof scores? The discussion above showed some weak points of proof scores, mainly its lack of advanced user interfaces, which has been also discussed in [118]. We also find inspiration for some issues:

- New protocols can be obtained from challenges like the ones proposed by others, including the ABZ conference.
- Software engineers do not proceed as researchers. Extending proof scores with features for refinement (already explored in CafeOBJ, as discussed above) and producing easier, human-readable proof (like Isar) would make proof scores more attractive and resolve (even if only partially) the need for experts
- Each proof starts from scratch, without libraries for easing proofs in certain contexts. Developing libraries and making them public would attract new users and ease automatization, hence being a piece of future work.

Moreover, we can argue that proof scores equal, or even surpass, several of the aspects described above:

- A very expressive syntax, especially including the support for equational axioms, such as commutativity and associativity.
- The operational semantics, as described in Section 3, are simple but very powerful.
- An efficient C++ implementation (for those Maude-based implementations, see Section 5.5).
- These three aspects together (expressive syntax, powerful and simple semantics, and efficient implementation) make proof scores very convenient. On the one hand, because we use the same language for specifying and proving, the more powerful is the language, the simpler the proof will be.
  On the other hand, only a few languages can handle efficiently equational axioms. There exist translations from Maude into Isabelle/HOL [29] and from Maude into Lean [121]. In both cases, the specifications and the proofs are more verbose in the target language than in Maude and the axioms need to be used carefully by hand to avoid infinite loops. This expressivity can be further improved by using symbolic analysis.
- The automation level reached by CiMPG+F (see Section 6.2.2) is comparable to Sledgehammer, as shown in [116]. Although future development should improve this tool, its current capabilities are state-of-the-art.
- The Maude-based implementations (see Section 5) support model checking of LTL formulas. The automatic nature of the model checker makes it very adequate for software engineers, allowing them to prove properties (e.g. safety properties, which are recurrent in industry) more easily. Moreover, CiMPG (see Section 6.2.2) provides a `proven` command [116] for asserting properties; connecting both tools is an interesting topic of future work.
- Although many proofs have been developed several years ago, recent advances in post-quantum protocols have been done recently, as discussed in Section [118]. As discussed above, more modern challenges should be achieved in order to attract new researchers, but these protocols prove that the proof score methodology is still relevant nowadays.

8 CONCLUSIONS

In this paper we have presented proof scores, a verification methodology that allows users to prove properties in systems in a flexible way. We have first defined the theoretical framework and the different tools supporting the approach, including those implementing novel features that ease the verification process. Then, we have analyzed the strong points of the approach, illustrated in many case studies that have been successfully analyzed, but also the weak points and open issues that should be addressed in order to make proof scores a widely used technique.

It is worth summarizing here how these open issues should be addressed. First, most modern systems provide an IDE and hence it is worth providing graphical support for proof scores. In this sense, it should help the user for both developing the specification and the proof. Because some of the latest CafeOBJ interpreters support interactive theorem proving and it is able to relate it to proof scores it would be worth integrating a graphical representation of the proof, so the user is aware of the remaining subgoals. Moreover, it would be nice to integrate graphical tools that help the user to find the most appropriate case splittings, possibly showing different runs of a (non-deterministic) protocol and presenting those terms that remain invariant and those that change.

Then, it is worth exploring how new Maude features can be used in theorem proving. First, it would be interesting to analyze how symbolic analysis, using unification and narrowing, can be applied in proof scores. Moreover, the latest Maude release supports meta-interpreters, which use Maude meta-level and work as a standard Maude terminal, supporting execution of terms in different processes. Using these features, we could even use potentially non-terminating computations in our proofs: they would be executed in a different process while the main proof continues by analyzing other subgoals; if the analysis does not finish after some time (chosen by the user), then the computation can be stopped and other analysis can be tried.

Regarding the properties that can be proven using proof scores, it is interesting to explore how to tackle liveness properties. In this case we should consider using a co-algebraic approach (or hidden algebra or behavioral specifications), because we need to have infinite sequences of states.

All these features should be applied to novel protocols, like quantum and blockchain protocols. Using proof scores for proving properties for these systems will possibly lead to new techniques, illustrate its power, and set the foundations for the future of proof scores.

REFERENCES

[1] Jean-Raymond Abrial. 2010. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, Cambridge, CB2 8BS, United Kingdom. DOI: http://dx.doi.org/10.1017/CBO9781139195881

[2] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. 2010. Rodin: an open toolset for modelling and reasoning in Event-B. *Int. J. Softw. Tools Technol. Transf.* 12, 6 (2010), 447–466. DOI: http://dx.doi.org/10.1007/s10009-010-0145-y

[3] Reynald Affeldt, Cyril Cohen, and Ayumu Saito. 2023. Semantics of Probabilistic Programs using s-Finite Kernels in Coq. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic (Eds.). ACM, New York, NY, USA, 3–16. DOI: http://dx.doi.org/10.1145/3573105.3575691

[4] Sharar Ahmadi, Brijesh Dongol, and Matt Griffin. 2024. Operationally proving memory access violations in Isabelle/HOL. *Sci. Comput. Program.* 234 (2024), 103088. DOI: http://dx.doi.org/10.1016/J.SCICO.2024.103088

[5] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification - The KeY Book - From Theory to Practice.* Lecture Notes in Computer Science, Vol. 10001. Springer, Berlin, Heidelberg. DOI: http://dx.doi.org/10.1007/978-3-319-49812-6

