The Journal of Systems & Software 189 (2022) 111302

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

An integrated tool set for verifying CafeOBJ specifications[☆],[☆☆]

Adrián Riesco [a,*], Kazuhiro Ogata [b,c]

[a] Software systems and computation, Fac. Informática, Universidad Complutense de Madrid, Spain
[b] School of Information Science, JAIST, Japan
[c] Research Center for Theoretical Computer Science, JAIST, Japan

ARTICLE INFO

Article history:
Received 21 July 2021
Received in revised form 3 February 2022
Accepted 12 March 2022
Available online 18 March 2022

Keywords:
CafeOBJ
Theorem proving
Proof scores
Script inference
Script generation

ABSTRACT

CafeOBJ is a language for specifying and verifying a wide variety of software and/or hardware systems. Traditionally, verification has been carried out via proof scores, which consist of reducing goal-related terms in user-defined modules. Although proof scores are semi-formal (the specifier is partially responsible for soundness), their flexibility makes them a useful approach to verification.

For the last years, we have developed different formal tools around the CafeInMaude interpreter, a CafeOBJ interpreter implemented in Maude. Besides supporting proof scores, we implemented a theorem prover, a proof script generator from proof scores, and the first stages of a proof script generator and fixer-upper. In this paper, we present (i) an improved and detailed version of our proof script generator and fixer-upper and (ii) a reimplementation of the CafeInMaude interpreter, which supports, among others, parallel execution, an improved tool integration, and an interactive user interface. The benchmarks used to evaluate the tools confirm the usefulness of the approach.

1. Introduction

CafeOBJ (Futatsugi and Diaconescu, 1998; Sawada et al., 2015) is a language for writing formal specifications for a wide variety of software and/or hardware systems, and verifying properties of them. CafeOBJ implements equational logic by rewriting and can be used as a powerful platform for proving properties of systems. CafeOBJ provides several features to ease the specification of systems, including a flexible mix-fix syntax, a powerful and clear typing system with ordered sorts, parameterized modules and views for instantiating the parameters, module expressions, operators for defining terms, equations for defining the (possibly conditional) equalities between terms, and (possibly conditional) transitions for specifying how a system evolves, among others. Equations and transitions can be understood as oriented from left to right, so they can be executed. In this paper we are interested in the equational part of CafeOBJ, which is used to specify the behavior of systems and *observe* it by means of equations. These equations stand for simplification rules that return the value of the relevant elements of the system in each step.

Regarding verification, specifiers can write proof scores (Futatsugi et al., 2012) also in CafeOBJ and perform proofs by executing these proof scores. Proof scores consist of reduction commands, standing for the (sub)goals we want to prove. These reduction commands are just terms standing for a property of the system in a particular state. We consider that the property holds if it is reduced to `true` by using equations as simplification rules. These reduction commands are introduced into user-defined modules (called open-close environments, because they "open" a previous module and add new information, which is lost once all commands have been executed and hence the environment becomes "closed"), including basically the constants required to instantiate the properties and the case splittings required for that (sub)goal. Proof scores are a very powerful verification technique because they allow specifiers to use all the features available in the language itself. However, they have the drawback of being semi-formal: the user is responsible for analyzing all possible cases and using sound case splittings and premises.

This semi-formal nature of proof scores led us to start the CafeInMaude project.[1] We first implemented the CafeInMaude interpreter (Riesco et al., 2016), a CafeOBJ interpreter implemented in Maude. This interpreter supported the same modules (including proof scores) as the standard CafeOBJ interpreter and showed a better performance, so in practice it allowed us to prove properties for a wider range of systems than the standard CafeOBJ interpreter, implemented in LISP. This new interpreter

[☆] Editor: Earl Barr.

[☆☆] Research partially supported by JSPS KAKENHI Grant Number 26240008, the MINECO Spanish project *ProCode-UCM* (PID2019-108528RB-C22), and by Comunidad de Madrid as part of the program *BLOQUES-CM* (S2018/TCS-4339) co-funded by EIE Funds of the European Union.

[*] Corresponding author.

E-mail addresses: ariesco@fdi.ucm.es (A. Riesco), ogata@jaist.ac.jp (K. Ogata).

[1] https://github.com/ariesco/CafeInMaude.

https://doi.org/10.1016/j.jss.2022.111302

did not add formality to proof scores, but was used as the starting point for the CafeInMaude Proof Assistant (CiMPA) and the CafeInMaude Proof Generator (CiMPG) (Riesco and Ogata, 2018). CiMPA provides a formal approach to theorem proving, but rules out the flexibility given by proof scores. On the other hand, CiMPG takes a proof score and generates (when possible) a CiMPA proof script. In this way, users could check whether their proof scores were correct while enjoying the flexibility of proof scores. As a weak point, CiMPG was only able to find the complete CiMPA script when the proof score was correct; otherwise, CiMPG just pointed out where the error was, without helping the user any further. The CafeInMaude Proof Generator & Fixer-Upper (CiMPG+F) (Riesco and Ogata, 2020) was proposed to solve this problem. CiMPG+F chooses possible case splittings guided by the current subgoal and follows a depth-first strategy to try to automatically discharge it. It is also worth noting that all these tools were static, in the sense that proof scores and CiMPA, CiMPG, and CiMPG+F commands must be given inside open-close environments and no interaction from the user was possible outside them. This limitation in the interaction affected also the functionality for loading modules, which could not be done from Maude but from Java.

Maude (Clavel et al., 2007) is a specification language based on rewriting logic. CafeOBJ and Maude are sister languages, descendants of OBJ. Although Maude does not support proof scores, and hence it cannot be used with the proof methodology in this paper, they share most of the syntax and the execution mechanism. For this reason, it makes sense to choose Maude as implementation language for our CafeOBJ interpreter. Moreover, taking advantage of the fact that rewriting logic is reflective (Clavel and Meseguer, 2002), Maude provides a powerful meta-level where Maude modules can be used as data. Using this feature, the CafeInMaude tools analyze the modules, reason about the current subgoals, and relate modules in proof scores with nodes in the proof tree. Full Maude (Clavel et al., 2007, Part II) is an extension of Maude written in Maude itself that provides a richer syntax, including new module types and commands not available in standard Maude (called Core Maude). Furthermore, it extends the Loop Mode module, which provides an I/O loop for interacting with the user. Full Maude uses the meta-level extensively: it has an explicit module database and provides several features for parsing modules, so it has been historically used as basis for many other applications requiring module manipulation. In particular, the CafeInMaude applications described above extended Full Maude.

The release of Maude 3 in December 2019 (Clavel et al., 2020) included several interesting features, including an external rewriting command for interacting with external objects, such as files (including the standard input and output) and meta-interpreters (complete Maude instances that interact via message-passing as a standard Maude terminal). Meta-interpreters were improved with the release of Maude 3.1 in October 2020, including support for execution in different processes, making possible the design of concurrent computation in a single computer. The new version of the CafeInMaude tool set takes advantage of external objects in several ways. Regarding user interaction, we use them for I/O with the user, both via text files and the standard I/O stream. Regarding the tool architecture, a meta-interpreter is used as database, where user modules are introduced. Moreover, a coordinator-worker architecture can be used to parallelize the computations required by CiMPG and CiMPG+F.

Summarizing, in this paper we present the following contributions:

1 The parsing process, the module database, and the input/output functionalities have been completely reimplemented to work with Maude 3.1. In particular:

   (a) The current version of CafeInMaude implements its own parsing functions and stores the resulting modules in its own database (implemented as a meta-interpreter).
   (b) The current version uses external objects to process text files, preprocessing them in Maude itself. Likewise, it can generate a text file with the current proof, so it is possible to continue with the proof later, or just store it when finished.
   (c) The current version of the tool is interactive. This feature makes dealing with subgoals with several case splittings easier, while trying different proof strategies is simpler and faster.
   (d) We use meta-interpreters for parallelizing CiMPG and CiMPG+F computations. Although we provide a single strategy for parallelizing the subgoals, the benchmarks suggest interesting lines of future work for this feature.

2 We fully integrate CiMPG and CiMPG+F. Although previous versions of CiMPG used CiMPG+F when the proof score was incomplete, the inference of the initial goal was different for both tools and different commands were required to start them. We found this integration very useful in large proofs, where some recursive cases can be generated automatically but others require the user to provide extra information.

3 We implemented new commands:

   (a) A "help" command that displays the case splittings considered by CiMPG+F for a given subgoal. As we discussed in Riesco and Ogata (2020), for really large systems/proofs it is not realistic to generate the proof in a completely automatic way and some assistance is required from the user. Making the tools interactive allows us to show the user some information that was being used internally before, guiding him/her towards the most convenient case splittings. In turn, the user employs his/her expert knowledge to choose the most appropriate among all the possibilities.
   (b) A "proven" command, which allows users to use previously proven lemmas in the current theorem. It can be used for two different purposes: (i) to simplify the current proof, which is especially useful when generating whole proofs but can be also helpful when proving interactively, as independent proofs are easier to handle; and (ii) to analyze whether a lemma is really required: in some cases the user might think that introducing a new lemma is required to discharge the current goal, but proving it beforehand is a waste of time if it does not really help with the proof.
   (c) Commands for dealing with the new functionalities: setting the path for storing text files, saving files, specifying the numbers of cores available for concurrent computations, and launching both CiMPG and CiMPG without enclosing it in an open-close environment.

4 We give a more detailed description of an improved version of the algorithm used by CiMPG+F, including heuristics that were not present in Riesco and Ogata (2020). We also describe how to discharge goals.

5 We present new benchmarks: in addition to analyzing new protocols, all previous benchmarks have been re-executed using the new parallelization features. The obtained results suggest that the combination of tools is required for large specifications and point out possible improvements.

The rest of the paper is organized as follows: Section 2 briefly presents CafeOBJ, proof scores, CiMPA, and CiMPG using a running example. Section 3 details the architecture of the system, emphasizing how CiMPG+F works, while Section 4 illustrates how to use the different tools. Section 5 presents the benchmarks used to analyze the tools. Finally, Section 6 discusses the related work and Section 7 concludes and presents some lines of ongoing work. The tool and several case studies are available at Riesco (2022a). The running example is available at Riesco (2022b).

2. Preliminaries

In this section we introduce CafeOBJ and the different tools implemented in CafeInMaude by means of a running example. Then, we illustrate how the different CafeInMaude tools can be used to verify it. All the details regarding the running example, including complete proofs and CiMPA, CiMPG, and CiMPG+F commands are available at Riesco (2022b).[2]

2.1. CafeOBJ

CafeOBJ (Futatsugi and Diaconescu, 1998; Sawada et al., 2015) implements order-sorted equational logic by rewriting. It supports the definition of datatypes by means of sorts and subsort relations, constructors for these datatypes, and functions whose behavior are specified by means of equations. We use the Needham–Schroeder–Löwe Public-Key protocol (NSLPK) (Lowe, 1995), a variant of the Needham–Schroeder Public-Key protocol (NSPK) (Needham and Schroeder, 1978), as running example. Because it is a public-key protocol, we assume all actors (called *principals*) have a pair of public (used to encrypt) and private (used to decrypt) keys; we will use syntax $\mathtt{enc}_p(m)$ to indicate that a message m is encrypted using the public key of principal p. Then, the protocol is used for verifying the identities of two principals, p and q, using the following messages:

1. p uses the public key of q to send it a nonce (a random number) and its own identifier, that is, $p \xrightarrow{\mathtt{enc}_q(n_p,p)} q$, with n_p the nonce generated by p.
2. q authenticates itself by using its private key to decrypt the message and uses p's public key to generate a new message that contains the nonce sent by p, a new nonce generated by q (n_q), and q identifier. That is, we have $q \xrightarrow{\mathtt{enc}_p(n_p,n_q,q)} p$.
3. Similarly, p replies decrypting n_q and sending it back encrypted with q's public key, that is, $p \xrightarrow{\mathtt{enc}_q(n_q)} q$.

We will assume that there is an intruder that can eavesdrop and fake messages, but cannot decrypt messages encoded with other principals' public key. We want to prove the nonce secrecy property, that is, the intruder cannot access the nonces generated by the rest of principals. We will first specify the system in CafeOBJ and then state the required properties to verify it.

