Chapter 19
On Combining Algebraic Specifications with First-Order Logic via Athena

Katerina Ksystra, Nikos Triantafyllou and Petros Stefaneas

Abstract We present a verification framework developed by researchers of the National Technical University of Athens as part of the Research Project Thalis “Algebraic Modeling of Topological and Computational Structures and Applications”. The proposed framework combines two different specification and theorem-proving systems, in order to facilitate the modeling and analysis of critical software systems. On the one hand, the CafeOBJ algebraic specification language offers executable, composable specifications, and insightful information about the proofs of desired invariant properties. On the other hand, Athena, an interactive theorem-proving system, provides automation and soundness guarantees for its results, as well as detailed structured proofs. Although having conducted complicated case studies (references to which are provided in the paper), here we focus on explaining the steps of the proposed hybrid methodology as clearly as possible, through an illustrative example of a simple mutual exclusion protocol.

Keywords Algebraic specifications · CafeOBJ
Observational transition systems · Proof scores · First-order logic · Athena

K. Ksystra (✉) · N. Triantafyllou · P. Stefaneas
Department of Applied Mathematics, National Technical University of Athens,
Zografou Campus, 15780 Athens, Greece
e-mail: katksy@central.ntua.gr

N. Triantafyllou
e-mail: nitriant@central.ntua.gr

P. Stefaneas
e-mail: petros@math.ntua.gr

S. Lambropoulou et al. (eds.), *Algebraic Modeling of Topological and Computational Structures and Applications*, Springer Proceedings in Mathematics & Statistics 219, https://doi.org/10.1007/978-3-319-68103-0_19

19.1 Introduction

Algebraic specification techniques have been developed in the area of formal methods and several algebraic specification languages and processors have been proposed. In algebraic specification methods, systems are specified based on algebraic modeling, and then the specifications are verified against requirements using algebraic techniques. Algebraic specification languages such as CafeOBJ [1], Maude [2] and CASL [3] have well-known advantages for modeling and reasoning about digital systems. The specifications are relatively simple, readable and writable, and can be executed and analyzed in various ways to provide valuable information to the modelers.

Sometimes, however, specification languages can be more effective when integrated with more conventional theorem-proving systems. CASL, for instance, has been interfaced with HOL/Isabelle [4], through HOL-CASL [5]. Also the Heterogeneous Tool Set (Hets) [6] has connections with the algebraic specification languages Maude and CASL and external theorem provers (SPASS, Vampire, Darwin, KRHyper and MathServe).

In this paper we present a methodology that combines the CafeOBJ algebraic specification language with the Athena theorem proving system [7], and a common interface that trasforms equational specifications written in CafeOBJ into Athena specifications. The aim of the proposed framework is to exploit both the nice properties of CafeOBJ specifications and the soundness and automation offered by Athena. This framework was developed by researchers of the National Technical University of Athens as part of the Research Project Thalis "Algebraic Modeling of Topological and Computational Structures and Applications".

In more details, CafeOBJ provides mechanized implementations of Observational Transition Systems (OTSs), a species of behavioral specifications, that allow users to specify distributed systems using multi-sorted conditional equational logic with subsorting [8]. The specifications are executable via rewriting, which is useful for building up computational intuitions about the underlying system. In addition, CafeOBJ allows users to compose proof scores that establish certain invariant properties, typically by induction. Athena on the other hand, is a system based on general polymorphic multi-sorted first-order logic. It integrates computation and deduction, allows for readable and highly structured proofs, guarantees the soundness of results that have been proved, and also has built-in mechanisms for general model-checking and theorem-proving, as well as seamless connections to state-of-the-art external systems for both.

By combining these two methodologies we wish to combine the strengths of CafeOBJ, most notably succinct, composable, executable specifications based on conditional equational logic with those of Athena, namely, structured and readable proofs, soundness of the results, and greater automation both for proof and for counterexample discovery. The goal of this paper is to give an overview of the theoretical foundations of our framework (Sect. 19.2) and to provide an easy to follow introduction to the proposed methodology (together with a tool that automates the transformation process from OTS/CafeOBJ into Athena specifications) using a simple and illustrative example (Sect. 19.3). Finally, we discuss related work and advantages of the proposed framework in Sect. 19.4 and future directions in Sect. 19.5.

We should mention here that the proposed methodology has been successfully applied to much larger and complicated use cases as well. To this end we encourage the interested reader to visit [9] for the application of the proposed methodology to the verification of the Alternating Bit Protocol, which is considered the traditional benchmark used for testing mechanical verification of protocols [10].

19.2 Theoretical Background

19.2.1 CafeOBJ Algebraic Specification Language

CafeOBJ [1] is an algebraic specification language and processor that can be used to specify abstract data types and abstract state machines. The basic units of a CafeOBJ specification are its modules. There are two kinds of modules in CafeOBJ, tight and loose modules. A tight module only accepts the smallest implementation that satisfies what are specified in the module, while a loose module can accept any implementations (that satisfy them). A tight module is declared with the keyword $mod!$, and a loose module with the keyword $mod*$.

In CafeOBJ modules, we can declare module imports, sorts, operators, variables and equations. Operators without arguments are called constants. Built-in operators denoting logical connectives can be used to declare negation, conjunction, disjunction, implication, and exclusive disjunction. Operators can have attributes such as *comm* that specifies that the binary operator is commutative. Operators are declared with the keyword *op* (or *ops* if there are many). The constructor operators of the sorts are declared with the attribute *constr*. The non-constructor operators, or some properties of the operators are defined in equations. Conditional equations can also be declared inside a module, using the keyword *ceq*.