[6] Elvira Albert, Samir Genaim, Daniel Kirchner, and Enrique Martin-Martin. 2023. Formally Verified EVM Block-Optimizations. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III (Lecture Notes in Computer Science)*, Constantin Enea and Akash Lal (Eds.), Vol. 13966. Springer, Berlin, Heidelberg, 176–189. DOI: http://dx.doi.org/10.1007/978-3-031-37709-9_9

[7] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. 2021. Extracting smart contracts tested and verified in Coq. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, Catalin Hritcu and Andrei

Popescu (Eds.). ACM, New York, NY, USA, 105–121. DOI: http://dx.doi.org/10.1145/3437992.3439934

[8] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. 2020. ConCert: a smart contract certification framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, New York, NY, USA, 215–228. DOI: http://dx.doi.org/10.1145/3372885.3373829

[9] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. 2002. CASL: the Common Algebraic Specification Language. *Theor. Comput. Sci.* 286, 2 (2002), 153–196. DOI: http://dx.doi.org/10.1016/S0304-3975(01)00368-1

[10] Edward W. Ayers, Mateja Jamnik, and William T. Gowers. 2021. A Graphical User Interface Framework for Formal Verification. In *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference) (LIPIcs)*, Liron Cohen and Cezary Kaliszyk (Eds.), Vol. 193. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:16. DOI: http://dx.doi.org/10.4230/LIPICS.ITP.2021.4

[11] Kyungmin Bae and José Meseguer. 2015. Model checking linear temporal logic of rewriting formulas under localized fairness. *Sci. Comput. Program.* 99 (2015), 193–234. DOI: http://dx.doi.org/10.1016/j.scico.2014.02.006

[12] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. CVC5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Proceedings, Part I (Lecture Notes in Computer Science)*, Dana Fisman and Grigore Rosu (Eds.), Vol. 13243. Springer, Berlin, Heidelberg, 415–442. DOI: http://dx.doi.org/10.1007/978-3-030-99524-9_24

[13] Haniel Barbosa, Chantal Keller, Andrew Reynolds, Arjun Viswanathan, Cesare Tinelli, and Clark W. Barrett. 2023. An Interactive SMT Tactic in Coq using Abductive Reasoning. In *LPAR 2023: Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Manizales, Colombia, 4-9th June 2023 (EPiC Series in Computing)*, Ruzica Piskac and Andrei Voronkov (Eds.), Vol. 94. EasyChair, United Kingdom, 11–22. DOI: http://dx.doi.org/10.29007/432M

[14] Jan A. Bergstra, Jan Heering, and Paul Klint. 1990. Module Algebra. *J. ACM* 37, 2 (1990), 335–372. DOI: http://dx.doi.org/10.1145/77600.77621

[15] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer, Berlin, Heidelberg. DOI: http://dx.doi.org/10.1007/978-3-662-07964-5

[16] Garrett Birkhoff. 1935. On the Structure of Abstract Algebras. *Mathematical Proceedings of the Cambridge Philosophical Society* 31 (October 1935), 433–454. Issue 04. DOI: http://dx.doi.org/10.1017/S0305004100013463

[17] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. 2000. Specification and Proof in Membership Equational Logic. *Theoretical Computer Science* 236 (2000), 35–132.

[18] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda - A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, Berlin, Heidelberg, 73–78. DOI: http://dx.doi.org/10.1007/978-3-642-03359-9_6

[19] Rod M. Burstall and Joseph A. Goguen. 1977. Putting Theories Together to Make Specifications. In *5th International Joint Conference on Artificial Intelligence*. William Kaufmann, Burlington, Massachusetts, 1045–1058.

[20] Rod M. Burstall and Joseph A. Goguen. 1979. The Semantics of CLEAR, A Specification Language. In *Abstract Software Specifications (Lecture Notes in Computer Science)*, Vol. 86. Springer, Berlin, Heidelberg, 292–332. DOI: http://dx.doi.org/10.1007/3-540-10007-5_41

[21] Rod M. Burstall, David B. MacQueen, and Donald Sannella. 1980. HOPE: An Experimental Applicative Language. In *LISP Conference*. University of Edinburgh, Department of Computer Science, Edinburgh, Scotland, 136–143.

[22] Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O'Hearn, and Francesco Zappa Nardelli. 2022. Applying formal verification to microkernel IPC at meta. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, New York, NY, USA, 116–129. DOI: http://dx.doi.org/10.1145/3497775.3503681

[23] Mario Carneiro. 2024. Lean4Lean: Towards a formalized metatheory for the Lean theorem prover. *CoRR* abs/2403.14064, Article 2024 (2024), 17 pages. DOI: http://dx.doi.org/10.48550/ARXIV.2403.14064

[24] Mario Carneiro, Chad E. Brown, and Josef Urban. 2023. Automated Theorem Proving for Metamath. In *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland (LIPIcs)*, Adam Naumowicz and René Thiemann (Eds.), Vol. 268. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:19. DOI: http://dx.doi.org/10.4230/LIPICS.ITP.2023.9

[25] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. MIT Press, Cambridge, Massachusetts.

[26] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. 2007a. *All About Maude: A High-Performance Logical Framework*. Lecture Notes in Computer Science, Vol. 4350. Springer, Berlin Heidelberg.