Principals are specified in the PRIN module, where the mod* syntax indicates that it has *loose semantics*, that is, many different implementations (models) for the sorts and operators in the specification satisfy the given axioms (Astesiano et al., 1999). Intuitively, modules with loose semantics pose the minimum requirements a specification must fulfill, although these specifications may also include more information (e.g. sorts or operators). In this case, we require the existence of a sort Prin for principals and a distinguished constructor (note the constr attribute in the operator definition,with keyword op, empty arity, and result sort Prin) intr, standing for the intruder. This means that, independently of how principals are defined (which is irrelevant for us and will not play any role in the verification process), acceptable implementations (models) are required to include this sort and this constructor.

```
mod* PRIN {
 [Prin]
 op intr : -> Prin {constr}
}
```

Similarly, the module RAND defines random numbers, which are built by using the constructors seed and next. We use equations to state equality between constructed terms by using the _=_ operator, where underscores are placeholders. This operator is defined for all CafeOBJ sorts by default with a single equation, stating that syntactically equal terms are equal, while the specifier is in charge of defining the rest of the cases. In this case note that we first indicate that terms built with different constructors must be different. Then, we indicate that an element built with next cannot be equal to its argument and, finally, indicate that two terms built with next are equal if their arguments are equal. Note that these equations use variables previously defined by using the keyword vars and the corresponding sort:

```
mod* RAND {
 [Rand]
 op seed : -> Rand {constr}
 op next : Rand -> Rand {constr}
 vars R1 R2 : Rand
 eq (seed = next(R1)) = false .
 eq (R1 = next(R1)) = false .
 eq (next(R1) = next(R2)) = (R1 = R2) .
}
```

Next, the module NONCE is used to define nonces. Note that it is defined with syntax mod!, which indicates that it has tight semantics, that is, it has a single model (the initial model), which is unique up to isomorphism (Astesiano et al., 1999). Intuitively, we are indicating that the details are closed and the implementation must be a model of the specification. This module imports the modules above by using the keyword pr, which stands for *protecting* and indicates that the imported modules cannot introduce new constructors nor identify terms that were previously different. In this case, we import a summation module by using the _+_ operator, which creates a new module including the information in its summands. Then, it defines the Nonce sort, with constructor n, which receives the principal that created it, the principal that should receive it, and a random number; and the projection functions p1, p2, and r. These functions are defined by means of equations, as well as the equality for nonces. This equality states that two nonces are equal if all their components are equal:

```
mod! NONCE {
  pr(PRIN + RAND)
  [Nonce]
  op n : Prin Prin Rand -> Nonce {constr}
  ops p1 p2 : Nonce -> Prin
  op r  : Nonce -> Rand
  vars P1 P2 P12 P22 : Prin
  vars R1 R2 : Rand
  eq p1(n(P1,P2,R1)) = P1 .
  eq p2(n(P1,P2,R1)) = P2 .
  eq r(n(P1,P2,R1))  = R1 .
 eq (n(P1,P2,R1) = n(P12,P22,R2)) = (P1 = P12 and P2 = P22 and R1 = R2) .
}
```

[2] Note that two properties of NSLPK (nonce secrecy property and authentication property) have been verified in CiMPA and CiMPG in Mon et al. (2021). In the current paper we use CiMPG+F instead of the previous versions of the tool, focusing on the nonce secrecy property. The proof in Mon et al. (2021) is discussed in the benchmarks.

The module CIPHER defines the sort Cipher for the different messages, with subsorts (note the < keyword) Cipher1, Cipher2, and Cipher3 for each particular message (built with the enc1, enc2, and enc3 constructors, respectively). It also defines projection functions for the different arguments of these messages (the corresponding equations, defined as shown in the module above, are avoided for simplicity), generalizing with the Cipher sort when possible:

```
mod! CIPHER {
 pr(NONCE)
 [Cipher1 Cipher2 Cipher3 < Cipher]
 op enc1 : Prin Nonce Prin -> Cipher1 {constr}
 op enc2 : Prin Nonce Nonce Prin -> Cipher2 {constr}
 op enc3 : Prin Nonce -> Cipher3 {constr}
 op k   : Cipher -> Prin
 op n1  : Cipher -> Nonce
 op n2  : Cipher2 -> Nonce
 op p1  : Cipher1 -> Prin
 op p2  : Cipher2 -> Prin
 ...
}
```

The parameterized module BAG defines a set of elements (defined with the constructors empty and empty syntax for union of sets; note that this operator is defined as associative and commutative by means of the assoc and comm attributes, respectively). It assumes the existence of the Elt sort in the parameter M (which is written as Elt.M), fulfilling the requirements from the predefined TRIV theory (which just defines this sort). It also defines the function _\in_ for checking whether a set contains an element, defined by means of equations distinguishing whether we have the empty set, a singleton, or a larger set:

```
mod! BAG (M :: TRIV) {
 [Elt.M < Bag]
 op empty : -> Bag {constr}
 op __ : Bag Bag -> Bag {constr assoc comm}
 op _\in_ : Elt.M Bag -> Bool

 vars X Y : Elt.M
 var S : Bag.

 eq X \in empty = false .
 eq X \in Y = (X = Y) .
 eq X \in (Y S) = (X = Y) or (X \in S) .
}
```

We define the views TRIV2NONCE and TRIV2CIPHER for instantiating the set of nonces (NonceBag) and the set of messages (Network). Note that views are just maps that ensure that modules satisfy the requirements posed by theories:

```
view TRIV2NONCE from TRIV to NONCE {sort Elt -> Nonce}
view TRIV2CIPHER from TRIV to CIPHER {sort Elt -> Cipher}

mod! NONCE-BAG {pr(BAG(M <= TRIV2NONCE) * {sort Bag -> NonceBag})}
mod! NETWORK {pr(BAG(M <= TRIV2CIPHER) * {sort Bag -> Network})}
```

Finally, we define in the NSLPK module the protocol above. It is defined as an observational transition system (OTS), so we will define the sort for the System (Sys), which will be built by means of the possible transitions. Then, we will use observer functions for collecting the value of the elements of interest. We first define the module, which imports the ones above, and the main sort, Sys:

```
mod* NSLPK {
 pr(NONCE-BAG + NETWORK)
 [Sys]
```

The possible transitions for Sys are init for the initial state, send1, send2, and send3 for sending the messages exchanged by non-intruder principals and fake1, fake2, and fake3 for the same messages when exchanged by the intruder. In this way, Sys is inductively defined:

```
 op init : -> Sys {constr}

 op send1 : Sys Prin Prin -> Sys {constr}
 op send2 : Sys Prin Prin Nonce -> Sys {constr}
 op send3 : Sys Prin Prin Nonce Nonce -> Sys {constr}
 op fake1 : Sys Prin Prin Nonce -> Sys {constr}
 op fake2 : Sys Prin Prin Nonce Nonce -> Sys {constr}
 op fake3 : Sys Prin Nonce -> Sys {constr}
```

The state of the system cannot be directly accessed. Instead, we can obtain information about it by *observing* the value of its components via functions. These observer functions are rand for obtaining the random number that has never been used and will be used next, nw for collecting the set of messages in the system, and nonces for gathering the nonces in the system that the intruder can see:

```
 op rand   : Sys -> Rand
 op nw     : Sys -> Network
 op nonces : Sys -> NonceBag
```

For the initial state, the random number that will be used is the seed, while the bags are empty:

```
 eq rand(init)   = seed .
 eq nw(init)     = empty .
 eq nonces(init) = empty .
```

Next, we analyze the send1 case (we assume variables of the appropriate sorts have been defined beforehand). Because sending the message enc1 requires generating a nonce, the random number to be used is the next to the one in the rest of the system. Likewise, we find the enc1 message in the network and, if the message is received by the intruder, then it can be decrypted with its own private key and added to the nonces set:

```
 eq rand(send1(S,P1,P2))   = next(rand(S)) .
 eq nw(send1(S,P1,P2))    = (enc1(P2,n(P1,P2,rand(S)),P1) nw(S)) .
 eq nonces(send1(S,P1,P2))
  = if P2 = intr then (n(P1,P2,rand(S)) nonces(S)) else nonces(S) fi .
```

The transition send2 requires an *effective condition* c-send2 to be defined. Effective conditions are used to check whether the transition can be applied; while the previous transitions can be always applied, this one requires an enc1 message to be in the network addressed to the appropriate principal:

```
 op c-send2 : Sys Prin Prin Nonce -> Bool
 eq c-send2(S,P1,P2,N1) = enc1(P1,N1,P2) \in nw(S) .
```

When this condition holds, we need to generate the next random number, we find the corresponding enc2 message in the network, and, if we are dealing with the intruder, we add the nonce to the nonces set. Note the use of the keyword ceq for conditional equations:

```
 ceq rand(send2(S,P1,P2,N1)) = next(rand(S))
  if c-send2(S,P1,P2,N1) .
 ceq nw(send2(S,P1,P2,N1)) = (enc2(P2,N1,n(P1,P2,rand(S)),P1) nw(S))
  if c-send2(S,P1,P2,N1) .
 ceq nonces(send2(S,P1,P2,N1))
  = if P2 = intr then (N1 n(P1,P2,rand(S)) nonces(S)) else nonces(S) fi
  if c-send2(S,P1,P2,N1) .
```

Otherwise, the transition is skipped:

```
ceq send2(S,P1,P2,N1) = S
 if not c-send2(S,P1,P2,N1) .
```

The observations for the rest of the transitions are defined in a similar way; we refer to the repository for more information. Finally, we define three inductive properties the system should verify: `inv1` is the nonce secrecy property, which states that the intruder only sees nonces generated or received by it. `inv2` indicates that, if a given `enc1` message is in the network, then the nonce was generated/received by the intruder or it can be found in the `nonces` set. Similarly, `inv3` states the same property for `enc2` messages:

```
 op inv1 : Sys Nonce -> Bool
 eq inv1(S,N1) = ((N1 \in nonces(S)) implies
                 (p1(N1) = intr or p2(N1) = intr)) .
 op inv2 : Sys Nonce Prin -> Bool
 eq inv2(S,N1,Q1)
   = ((enc1(Q1,N1,intr) \in nw(S))
      implies (p1(N1) = intr or p2(N1) = intr or N1 \in nonces(S))) .
 op inv3 : Sys Nonce Nonce Prin -> Bool
 eq inv3(S,N1,N2,Q1)
   = ((enc2(Q1,N1,N2,intr) \in nw(S))
      implies (p1(N2) = intr or p2(N2) = intr or N2 \in nonces(S))) .
}
```

In the next sections we will present different ways to verify these properties. We will focus on the main property, `inv1`, for the `send2` transition, which allows us to introduce different features of our tools.

2.2. Verifying with proof scores

We use proof scores to verify the properties as follows. First, we create an open-close environment extending the NSLPK module. We use an `:id(nslpk)` label to indicate that it is part of the `nslpk` proof; this label, used for the verification of the three properties above, will be used by CiMPG later. Note in the open-close environment below how we define fresh constants for using them in the equations and in the reduction command. In turn, the equations stand for case-splittings; in this case we have three equations providing extra information about the constants above: the message `enc1(p1,m,p2)` is in the network, `p2` is the intruder, and `n1` has the form `n(p1,p2,rand(s))`. Finally, the reduction command (which is reduced to `true`) uses the induction hypothesis as premise for discharging a subgoal of the recursive casethat we are dealing with, `send2`:

```
open NSLPK .
 :id(nslpk)
 -- fresh constants
 op s : -> Sys .
 op n1 : -> Nonce .
 ops p1 p2 : -> Prin .
 op m : -> Nonce .
 -- assumptions
 eq enc1(p1,m,p2) \in nw(s) = true .
 eq intr = p2 .
 eq n1 = n(p1,p2,rand(s)) .

 red inv1(s,n1) implies inv1(send2(s,p1,p2,m),n1) .
close
```

We say that proof scores are semi-formal because CafeOBJ does not check whether: (i) the equations standing for case splittings are valid (see Futatsugi et al. (2012), Găină et al. (2012) for the technical details); (ii) the premises used in the reduction command correspond to the induction hypotheses; and (iii) all the open-close environments for the corresponding case are present. In fact, we need three more open-close environments for discharging this subgoal; if any of them would be missing no notification would be generated by the system. Note, however, that they are defined in a very simple way, just using CafeOBJ syntax, which greatly eases the verification process.

2.3. Verifying with CiMPA

In contrast to proof scores, a proof assistant as CiMPA provides a strict syntax but ensures the soundness of the verification process. CiMPA supports (i) goals stated as Boolean equations; (ii) simultaneous induction; (iii) application of the theorem of constants; (iv) case splittings by constructors (in particular by true/false when dealing with Boolean terms like equalities) and special splittings for associative sequences with unit element; (v) implication with the induction hypotheses; and (vi) discharging goals by means of reduction. See Riesco and Ogata (2018) for details.