19.2.2 Observational Transition Systems (OTSs)

An Observational Transition System (OTS) is a transition system written in terms of equations and is a proper subclass of behavioral specifications [11]. Assuming that there exists a universal state space Y and that each datatype we need to use has been declared in advance, an OTS S is defined as the triplet $S = \langle O, I, T \rangle$ where [12]:

- O is a set of observers. Each $o \in O$ is a function $o : Y D_{o1} \ldots D_{om} \rightarrow Do$, that takes as input a state of the system and maybe other datatypes (not necessarily in that order) and return a datatype value. Given an OTS S and two states u_1, u_2 the equivalence between them is defined with respect to the values returned by the observers, i.e. $u_1 =_S u_2$ if and only if for each $o \in O$, $o(u_1, x_1, \ldots, x_m) = o(u_2, x_1, \ldots, x_m)$ for all $x_1 \in Do_1, \ldots, x_m \in Do_m$.

- I is the set of initial states, such that $I \subseteq Y$.
- T is a set of conditional transitions. Each $t \in T$ is a function $t : Y\ D_{t1} \ldots D_{tn} \rightarrow Y$. Each transition t, together with any other parameters, preserves the equivalence between two states, i.e. if $u_1 =_S u_2$, then for each $t \in T$, $t(u_1, y_1, \ldots, y_n) =_S t(u_2, y_1, \ldots, y_n)$ for all $y_1 \in D_{t1}, \ldots, y_n \in D_{tn}$. Each t has an effective condition $c\text{-}t : Y\ D_{t1} \ldots D_{tn} \rightarrow Bool$. If $\sim c\text{-}t(u, y_1, \ldots, y_n)$, then $t(u, y_1, \ldots, y_n) =_S u$. $t(u, y_1, \ldots, y_n)$ is called a successor state of a state u. We write $u \leadsto_S u'$ iff a state u' ∈Y is a successor state of a state $u \in Y$.

An *execution* of an OTS S is an infinite sequence $u_0, u_1, \ldots$ of states satisfying:

– *Initiation*: $u_0 \in I$ and
– *Consecution*: For each $i \in N$, there exists $t \in T$ such that $u_{i+1} =_S t(u_i)$.

Let ϵ_s be the set of all executions obtained from S. A state $u \in Y$ appears in an execution $u_0, u_1, \ldots$ of an OTS S, denoted by $u \in u_0, u_1, \ldots$ if there exists $i \in N$ such that $u =_S u_i$. A state $u \in Y$ is called reachable with respect to an OTS S, if and only if there exists an execution $e \in \epsilon_s$ such that $u \in e$. Let R_S be the set of all reachable states with respect to S.[1]

Roughly speaking, in the OTS/CafeOBJ method the *transitions* between the states of the system are modelled with constructor operators. The structure of a state is abstracted by the *observers*, each one returning some observable information about the state. The meaning of an observer is formally described by means of (conditional) equations [14].

19.2.3 Proof Scores in CafeOBJ

CafeOBJ is equipped with its processor called the CafeOBJ system, which is used as an interactive theorem prover. The CafeOBJ system verifies the desired properties by using the equations of the theory that defines an OTS as left to right re-write rules. This method is called proof score approach and is computer-human interactive [15]. A proof score is a plan to verify that a property holds for a specification. This is implemented as a set of instructions written by a human to a proof engine, such that when executed, and if everything evaluates as expected, a desired theorem is proved.

In order to prove an invariant property using CafeOBJ the following steps need to be taken. First, we formally express the property we want to prove as a predicate in CafeOBJ terms in a module. Next, we write the inductive step as a predicate that defines that if the invariant holds in an arbitrary state s then that implies that it holds in its successor state s'. Then we ask CafeOBJ to prove via term rewriting (using the

[1] R_S is the type denoting the set of all reachable states wrt S. Also Sys denotes R_S but not Y if the constructor-based logic is adopted, which is the current logic underlying the OTS/CafeOBJ method [13].

reduce command), if the property holds for an arbitrary initial state. Finally, using all the transition rules in turn, we must instantiate s' and ask CafeOBJ to prove the inductive step for each case.

After asking CafeOBJ to prove such an expression three results might be returned by the system. If true is returned this means that the proof is successful. If a CafeOBJ term is returned, this means that there exist some terms that the system cannot fully reduce. The user must then split the case, by stating that the returned term equals to true and false in turn (computer-human interactive method). This creates two new proof obligations and is known as case splitting. Finally, if false is returned, either the property does not hold, or the case that returned false is unreachable for our system.

19.2.4 Athena

Athena [7] is an interactive theorem proving environment for polymorphic multi-sorted first-order logic, with separate (but intertwined) languages for computation and deduction. The main tool for constructing proofs is the method call. A method call represents an inference step and can be primitive or complex. Methods can accept as arguments other methods and/or procedures, and thus are higher-order.

Athena proofs are machine-checkable and expressed in a true natural-deduction style, a style of proof that was explicitly designed to capture the essential aspects of mathematical reasoning as it has been practiced for thousands of years [16], whose semantics are formalized in terms of *assumption bases*. An assumption base is a set of sentences serving as working hypotheses—typically axioms and previously derived theorems. Initially the system starts with a small assumption base, containing defining axioms for some built-in function symbols. When an axiom is postulated or a theorem is proved, the corresponding sentence is inserted into the assumption base. During the evaluation of a proof, methods interact with the assumption base, checking to see if some arguments are present in the set and/or making new entries [17].

Another important point here is that Athena guarantees the soundness of every defined method, meaning that if and when a method call manages to derive a conclusion p, we can be assured that p is a logical consequence of the assumption base in which the call occurred. This guarantee stems from the formal semantics of Athena. The larger point to keep in mind is that defined methods can never produce a result that is not logically entailed by the assumption base [16].

Additionally, Athena is integrated with model builders. Model generation is useful for consistency checking, and in particular for falsifying conjectures: If we are not sure whether a formula follows from a given set of premises, we can try to find a counter-model, i.e., a model in which all the premises are true but the formula in question is false. Model generators can be used to find such models automatically. If we manage to find a counterexample in Athena, then any attempt to prove the property would be pointless and therefore model checking can be quite a time saver [16].

Athena provides also a significant degree of proof automation, through seamless interfaces to powerful automated theorem provers like Vampire [18] and SPASS [19], as well as SAT and SMT solvers.