[27] Manuel Clavel, Francisco Durán, Joe Hendrix, Salvador Lucas, José Meseguer, and Peter Csaba Ölveczky. 2007b. The Maude Formal Tool Environment. In *Second International Conference on Algebra and Coalgebra in Computer Science, CALCO 2007 (Lecture Notes in Computer Science)*, Till Mossakowski, Ugo Montanari, and Magne Haveraaen (Eds.), Vol. 4624. Springer, Berlin, Heidelberg, 173–178.

[28] Manuel Clavel, Miguel Palomino, and Adrián Riesco. 2006. Introducing the ITP Tool: a Tutorial. *J. Univers. Comput. Sci.* 12, 11 (2006), 1618–1650. DOI: http://dx.doi.org/10.3217/jucs-012-11-1618

[29] Mihai Codescu, Till Mossakowski, Adrián Riesco, and Christian Maeder. 2011. Integrating Maude into Hets. In *Proceedings of the 13th International Conference on Algebraic Methodology and Software Technology, AMAST 2010 (Lecture Notes in Computer Science)*, Michael Johnson and Dusko Pavlovic (Eds.), Vol. 6486. Springer, Berlin, Heidelberg, 60–75.

[30] Ionuţ Ţuţu. 2014. Parameterisation for abstract structured specifications. *Theor. Comput. Sci.* 517 (2014), 102–142. DOI: http://dx.doi.org/10.1016/J.TCS.2013.11.008

[31] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008 (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, Berlin, Heidelberg, 337–340. DOI: http://dx.doi.org/10.1007/978-3-540-78800-3_24

[32] Leonardo Mendonça de Moura, Sam Owre, Harald Rueß, John M. Rushby, Natarajan Shankar, Maria Sorea, and Ashish Tiwari. 2004. SAL 2. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings (Lecture Notes in Computer Science)*, Rajeev Alur and Doron A. Peled (Eds.), Vol. 3114. Springer, Berlin, Heidelberg, 496–500. DOI: http://dx.doi.org/10.1007/978-3-540-27813-9_45

[33] Razvan Diaconescu and Ionuţ Ţuţu. 2011. On the algebra of structured specifications. *Theor. Comput. Sci.* 412, 28 (2011), 3145–3174. DOI: http://dx.doi.org/10.1016/j.tcs.2011.04.008

[34] Razvan Diaconescu and Ionuţ Ţuţu. 2014. Foundations for structuring behavioural specifications. *J. Log. Algebraic Methods Program.* 83, 3-4 (2014), 319–338. DOI: http://dx.doi.org/10.1016/J.JLAMP.2014.03.001

[35] Razvan Diaconescu and Kokichi Futatsugi. 1998. *CafeOBJ Report - The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification.* AMAST Series in Computing, Vol. 6. World Scientific, Toh Tuck Link, Singapur. DOI: http://dx.doi.org/10.1142/3831

[36] Razvan Diaconescu and Kokichi Futatsugi. 2000. Behavioural Coherence in Object-Oriented Algebraic Specification. *J. UCS* 6, 1 (2000), 74–96.

[37] Razvan Diaconescu and Kokichi Futatsugi. 2002. Logical foundations of CafeOBJ. *Theoretical Computer Science* 285, 2 (2002), 289–318. DOI: http://dx.doi.org/10.1016/S0304-3975(01)00361-9

[38] Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. 2022. A drag-and-drop proof tactic. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, New York, NY, USA, 197–209. DOI: http://dx.doi.org/10.1145/3497775.3503692

[39] Pablo Donato, Benjamin Werner, and Kaustuv Chaudhuri. 2023. Integrating graphical proofs in Coq. (2023).

[40] Francisco Durán, Santiago Escobar, José Meseguer, and Julia Sapi na. 2024. *NuITP alpha 28 - An Inductive Theorem Prover for Maude Equational Theories.* Technical Report. Universitat Politècnica de València.

[41] Bruno Dutertre. 2014. Yices 2.2. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, Berlin, Heidelberg, 737–744. DOI: http://dx.doi.org/10.1007/978-3-319-08867-9_49

[42] Hartmut Ehrig, Werner Fey, and Horst Hansen. 1983. ACT ONE - An Algebraic Specification Language with two Levels of Semantics. In *2nd Workshop on Abstract Data Type*, Manfred Broy and Martin Wirsing (Eds.). University of Passau, Department of Computer Science, Germany, 1 – ff.

[43] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. 2002. The Maude LTL Model Checker. *Electron. Notes Theor. Comput. Sci.* 71 (2002), 162–187. DOI: http://dx.doi.org/10.1016/S1571-0661(05)82534-4

[44] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kuncak (Eds.), Vol. 10427. Springer, Berlin, Heidelberg, 126–133. DOI: http://dx.doi.org/10.1007/978-3-319-63390-9_7

[45] Michael Joseph Fay. 1979. First-order unification in an equational theory. In *Proceedings of the 4th Workshop on Automated deduction (Academic Press)*, W. H. Joyner (Ed.). University of California, Santa Cruz, California, 161–167.