For verifying the properties with CiMPA we would state the three equations standing for the goal; note that the equations have a label for identifying them and they use the `:nonexec` attribute, so they cannot be used for execution. Then, we apply simultaneous induction (`si`) on the variable `S:Sys`. This command builds terms using the constructors for the sort `Sys` (possibly creating fresh constants). Then, it substitutes, in all goals, the variable `S:Sys` by each one of these terms, hence creating as many new goals as constructors defined for the sort (seven in this case). Because we are interested in the sixth goal, corresponding to `send2` (recursive cases appear in alphabetic order), we use the command `:sel(6)` to select it. Finally, we apply the theorem of constants (`tc`):

```
open NSLPK .
 :goal{eq [nslpk :nonexec] : inv1(S:Sys, N:Nonce) = true .
      eq [nslpk1 :nonexec] : inv2(S:Sys, N:Nonce, P:Prin) = true .
      eq [nslpk2 :nonexec] : inv3(S:Sys, N:Nonce, N0:Nonce, P:Prin)
                           = true .
 }
 :ind on (S:Sys)
 :apply(si)

 :sel(6)
 :apply(tc)
```

In particular, the theorem of constants (Goguen, 2021) substitutes variables by fresh constants and separates the equations in the goal, allowing us to prove each of them separately. The current goal is

```
 inv1(send2(S#Sys, P#Prin, P0#Prin, N#Nonce), n1@Nonce)
```

where constants generated when applying induction use the postfix `#Sort`, while the ones generated by the theorem of constant have the postfix `@Sort`. Note that we are trying to prove

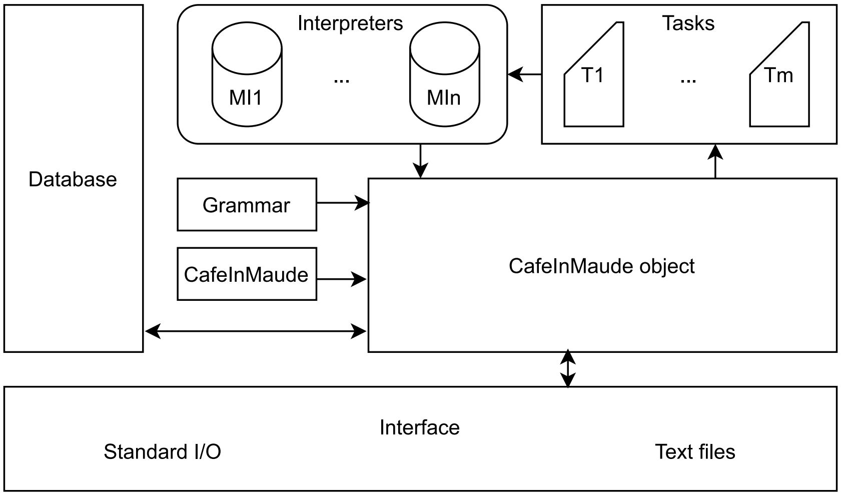

Fig. 1. CafeInMaude architecture.

the property `inv1` (the nonce secrecy property) for the `send2` message (the second message exchanged by non-intruders). We would then proceed by defining and applying the same case splittings defined in the open-close environment above, but using these constants:

```
:def csb1 = :ctf [enc1(P#Prin, N#Nonce, P0#Prin) \in nw(S#Sys)  .]
:apply(csb1)
:def csb2 = :ctf {eq intr  = P0#Prin .}
:apply(csb2)
:def csb3 = :ctf {eq N@Nonce  = n(P#Prin, P0#Prin, rand(S#Sys)) .}
:apply(csb3)
```

And, finally, we would discharge the subgoal by using implication with the corresponding induction hypothesis (with label `nslpk` when we defined the goal) with a particular substitution:

```
:imp [nslpk] by {N:Nonce <- N@Nonce ;}
:apply (rd)
```

In this case, CiMPA ensures that all cases are generated and the premise in the implication is an induction hypothesis. Once these commands have been loaded, it shows the next subgoal to be discharged, corresponding to the `false` case of the last case splitting. However, we are forced to use a strict syntax and to deal with difficult constant names (in the sense that they are not self-explaining and are built with characters such as `@` and `#`), which might be difficult for a human user.

2.4. Verifying with CiMPG

The proof in Section 2.3 can be automatically generated using CiMPG from the proof scores in Section 2.2 using CiMPG as follows:

```
open NSLPK .
 :proof(nslpk)
close
```

where the `:proof(nslpk)` command indicates that only proof scores with the `:id(nslpk)` label must be used. The basic idea when using CiMPG is that each open-close environment from a proof score corresponds to a leaf of the CiMPA proof tree. Hence, it is possible to reproduce the proof if the equations in the proof score correspond to the case splittings supported by CiMPA, presented in the previous section.

It is important to note that all the commands used in the tools in this section are enclosed in open-close environments. Moreover, the CiMPA script will not be completely generated if any open-close environment is missing or incorrect, leaving an open subgoal in the proof.

3. The CafeInMaude tool set

We present in this section the new architecture of CafeInMaude. This new architecture, described in Section 3.1, involves external objects, which allowed us to make the tool interactive, read, write, and process text files, and improve the performance of the tool. Then, in Section 3.2 we explain how we extended the ideas in Riesco and Ogata (2020) for fixing incomplete proof scores and illustrate it using our running example.

3.1. CafeInMaude architecture

We summarize in Fig. 1 the architecture of CafeInMaude. It makes extensive use of Maude 3's new features, in particular external objects and meta-interpreters as remarkable case of external objects (Clavel et al., 2020). Meta-interpreters are complete Maude instances and are able to mimic all the functionality available in a standard Maude terminal via message-passing using meta-level syntax. That is, it is possible to send them messages asking for introducing and showing modules, reducing/rewriting terms, etc. Each of these messages has the corresponding counterpart, informing about the result of the operation (different messages may distinguish success/failure states). Meta-interpreters were extended in Maude 3.1 to support creation in different processes, which allows developers to parallelize tasks in a single computer. It is worth noting that the introduction of meta-interpreters (which can be used as module database) and external rewriting via standard I/O made the Loop Mode deprecated. Among the current limitations of the external objects we highlight the blocking nature of the message for reading from the standard input, which prevents specifiers from using complex combinations of interactive applications and concurrency via meta-interpreters. To the best of our knowledge, this is the largest Maude application using these features thus far. Hence, the ideas in this section are presented in a general way, so they can be applied to other applications.

The main idea when working with external objects is that we deal with a multi-set of objects and messages. The system evolves either when (i) one object receives and processes a message, possibly generating new messages or (ii) when an object performs a computation and generates new messages. Following these ideas, the main actor in our architecture is the CafeInMaude object, which keeps a set of attributes storing context information. Although the complete list of attributes is too long for a complete enumeration, it is worth mentioning that it contains the current state, the default module (updated when a new module is introduced), the current proof tree, the current proof script, the open-close environments introduced thus far, and the number of cores the user wants to use for concurrent computation.

The CafeInMaude object interacts with the user via the standard input/output and text files (both built-in external objects). It is important to note that currently waiting for input from the user in the standard input blocks the rest of actions,[3] so input is only required when all inner actions have been completed. Once a command (either from the standard input or from a text file) is received by the CafeInMaude object, it is parsed according to the grammar and, if correct, it will be executed. CafeInMaude provides the following commands:

- Commands concerning only the current state, like setting the number of cores to be used, setting the path for storing the current proof, reading/writing a file (once the file is parsed the input it contains will be processed independently). These "basic" commands just update the internal state of the CafeInMaude object and output some information.
- Parsing CafeOBJ input (modules, views, etc.). In this case the parsing process is performed in two steps. The first step extracts the information that only depends of the term being parsed (e.g. module name, parameters, imports, and sorts). Once this information is obtained we use the module database in the left of the figure, which is a meta-interpreter in charge of storing CafeOBJ modules and views, to obtain a module extended with the information from the importations. This extended module will be used in the second step, which is in charge of solving subsorts, operators, and equations declarations.
  This step is required because we need to parse terms possibly using syntax defined in the imported modules or in the parameters, which is stored in the database. Note that these two parsing steps are the Maude 3 counterpart of Full Maude (Clavel et al., 2007, Part II) and would be required by any interactive Maude specification keeping its own module database.
- CiMPA commands. These commands just use the corresponding CiMPA functions to update the proof tree and output the corresponding information.
- CiMPG and CiMPG+F commands. These commands involve heavy computation in general, so they are implemented to work concurrently following a coordinator-worker schema. Hence, these commands must be easily split and the results easily combined. CafeInMaude currently supports a command for generating a CiMPA script from scratch, possibly using open-close environments to guide the generation of the script. In this case the goal and the variable for induction are inferred (or given by the user in some cases) and simultaneous induction (Dybjer, 1994) applied.
  Each recursive case is independent from the rest (because they correspond to different constructors), so they are a perfect choice for distributing to the workers. Hence, we create a list of tasks, as shown in the upper-right corner of the figure. Each task is composed of an identifier (its index in the proof tree), which will be used when finished to obtain the final proof, the proof tree restricted to that subgoal, and the open-close environments related to the subgoal. This list of tasks is connected with a set of meta-interpreters (top of the figure, middle). An idle meta-interpreter will take the first available task in the list and start it, updating its state to indicate it cannot admit new tasks. Once finished, it will send a message to the CafeInMaude object and become idle again, so it can start the next task (if any). Combining the results just consists of introducing the `:sel(N)` command from CiMPA, with `N` the identifier of the task, which selects the goal `N` as current one, and then concatenating the computed proof. Note that we must rename case splittings using the corresponding identifier to avoid clashes with other concurrent computation. Splitting and combining tasks is not that difficult, and thus, we use the coordinator to perform worker tasks as well.

Even though we presented the intuition about how tasks are distributed above, it is worth discussing the details of the system. First, note that the proof consists of the list of open goals identifiers and the proof tree, where each node contains a node identifier, a set of equations (the goals), and a module (the initial module possibly enriched with fresh constants, the induction hypotheses, and case splittings). The list of open goals is modified when different actions are applied to the tree and the proof is finished when it becomes empty. Fig. 2 (bottom) presents the CafeInMaude object containing a proof script (which would contain the goal and the induction command), the proof tree that we would obtain after applying induction, and the open-close environments introduced into the system. Note that the tree consists of the root and n children, each of them corresponding to one recursive case. We have colored the nodes to indicate that they are part of the list of open goals to simplify the presentation. Then, each task in the tasks list (top-right corner of the figure) is generated by creating a term with the function `generateProof` (from CiMPG) at the top and the proof tree and the open-close environments restricted to each particular case, as arguments.[4] This restriction removes all subgoals, except for the selected one, from the list of open goals, so the proof will be generated only for one recursive case. Finally, each task has an identifier, which corresponds to the index of the recursive case being proven. The meta-interpreters, represented in the top-left corner of the figure, are stored as a set of pairs containing the meta-interpreter identifier and its state (either idle or busy). As soon as a meta-interpreter is available (state idle) and there is at least one task in the list the following happens: (i) a message is sent to the meta-interpreter, asking it to complete the task; (ii) the state of the meta-interpreter becomes busy; and (iii) the task is removed from the list. Once the task is finished the meta-interpreter answers with another message including a proof script starting with the `:sel(i)` command. Note that it might be the case that the proof is not successful. In this case there is no error message: instead, the script will contain a `:postpone` command indicating that some subgoals could not be discharged. This means that the proof script will not be complete and the user will be in charge of finishing the postponed subgoals. This proof script is appended to the current proof in the CafeInMaude object, possibly renaming case splitting to avoid name clashes.

[3] As explained in the conclusions, this restriction has been solved in the last alpha release of Maude. Because this feature is still experimental, in this section we only discuss the stable features.

[4] Note that the proof tree already contains the module and the goals. As we will explain in the next section, some extra parameters are required but we skip them here for the sake of simplicity.