19.3 Proposed Framework: Combining CafeOBJ and Athena Environments

The OTS/CafeOBJ approach presents many advantages, the most important of which, in our opinion and experience, is that the proof scores can easily and effectively guide the user to discover the required case splittings and occasional lemmas for the proof. However, we believe that it could benefit by being combined with proof systems like Athena, for the following reasons.

At first, each proof conducted in Athena is checked for soundness by the system. Thus, Athena could act as a validator for the proofs conducted in CafeOBJ. Also, Athena as we have already mentioned is integrated with some powerful automated theorem provers (ATPs). Thus, via Athena, access to these highly efficient ATPs can be enabled for OTS/CafeOBJ specifications as well, which could help to streamline the verification process. Finally, Athena uses a Fitch-style natural-deduction proof system. Expressing OTS/CafeOBJ in a high-level and structured natural-deduction style such as Athena's could help to make them comprehensible and accessible to a wider audience.

Accordingly, we propose a methodology which combines both environments (CafeOBJ and Athena) and consists of the following steps:

- Step 1: Create the OTS specification in CafeOBJ.
- Step 2: Generate an equivalent Athena specification, automatically.
- Step 3: Attempt to falsify the property of interest in Athena. If unsuccessful proceed to the next step, else either the property or the specification should be revised.
- Step 4: Apply the proof score methodology in CafeOBJ until a lemma is required. Discover a candidate lemma in CafeOBJ that can discard the problematic case.
- Step 5: Attempt to falsify the candidate lemma using Athena. If unsuccessful proceed to the next step, else either the property or the specification should be revised.
- Step 6: Iterate steps 4 to 5 until the proof scores methodology is completed for the property in question and also for all the lemmas used.
- Step 7: Using the insights gained by the proof scores (case splittings and lemmas) generate an Athena proof and check its soundness.

With the proposed methodology (steps 1 to 3), the user could save considerable time by first attempting to falsify the property in question. If indeed a counter example is returned by Athena, then either the property in question is not invariant for the specification, or there might be a bug in the specification itself. In the first case, the proof is completed and the property falsified. In the second case the output of Athena usually provides sufficient information for the discovery and correction of the bug.

During the verification of complex systems with the proof score methodology, it is possible that the user will consider as lemmas, properties which while successfully discard the problematic cases, are not invariant for the specification. This is usually discovered at a late stage of the verification of the lemmas, in which case the proof needs to be recreated using a different candidate lemma. This, can at times become an important time sink for the verification effort. Athena, could potentially inform the user of the error in the candidate lemmas at a much earlier stage (steps 4 to 5) and thus save the user valuable time.

Also once the proof score methodology is completed, by transferring the insights gained by it to Athena we can easily create a formal proof for the property in question. Athena can thus inform us about the soundness of the proof or point to reasoning errors.

Finally, as we mentioned earlier the final proof constructed in Athena will be in a style easy to understand and thus could help provide explanation as to why the property is invariant or not for the system, in other words act as a sort of software documentation.

19.3.1 Rules of Translation

In order to obtain an Athena specification from an OTS/CafeOBJ specification we have defined an appropriate translation schema. The basic units of OTS/CafeOBJ specifications and their transformation into Athena notation are shown in Table 19.1.

In more details, in Athena initial algebras (with tight semantics) are specified using the keyword *datatype*, whereas an arbitrary carrier (with loose semantics) is introduced with the *domain* keyword. An induction principle is automatically generated for every new datatype. States are formalized as a *structure* and state transitions as its constructors. Structures are very much like datatypes, except that there may be confusion, i.e. different constructor terms might denote one and the same object. An induction principle is also automatically generated (and, of course, valid). We continue with the declaration of the observers. They are defined as functions with the constraint that they take as input a state (and maybe additional input) and return

Table 19.1 The basic units of OTS/CafeOBJ specifications and their transformation into Athena notation

OTS/CafeOBJ notation	Athena notation
Tight modules	Datatypes
Loose modules	Domains
State	Structure
Initial state	State constructor
State transitions	State constructors
Observers	Functions
Equations	Axioms

some datatype. Finally, the equations that define the initial state, as well as the pre- and post-conditions of the state transitions are defined as axioms.

Informally, an arbitrary OTS specification written in CafeOBJ terms is translated into an Athena specification through the operator `cafe2athena` as follows:

Datatype modules:

```
cafe2athena(mod! M1 {[m1] ...})
=
datatype m1 cafe2athena(...)

cafe2athena(mod* M2 {[m2] ...})
=
domain m2 cafe2athena(...)
```

OTS modules (sort representing state space, initial state, transitions):

```
cafe2athena(mod S {*[Sys]* op init : → Sys .
bop a : Sys Vj1 ... Vjn → Sys ...})
=
structure Sys := init | (a Sys Vj1 ... Vjn)
```

Observers:

```
cafe2athena(bop o : Sys Vi1 ... Vin → V)
=
declare o := [Sys Vi1 ... Vin] → V
```

Variables:

```
cafe2athena(Var xi1 : Vi1 ... Var xin : Vin)
=
define [xi1 ... xin] := [?Vi1 ... ?Vin]
```

Init axiom:

```
cafe2athena(eq o(init, xi1, ..., xin) = f(xi1, ..., xin))
=
assert* init-axiom := ((o init xi1 ... xin) = f xi1 ... xin)
```

where x_{i1},...,x_{in} are V_{i1} ...V_{in} sorted CafeOBJ variables and $f(x_{i1}, ..., x_{in})$ is the value of the observer at the initial state.

Effective condition:

```
cafe2athena(op c-a : Sys Vj1 ... Vjn → Bool .
eq c-a(w, xj1, ..., xjn) = g(w, xj1, ..., xjn) .)
=
define c-a := lambda (w xj1 ... xjn) (g w xj1 ... xjn)
```

where w is a hidden sorted variable and x_{j1}, ..., x_{jn} are V_{j1}...V_{jn} sorted CafeOBJ variables.