[46] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, Berlin, Heidelberg, 125–128. DOI: http://dx.doi.org/10.1007/978-3-642-37036-6_8

[47] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. 2015. KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (Lecture Notes in Computer Science)*, Amy P. Felty and Aart Middeldorp (Eds.), Vol. 9195. Springer, Berlin, Heidelberg, 527–538. DOI: http://dx.doi.org/10.1007/978-3-319-21401-6_36

[48] Kokichi Futatsugi. 2015. Generate & Check Method for Verifying Transition Systems in CafeOBJ. In *Software, Services, and Systems - Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering (Lecture Notes in Computer Science)*, Rocco De Nicola and Rolf Hennicker (Eds.), Vol. 8950. Springer, Berlin, Heidelberg, 171–192. DOI: http://dx.doi.org/10.1007/978-3-319-15545-6_13

[49] Kokichi Futatsugi. 2022. Advances of proof scores in CafeOBJ. *Science of Computer Programming* 224 (2022), 102893. DOI: http://dx.doi.org/10.1016/j.scico.2022.102893

[50] Kokichi Futatsugi and Razvan Diaconescu. 1998. *CafeOBJ Report.* World Scientific, Singapore.

[51] Kokichi Futatsugi, Daniel Gâinâ, and Kazuhiro Ogata. 2012. Principles of proof scores in CafeOBJ. *Theoretical Computer Science* 464 (2012), 90–112.

[52] Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and José Meseguer. 1985. Principles of OBJ2. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, 1985*, Mary S. Van Deusen, Zvi Galil, and Brian K. Reid (Eds.). ACM Press, New York, NY, USA, 52–66. DOI: http://dx.doi.org/10.1145/318593.318610

[53] Kokichi Futatsugi and Koji Okada. 1980. Specification Writing as Construction of Hierarchically Structured Clusters of Operators. In *Information*

Processing, Proceedings of the 8th IFIP Congress 1980, Tokyo, Japan - October 6-9, 1980 and Melbourne, Australia - October 14-17, 1980, Simon H. Lavington (Ed.). North-Holland/IFIP, Amsterdam, 287–292.

[54] Daniel Gâinâ, Ionuț Țuțu, and Adrián Riesco. 2018. Specification and Verification of Invariant Properties of Transition Systems. In *25th Asia-Pacific Software Engineering Conference, APSEC 2018*. IEEE, Berlin, Heidelberg, 99–108. DOI: http://dx.doi.org/10.1109/APSEC.2018.00024

[55] Daniel Gâinâ, Kokichi Futatsugi, and Kazuhiro Ogata. 2012. Constructor-based Logics. *J. Univers. Comput. Sci.* 18, 16 (2012), 2204–2233. DOI: http://dx.doi.org/10.3217/jucs-018-16-2204

[56] Daniel Gâinâ, Min Zhang, Yuki Chiba, and Yasuhito Arimoto. 2013. Constructor-Based Inductive Theorem Prover. In *Algebra and Coalgebra in Computer Science - 5th International Conference, CALCO 2013, Warsaw, Poland, September 3-6, 2013. Proceedings (Lecture Notes in Computer Science)*, Reiko Heckel and Stefan Milius (Eds.), Vol. 8089. Springer, Berlin, Heidelberg, 328–333. DOI: http://dx.doi.org/10.1007/978-3-642-40206-7_26

[57] Ruben Gamboa, Panagiotis Manolios, Eric Whitman Smith, and Kyle Thompson. 2023. Using Counterexample Generation and Theory Exploration to Suggest Missing Hypotheses. In *Proceedings of the 18th International Workshop on the ACL2 Theorem Prover and Its Applications, 2023 (EPTCS)*, Alessandro Coglio and Sol Swords (Eds.), Vol. 393. Open Publishing Association, Wroclaw, Poland, 82–93. DOI: http://dx.doi.org/10.4204/EPTCS.393.8

[58] Stephen J. Garland and John V. Guttag. 1989. An Overview of LP, The Larch Power. In *Rewriting Techniques and Applications, 3rd International Conference, RTA-89, Chapel Hill, North Carolina, USA, April 3-5, 1989, Proceedings (Lecture Notes in Computer Science)*, Nachum Dershowitz (Ed.), Vol. 355. Springer, Berlin, Heidelberg, 137–151. DOI: http://dx.doi.org/10.1007/3-540-51081-8_105

[59] Joseph Goguen and Jose Meseguer. 1985. Completeness of many-sorted equational logic. *Houston Journal of Mathematics* 11, 3 (1985), 307–334.

[60] Joseph A. Goguen. 1968. *Categories of fuzzy sets: Applications of non-Cantorian set theory.* Ph.D. Dissertation. University of California, Berkeley.

[61] Joseph A. Goguen. 1984. Parameterized Programming. *IEEE Trans. Software Eng.* 10, 5 (1984), 528–544. DOI: http://dx.doi.org/10.1109/TSE.1984.5010277

[62] Joseph A. Goguen. 1990. Proving and Rewriting. In *2nd International Conference on Algebraic and Logic Programming (Lecture Notes in Computer Science)*, Vol. 463. Springer, Berlin, Heidelberg, 1–24. DOI: http://dx.doi.org/10.1007/3-540-53162-9_27

[63] Joseph A. Goguen. 1993. Memories of ADJ. In *Current Trends in Theoretical Computer Science*. World Scientific Series in Computer Science, Vol. 40. World Scientific, Toh Tuck Link, Singapur, 76–81. DOI: http://dx.doi.org/10.1142/9789812794499_0004

[64] Joseph A. Goguen. 1997. Tossing Algebraic Flowers Down the Great Divide. (1997). http://cseweb.ucsd.edu/~goguen/pps/tcs97.pdf

[65] Joseph A. Goguen. 2021. Theorem Proving and Algebra. *CoRR* abs/2101.02690 (2021), 1–427. https://arxiv.org/abs/2101.02690