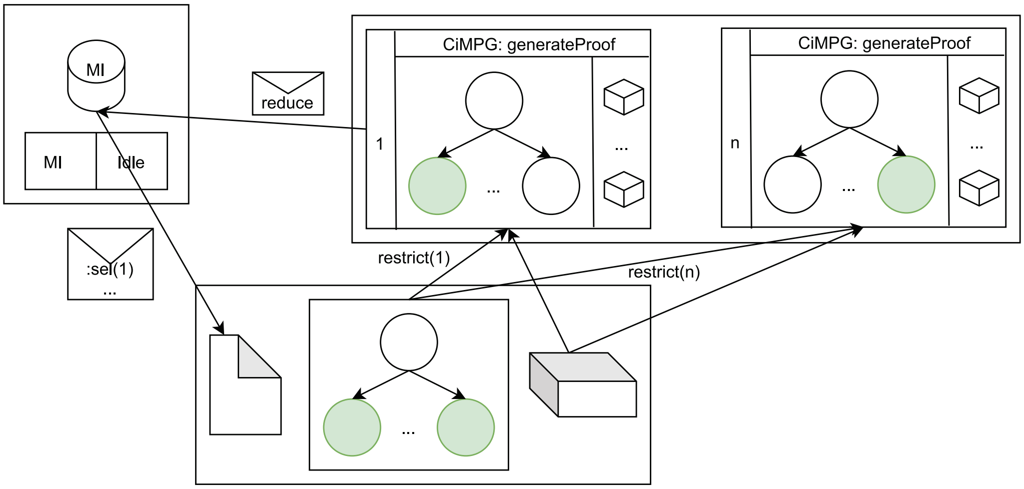

Fig. 2. Interaction between CafeInMaude, the tasks list, and the meta-interpreters. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Although we will present in Section 4 the CafeInMaude commands, the basic, non-interactive methodology of CiMPA and CiMPG has been already presented in Riesco and Ogata (2018). The main novelties in CiMPA are its interactive nature and a new command that suggests possible case splittings to the user; on the other hand, CiMPG is interactive, concurrent, and shares the same command with CiMPG+F, so proofs can be partially generated and partially inferred from proof scores. Its concurrent execution improves the previous version of CiMPG not only because it is concurrent, also thanks to the splitting process: picking the open-close environments corresponding to the subgoal beforehand speeds-up the later operations, because the search space becomes much smaller. Regarding CiMPG+F, besides using the same command as CiMPG we have improved its behavior and included new decision procedures and heuristics, as explained in the next section.

3.2. CiMPG+F

In Riesco and Ogata (2020) we presented the main idea underlying CiMPG+F: it consists of a bounded depth-first algorithm where we check whether the current subgoal can be reduced to `true` (possibly using the induction hypotheses as premises) or, otherwise, the possible case-splittings are extracted from the current subgoal. Then, they are applied and, for each one, recursion is used to try to discharge the subgoal in the new context. If terms for case splittings are extracted, it also indicates which type of case splitting should be applied, considering the ones supported by CiMPA: by true-false for Boolean terms, by constructors, and special case splittings for associative sequences. Because backtracking algorithms are very expensive, using heuristics to improve its performance in practice is a very important issue. We present here some of these heuristics, as well as decision procedures for lists for improving the generation of case splittings. Finally, we show that using the induction hypotheses is useful not just when trying to discharge the subgoal but also when generating case splittings.

Fig. 3 presents CiMPG+F algorithm. Note that it requires, besides the current goal and the corresponding module, three parameters: the bound d in the depth of the search tree; the number of implications i; and the free variables v that can be taken into consideration when trying to discharge the goal. Note that choosing the appropriate configuration for these parameters heavily affects the performance of the tool. Right now we compute the best values experimentally. Our experience indicates that i and v can be low (1 or 2), because it is unlikely to use many different implications when discharging goals, while d must be higher (around 7 for medium examples). The result p is a sequence of CiMPA commands. The algorithm applies induction if required (line 2) and then calls GENPROOF (lines 5–30), which is in charge of the backtracking algorithm. GENPROOF first applies the theorem of constants (line 6) if the goal contains variables. The theorem of constants is also applied if the goal consists of more than one subgoals, even if there are no variables, in order to prove each of them separately. Then, we use the auxiliary function DISCHARGE (line 8), explained below, to check whether no more case splittings are required and the subgoal can be discharged by using implications and reduction. If it is not the case and we can still explore more levels (that is, d is greater than 0), then we compute a list of candidates for case splitting and try to use them to discharge the subgoal (lines 9–26). We first compute the candidates for case splitting (cs) related to the current goal (line 10) and prioritize them (line 11). The GENERATE function, explained below, generates case splittings related to those terms that, if reduced, would force the goal to be further reduced, progressively approaching it to `true`. In turn, the PRIORITIZE function takes into account that case splittings related to associative sequences work better in practice than case splittings by constructors, which work better than case splittings by true-false, so it orders the list of case splittings following this idea. Then, we compute the case splittings generated when using implication with the hypotheses in the same way (lines 12–16) and append them to the case splittings generated previously (line 17). It is important to note that the order of the composition in line 17 matters, because in practice case splittings obtained from the current goal are more often required than those generated when using the hypotheses. Note that we also remove potential repetitions before concatenating. Then, the loop in lines 18–25 applies, using CiMPA, the corresponding case splitting and recursively continues the search. Because applying case splitting generates in general several new modules where the goal needs to be discharged, we need an extra loop (lines 21–23) to discharge each of them. The original goal is discharged if all of them are discharged as well (line 24). Finally, in line 27 we indicate that if we have reached the maximum depth allowed by the d parameter or no case splitting allowed us to discharge the subgoal then a `:postpone` command is generated, which indicates that the proof for this subgoal is skipped (and, in practice, left for the user). Note that we cannot ensure the proof will be generated, independently of the value

```
Require: Module $m$, goal $g$, depth $d$, implication bound $i$, variables bound $v$.
Ensure: $found$ implies that $p$ contains the proof for $g$; $\neg found$ implies the proof
    is composed of exactly a :postpone command.
 1: $p \leftarrow \emptyset$
 2: if INDUCTIONREQUIRED($g$) then $(g, m, p) \leftarrow$APPLYINDUCTION($m$, $g$)
 3: end if
 4: return GENPROOF($m$, $g$, $d$, $i$, $v$, $p$)
 5: function GENPROOF($m$, $g$, $d$, $i$, $v$, $p$)
 6:     if HASVARS($g$) or MULTIPLEGOALS($g$) then $(g, m, p) \leftarrow$THOFCONS($m$, $g$, $p$)
 7:     end if
 8:     $(found, p') \leftarrow$ DISCHARGE$(m, g, i, v, p)$
 9:     if $\neg found$ & $(d > 0)$ then
10:         $cs \leftarrow$ GENERATE$(m, g)$
11:         $cs \leftarrow$ PRIORITIZE$(cs)$
12:         $cs' \leftarrow []$
13:         for each $hypothesis \in m$ do
14:             $cs' \leftarrow cs' +$ GENERATE$(m, hypothesis\ implies\ g)$
15:         end for
16:         $cs' \leftarrow$ PRIORITIZE$(cs')$
17:         $cs \leftarrow cs + (cs' \backslash cs)$
18:         while $cs \neq \emptyset$ and $\neg$ $found$ do
19:             $c \leftarrow$ POP$(cs)$
20:             $([m_1, \ldots, m_n], g', p') \leftarrow$ APPLY$(m, p, c)$
21:             for each $m_i$ do
22:                 $(found_i, p') \leftarrow$ GENPROOF$(m_i, g', d-1, i, v, p')$
23:             end for
24:             $found \leftarrow found_1$ and $\ldots$ and $found_n$
25:         end while
26:     end if
27:     if $(\neg found)$ then $p' \leftarrow$:postpone
28:     end if
29:     return $(found, p')$
30: end function
```

Fig. 3. Main algorithm for CiMPG+F.

of the parameters, because it is possible that the case splitting required for discharging a goal is not generated by CiMPG+F or even supported by CiMPA. In this sense, CiMPG+F is not complete.

Fig. 4 presents the DISCHARGE function, used to check whether the current goal can be discharged by using implications and reduction. It receives the current module and goal, the bounds i and v, and the current proof and returns a pair where the first element indicates if the goal was discharged. If this first element is `true` then the second element of the pair contains the proof extended with the required commands to discharge it. Otherwise it contains an empty proof, in order to make sure it will not be used. The main function (lines 1–11) checks whether the goal can be directly reduced to `true`. If it is the case, it returns `true` and the current proof extended with a single reduction command (line 2). Otherwise, we use the induction hypothesis corresponding to the current property, trivially instantiating the free variables with the same fresh constants as the goal (line 4). If this new term is reduced to `true` we finish, returning the implication and the reduction commands. It is worth noting that this implication is always used, independently of the bound i. We took this decision because this implication is often required and very cheap to compute (free variables are easily bound), so the i bound is used to check whether we should continue trying the rest of hypotheses. In that case (line 9) we call the auxiliary function $DISCHARGE; otherwise (line 8) we return `false` and an empty list, so we make sure the returned sequence will not be used.

The function $DISCHARGE (lines 12–32) traverses all the hypotheses in the module (lines 13–30). For each possible implication, it computes (line 14) a preliminary substitution using a *pre-instantiation*. This method helps us to bind free variables by reducing the term and looking for terms of the form $c(\ldots, var_j, \ldots) = c(\ldots, t_j, \ldots)$, with c a constructor symbol, var_j a variable at position j, and t_j a term built with constructors and fresh constants at position j. If found, we would generate the substitution $var_j \mapsto t_j$. Although it is possible that not all variables are bound using pre-instantiations, we have found experimentally that this method greatly reduces the number of free variables. Because otherwise we need to compute all possible instantiations (explained below), reducing the state space greatly speeds up the performance of the tool for large examples. The function continues by checking (line 15) if the number of free variables remaining in the term is less than or equal to v. In that case, we continue by generating all possible instances of the subgoal by assigning fresh constants or constructors (arguments of non-constant constructors are fresh constants and constant constructors; otherwise, we would find non-termination problems) to these variables. If any of these terms can be reduced

```
Require: Module $m$, goal $g$, implication bound $i$, variables bound $v$, and sequence of CiMPA commands $p$.
Ensure: A pair $(found, p')$ is returned. $found$ implies that $g$ can be discharged using $p' \setminus p$; $\neg found$ implies $g$ cannot be discharged and $p$ is empty.
 1: function DISCHARGE($m$, $g$, $i$, $v$, $p$)
 2:     if $red(m, g) = \texttt{true}$ then return $(\texttt{true}, p +: \texttt{apply(red)})$
 3:     end if
 4:     $(g, sb) \leftarrow hyp\ implies\ g$ ▷ $hyp$ is the induction hypothesis corresponding to $g$
 5:     $p \leftarrow p +: imp[hyp]\{sb\}$
 6:     if $red(m, g) = \texttt{true}$ then return $(\texttt{true}, p +: \texttt{apply(red)})$
 7:     end if
 8:     if $i = 0$ then return $(\texttt{false}, p)$
 9:     else return $DISCHARGE($m, g, i - 1, v, p$)
10:     end if
11: end function
12: function $DISCHARGE($m$, $g$, $i$, $v$, $p$)
13:     for each $hyp \in m$ do
14:         $(g, sb) \leftarrow hyp\ implies\ g$ ▷ $sb$ computed with pre-instantiation, $g$ might contain free variables
15:         if FREEVARS($g$) $\leq v$ then
16:             $instances \leftarrow$ ALLINSTANCES($g$)
17:             if $\exists (g', sb') \in instances\ .\ red(m, g') = \texttt{true}$ then
18:                 $p \leftarrow p +: imp[hyp]\{sb \cup sb'\} +: \texttt{apply(red)}$
19:                 return $(\texttt{true}, p)$
20:             end if
21:             if $i > 0$ then
22:                 for each $(g', sb') \in instances$ do
23:                     $p' \leftarrow p +: imp[hyp]\{sb \cup sb'\}$
24:                     $(found, p') \leftarrow$ $DISCHARGE($m, g, i - 1, v, p'$)
25:                     if $found$ then return $(found, p')$
26:                     end if
27:                 end for
28:             end if
29:         end if
30:     end for
31:     return $(\texttt{false}, [])$
32: end function
```

Fig. 4. DISCHARGE function.

to `true` (line 17) then the process finishes and the corresponding proof, combining the substitution from the pre-instantiation and the substitution from the instantiation, is returned (lines 18–19). Otherwise, if we have not reached the bound *i* (line 21), then we recursively call $DISCHARGE to apply more hypotheses. Finally, if we traverse all hypotheses and the subgoal could not be discharged, the function returns `false` (line 31). This indicates that the current subgoal (and hence the whole goal) cannot be discharged.

Finally, Fig. 5 shows how the case splittings are generated by the GENERATE function (lines 1–23). It starts by initializing the variables *cs* (the list of case splittings, line 2) and *candidates* (the set of equalities that are candidates for case splitting, line 3). Then, we reduce the current goal and extract the equalities (lines 4–5), which are stored in *equals*. We traverse these equalities (lines 6–14) to check whether their terms are completely reduced or there exists at least one equation whose lefthand side matches them, given a substitution θ. If there exists such an equation, then the term was not reduced because the equation is conditional and some condition does not hold, possibly because case splittings for making it true are missing. Then, we remove the current equality from the set and introduce the first condition that does not hold (which must be an equality) into *equals*. Because of equational axioms such associativity and commutativity many different matchings can be found. In this step we introduce into *equals* all the possible instantiations (loop in lines 7–10). Likewise, we introduce matchings with all possible equations. Note also that this checking is performed for the two terms in the equality, although the algorithm only shows one of the loops for simplicity. If such an equation does not exist, then this equality is a candidate for case splitting and it is stored in *candidates*.