Transition axioms:

```
cafe2athena(eq $o(a(w, x_{j1}, ..., x_{jn}), x_{i1}, ..., x_{in})$ = $e$-$a(w, x_{j1}, ..., x_{jn}, x_{i1}, ..., x_{in})$
     if $c$-$a$($w, x_{j1}, ..., x_{jn}$) .)
=
assert a-axiom :=
(($o$($a$ $w$ $x_{j1}$ ...$x_{jn}$) $x_{i1}$ ... $x_{in}$) = (e-a $w$ $x_{j1}$ ... $x_{jn}$ $x_{i1}$ ... $x_{in}$)
     if ($c$-$a$ $w$ $x_{j1}$ ... $x_{jn}$))
```

where e-$a(w, x_{j1}, ..., x_{jn}, x_{i1}, ..., x_{in})$ is the changed value of the observer after the application of the transition. The definition of a proof score in Athena terms can be found in Appendix 5

19.3.1.1 Semantic Correctness of the Translation.

Some readers may have noticed that we restricted the translation to non-parametric modules. The parametric module system does not add to the expressiveness of the language, however not supporting it may result in an overhead in specification code. By enforcing this restriction on the style of the OTS/CafeOBJ specifications it is safe to claim the semantic correctness of the translation, since in both languages the underlying semantics is basically Order-sorted Conditional Equational Logic (in which constructors are explicitly used). For details about the semantic correctness of the translation we refer readers to [9].

19.3.1.2 Cafe2Athena Tool.

In order to make the proposed methodology more agile, we have developed a tool that takes as input an OTS/CafeOBJ specification and automatically produces an Athena specification, implementing the rules of translation we previously presented. The tool is written in Java and hides the details of the translation, since the user loads a CafeOBJ specification file, presses the "Translate to Athena" button and gets the corresponding Athena specification. Cafe2Athena tool was a deliverable of the Research Program Thalis "Algebraic modeling of topological and computational structures" and can be downloaded from [20].

19.3.2 Illustrating Example: A Mutual Exclusion Protocol Using an Atomic Instruction (Mutex)

We present in this section the application of the proposed methodology to a mutex algorithm so as to explain it better and demonstrate its effectiveness. In this system, we have a set of processes, each of which is executing code. A process, at any system state, is either in some critical section of the code or in some remainder (waiting) section. Also, we have two transitions; the first corresponds to a process entering the critical section and the second to a process exiting the critical section (and entering the remainder section).

When a process p enters its critical section, the resulting state becomes *locked*. When p exits the critical section, the resulting state is unlocked. For p to enter its critical section in some state s, p must be *enabled* in s. A process p is *enabled* in s iff p is in its remainder section in s and s is not locked. This is, therefore, the effective condition of the *enter* state transition for a given process. The effective condition of the *exit* transition is for the process to be in its critical section. We also have two observer functions, one that takes a state s and a process id p and tells us what section of the code p is executing in s (critical or remainder), and a function that takes a state s and tells us whether s is locked.

19.3.2.1 Step 1. Specification in CafeOBJ.

The specification of the mutex OTS in CafeOBJ can be seen in Appendix 5

19.3.2.2 Step 2. Specification in Athena.

Using the ‘Cafe2Athena’ tool we obtain the specification of the mutex OTS in Athena. Some parts of the specification are explained below.

A domain of process identifiers (`Pid`) and a datatype for code labels (`cs` and `rs`, for critical and remainder section, respectively) have been introduced:

```
domain Pid
datatype Label := rs | cs
assert label-axioms := (datatype-axioms"Label")
```

Here, `rs` and `cs` are the (nullary) *constructors* of the `Label` algebra. The `datatype-axioms` of `Label` are quantified sentences that assert no-confusion and no-junk conditions for the constructors. The effect of the `assert` command is to insert those conditions into the global *assumption base*. States are formalized as a structure and the state transitions as constructors of this structure, as follows:

```
structure State := init | (enter Pid State)
| (exit Pid State)
```

The declaration of the observer functions can be seen below:

```
declare at: [Pid State] -> Label
declare locked: [State] -> Boolean
```

The “*at*” function tells us the label of a given process in a given state. Binary function symbols can be used in infix in Athena (the default notation is prefix), so if *p* and *s* are terms of sort *Pid* and *State*, respectively, then (*p ats*) gives us the label of *p* in state *s*. The term *(lockeds)* tells us whether or not *s* is locked.

We now present the axioms that define the initial state, as well as the pre- and post-conditions of the two state transitions (entering and exiting). First, appropriate variables for the given sorts are defined.

```
define [i j s s'] := [?i:Pid ?j:Pid ?s s':State]
assert* init-axioms := [(_ at init = rs)
                        (~ locked init)]
```

The initial-state axioms are simple enough: every process in the initial state is in the remainder section, and the initial state is not locked. For the transition axioms of *enter* we see it is helpful to define the effective condition as a separate procedure, called `enabled-at`:

```
define (enabled-at i s) := (s at i = rs & ~ locked s)
```

The axioms for the *enter* and *exit* transitions, respectively, are presented below:

```
assert* enter-axioms :=
 [(i enabled-at s ==> locked i enter s)
  (i enabled-at s ==> i at i enter s = cs)
  (i enabled-at s & j =/= i ==> j at i enter s = j at s)
  (~ i enabled-at s ==> i enter s = s)]

assert* exit-axioms :=
 [(i at s = cs ==> i at i exit s = cs)
  (i at s = cs & j =/= i ==> j at i enter s = j at s)
  (i at s = cs ==> ~ locked i exit s)
  (i at s =/= cs ==> i exit s = s)]
```

19.3.2.3 Step 3. Define the Desired Goal and Falsify It with Athena.

The desired mutual exclusion property satisfied by the algorithm, is that there is at most one process in the critical section at any given moment. This property can be rephrased as *"if there are two processes in the critical section, then those processes are identical"*. In Athena, the desired goal is defined as follows:

```
define (goal-property s) :=
    (forall i j . i at s = cs & j at s = cs ==> i = j)
define goal := (forall s . goal-property s)
```

Before we try to prove a conjecture *p*, it is often useful to first try to falsify it in Athena by finding a counterexample to it. This falls under a class of techniques collectively known as model checking, or model building. If we manage to find a counterexample, then any attempt to prove *p* would be useless (because Athena's proofs are guaranteed to be sound, so we can never prove something that doesnt follow logically from the assumption base, and if we find a counterexample to p then p does not follow from the assumption base), and therefore model checking can be quite a time saver. Moreover, the feedback provided by the model checker, given in the form of a specific counterexample to the conjecture, is often very valuable in helping us to debug and, in general, to better understand the theory or system we are developing [16].