[66] Joseph A. Goguen and Rod M. Burstall. 1992. Institutions: Abstract Model Theory for Specification and Programming. *J. ACM* 39, 1 (1992), 95–146. DOI: http://dx.doi.org/10.1145/147508.147524

[67] Joseph A. Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, José Meseguer, and Timothy C. Winkler. 1987. An Introduction to OBJ 3. In *Proceedings of the 1st International Workshop on Conditional Term Rewriting Systems (Lecture Notes in Computer Science)*, Vol. 308. Springer, Berlin, Heidelberg, 258–263. DOI: http://dx.doi.org/10.1007/3-540-19242-5_22

[68] Joseph A. Goguen and Grant Malcolm. 2000. A hidden agenda. *Theoretical Computer Science* 245, 1 (2000), 55–101. DOI: http://dx.doi.org/10.1016/S0304-3975(99)00275-3

[69] Joseph A. Goguen and José Meseguer. 1992. Order-sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. *Theoretical Computer Science* 105, 2 (Nov. 1992), 217–273. DOI: http://dx.doi.org/10.1016/0304-3975(92)90302-V

[70] Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. 1999. Introducing OBJ. In *Software Engineering with OBJ*, Joseph A. Goguen and Grant Malcolm (Eds.). Algebraic Specification in Action, Vol. 2. Springer, Berlin, Heidelberg, 3–167. DOI: http://dx.doi.org/10.1007/978-1-4757-6541-0_1

[71] Georges Gonthier. 2008. Formal Proof–The Four-Color Theorem. *Notices of the AMS* 55, 11 (2008), 1382–1393. https://www.ams.org/notices/200811/tx081101382p.pdf

[72] Daniel Găină, Masaki Nakamura, Kazuhiro Ogata, and Kokichi Futatsugi. 2020. Stability of termination and sufficient-completeness under pushouts via amalgamation. *Theor. Comput. Sci.* 848 (2020), 82–105. DOI: http://dx.doi.org/10.1016/j.tcs.2020.09.024

[73] John V. Guttag, James J. Horning, Stephen J. Garland, Kevin D. Jones, A. Modet, and Jeannette M. Wing. 1993. *Larch: Languages and Tools for Formal Specification.* Springer, Berlin, Heidelberg. DOI: http://dx.doi.org/10.1007/978-1-4612-2704-5

[74] Xuan-Linh Ha and Kazuhiro Ogata. 2017. Writing Concurrent Java Programs Based on CafeOBJ Specifications. In *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*, Jian Lv, He Jason Zhang, Mike Hinchey, and Xiao Liu (Eds.). IEEE Computer Society, Piscataway, New Jersey, USA, 618–623. DOI: http://dx.doi.org/10.1109/APSEC.2017.75

[75] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. 2017. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi* 5 (2017), e2. DOI: http://dx.doi.org/10.1017/fmp.2017.1

[76] David S. Hardin. 2022. Hardware/Software Co-Assurance using the Rust Programming Language and ACL2. In *Proceedings Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, 26th-27th May 2022 (EPTCS)*, Rob Sumners and Cuong Chau (Eds.), Vol. 359. Open Publishing Association, Wroclaw, Poland, 202–216. DOI: http://dx.doi.org/10.4204/EPTCS.359.16

[77] David S. Hardin. 2023. Verification of a Rust Implementation of Knuth's Dancing Links using ACL2. In *Proceedings of the 18th International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, TX, USA and online, November 13-14, 2023 (EPTCS)*, Alessandro Coglio and Sol Swords (Eds.), Vol. 393. Open Publishing Association, Wroclaw, Poland, 161–174. DOI: http://dx.doi.org/10.4204/EPTCS.393.13

[78] John Harrison. 1996. HOL Light: A Tutorial Introduction. In *Formal Methods in Computer-Aided Design, First International Conference, FMCAD '96, Palo Alto, California, USA, November 6-8, 1996, Proceedings (Lecture Notes in Computer Science)*, Mandayam K. Srivas and Albert John Camilleri (Eds.), Vol. 1166. Springer, Berlin, Heidelberg, 265–269. DOI: http://dx.doi.org/10.1007/BFB0031814

[79] John Harrison. 2000. Floating Point Verification in HOL Light: The Exponential Function. *Formal Methods Syst. Des.* 16, 3 (2000), 271–305. DOI: http://dx.doi.org/10.1023/A:1008712907154

[80] Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Catalin Hritcu, Kenji Maillard, and Bas Spitters. 2023. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. *ACM Trans. Program. Lang. Syst.* 45, 3 (2023), 15:1–15:61. DOI: http://dx.doi.org/10.1145/3594735

[81] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. DOI: http://dx.doi.org/10.1145/363235.363259

[82] Jieh Hsiang. 1985. Refutational Theorem Proving Using Term-Rewriting Systems. *Artif. Intell.* 25, 3 (1985), 255–300. DOI: http://dx.doi.org/10.1016/0004-3702(85)90074-8

[83] Isabelle/HOL. 2024a. Isar. https://isabelle.in.tum.de/Isar/. (2024).

[84] Isabelle/HOL. 2024b. Sledgehammer. https://isabelle.in.tum.de/website-Isabelle2009-1/sledgehammer.html. (2024).