Once all equalities have been checked, we traverse the *candidates* (lines 15–19) to decide the case splittings that will be generated. Note that we assume an order in the equalities to ensure that the specification, once the case splitting is added, is terminating. This order is automatically computed by CiMPG+F by considering, intuitively, that constant terms are smaller than non-constant terms, and that function symbols are greater than

```
Require: Module $m$ and goal $g$.
Ensure: $cs$ a list of pairs $(c, type)$, where $c$ a term and $type$ the type of case
    splitting. The terms in this list are valid candidates for case splitting.
 1: function GENERATE($m$, $g$)
 2:     $cs \leftarrow []$
 3:     $candidates \leftarrow \emptyset$
 4:     $red \leftarrow$ REDUCE($m, g$)
 5:     $equals \leftarrow$ GETEQUALITIES($red$)
 6:     for each $t = t' \in equals$ do ▷ The following loop would be repeated for t'
 7:         for each $\theta \;\mid \theta(l) = t\mid_{pos}$ & $ceq\ l = r\ if\, c_1 = c'_1 \ \wedge \cdots \wedge c_n = c'_n \in m$ do
 8:             $cond \leftarrow$ REDUCE($m, \theta c_i = \theta c'_i$)   ▷ for $i$ the first condition that fails
 9:             $equals \leftarrow (equals \backslash (t = t')) \cup$ GETEQUALITIES($cond$)
10:         end for
11:         if not entered any of the previous loops then
12:             $candidates \leftarrow candidates \cup t = t'$
13:         end if
14:     end for
15:     for each $t = t' \in candidates$ do                    ▷ We assume $t > t'$
16:         if ONLYCTORS($t'$) then $cs \leftarrow cs +$ SUBTERMS($t$)
17:         else if SEQEQ($t = t'$) then  $cs \leftarrow cs +$ DECPROC($t = t'$)
18:         else $cs \leftarrow cs + (t = t', eq) +$ SUBTERMS($t$) + SUBTERMS($t'$)
19:         end if
20:     end for
21:     return $cs$
22: end function
23: function SUBTERMS($t$)
24:     $cs \leftarrow []$
25:     for each subterm $t'$ of $t$ s.t. $t'$ has a function symbol at the top do
26:         if ISASSOCIDSEQ($t'$) then $cs \leftarrow cs + (t, seq)$
27:         else $cs \leftarrow cs + (t, term)$
28:         end if
29:     end for
30:     return $cs$
31: end function
```

Fig. 5. GENERATE and SUBTERMS functions.

fresh constants, which are greater than constructors. In line 16 we indicate that if the right hand side is only built with constructors (we assume that if both terms are built only with constructors the term would be reduced, so the lefthand side must contain function symbols and/or fresh constants) then we compute the case splittings for the lefthand side with the auxiliary function SUBTERMS, explained below. Otherwise (line 17), if the equality is between associative sequences with identity, checked by SEQEQ, we use specific decision procedures by using DECPROC. These procedures are not general, but a series of rules to identify particular cases and simplify the equalities (if an equality does not fit into any rule then the SUBTERMS function is used). As an example of these rules, if we have the equality `s1 e1 e2 s2 = s3 e3` (note the empty syntax for concatenation of lists), for `e1`, `e2`, and `e3` particular elements and `s1`, `s2`, and `s3` sequences (associative and with unit element) then CiMPG+F suggests `s1 e1 = s3`, assuming that in the next step `e2 = e3` can be suggested (and `s2` will work as unit element). Note that not all combinations will be tried: for the time being, we only included those matchings that are more often used in our proof scores. Once more examples are developed we plan to analyze the impact of not including these solutions and will consider if they must be added. In other case (line 18), CiMPG+F suggests case splitting by true-false for the equality (indicated with the keyword `eq`) and uses the SUBTERMS function in both sides of the equality.

The SUBTERMS function (lines 23–31) traverses all the subterms of the given term (including the whole term) and first checks whether the type of the (sub)term is an associative sequence with identity. In that case (line 26) case splitting for sequences is returned (note the `seq` keyword). Otherwise, case splitting by constructors is returned using the keyword `term`.

Soundness of the approach. The soundness of the approach relies on the correctness of the inference rules used by CiMPA (Riesco and Ogata, 2018). These inference rules are the ones for constructor-based logics (see e.g. Găină et al. (2012, 2013)). It is easy to see that discharging a goal by means of reduction and implications and applying induction and the theorem of constants are guaranteed to be correct in this context. Regarding case splittings, the algorithms in this section look for the most appropriate terms for case splitting, taking into account how they are built to choose the adequate type. Intuitively, they are correct because we select case splitting by true-false for Boolean terms, by associative sequences once we check the constructors for the term build an associative sequence with identity, and by terms otherwise. Once the terms and the type of case splitting are decided, it is in charge of CiMPA to develop the proof following the appropriate inference rules. Hence, the correctness of the approach is not modified by CiMPG+F.

3.2.1. Illustrating the CiMPG+F algorithm

We use the example from the previous section to illustrate how CiMPG+F works. We will use $d = 6$, $i = 1$, and $v = 2$ for the bounds in the algorithms above. Assume we want to discharge the `send2` recursive case for the `inv1` property, as we have discussed above. More specifically, the goal has the form `inv1(send2(s, p1, p2, m), n1)` (we assume the arguments have been defined as constants of the appropriate types). We would first use DISCHARGE to check whether it can be directly discharged, which is not the case, but it is reduced to:

```
true xor n1 \in nonces(send2(s, p1, p2, m)) xor
n1 \in nonces(send2(s, p1, p2, m)) and intr = p1(n1) xor
n1 \in nonces(send2(s, p1, p2, m)) and intr = p2(n1) xor
n1 \in nonces(send2(s, p1, p2, m)) and intr = p1(n1) and
intr = p2(n1) : Bool
```

Similarly, it is not reduced to `true` by using implication `(inv1(s, n1) implies inv1(send2(s, p1, p2, m), n1)`, where we trivially use the same constant as the goal, `n1`, to instantiate the induction hypothesis, `inv1(s, N:Nonce))`.

Because $i = 1$ it is possible to use $DISCHARGE, but it will not find a way to reduce the goal to `true` using the rest of hypotheses. For example, it would use the `inv2(s,N1:Nonce,Q1:Prin)` hypothesis. Although pre-instantiation cannot bind any variable, because we have $v = 2$ we can compute all possible instantiations of the variables and try to reduce the goal. Although not exhaustive, some of the instantiations that CiMPG+F considers are:

```
inv2(s,m,p1)        inv2(s,m,p2)        inv2(s,m,intr)
inv2(s,n1,p1)       inv2(s,n1,p2)       inv2(s,n1,intr)
inv2(s, n(p1,p1, seed), p1)             inv2(s, n(p1,p1, seed), p2)
inv2(s, n(p1,p1, seed), intr)           inv2(s, n(p2,p2, seed), p1)
inv2(s, n(p2,p2, seed), p2)             inv2(s, n(p2,p2, seed), intr)
```

Note that the variables have been first instantiated with the fresh constants available from the goal, then with constant constructors (such as `intr` and `seed`), and then with non-constant constructors such as `n`, which can only receive constant values. Even trying all these possible instantiations the goal cannot be discharged. Although we could try using recursively more implications we had $i = 1$, so we stop here and this case fails. Finally, the `inv3(s,N1:Nonce,N2:Nonce,Q1:Prin)` case is easier, because pre-instantiation does not work and hence the number of free variables is greater than v and not even tried.

Because the goal cannot be discharged using only implications and reduction we use GENERATE to compute all the possible case splittings. The initial value for *equals* contains `n1 \in nonces(send2(s, p1, p2, m))` (we assume Boolean terms remove the `= true` part for simplicity), `intr = p1(n1)`, and `intr = p2(n1)`. The terms `intr = p1(n1)`, and `intr = p2(n1)` cannot be further reduced, but for `nonces(send2(s, p1, p2, m))` we have the following equation from Section 2.1:

```
ceq nonces(send2(S,P1,P2,N1))
  = (if P2 = intr then (N1 n(P1,P2,rand(S)) nonces(S))
            else nonces(S) fi)
 if c-send2(S,P1,P2,N1) .
```

The term matches the lefthand side of the equation with the substitution `S` ↦ `s`, `P1` ↦ `p1`, `P2` ↦ `p2`, `N1` ↦ `m` but it was not reduced, so it means the effective condition does not hold. We reduce `c-send2(s,p1,p2,m)` and add the result (see Section 2.1), `enc1(p1,m,p2) \in nw(s)` (we omit the `= true` part again), to *equals*. Note that there are no equations for reducing this term, so the *candidates* variable from the GENERATE algorithm contains `{enc1(p1,m,p2) \in nw(s) = true, intr = p1(n1), intr = p2(n1)}` once all equalities have been traversed.

When traversing this set, we notice that `intr` and `true` are constructors, so case splittings are generated for `p1(n1)`, `p2(n1)`, `enc1(p1,m,p2) \in nw(s)`. Because they are not associative sequences with identity, this last step is easy: we get the pairs `(p1(n1), term)`, `(p2(n1), term)`, `(enc1(p1,m,p2) \in nw(s), term)`, and `(nw(s), term)`, where the keyword `term` indicates that all case splittings are by constructors on those terms. Note that we include `nw(s)` because it is a term with a function symbol at the top but it is not an associative sequence *with identity*. If the `id:` attribute, used for stating identity, were used in the specification in Section 2.1, then case splitting for sequences would be suggested. Likewise `enc1(p1,m,p2)` is not included because `enc1` is a constructor. Also note that we showed in Section 2.2 that `(enc1(p1,m,p2) \in nw(s), term)` is the first splitting required to discharge the goal. If chosen, the algorithm would continue finding the appropriate case splittings until the subgoal is discharged.

However, the case splittings above are not the only ones considered by CiMPG+F. Although we know that in this case they will not be required (and in fact the case splittings above will be prioritized, as explained in the previous section), CiMPG+F also computes possible case splittings when using the rest of the hypotheses for implication. In our case, we would use `inv2(s,N,Q)` and `inv3(s,N1,N2,Q)`, with `N`, `N1`, and `N2` variables of sort `Nonce` and `Q` a variable of sort `Prin`, as hypotheses. The steps explained above would be repeated with the terms obtained when reducing the corresponding implications. These terms are too big for presentation and we cannot display them here, but it is worth mentioning that among the equalities we find `n1 \in nonces(s)` (for `n1` the variable from `inv1`), so new case splittings from this implication would be `(n1 \in nonces(s), term)` and `(nonces(s), term)`. We will see more details about these case splittings in the next section.

4. Using the tools

We present in this section how to use the new CafeInMaude tool set to prove the properties on NSLPK presented in Section 2.1. We would start the tool launching Maude and loading the `cafeInMaude.maude` file, available in Riesco (2022a):

```
$ maude -allow-files src/cafeInMaude.maude
```

Note that we need the `-allow-files` option for allowing Maude to read and write text files. Once loaded, the prompt will show `CafeInMaude>` and all CafeInMaude commands will be accepted. In particular, we use the `load` command to load text files. In our case we load the NSLPK specification (Riesco, 2022b). Optionally, we can also load a (possibly partial) proof score, although in this case this step is not required and hence we do not show it:

```
CafeInMaude> load nslpk.cafe .
```

Then, we set the number of cores we want our computer to use (e.g. four), so four meta-interpreters are created with state idle, and specify the path for writing text files:

```
CafeInMaude> set-cores 4 .
CafeInMaude> set-output nslpk-proof.cafe .
```

We distinguish in the next sections between automatic inference and interactive sessions.

4.1. Automatic inference

In this section we assume the user is interested in automatically generating the proof for `inv1` from Section 2.1. First, because he/she knows that `inv2` and `inv3` might be required, they are added as follows:

```
CafeInMaude> :proven(inv2(S:Sys, N1:Nonce, Q:Prin))
CafeInMaude> :proven(inv3(S:Sys, N1:Nonce, N2:Nonce, Q:Prin))
```

which indicates that these properties can be used as premises when requiring induction hypotheses (the user is in charge of proving them in a separate session). Then, the user must introduce the goal to be proven. Although it can be inferred when a proof score is given, because in this case we assume only partial information is given (if any), we need to introduce:

```
open NSLPK .
 :id(nslpk)

 op n1 : -> Nonce .

 red inv1(S:Sys, n1) .
close
```

Implicitly, this open-close environment indicates that induction must be performed on `S:Sys`, because it is the only variable in the environment. Notice that the `:id` label has the same name as the environments shown in Section 2.2, so they can be used together. It is also important, for the induction hypothesis, to use the same variable name in both the goal and the `proven` commands. Once this environment has been loaded it is enough to execute the following command to start the generation of the proof:

```
CafeInMaude> :infer-proof nslpk .
```

Note that an equivalent command was introduced into an open-close environment in the previous version of the tool. When CiMPG+F receives this command it stops the interaction with the user and takes all the open-close environments with the `nslpk` label, applies induction, and generates seven recursive cases, one for each constructor of the sort `Sys` in alphabetical order (`fake1`, `fake2`, `fake3`, `init`, `send1`, `send2`, and `send3`). Hence, seven tasks, including the open-close environments related to them (if any, as explained in Section 3.1), are generated. It is important to remember that the open-close environments in each task are only those related to the particular recursive case, so if we consider some case is particularly complex we can give extra information about it, without writing the environments for the rest of cases.