Athena is integrated with a number of external systems that can be used for model building/checking, most notably SMT and SAT solvers, but there is also a built-in model checker which is perhaps the simplest to use and can be surprisingly effective [16].

To falsify a conjecture p, we simply call (`falsify` p N). Here N is the desired *quantifier bound*, namely, the number of values of the corresponding sort that we wish to examine (in connection with the truth value of p) at each quantifier of p. When falsification fails within the given bound, the term *'failure* is returned. When falsification succeeds, it returns specific values for the quantified variables that make the conjecture false. Let us see in our example, if we can falsify the goal by examining 100 states:

```
> (falsify  goal 100)
List: ['success
|{
?i:Pid  := Pid_1
?j:Pid  := Pid_2
?s:State := (enter Pid_2
       (exit Pid_1
             (enter Pid_1 init)))
}|]
```

As we can see, Athena falsified the property in question and returned a state in which the goal is violated. This state can be reached after the application of the transitions *enter*, *exit* and *enter* in the initial state of our system, as Athena informed us `(enter p2 exit p1 enter p1 init).`

In order to understand better why the property is violated in a particular state we have written a procedure, `simulate`, that takes as input a sequence of states $s_1, \ldots, s_n$, and prints out each state in the sequence by applying all observer functions to the given state, and specifically by evaluating the applications of those observer functions (by using the relevant axioms as rewrite rules).

Since we know which transitions cause the falsification of the goal, we can use the simulate method to see the value of the observers in this problematic state. In our case, calling `simulate (p2 enter p1 exit p1 enter init)`, results in the following output:

```
State after (p1 enter init):
locked:                                    p1 at:
------------                               --------------------
true                                         cs

State after (p1 exit (p1 enter init)):
locked:                                    p1 at:
------------                               --------------------
false                                        cs

State after (p2 enter (p1 exit (p1 enter init))):
locked:                                    p2 at:
------------                               --------------------
true                                         cs
```

If we observe the returned values in this state we see that while $p2$ exits the critical section it basically remains in the critical section. Thus we understand that there must exist an error in the definition of the exit axioms (in particular, the first axiom was mis-written as `(i at s = cs ==> i at i exit s = cs)` instead of `(i at s = cs ==> i at i exit s = rs)`). After redefining the axiom, we falsify again the desired goal, and Athena returns the term *'failure*. Thus we proceed to the next step of our methodology.

19.3.2.4 Step 4. Start the Proof of the Desired Goal Using Proof Scores Approach (until a Lemma Is Needed).

We start to work with proof scores in CafeOBJ, until we reach the following case where CafeOBJ returns false:

```
open ISTEP .
-- arbitrary values
op k : -> Pid .
-- assumptions
-- eq c-enter(s,k) = true .
eq at(s,k) = rs .
eq locked(s) = false .
eq i = k .
eq (j = k) = false .
eq at(s,j) = cs .
-- successor state
eq s' = enter(s,k) .
red istep1 .
close
```

This means that in order to discard this case we must use a lemma. By taking a close look at the equations that define this state we can observe that `eq at(s,j) = cs` and `eq locked(s) = false` cannot hold together. This means that a possible lemma could be the following: `inv2(S,J) = (at(S,J) = cs implies locked(S) = true)`. To test if this lemma actually discards the problematic case we use the above proof score with the reduction `red inv2(s,j) implies istep1` and CafeOBJ returns true.

19.3.2.5 Step 5. Falsify the Discovered Lemma with Athena.

After defining the lemma in Athena, we try to falsify it and Athena returns the term *'failure*, so we continue the proof score using this lemma.

19.3.2.6 Step 6. Continue the Proof Using Proof Scores in CafeOBJ.

Checking the rest of the cases in CafeOBJ will result in the completion of the proof score. It is interesting to point out that in our example, during the proof score of the lemma the original property under verification was required as part of the inductive hypothesis. This case is shown below.

```
open ISTEP .
-- arbitrary values
op k : -> Pid .
-- assumptions
eq at(s,k) = cs .
eq (j = k) = false .
eq at(s,j) = cs .
-- successor state
eq s' = exit(s,k) .
red inv1(s,j,k) implies istep2 .
close
```

19.3.2.7 Step 7. Create an Athena Proof Based on the Gained Insights.

The above pattern in a proof score denotes a situation where simultaneous induction must performed. Together with the case splits and lemmas used, such information is essential to the construction of the Athena proof. Also, by taking a closer look at the invariant and the lemma used ((`i at s = cs and j at s = cs ==> i = j` and `j at s = cs ==> locked s`, respectively) it is not difficult to understand that a strengthened goal can be formulated out of them: `i at s = cs & j at s = cs ==> i = j & locked s`, which we will attempt to verify in Athena. The new goal is defined as follows:

```
define (new-goal-property s) :=
(forall i j . i at s = cs & j at s = cs ==> i = j & locked s)
define new-goal := (forall s . new-goal-property s)
```

A method for completely automated inductive reasoning is automatically defined whenever a new structure or datatype is introduced, as we have already mentioned. The name of the method is the name of the corresponding structure joined with the string `-induction`, in lower case. If we try to prove the new goal using the `state-induction` we get the following output, which means that the goal is automatically proved.

```
> (!state-induction new-goal)
Theorem: (forall ?s:State
             (forall ?i:Pid
               (forall ?j:Pid
                 (if (and (= (at ?i:Pid ?s:State)
                             cs)
                          (= (at ?j:Pid ?s:State)
                             cs))
                     (and (= ?i:Pid ?j:Pid)
                          (locked ?s:State))))))
```

However, a completely automatic proof does not shed much light on a system's workings. A structured proof that derives the desired result in a piecemeal fashion can be much more valuable in *explaining* the underlying system, i.e., in explaining why a given property holds. In addition, the effort invested in constructing the proof often pays off in increased understanding, and also in the discovery of errors, unintended consequences of design constraints, and so on.