[85] Paul B. Jackson. 2006. NuPRL. In *The Seventeen Provers of the World, Foreword by Dana S. Scott*, Freek Wiedijk (Ed.). Lecture Notes in Computer Science, Vol. 3600. Springer, Berlin, Heidelberg, 116–126. DOI: http://dx.doi.org/10.1007/11542384_16

[86] Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. 2023. Modular Verification of Safe Memory Reclamation in Concurrent Separation Logic. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 828–856. DOI: http://dx.doi.org/10.1145/3622827

[87] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. DOI: http://dx.doi.org/10.1017/S0956796818000151

[88] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. 2000. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, MA, USA. https://dl.acm.org/doi/book/10.5555/555902

[89] Donald E. Knuth. 2022. *The Art of Computer Programming, Volumes 1-4B.* Addison-Wesley, Boston, American.

[90] Joomy Korkut. 2022. A Proof Tree Builder for Sequent Calculus and Hoare Logic. In *Proceedings 11th International Workshop on Theorem Proving Components for Educational Software, ThEdu@FLoC 2022, Haifa, Israel, 11 August 2022 (EPTCS)*, Pedro Quaresma, João Marcos, and Walther Neuper (Eds.), Vol. 375. Open Publishing Association, Wroclaw, Poland, 54–62. DOI: http://dx.doi.org/10.4204/EPTCS.375.5

[91] Philipp Körner, Michael Leuschel, and Jeroen Meijer. 2018. State-of-the-Art Model Checking for B and Event-B Using ProB and LTSmin. In *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings (Lecture Notes in Computer Science)*, Carlo A. Furia and Kirsten Winter (Eds.), Vol. 11023. Springer, Berlin, Heidelberg, 275–295. DOI: http://dx.doi.org/10.1007/978-3-319-98938-9_16

[92] Laura Kovács and Andrei Voronkov. 2013. First-Order Theorem Proving and Vampire. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, Berlin, Heidelberg, 1–35. DOI: http://dx.doi.org/10.1007/978-3-642-39799-8_1

[93] Ankit Kumar, Max von Hippel, Panagiotis Manolios, and Cristina Nita-Rotaru. 2023. Verification of GossipSub in ACL2s. In *Proceedings of the 18th International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, TX, USA and online, November 13-14, 2023 (EPTCS)*, Alessandro Coglio and Sol Swords (Eds.), Vol. 393. Open Publishing Association, Wroclaw, Poland, 113–132. DOI: http://dx.doi.org/10.4204/EPTCS.393.10

[94] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (Lecture Notes in Computer Science)*, Edmund M. Clarke and Andrei Voronkov (Eds.), Vol. 6355. Springer, Berlin, Heidelberg, 348–370. DOI: http://dx.doi.org/10.1007/978-3-642-17511-4_20

[95] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. DOI: http://dx.doi.org/10.1145/1538788.1538814

[96] Zhaohui Luo and Robert Pollack. 1992. *The LEGO Proof Development System: A User's Manual.* Technical Report ECS-LFCS-92-211. University of Edinburgh.

[97] David B. MacQueen. 1984. Modules for Standard ML. In *1984 ACM Conference on LISP and Functional Programming*. ACM, New York, NY, USA, 198–207.

[98] Amel Mammar. 2024. An Event-B Model of a Mechanical Lung Ventilator. In *Rigorous State-Based Methods - 10th International Conference, ABZ 2024, Bergamo, Italy, June 25-28, 2024, Proceedings (Lecture Notes in Computer Science)*, Silvia Bonfanti, Angelo Gargantini, Michael Leuschel, Elvinia Riccobene, and Patrizia Scandurra (Eds.), Vol. 14759. Springer, Berlin, Heidelberg, 307–323. DOI: http://dx.doi.org/10.1007/978-3-031-63790-2_25

[99] Amel Mammar, Marc Frappier, and Régine Laleau. 2024. An Event-B model of an automotive adaptive exterior light system. *Int. J. Softw. Tools Technol. Transf.* 26, 3 (2024), 331–346. DOI: http://dx.doi.org/10.1007/S10009-024-00748-Z

[100] Narciso Martí-Oliet and José Meseguer. 2002. Rewriting logic as a logical and semantic framework. In *Handbook of Philosophical Logic, Second Edition, Volume 9*, D. M. Gabbay and F. Guenthner (Eds.). Kluwer Academic Publishers, Dordrecht, The Netherlands, 1–87. First published as SRI Technical Report SRI-CSL-93-05, August 1993.

[101] José Meseguer and Prasanna Thati. 2004. Symbolic Reachability Analysis Using Narrowing and Its Application to Verification of Cryptographic Protocols. In *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27 – April 4, 2004 (Electronic Notes in Theoretical Computer Science)*, Narciso Martí-Oliet (Ed.). Elsevier, Essex, UK, 147–174.

[102] Glen Mével and Jacques-Henri Jourdan. 2021. Formal verification of a concurrent bounded queue in a weak memory model. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. DOI: http://dx.doi.org/10.1145/3473571

[103] J. Strother Moore, Thomas W. Lynch, and Matt Kaufmann. 1998. A Mechanically Checked Proof of the AMD5$_K$86^{TM} Floating Point Division Program. *IEEE Trans. Computers* 47, 9 (1998), 913–926. DOI: http://dx.doi.org/10.1109/12.713311

[104] Till Mossakowski, Christian Maeder, and Klaus Lüttich. 2007. The Heterogeneous Tool Set (Hets). In *Proceedings of 4th International Verification Workshop in connection with CADE-21 (CEUR Workshop Proceedings)*, Bernhard Beckert (Ed.), Vol. 259. CEUR-WS.org, Aachen, 1–17. http://ceur-ws.org/Vol-259/paper11.pdf