Because four cores can be used, the first four tasks are started; as tasks are finished they are reported to the CafeInMaude object and the first of the remaining ones is started until the proof is completely finished. For example, in our case the first task that is finished is the fourth one, corresponding to `init`, which just starts with `:sel(4)` to indicate that it corresponds to the fourth recursive case and then requires reduction (`:apply(rd)`) to discharge the goal. Finishing this task allows the fifth case, `send1`, to start. Hence, note that tasks are finished in a different order to the one they started. Once the task list is empty and all meta-interpreters are idle the following message is shown

```
Execution finished.
```

and the interaction with the user is restarted. Then, it is enough to use the following command to store the proof in the text file introduced above:

```
CafeInMaude> :save-proof .
```

It is important to remember that this script might not be complete, in the sense that some subgoals could not be discharged; in this case the script contains `:postpone` commands and loading the proof in CiMPA will provide extra information about these subgoals.

Finally, remember that the `:infer-proof` command is used and executed in the same way if we just want to use CiMPG. Once the proof score has been introduced, it would not be required to introduce the goal because CiMPG can infer the goal and the variable used for induction from the reductions in the open-close environments. Then, the tasks generated from the command will include a non-empty set of environments related to the goal and, before applying the algorithm from the previous section, will try to use them to generate a (possibly partial) proof for the recursive case. In case the proof score introduced by the user was not correct CiMPG would try to fix the proof using CiMPG+F. The corresponding proof script will include comments indicating those commands generated by CiMPG+F.

4.2. Interactive session

In this section we briefly describe how the new version of CiMPA is used. We refer to Riesco and Ogata (2018) and the tool repository for an exhaustive list of commands. We start the proof by introducing the goal; although we can use the `:proven` command shown in the previous section, in this case we will show how to prove the three goals simultaneously by introducing the following goal (which can be outside an open-close environment, as the rest of commands):

```
:goal{
 eq [proofNSLPK :nonexec]  : inv1(S:Sys, n1:Nonce) = true .
 eq [proofNSLPK1 :nonexec] : inv2(S:Sys, n1:Nonce, q:Prin) = true .
 eq [proofNSLPK2 :nonexec] : inv3(S:Sys, n1:Nonce, n2:Nonce, q:Prin)
                           = true .
}
```

Following the same ideas presented in Section 2.3 we apply induction, select the goal for `send2`, and apply the theorem of constants:

```
:ind on (S:Sys)
:apply(si)

:sel(6)
:apply(tc)
```

If we print the information of the proof we obtain

```
CafeInMaude> :desc proof .
 6. SI  eq [proofNSLPK :nonexec] :
     inv1(send2(S#Sys, P#Prin, P0#Prin, N#Nonce), n1:Nonce) = true .

  -- Assumption:
  eq [proofNSLPK :nonexec] : inv1(S#Sys, n1:Nonce)  = true .
  -- Assumption:
  eq [proven-goal0 :nonexec] : inv2(S#Sys, N1:Nonce, Q:Prin)  = true .
  -- Assumption:
  eq [proven-goal1 :nonexec] : inv3(S#Sys, N1:Nonce, N2:Nonce, Q:Prin)  = true .
  6-1. > TC  eq [proofNSLPK :nonexec] :
            inv1(send2(S#Sys, P#Prin, P0#Prin, N#Nonce), n1@Nonce) = true .
```

Table 1
Benchmarks for CiMPG and CiMPG+F.

Protocol	Spec. size	Proof score	Proof script	CiMPG (old)	CiMPG	CiMPG (conc.)	CiMPG+F (old)	CiMPG+F	CiMPG+F (conc.)	Auto.
2p-mutex	58	73	23	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s	100%
TAS	73	341	88	10 s	9 s	7 s	<1 s	<1 s	<1 s	100%
Qlock	124	444	88	24 s	23 s	12 s	<1 s	<1 s	<1 s	100%
Cloud	127	1 715	530	132 s	82 s	27 s	<1 s	<1 s	1 s	100%
SCP	182	664	200	202 s	62 s	31 s	340 s	51 s	34 s	100%
NSLPK$_1$	188	1 173	342	284 s	118 s	59 s	617 s	26 s	14 s	100%
ABP	320	8 655	1370	5 h 13 m	5 h 13 m	2 h 25 m	2 h 35 m	2 h 30 m	2 h 1 m	100%
MCS	342	6 517	2484	2 h 21 m	2 h 14 m	53 m 50 s	–	3 h 16 m	59 m 14 s	52,5%
NSLPK$_2$	568	15 713	3829	12 h	2 h 53 m	2 h 1 m	–	4 h 59 m	2 h 57 m	90%

where we have omitted the information unrelated to the current subgoal. Note that fresh constants have been generated as explained in Section 2.3 and that the induction hypotheses are the only assumptions; case splittings will be added to these assumptions when introduced by the user.

At this point, we can ask CiMPG+F about the case splittings computed by the GENERATE algorithm in Section 3.2 with the `:help` command:

```
CafeInMaude> :help .
 The following case splittings are recommended:
  Case splitting by constructors for
                 enc1(P#Prin, N#Nonce, PO#Prin) \in nw(S#Sys)
  Case splitting by constructors for nw(S#Sys)
  Case splitting by constructors for p1(n1@Nonce)
  Case splitting by constructors for p2(n1@Nonce)
 The following case splittings are recommended when using premises:
  Case splitting by constructors for N#Nonce \in nonces(S#Sys)
  Case splitting by constructors for n1@Nonce \in nonces(S#Sys)
  Case splitting by constructors for
                 enc1(P#Prin, N#Nonce, intr) \in nw(S#Sys)
  Case splitting by constructors for
                 enc1(P#Prin, n1@Nonce, intr) \in nw(S#Sys)
  Case splitting by constructors for
                 enc1(PO#Prin, N#Nonce, intr) \in nw(S#Sys)
  Case splitting by constructors for
                 enc1(PO#Prin, n1@Nonce, intr) \in nw(S#Sys)
  Case splitting by constructors for nonces(S#Sys)
  Case splitting by constructors for p1(N#Nonce)
  Case splitting by constructors for p2(N#Nonce)
```

This novel feature makes available for the user the internal knowledge that was hidden in the previous version of the tool. As explained in the previous section, the information distinguishes between the case splittings obtained from the current goal and the ones computed when using implications. In this case we would use the first one and continue with the proof. If we ask for case splittings in the next step we obtain:

```
CafeInMaude> :help .
The following case splittings are recommended:
 Case splitting by constructors recommended for N@Nonce
 Case splitting by constructors recommended for PO#Prin
 Case splitting by constructors recommended for p1(N@Nonce)
 Case splitting by constructors recommended for p2(N@Nonce)
The following case splittings are recommended when using premises:
 ...
```

In this case, we find case splitting by constructors using `PO#Prin`, which in fact is the second case splitting (the constructor for `Prin` is `intr`) that we used for our proof. CiMPG+F suggests case splitting by constructors because it is more powerful, assuming equalities are defined for all constructor symbols, than using equalities. After applying this case splitting we ask for help again and we find:

```
CafeInMaude> :help .
The following case splittings are recommended:
 Case splitting by true-false recommended for N#Nonce = N@Nonce
 Case splitting by true-false recommended for
            n(P#Prin, PO#Prin, rand(S#Sys)) = N@Nonce
 Case splitting by true-false recommended for p1(N@Nonce) = PO#Prin
 Case splitting by true-false recommended for p2(N@Nonce) = PO#Prin
 Case splitting by constructors recommended for N@Nonce \in nonces(S#Sys)
 Case splitting by constructors recommended for nonces(S#Sys)
 Case splitting by constructors recommended for p1(N@Nonce)
 Case splitting by constructors recommended for p2(N@Nonce)
 Case splitting by constructors recommended for rand(S#Sys)
The following case splittings are recommended when using premises:
 ...
```

The last case splitting that we require, `n(P#Prin, PO#Prin, rand(S#Sys)) = N@Nonce`, is also suggested by the tool. In this case the right hand side of the equality is `N@Nonce` because it is a constant. Once these three case splittings have been used we can discharge the subgoal and continue with the rest of the proof using the same features described thus far. Finally, it is worth remembering that incomplete proofs can be also saved in order to continue with them later.

5. Benchmarks

In this section we present the results obtained when using the CafeInMaude tool set to prove properties of different systems specified in CafeOBJ. All benchmarks were executed on a MacBook Pro with 16 GB of memory and a 2,4 GHZ CPU with four cores. We decided to set the number of workers when using concurrent computation to four; even though in some cases the same core might be busy with both the coordinator and one worker, it is worth for large proofs because the time the coordinator is working is much smaller than the worker time. It is worth noting that the values for the parameters in Section 3.2 have been chosen experimentally, using the lowest values that generate the proof.

Table 1 summarizes the information, where the columns have the following meaning:

- The *Protocol* column indicates the protocol name.
- The *Spec. size* column displays the lines of code of the protocol specification.
- The *Proof score* column presents the lines of code of the proof scores verifying the protocol.
- The *Proof script* column indicates the size, in lines of code, of the CiMPA proof scripts generated for verifying the protocol.
- The *CiMPG (old)* column presents the time required to generate the proof script from the proof scores with the previous version of the tool.
- The *CiMPG* and *CiMPG (conc.)* columns show the time required to generate the proof script from the proof scores with the new version of the tool using one and four cores, respectively.
- The *CiMPG+F (old)* column indicates the time required to generate the proof script using automatic generation with the previous version of the tool.

- The *CiMPG+F* and *CiMPG+F (conc.)* columns depict the time required to generate the proof script combining automatic generation and proof scores with the new version of the tool using one and four cores, respectively.
- The *Auto.* column indicates the degree of automation for the *CiMPG+F* and *CiMPG+F (conc.)* columns. That is, the percentage of proof scripts generated automatically, without using proof scores.

The protocols that we have used for benchmarking, ordered by specification size, are:

- *2p-mutex* is a simple protocol for mutual exclusion of two processes. We verify mutual exclusion between them.
- *TAS* (Test And Set) is a spinlock mutual exclusion protocol. The proof ensures that at most one process enters the critical section.
- *Qlock* is a variant of Dijkstra's binary semaphore. The properties in this case are mutual exclusion and that only the process on the top of the queue can go into the critical section.
- *Cloud* is a simplified cloud synchronization protocol, where one server stores a value and all clients must synchronize when a new value is uploaded. The properties verify that, when updated, the clients and the server share the same value.
- *SCP* (Simple Communication Protocol) is a simplified version of the ABP communication protocol, which uses unreliable cells as communication channels. We analyze in this case the reliable communication property: if a receiver gets n packets, then they are the first n packets dispatched by the sender, and they have been received in the same order they were sent.
- *NSLPK*$_1$ is the protocol used as running example in this paper.
- *ABP* is a communication protocol between two agents that can be understood as a simplified version of TCP. In ABP the channels are unreliable, so information might be repeated or lost, and we verify that all messages are delivered in the proper order.
- *MCS* is a list-based queuing lock algorithm for spinlocks. This protocol must verify the safety and liveness properties.
- *NSLPK*$_2$ is a more complex NSLPK specification that includes the details required for proving that the protocol satisfies both the nonce secrecy and one-to-many correspondence (which verifies that answers to messages in fact are sent by those principals that were addressed the previous message) properties. More details on this proof are presented in Mon et al. (2021).