Athena uses a natural-deduction style of proof, as we have already mentioned, which allows for structured and readable proofs that resemble in certain key respects the informal proofs one encounters in practice [16]. Part of a detailed structured Athena proof of the goal property for our example is shown below.[2]

```
by-induction (forall s . new-goal s) {
  init => (!spf (new-goal init) (ab))
| (s' as (k enter s)) =>
    pick-any i:Pid j:Pid
     let {IH := (forall i j . i at s = cs & j at s = cs
                      ==> i = j & locked s)}
     assume hyp := (i at s' = cs & j at s' = cs)
      conclude goal := (i = j & locked s')
       (!two-cases
        assume case1 := (k enabled-at s)
         let {s'-locked := (!chain-> [case1
                              ==> (locked s') [enter-axioms]]);
              s-not-locked := (!chain-> [case1
                              ==> (~ locked s) [right-and]]);
              i=j := (!by-contradiction (i = j)
                      assume h := (i =/= j)
                   let {D := {(i =/= k | j =/= k) from h}}
                       (!cases D
                         assume i=/=k := (i =/= k)
                          (!M i=/=k case1 IH)
                         assume j=/=k := (j =/= k)
                          (!M j=/=k case1 IH)))}
```

[2]For the full proof we refer readers to Appendix 5.

```
        (!both i=j s'-locked)
      assume case2 := (~ k enabled-at s)
        (!direct-ih hyp s case2 IH enter-axioms))
...
}
```

In this way, Athena takes us one step closer to the goal of enabling proofs that can serve to explain and communicate our reasoning, but which are nevertheless entirely formal and checkable by machine [16].

19.4 Discussion

Some approaches that deal with the integration of algebraic specifications with more conventional proving systems are briefly presented. A methodology that offers parsing, static analysis and proof management is the tool Hets - the Heterogeneous Tool Set [6]. Hets has among others, connections with the algebraic specification languages Maude and CASL and external theorem provers and is based on a graph of logics and logic translations. Another interesting methodology for proving inductive properties of OTSs that aims at automating the proof scores approach to verification, can be found in [14]. In this paper, authors revise the entailment system of proof scores and enrich it with proof rules and tactics. Also, a prototype tool (Constructor-based Inductive Theorem Prover - CITP) implementing the methodology is demonstrated. CITP is implemented in Maude.

This last approach is the closest to ours but with a different point of view. While both methodologies aim at providing soundness to the OTS/CafeOBJ method, CITP focuses on the automation of the proofs whereas we are more concerned with combining the benefits of CafeOBJ and Athena specification and verification methodologies into one. For example, in [14] authors discharge automatically a desired property and the required lemmas but they do not describe how the lemmas are formulated. Also they do not support detailed proofs or simulation of the system's behavior. Thus, we believe that our approach offers better explanation and understanding of the verification of the desired properties and the behavior of the specified system.

In the following we summarize the benefits of the proposed methodology; One advantage is that the proof scores approach can provide feedback to the user when a proof fails, and can be used to discover the required case splittings and occasional lemmas for the proof, in contrast with most automated theorem provers including Athena. Another advantage is that you can use the model-checking tools of Athena to obtain some first insights about the specified system and thus save valuable time. Also, the simulate procedure, can become really helpful in understanding how the specified system behaves. Especially when you deal with a complex system where it is almost impossible to "follow" its execution process, such visualization techniques can provide a clear overview of the system and help in the discovery of possible errors. For example in [21] authors state that among the lessons learned from a non trivial, real life protocol case study of the OTS/CafeOBJ method, is the delay arose from errors in the specification and the expression of system's property. These errors would be found easily with the proposed integration. In addition, the structured proofs supported by Athena can provide valuable information and explanation as to why a property is invariant or not for the specified system. Another important advantage of our approach is that Athena does a thorough check of the overall proof and provides a guarantee that if and when you get a theorem, that result follows logically from the assumption base and the primitive methods (which in this case include the external ATP). This is really helpful because on the other hand, with CafeOBJ's proof scores approach there is much greater room for human oversight. Finally, the automation offered by Athena through its connection with external systems is another advantage of the proposed framework.

19.5 Conclusion

We proposed a hybrid methodology that combines the proof scores approach of CafeOBJ with Athena's reasoning and verification tools, together with a tool that translates equational CafeOBJ specifications into Athena code, and demonstrated our approach with a mutex algorithm. Also we presented several features of the methodology that can be used to: better understand the specified system, model-check desired properties and verify them via theorem proving, both automatically and in a structured and more detailed way. The proposed method aims at combining the strengths of the languages, CafeOBJ and Athena, by working with proof scores and CafeOBJ but also taking advantage of more conventional formal-methods techniques that have traditionally lied outside of the rewriting community.

This methodology has been applied in larger case studies as well [9] and as future work we plan to push the automation level further for complex systems and to investigate possible connections with tools incorporating various provers and different specification languages. Also, we plan to extend the proposed framework in order to include the behavioral aspects of CafeOBJ as well.

Acknowledgements This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALIS. The authors would like to acknowledge the insightful feedback provided by Dr. Konstantine Arkoudas. The authors would also like to warmly thank Prof. Dr. Sofia Lambropoulou, Project Coordinator of the Research Project Thalis "Algebraic Modeling of Topological and Computational Structures and Applications" for their excellent collaboration.