[105] Masaki Nakamura, Kazutoshi Sakakibara, and Kazuhiro Ogata. 2020. Specification description and verification of multitask hybrid systems in the OTS/CafeOBJ method. *CoRR* abs/2010.15280 (2020), 1–25. https://arxiv.org/abs/2010.15280

[106] Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. 2023. An Extensible User Interface for Lean 4. In *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Bialystok, Poland (LIPIcs)*, Adam Naumowicz and René Thiemann (Eds.), Vol. 268. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 24:1–24:20. DOI: http://dx.doi.org/10.4230/LIPICS.ITP.2023.24

[107] Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. 2023. Formalising Decentralised Exchanges in Coq. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic (Eds.). ACM, New York, NY, USA, 290–302. DOI: http://dx.doi.org/10.1145/3573105.3575685

[108] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer, Berlin, Heidelberg. DOI: http://dx.doi.org/10.1007/3-540-45949-9

[109] Kazuhiro Ogata and Kokichi Futatsugi. 2003. Proof Scores in the OTS/CafeOBJ Method. In *6th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (Lecture Notes in Computer Science)*, Vol. 2884. Springer, Berlin, Heidelberg, 170–184. DOI: http://dx.doi.org/10.1007/978-3-540-39958-2_12

[110] Kazuhiro Ogata and Kokichi Futatsugi. 2006. Some Tips on Writing Proof Scores in the OTS/CafeOBJ Method. In *Essays Dedicated to Joseph A. Goguen (Lecture Notes in Computer Science)*, Vol. 4060. Springer, Berlin, Heidelberg, 596–615. DOI: http://dx.doi.org/10.1007/11780274_31

[111] Kazuhiro Ogata and Kokichi Futatsugi. 2008. Simulation-based Verification for Invariant Properties in the OTS/CafeOBJ Method. *Electron. Notes Theor. Comput. Sci.* 201 (2008), 127–154. DOI: http://dx.doi.org/10.1016/j.entcs.2008.02.018

[112] Kazuhiro Ogata, Daigo Yamagishi, Takahiro Seino, and Kokichi Futatsugi. 2004. Modeling and Verification of Hybrid Systems Based on Equations. In *Design Methods and Applications for Distributed Embedded Systems, IFIP 18th World Computer Congress, TC10 Working Conference on Distributed and Parallel Embedded Systems, DIPES 2004 (IFIP)*, Bernd Kleinjohann, Guang R. Gao, Hermann Kopetz, Lisa Kleinjohann, and Achim Rettberg (Eds.), Vol. 150. Kluwer/Springer, Berlin, Heidelberg, 43–52. DOI: http://dx.doi.org/10.1007/1-4020-8149-9_5

[113] Sam Owre, John M. Rushby, and Natarajan Shankar. 1992. PVS: A Prototype Verification System. In *Proceedings of the 11th International Conference on Automated on Automated Deduction, CADE-11 (Lecture Notes in Computer Science)*, Deepak Kapur (Ed.), Vol. 607. Springer, Berlin, Heidelberg, 748–752. DOI: http://dx.doi.org/10.1007/3-540-55602-8_217

[114] Yan Peng and Mark R. Greenstreet. 2015. Extending ACL2 with SMT Solvers. In *Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications (EPTCS)*, Matt Kaufmann and David L. Rager (Eds.), Vol. 192. Open Publishing Association, Waterloo, Australia, 61–77. DOI: http://dx.doi.org/10.4204/EPTCS.192.6

[115] Adrián Riesco and Kazuhiro Ogata. 2018. Prove it! Inferring Formal Proof Scripts from CafeOBJ Proof Scores. *ACM Transactions on Software Engineering and Methodology* 27, 2 (2018), 6:1 – 6:32.

[116] Adrián Riesco and Kazuhiro Ogata. 2022. An integrated tool set for verifying CafeOBJ specifications. *J. Syst. Softw.* 189 (2022), 111302. DOI: http://dx.doi.org/10.1016/J.JSS.2022.111302

[117] Adrián Riesco, Kazuhiro Ogata, and Kokichi Futatsugi. 2016. A Maude environment for CafeOBJ. *Formal Aspects of Computing* 29, 2 (2016), 309–334. DOI: http://dx.doi.org/10.1007/s00165-016-0398-7

[118] A. Riesco, K. Ogata, M. Nakamura, D. Găină, D. Dinh Tran, and K. Futatsugi. 2024. Open issues in theorem proving - Supplemental materials for "Proof Scores: A Survey". (2024).

[119] Camilo Rocha and José Meseguer. 2014. Mechanical Analysis of Reliable Communication in the Alternating Bit Protocol Using the Maude Invariant Analyzer Tool. In *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi (Lecture Notes in Computer Science)*, Shusaku Iida, José Meseguer, and Kazuhiro Ogata (Eds.), Vol. 8373. Springer, Berlin, Heidelberg, 603–629. DOI: http://dx.doi.org/10.1007/978-3-642-54624-2_30

[120] Camilo Rocha, José Meseguer, and César A. Muñoz. 2017. Rewriting modulo SMT and open system analysis. *J. Log. Algebraic Methods Program.* 86, 1 (2017), 269–297. DOI: http://dx.doi.org/10.1016/j.jlamp.2016.10.001

[121] Rubén Rubio and Adrián Riesco. 2025. Maude2Lean: Theorem proving for Maude specifications using Lean. *J. Log. Algebraic Methods Program.* 142 (2025), 101005. DOI: http://dx.doi.org/10.1016/J.JLAMP.2024.101005