The values in Table 1 show some relevant points. First, the new version with one core behaves in all cases at least as well as the old version, and better in general. Second, some case studies cannot be generated by CiMPG+F, while CiMPG behaves much worse for NSLPK$_2$; these issues are due to the automation level and the new features developed for the new version, as we will discuss next. It is illustrative the ABP case, which behaves almost in the same way in both cases. We consider this is due to the fact that new heuristics are used in the simplest cases, where not much improvement can be obtained. Second, using the concurrent version of the tools is more useful with bigger specifications, because for smaller ones the time required for distributing and collecting the tasks is similar to the one required to generate the whole proof. Then, the automation level of MCS and NSLPK$_2$ is not 100%. In the MCS case, we found out an interesting scalability problem: some subgoals are huge and just reducing them is a very expensive task that takes up to minutes even when reduced standalone (from the terminal, without involving CiMPG+F). Because reduction is used in all nodes of the search tree for checking whether the current subgoal can be discharged, as explained in Section 3.2, it is not possible in practice to generate the proof for them. This problem emphasizes the importance of the interaction with the user, because he/she can guide the proof in these cases. In this sense, we consider that helping the user with the case splittings that can be used, as we showed in the previous sections, is useful in systems like this. On the other hand, NSLPK$_2$ has a particular recursive case (out of ten) that requires a large number of case splittings, much larger in particular than the rest of recursive cases. Hence, it is worth using the integration between the tools to solve this case while automatically generating the rest. It is also interesting mentioning that all NSLPK$_2$ properties (17 properties in total) are not recursively dependent and can be proved one by one using the `:proven` command presented in the previous section, which greatly improves the performance of the proof; previous versions of the tool required up to 12 h to completely generate the proof with CiMPG, while we stopped CiMPG+F after this time, because the number of possible combinations of induction hypotheses is too big, hence making the state space too large for a complete traversal. Hence, the improvement is mainly due to the fact that using less hypotheses ease both the function in charge of checking whether a goal can be discharged and the list of case splittings to be tried.

We also note that CiMPG+F is faster than CiMPG mainly for smaller examples, while CiMPG is faster for larger proofs. This indicates that, for small examples, it is easier to generate the proof than to process all open-close environments and infer the proof from them. However, as the complexity of the proof grows the generation process becomes more time consuming and processing the open-close environments becomes the most adequate option for obtaining a proof. Note, however, that the time required for creating the proof scores by hand is not included in the table and it is possibly much larger than the one required to generate the proof automatically.

It is also important to note that the time for the concurrent version is not reduced by four, as could be expected from using four cores: we noticed that some recursive cases take much more time than others, so implementing a method for allowing the user to customize the particular subgoal to be parallelized is an interesting topic of future work. Moreover, it would allow us to define different parameters for each call, so proofs requiring a large number of case splittings would have a larger d value (see Section 3.2 for details) than the rest of case splittings, improving the performance of smaller case splittings while using a large enough value for generating more complex cases.

6. Related work

Theorem proving is a well established field, with several state-of-the-art tools. Among the automatic theorem provers we highlight Spass (Weidenbach et al., 2009) and Vampire (Kovács and Voronkov, 2013). Although these tools are less powerful in general than interactive theorem provers, they are very efficient for a large number of problems, and hence it is worth considering their features for automating parts of interactive theorem provers. On the other hand, possibly three of the most used interactive theorem provers are Isabelle/HOL (Nipkow et al., 2002), Coq (Huet et al., 2002), and PVS (Owre et al., 1992). These theorem provers implement powerful commands combining several heuristics for trying to automatically discharge goals. The advantages of CiMPG+F with respect to these provers are: (i) CafeOBJ is a high-level language for specifying systems, not for specifically proving theorems, which provides users with a much richer syntax, and (ii) while heuristics are general, CiMPG+F strategies are specific for each goal, making them appropriate for novel situations. However, we consider that the user interface provided by these theorem provers is far superior to the one currently used by CafeInMaude tools, so an interesting topic of future work is improving our interface, possibly integrating some graphical aspects and allowing the definition of high-level strategies. It is important to note that our generation algorithm is different from the simplification rules supported by theorem provers like Isabelle/HOL. It is also different from the Sledgehammer tool (Isabelle/HOL, 2009), a tool integrated into Isabelle/HOL that uses external automated reasoning tools to discharge a goal, which is passed to the tool together with a "smart selection of lemmas from the current theory context" (Wenzel and Berghofer, 2009). In our case, the simplification rules are directly obtained from the specification and do not rely just in matching, but in a more complex relation guided by the goal being proven. On the other hand, in Nagashima and Kumar (2017) the authors introduce a Proof Strategy Language (PSL) that is used by a proof script generator to produce tactics combinations for Isabelle/HOL. This approach is similar to ours, since they work in an integrated way as CiMPG+F when extra open-close environments are introduced. As an advantage, in our case no extra language is required, because CafeOBJ syntax is used. However, using a particular language is more flexible and may help the tool to reduce the state space and produce better results.

Another tool for automatic proof generation related to Isabelle/HOL is Zeno (Sonnex et al., 2012). Although the tool is limited to proofs of properties of recursive data structures specified in Haskell, the authors implement several heuristics for generating these proofs, including general ones like prioritizing some tactics, and some that depend of the current goal. It is worth studying whether some of these heuristics can be generalized to other programming languages and other contexts beyond recursive data structures.

It is worth discussing the NSLPK proof carried out in Isabelle/HOL (Paulson, 1998). On the one hand, the specification is much shorter in Isabelle/HOL, mainly because it just defines what happens in the set of messages (which in fact is implemented as a list), while not making observations explicit and using built-in notation for lists. In this case that the CafeInMaude specification is executable might help the user understand what is going on, although it requires a higher specification effort. On the other hand, the proof is shorter but must be done by hand, while the CafeInMaude is larger but automatically generated. Although larger proofs are worser in general when generated by hand, in an automatic setting they provide more detailed information of the proofs, which might help users to understand how they are carried out. Note that Sledgehammer has been tried with NSPK. Although the results are positive, it cannot fully verify the protocol, discharging up to 68% of the goals (Paulson, 2010). This result gives us confidence in the usability of our approach.

There exist other tools that have been used in combination with theorem provers that are worth discussing. Boogie (Leino, 2008) is an intermediate verification language designed to be used as middle layer in the construction of other verifiers. It is able to generate verification conditions, which are solved by external SMT solvers, being Z3 (de Moura and Bjørner, 2008) the default one. Dafny (Leino, 2010) is an imperative language that aims C#. Dafny has been built on top of Boogie, which is used to generate verification conditions from pre and postconditions, as well as for loop invariants. KeY (Ahrendt et al., 2016) follows a similar approach: it supports (a subset of) Java programs annotated using the Java Modeling Language and generate proof obligations that can be later discharged using their theorem prover. Why3 (Filliâtre and Paskevich, 2013) provides a language for specifying programs; then, verification conditions can be extracted from these programs and discharged using external theorem provers. Frama-C (Kirchner et al., 2015) is a platform for analyzing and proving properties of C programs. Although it has an internal prover, it is only used for simplification before sending the proof obligations to external solvers. In particular, Why3 can be used as front-end for Frama-C programs.

Automatic software repair (Gazzola et al., 2019) is a field of growing interest. For example, in model checking a logic-based machine learning technique (inductive logic programming) has been used to automatically repair models (Alrajeh et al., 2013). We propose the complementary approach, where the model is assumed correct and we try to "fix" the proof by closing gaps. Note that this fixing includes automatically generating new case splittings, which is beyond the standard automatic strategies integrated in other theorem provers, where a fixed set of previously generated lemmas is applied.

A similar approach to CiMPG+F was proposed in Nakano et al. (2007), whose authors present Crème, a tool checking invariants in CafeOBJ OTS specifications by trying case splitting. Crème also automatically generates lemma candidates and then it is unnecessary for users to prepare them in advance. Its main advantage with respect to CiMPG+F is that Crème is able to generate a counterexample when the invariant does not hold. However, it presents a number of problems: it can only use case splittings for Boolean terms, greatly limiting the proofs it can tackle; it is not directed by the current goal and the module, hence generating a much bigger state space, which makes very time-consuming to automatically generate lemma candidates and highly reduces its scalability; and it is not directly implemented in Maude but in Common Lisp, which greatly worsens its efficiency. For these reasons Crème is restricted to particular examples, while CiMPG+F has been applied to many protocols implemented in CafeOBJ.

Regarding theorem provers implemented for Maude specifications, the Interactive Theorem Prover (ITP) (Clavel et al., 2006) allowed users to prove inductive properties in Maude specifications. Although partially updated for inclusion into the Maude Formal Environment (Durán et al., 2011), the new features were not documented and currently it is discontinued. The Constructor-based Interactive Theorem Prover (Gâinâ et al., 2013) is also a theorem prover for inductive properties of Maude specifications (updated in Gâinâ et al. (2018) to deal with a novel induction scheme), which has been partially updated to use some Maude 3 features. In particular, it supports external objects for interaction with the standard I/O, although it does not keep its own database, use meta-interpreters, nor interact with text files. Regarding its features, it incorporates many general decision procedures for optimizing proofs but, as happened with other theorem provers, they are not guided by the current goal. Finally, as it does not consider any integration with CafeOBJ it cannot be externally guided by proof score-like information.

The K framework (Rosu and Serbanuta, 2010), which was initially implemented in Maude (the current implementation language is Java), is a semantic framework for the specification and analysis of programming languages. K provides theorem proving via matching logic (Chen et al., 2021), a logic proposed to reason about the static structure and dynamic behavior of programs. The main difficulty to adapt these techniques to our tool is the specific target of the K framework, which is not designed for general systems but for the semantics of programming languages.

It is interesting to consider other specification languages like Alloy (Jackson, 2012). In Alloy systems are described by means of constraints, while verification, performed by the Alloy Analyzer, follows a "lightweight" approach based in a Boolean SAT solver used to find structures that satisfy these constraints. This analyzer can be used to both explore the model and to generate counterexamples. This approach is interesting for us because Maude supports SMT annotations, that are later solved by external tools. Following Alloy ideas, specific parts of the system could be annotated, so SMT solvers can be used to find a counterexample, which would indicate the property does not hold.

Some other approaches are being developed to prove particular systems. The CompCert (Kästner et al., 2018) project has introduced different features into Coq, so compiler verification is more easily achieved. Similarly, seL4 (Trustworthy Systems Team, 2021) is a microkernel that has, as main feature, a full code-level functional correctness proof, performed in Isabelle/HOL. This enforces integrity, confidentiality, and availability, given some hardware assumptions hold. Experience in these languages might be useful for implementing new heuristics for particular classes of proofs.

Besides theorem proving, *model checking* has been used for verifying several properties of systems. Maude and Spin (Ben-Ari, 2008) support LTL model checking, while NuSMV (Cavada et al., 2010) supports CTL model checking. These tools have been used to verify, automatically, temporal properties in several systems. In particular, Spin has been used to find the error in NSPK (Dong-huo, 2008) later fixed in NSLPK. Although this verification technique is different to the one in this paper, and in fact can be used to analyze different properties (which are weaker in general, because model checking is applied to a particular initial state), it is interesting to consider how to combine them. In fact, searches have been used to analyze reachability (see e.g. Riesco et al., 2016), so our tool set could take this strategy into account and extend it with model checking in the future for some classes of properties. This would be particularly interesting (i) when looking for counterexamples to falsify a property and (ii) using an independent meta-interpreter, so this computation is performed without affecting the rest of the proof.

7. Concluding remarks and ongoing work

In this paper we have presented a new architecture for the CafeInMaude tools implemented using the last features from Maude 3.1, including meta-interpreters and rewriting with external objects. Using these features we implemented our own module database, made the tools interactive, integrated the functionality of all tools, added new commands, and implemented concurrent computation. We have also described new heuristics in the CiMPG+F tool and shown how they work. The benchmarks confirm that the tools are promising and point out some lines of future work. We are currently working on a flexible infrastructure for allowing users to send subgoals interactively for background execution, modifying on the fly the corresponding parameters. The user would continue with the proof and eventually he/she would receive a notification informing whether the subgoal was discharged or must be discharged by hand (or automatically with new parameters or after manually introducing a new case splitting). It would also allow us to define proof scripts for automatizing the proof generation with different depths for each recursive case, further splitting the most complex cases. We are experimenting with the latest Maude alpha release, which provides for the first time non-blocking interaction with the standard input (required to interact with the user), which previously prevented us from using the meta-interpreter at the same time.

We are also studying how to extend CiMPG+F to detect false properties, so the rest of the proof (if computed concurrently) would be canceled and the user informed.

It is also interesting to extend the number and the variety benchmarks, possibly using already developed sets of examples like the ones in Claessen et al. (2015) and Hajdú et al. (2021). Using these new benchmarks we could compare with other tools and analyze the usefulness of new features.

Finally, it would be interesting to use external objects to integrate the tool with a graphical interface, so the user could visualize the proof tree and select in a graphical way which case splittings he/she wants to proof by hand and which ones to proof automatically.

CRediT authorship contribution statement

Adrián Riesco: Conceptualization, Methodology, Investigation, Software, Writing – original draft. **Kazuhiro Ogata:** Conceptualization, Methodology, Investigation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Research partially supported by JSPS KAKENHI Grant Number 26240008, the MINECO Spanish project *ProCode-UCM* (PID2019-108528RB-C22), and by Comunidad de Madrid as part of the program *BLOQUES-CM* (S2018/TCS-4339) co-funded by EIE Funds of the European Union. We thank the anonymous reviewers for their comments on the first version of this manuscript.