Appendix A

Here we present the CafeOBJ specification of the mutex system.

```
mod* PID {
  [Pid]
  op _=_ : Pid Pid -> Bool {comm}
  var I : Pid
  eq (I = I) = true .}
mod! LABEL {
  [Label]
  ops rs cs : -> Label
  op _=_ : Label Label -> Bool {comm}
  var L : Label
  eq (L = L) = true .
  eq (rs = cs) = false .}
mod* MUTEX {
  pr(LABEL + PID)
  *[Sys]*
  -- an arbitrary initial state
  op init : -> Sys
  -- observation functions
  bop at : Sys Pid -> Label
  bop locked : Sys -> Bool
  -- transition functions
  bop enter : Sys Pid -> Sys
  bop exit : Sys Pid -> Sys
  -- CafeOBJ variables
  var S : Sys
  vars I J : Pid
  -- init
  eq at(init,I) = rs .
  eq locked(init) = false .
  -- enter
  op c-enter : Sys Pid -> Bool
  eq c-enter(S,I) = ((at(S,I) = rs) and not locked(S)) .
  ceq at(enter(S,I),J) = cs if (I = J) and c-enter(S,I) .
  ceq at(enter(S,I),J) = at(S,J) if not((I = J) and c-enter(S,I)) .
  ceq locked(enter(S,I)) = true if c-enter(S,I) .
  ceq enter(S,I) = S if not c-enter(S,I) .
  -- exit
  op c-exit : Sys Pid -> Bool
  eq c-exit(S,I) = (at(S,I) = cs) .
  ceq at(exit(S,I),J) = rs if (I = J) and c-exit(S,I) .
  ceq at(exit(S,I),J) = at(S,J) if not((I = J) and c-exit(S,I)) .
  ceq at(exit(S,I),J) = at(S,J) if not(I = J) and not c-exit(S,I) .
  ceq locked(exit(S,I)) = false if c-exit(S,I) .
  ceq exit(S,I) = S if not c-exit(S,I) .}
```

Appendix B

The declaration of an invariant property in CafeOBJ terms and the definition of the induction schema, are shown below:

```
-- declaration of the invariant property
mod INV {
-- arbitrary values
op s : -> Sys .
ops i j : -> Pid .
-- name of invariant to prove
op inv1 : Sys Pid Pid -> Bool
-- CafeOBJ variables
var S : Sys
vars I J : Pid
-- invariant to prove
eq inv1(S,I,J) = (at(S,I) = cs and at(S,J) = cs implies I = J) .
}
-- declaration of the inductive step
mod ISTEP {
pr(INV)
-- arbitrary values
 op s' : -> Sys
-- name of formula to prove in each inducion case
op istep1 : -> Bool
-- formula to prove in each induction case
eq istep1 = inv1(s,i,j) implies inv1(s',i,j) .
}
```

The definition of the corresponding invariant in Athena is presented here (the induction schema is automatically defined in Athena).

```
define (inv1 s) := (forall ?i ?j . at s ?i = cs & at s ?j = cs
==> ?i = ?j)
```

The proof score of the desired invariant property in CafeOBJ, for the initial state and when a transition, called `enter(s,k)`, is applied can be seen below:

```
-- proof score
-- I. Base case
open INV .
  red inv1(init,i,j) .
close .

-- II. Induction case
-- 1. enter(s,k)
open ISTEP  .
-- arbitrary values
  op k : -> Pid .
-- successor state
  eq s' = enter(s,k) .
-- check
  red istep1 .
close
```

The corresponding proof skeleton in Athena can be defined as follows.

```
by-induction (forall ?s . goal-property ?s) {
    init => (!prove (goal-property init))
    | (state as (enter s k)) =>
    pick-any i:Pid j:Pid
        assume hyp := (state at i = cs & state at j = cs) {
                goal := (i = j);
            goal from (ab)
        }
    }
```

Finally, the following proof scores present a case splitting in CafeOBJ. In the first case we assume that `at(s,k) = cs` while the second proof score assumes its symmetrical case, i.e. `(at(s,k) = cs) = false`.

```
open ISTEP  .
-- arbitrary values
  op k : -> Pid .
-- case 1
eq at(s,k) = cs .
-- successor state
  eq s' = enter(s,k) .
-- check
  red istep1 .
close

open ISTEP  .
-- arbitrary values
  op k : -> Pid .
-- case 2
eq (at(s,k) = cs) = false .
-- successor state
  eq s' = enter(s,k) .
-- check
  red istep1 .
close
```

The same case splitting can be defined in Athena terms as follows:

```
(! two-cases
assume case1 := (k at s = cs)
...
assume case2 := (k at s =/= cs))
```

Appendix C

A detailed structured Athena proof of the (strengthened) goal for our example is shown below.

Theorem 19.1 *For all states s' and processes i and j, if i and j are in their critical sections in s', then $i = j$ and s' is locked.* ■

Proof By structural induction on s'. When s' is the initial state the result is trivial because the antecedent is false, as all processes are in their remainder sections initially. Suppose now that s' is of the form (k *enter* s). Pick any processes i and j and assume both are in their critical sections in s'. We then need to show that $i = j$ and that $s' =$ (k*enter*s) is locked. The inductive hypothesis here is:

$$i\ at\ s = cs\ \&\ j\ at\ s\ = cs ==> i = j\ \&\ locked\ s \tag{19.1}$$

We distinguish two cases:

1. *Case 1*: k is enabled at s. Then (k *ats'* = *cs)*and *(lockeds')* follow from the *enter* axioms. Thus, we only need to show $i = j$. By contradiction, suppose that $i \neq j$. Then either $i \neq k$ or $j \neq k$. [3] So assume first that $i \neq k$ (the reasoning for the case $j \neq k$ is symmetric). Then, from the *enter* axioms and the assumption that k is enabled at s, we conclude i *ats'* = i *ats*, hence i *at s* = *cs*. Now applying the inductive hypothesis to the above assumption, we conclude *(locked s*). However, that contradicts the assumption that k is enabled at s, as that assumption means that s is *not* locked.
2. *Case 2*: k is not enabled at s. In that case, by the *enter* axioms, we get

$$(k\ enter\ s = s)$$

i.e., $s' = s$, and the result now follows directly from the inductive hypothesis.