[122] David M. Russinoff. 2022. *Formal Verification of Floating-Point Hardware Design - A Mathematical Approach, Second Edition*. Springer, Berlin, Heidelberg. DOI: http://dx.doi.org/10.1007/978-3-030-87181-9

[123] Mark Saaltink. 1997. The Z/EVES System. In *ZUM '97: The Z Formal Specification Notation, 10th International Conference of Z Users, Reading, UK, April 3-4, 1997, Proceedings (Lecture Notes in Computer Science)*, Jonathan P. Bowen, Michael G. Hinchey, and David Till (Eds.), Vol. 1212. Springer, Berlin, Heidelberg, 72–85. DOI: http://dx.doi.org/10.1007/BFb0027284

[124] Donald Sannella and Andrzej Tarlecki. 1988. Specifications in an Arbitrary Institution. *Inf. Comput.* 76, 2/3 (1988), 165–210. DOI: http://dx.doi.org/10.1016/0890-5401(88)90008-9

[125] Donald Sannella and Andrzej Tarlecki. 2012. *Foundations of Algebraic Specification and Formal Software Development*. Springer, Berlin, Heidelberg. DOI: http://dx.doi.org/10.1007/978-3-642-17336-3

[126] Toshimi Sawada, Kokichi Futatsugi, and Norbert Preining. 2018. *CafeOBJ Reference Manual (version 1.5.7)*. JAIST.

[127] Stephan Schulz, Simon Cruanes, and Petar Vukmirovic. 2019. Faster, Higher, Stronger: E 2.3. In *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings (Lecture Notes in Computer Science)*, Pascal Fontaine (Ed.), Vol. 11716. Springer, Berlin, Heidelberg, 495–507. DOI: http://dx.doi.org/10.1007/978-3-030-29436-6_29

[128] Jittisak Senachak, Takahiro Seino, Kazuhiro Ogata, and Kokichi Futatsugi. 2005. Provably Correct Translation from CafeOBJ into Java. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'2005), Taipei, Taiwan, Republic of China, July 14-16, 2005*, William C. Chu, Natalia Juristo Juzgado, and W. Eric Wong (Eds.). KSI Research Inc. and Knowledge Systems Institute Graduate School, Pittsburgh, USA, 614–619.

[129] Wenjun Shi, Qinxiang Cao, and Yuxin Deng. 2024. Formalizing the Semantics of a Classical-Quantum Imperative Language in Coq. *J. Circuits Syst. Comput.* 33, 6 (2024), 2450112:1–2450112:25. DOI: http://dx.doi.org/10.1142/S0218126624501123

[130] Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings (Lecture Notes in Computer Science)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.), Vol. 5170. Springer, Berlin, Heidelberg, 28–32. DOI: http://dx.doi.org/10.1007/978-3-540-71067-7_6

[131] Terese. 2003. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science, Vol. 55. Cambridge University Press, Cambridge, CB2 8BS, United Kingdom.

[132] Duong Dinh Tran, Dang Duy Bui, and Kazuhiro Ogata. 2021. Simulation-Based Invariant Verification Technique for the OTS/CafeOBJ Method. *IEEE Access* 9 (2021), 93847–93870.

[133] Duong Dinh Tran and Kazuhiro Ogata. 2022. Formal verification of TLS 1.2 by automatically generating proof scores. *Computers & Security* 123 (2022), 102909. DOI: http://dx.doi.org/10.1016/j.cose.2022.102909

[134] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2022. Mechanized verification of a fine-grained concurrent queue from meta's folly library. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, New York, NY, USA, 100–115. DOI: http://dx.doi.org/10.1145/3497775.3503689

[135] Eric G. Wagner. 2002. Algebraic Specifications: some old history and new thoughts. *Nord. J. Comput.* 9, 4 (2002), 373–404.

[136] Michael Winter. 2007. *Goguen Categories – A Categorical Approach to L-fuzzy Relations*. Trends in Logic, Vol. 25. Springer, Berlin, Heidelberg. DOI: http://dx.doi.org/10.1007/978-1-4020-6164-6

[137] J. C. P. Woodcock and Jim Davies. 1996. *Using Z - specification, refinement, and proof*. Prentice Hall, New Jersey, USA.

[138] Hao Xu and Yongwang Zhao. 2023. Isabelle/Cloud: Delivering Isabelle/HOL as a Cloud IDE for Theorem Proving. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware, Internetware 2023, Hangzhou, China, August 4-6, 2023*, Hong Mei, Jian Lv, Zhi Jin, Xuandong Li, Xiaohu Yang, and Xin Xia (Eds.). ACM, New York, NY, USA, 313–322. DOI: http://dx.doi.org/10.1145/3609437.3609460

[139] Jonathan Julián Huerta y Munive, Simon Foster, Mario Gleirscher, Georg Struth, Christian Pardillo Laursen, and Thomas Hickman. 2024. IsaVODEs: Interactive Verification of Cyber-Physical Systems at Scale. *J. Autom. Reason.* 68, 4 (2024), 21. DOI: http://dx.doi.org/10.1007/S10817-024-09709-2

[140] Yizhou Zhang and Nada Amin. 2022. Reasoning about "reasoning about reasoning": semantics and contextual equivalence for probabilistic programs with nested queries and recursion. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. DOI: http://dx.doi.org/10.1145/3498677

Month YearMonth Year

Received Month Year