References

Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (Eds.), 2016. Deductive Software Verification – The KeY Book, second ed. In: Lecture Notes in Computer Science, vol. 10001, Springer.

Alrajeh, D., Kramer, J., Russo, A., Uchitel, S., 2013. Elaborating requirements using model checking and inductive learning. IEEE Trans. Softw. Eng. 39 (3), 361–383. http://dx.doi.org/10.1109/TSE.2012.41.

Astesiano, E., Kreowski, H.-J., Krieg-Brueckner, B. (Eds.), 1999. Algebraic Foundations of Systems Specification, first ed. Springer, Secaucus, NJ, USA.

Ben-Ari, M., 2008. Principles of the Spin Model Checker. Springer.

Cavada, R., Cimatti, A., Jochim, C.A., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., Tchaltsev, A., 2010. NuSMV 2.6 User Manual. FBK-irst.

Chen, X., Lucanu, D., Rosu, G., 2021. Matching logic explained. J. Log. Algebr. Methods Program 120, 100638. http://dx.doi.org/10.1016/j.jlamp.2021.100638.

Claessen, K., Johansson, M., Rosén, D., Smallbone, N., 2015. TIP: tons of inductive problems. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (Eds.), Proceedings of the International Conference on Intelligent Computer Mathematics, CICM 2015. In: Lecture Notes in Computer Science, vol. 9150, Springer, pp. 333–337. http://dx.doi.org/10.1007/978-3-319-20615-8_23.

Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C., 2020. Maude Manual (Version 3). Maude Team, http://maude.lcc.uma.es/maude30-manual-html/maude-manual.html.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C., 2007. All About Maude: A High-Performance Logical Framework. In: Lecture Notes in Computer Science, vol. 4350, Springer, Berlin Heidelberg.

Clavel, M., Meseguer, J., 2002. Reflection in conditional rewriting logic. Theoret. Comput. Sci. 285 (2), 245–288.

Clavel, M., Palomino, M., Riesco, A., 2006. Introducing the ITP tool: a tutorial. J. UCS 12 (11), 1618–1650, programming and Languages. Special Issue with Extended Versions of Selected Papers from PROLE 2005: The Fifth Spanish Conference on Programming and Languages.

de Moura, L.M., Bjørner, N., 2008. Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (Eds.), 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. In: Lecture Notes in Computer Science, vol. 4963, Springer, Berlin, Heidelberg, pp. 337–340. http://dx.doi.org/10.1007/978-3-540-78800-3_24.

Dong-huo, C., 2008. Model checking of the NSPK protocol with Spin, Microelectron. Comput..

Durán, F., Rocha, C., Álvarez, J.M., 2011. Towards a Maude formal environment. In: Agha, G., Meseguer, J., Danvy, O. (Eds.), Formal Modeling: Actors, Open Systems, Biological Systems. In: Lecture Notes in Computer Science, vol. 7000, Springer, Berlin, Heidelberg, pp. 329–351.

Dybjer, P., 1994. Inductive families. Form. Asp. Comput. 6 (4), 440–465. http://dx.doi.org/10.1007/BF01211308.

Filliâtre, J., Paskevich, A., 2013. Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (Eds.), Proceedings of the 22nd European Symposium on Programming Languages and Systems, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013. In: Lecture Notes in Computer Science, vol. 7792, Springer, Berlin, Heidelberg, pp. 125–128. http://dx.doi.org/10.1007/978-3-642-37036-6_8.

Futatsugi, K., Diaconescu, R., 1998. CafeOBJ Report. In: AMAST Series, World Scientific, Singapore.

Futatsugi, K., Gâinâ, D., Ogata, K., 2012. Principles of proof scores in CafeOBJ. Theoret. Comput. Sci. 464, 90–112.

Gâinâ, D., Futatsugi, K., Ogata, K., 2012. Constructor-based logics. J. Univ. Comput. Sci. 18 (16), 2204–2233. http://dx.doi.org/10.3217/jucs-018-16-2204.

Gâinâ, D., Tutu, I., Riesco, A., 2018. Specification and verification of invariant properties of transition systems. In: 25th Asia-Pacific Software Engineering Conference, APSEC 2018. IEEE, Berlin, Heidelberg, pp. 99–108. http://dx.doi.org/10.1109/APSEC.2018.00024.

Gâinâ, D., Zhang, M., Chiba, Y., Arimoto, Y., 2013. Constructor-based inductive theorem prover. In: Heckel, R., Milius, S. (Eds.), Proceedings of the 5th International Conference in Algebra and Coalgebra in Computer Science, CALCO 2013. In: Lecture Notes in Computer Science, vol. 8089, Springer, Berlin Heidelberg, pp. 328–333.

Gazzola, L., Micucci, D., Mariani, L., 2019. Automatic software repair: A survey. IEEE Trans. Softw. Eng. 45 (1), 34–67. http://dx.doi.org/10.1109/TSE.2017.2755013.

Goguen, J.A., 2021. Theorem proving and algebra. pp. 1–427, CoRR abs/2101.02690. arXiv:2101.02690.

Hajdú, M., Hozzová, P., Kovács, L., Schoisswohl, J., Voronkov, A., 2021. Inductive benchmarks for automated reasoning. In: Kamareddine, F., Coen, C.S. (Eds.), Proceedings of the 14th International Conference on Intelligent Computer Mathematics, CICM 2021. In: Lecture Notes in Computer Science, vol. 12833, Springer, pp. 124–129. http://dx.doi.org/10.1007/978-3-030-81097-9_9.

Huet, G., Kahn, G., Paulin-Mohring, C., 2002. The Coq Proof Assistant: A Tutorial: Version 7.2. Technical Report 256, INRIA, URL ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RT/RT-0256.pdf.

2009. Isabelle/HOL, Sledgehammer, https://isabelle.in.tum.de/website-Isabelle2009-1/sledgehammer.html.

Jackson, D., 2012. Software Abstraction (Revised Edition). The MIT Press.

Kästner, D., Wünsche, U., Barrho, J., Schlickling, M., Schommer, B., Schmidt, M., Ferdinand, C., Leroy, X., Blazy, S., 2018. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In: ERTS 2018: Embedded Real Time Software and Systems, SEE. pp. 1–9, URL http://xavierleroy.org/publi/erts2018_compcert.pdf.

Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B., 2015. Frama-C: A software analysis perspective. Form. Asp. Comput. 27 (3), 573–609. http://dx.doi.org/10.1007/s00165-014-0326-7.

Kovács, L., Voronkov, A., 2013. First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (Eds.), Proceedings of the 25th International Conference on Computer Aided Verification, CAV 2013. In: Lecture Notes in Computer Science, vol. 8044, Springer, Berlin, Heidelberg, pp. 1–35. http://dx.doi.org/10.1007/978-3-642-39799-8_1.

Leino, K.R.M., 2008. This is boogie 2. unpublished. URL https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/.

Leino, K.R.M., 2010. Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. In: Lecture Notes in Computer Science, vol. 6355, Springer, Berlin, Heidelberg, pp. 348–370. http://dx.doi.org/10.1007/978-3-642-17511-4_20.

Lowe, G., 1995. An attack on the Needham-Schroeder public-key authentication protocol. Inform. Process. Lett. 56 (3), 131–133.

Mon, T.W., Fujii, S., Tran, D.D., Ogata, K., 2021. Formal verification of iff & nslpk authentication protocols with CiMPG. In: Proceedings of the 33rd International Conference on Software Engineering & Knowledge Engineering, SEKE 2021. pp. 120–125.

Nagashima, Y., Kumar, R., 2017. A proof strategy language and proof script generation for isabelle/HOL. In: de Moura, L. (Ed.), Proceedings of the 26th International Conference on Automated Deduction, CADE 26. In: Lecture Notes in Computer Science, vol. 10395, Springer, pp. 528–545. http://dx.doi.org/10.1007/978-3-319-63046-5_32.

Nakano, M., Ogata, K., Nakamura, M., Futatsugi, K., 2007. Creme: An automatic invariant prover of behavioral specifications. Int. J. Softw. Eng. Knowl. Eng. 17 (6), 783–804.

Needham, R.M., Schroeder, M.D., 1978. Using encryption for authentication in large networks of computers. Commun. ACM 21 (12), 993–999. http://dx.doi.org/10.1145/359657.359659.

Nipkow, T., Paulson, L.C., Wenzel, M., 2002. Isabelle/HOL: A Proof Assistant for Higher-Order Logic. In: Lecture Notes in Computer Science, vol. 2283, Springer, Berlin Heidelberg.

Owre, S., Rushby, J.M., Shankar, N., 1992. PVS: A prototype verification system. In: Kapur, D. (Ed.), Proceedings of the 11th International Conference on Automated Deduction, CADE 11. In: Lecture Notes in Computer Science, vol. 607, Springer, Berlin, Heidelberg, pp. 748–752. http://dx.doi.org/10.1007/3-540-55602-8_217.

Paulson, L.C., 1998. The inductive approach to verifying cryptographic protocols. J. Comput. Secur. 6 (1–2), 85–128.

Paulson, L.C., 2010. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In: Schmidt, R.A., Schulz, S., Konev, B. (Eds.), Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning, PAAR-2010. In: EPiC Series in Computing, vol. 9, EasyChair, pp. 1–10. http://dx.doi.org/10.29007/tnfd.

Riesco, A., 2022a. CafeInMaude GitHub repositoty, https://github.com/ariesco/CafeInMaude.

Riesco, A., 2022b. CafeInMaude GitHub repositoty, https://github.com/ariesco/CafeInMaude/tree/master/examples/CCiMPG/NSLPK.

Riesco, A., Ogata, K., 2018. Prove it! inferring formal proof scripts from CafeOBJ proof scores. ACM Trans. Softw. Eng. Methodol. 27 (2), 6:1–6:32.

Riesco, A., Ogata, K., 2020. CiMPG+F: A proof generator and fixer-upper for CafeOBJ specifications. In: Pun, V.K.I., Stolz, V., Simão, A. (Eds.), Proceedings of the 17th International Colloquium on Theoretical Aspects of Computing, ICTAC 2020. In: Lecture Notes in Computer Science, vol. 12545, Springer, Berlin, Heidelberg, pp. 64–82. http://dx.doi.org/10.1007/978-3-030-64276-1_4.

Riesco, A., Ogata, K., Futatsugi, K., 2016. A Maude environment for CafeOBJ. Form. Asp. Comput. 29 (2), 309–334. http://dx.doi.org/10.1007/s00165-016-0398-7.

Rosu, G., Serbanuta, T., 2010. An overview of the k semantic framework. J. Log. Algebr. Methods Program. 79 (6), 397–434. http://dx.doi.org/10.1016/j.jlap.2010.03.012.

Sawada, T., Futatsugi, K., Preining, N., 2015. CafeOBJ Reference Manual (Version 1.5.3). Japan Advanced Institute of Science and Technology.

Sonnex, W., Drossopoulou, S., Eisenbach, S., 2012. Zeno: An automated prover for properties of recursive data structures. In: Flanagan, C., König, B. (Eds.), Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012. In: Lecture Notes in Computer Science, vol. 7214, Springer, pp. 407–421. http://dx.doi.org/10.1007/978-3-642-28756-5_28.

Trustworthy Systems Team, 2021. Data61, SeL4 Reference Manual - Version 12.1.0.

Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P., 2009. SPASS version 3.5. In: Schmidt, R.A. (Ed.), Proceedings of the 22nd International Conference on Automated Deduction, CADE 2009. In: Lecture Notes in Artificial Intelligence, vol. 5663, Springer, Berlin, Heidelberg, pp. 140–145.

Wenzel, M., Berghofer, S., 2009. The Isabelle System Manual. TU München.

Adrián Riesco is Associate Professor at Facultad de Informática – Universidad Complutense de Madrid (UCM). He has worked mainly on formal methods in rewriting logic, metaprogramming, theorem proving, symbolic biological systems, debugging, testing and on analysis of emerging technologies, such as Apache Flink/Spark or sentiment analysis on Twitter. He has collaborated with 38 other researchers, both nationally and internationally, and including both academia and industry. He has been part of several program committees and collaborated as reviewer for several international conferences and journals. More information is available at http://maude.sip.ucm.es/~adrian/.

Kazuhiro Ogata is a Professor at the School of Information Science - Japan Advanced Institute of Science and Technology (JAIST). His main research interests are verification, theorem proving, specification, rewriting, model checking, having collaborated in the analysis of several protocols using both CafeOBJ and Maude. He has being chair of several international conferences and principal investigator of several research projects focused on the specification and verification of systems. More information is available at http://www.jaist.ac.jp/~ogata/.