Finally, suppose that s' is of the form (k *exit* s). Again pick any processes i and j and assume both are in their critical sections in s'. We again need to show that $i = j$ and that $s' =$ (k*exit*s) is locked. The inductive hypothesis here is the same as before, (19.1). We distinguish two cases again, depending on whether or not the effective condition of the *exit* transition holds:

1. *Case 1*: (k *at s* = cs). We proceed by contradiction. First, by applying the inductive hypothesis to the conjunction of (k *at s = cs)* with itself, we obtain *(locked s*). Also, by the *exit* axioms, we get

$$k\ at\ s' = (k\ exit\ s) = rs$$

i.e.,

$$k\ at\ s' = rs. \tag{19.2}$$

The *exit* axioms also imply that s' is *not* locked. We can now conclude that

$$i \neq k \tag{19.3}$$

[3]Clearly, if neither of these hold, i.e., if $i = k$ and $j = k$, then we could also have $i = j$, contradicting our hypothesis.

because otherwise, if $i = k$, the assumption that i is in cs in state s' would contradict (19.2). Hence, by the *exit* axioms, we get

$$i\ at\ s' = i\ at\ s. \tag{19.4}$$

Therefore, from (19.4) and the assumption that i is in cs in s', we get $i\ at\ s\ = cs$. But now applying the inductive hypothesis to $i\ at\ s = cs$ and to $(k\ at\ s = cs)$ yields $i = k$, contradicting (19.3).

2. *Case 2*: ($k\ at\ s \neq$ cs). In that case the *exit* axioms give ($k\ exit\ s = s$, i.e., $s' = s$, and the result follows directly from the inductive hypothesis.

The above informal proof can be formulated in Athena at the same level of abstraction and with the exact same structure. Moreover, the proof colloquialism "the reasoning for that case is symmetric" that appears in the *enter* transition can be directly accommodated by abstracting the symmetric reasoning into a *method* and then applying that method to multiple instances. Likewise, the treatment of *enter* and *exit* is symmetric when their effective conditions are violated, in which case the result follows directly from the inductive hypothesis, and this commonality too can be easily factored out into a general method. The entire proof, along with these two methods, can be seen below. Note that the proof doesn't use external theorem provers. Instead, it uses Athena's own library *chain* method, which allows for limited proof search. The *chain* method extends the readability benefits of equational chains into arbitrary implication chains.

```
# This is the ``symmetric'' reasoning that appears in two
# places in the treatment of enter. We abstract it here
# into one single generic method M.

define (M inequality enabled-premise IH) :=
  match [inequality enabled-premise] {
    [(~ (i = k)) (((k at s) = rs) & (~ (locked s)))] =>
      (!chain->
       [inequality
    ==> (enabled-premise & inequality) [augment]
    ==> (i at k enter s = i at s)      [enter-axioms]
    ==> (i at s = i at k enter s)      [sym]
    ==> (i at s = cs)                  [(i at k enter s = cs)]
    ==> (i at s = cs & i at s = cs)    [augment]
    ==> (locked s)                     [IH]
    ==> (locked s & ~ locked s)        [augment]
    ==> false                          [prop-taut]])
  }

# This method handles all cases where the effective condition
# is violated.

define (direct-ih hyp s failed-ec IH transition-axioms) :=
 match hyp {
   (((i at s') = cs) & ((j at s') = cs)) =>
     let {s=s' := (!chain->
                   [failed-ec ==> (s' = s) [transition-
                                            axioms]])}
       (!chain-> [hyp ==> (i at s = cs & j at s = cs) [s=s']
                      ==> (i = j & locked s)          [IH]
                      ==> (i = j & locked s')         [s=s']])
  }

# The main proof:

by-induction (forall s . gp s) {
  init => (!spf (gp init) (ab))
| (s' as (k enter s)) =>
   pick-any i:Pid j:Pid
    let {IH := (forall i j . i at s = cs & j at s = cs
                     ==> i = j & locked s)}
    assume hyp := (i at s' = cs & j at s' = cs)
     conclude goal := (i = j & locked s')
      (!two-cases
       assume case1 := (k enabled-at s)
        let {s'-locked := (!chain-> [case1
                              ==> (locked s') [enter-axioms]]);
             s-not-locked := (!chain-> [case1
                              ==> (~ locked s) [right-and]]);
             i=j := (!by-contradiction (i = j)
                     assume h := (i =/= j)
                  let {D := {(i =/= k | j =/= k) from h}}
                       (!cases D
                        assume i=/=k := (i =/= k)
                         (!M i=/=k case1 IH)
                        assume j=/=k := (j =/= k)
                         (!M j=/=k case1 IH)))}
        (!both i=j s'-locked)
       assume case2 := (~ k enabled-at s)
        (!direct-ih hyp s case2 IH enter-axioms))
| (s' as (k exit s)) =>
  pick-any i:Pid j:Pid
   let {IH := (forall i j . i at s = cs & j at s = cs
                    ==> i = j & locked s)}
    assume hyp := (i at s' = cs & j at s' = cs)
     conclude goal := (i = j & locked s')
      (!two-cases
       assume case1 := (k at s = cs)
        (!by-contradiction goal
         assume -goal := (~ goal)
          let {locked-s :=
                (!chain-> [case1
                       ==> (case1 & case1) [augment]
                       ==> (locked s)      [IH]]);
               p2 := (!chain-> [case1
                           ==> (k at s' = rs) [exit-axioms]]);
               i=/=k :=
                 (!by-contradiction (i =/= k)
                  assume h := (i = k)
                   (!chain-> [p2
                          ==> (i at s' = rs) [h]
                          ==> (cs = rs)       [(i at s' = cs)]
                          ==> (cs = rs & cs =/= rs)  [augment]
                          ==> false              [prop-taut]]))}
           (!chain-> [i=/=k
                  ==> (i at s' = i at s)  [exit-axioms]
                  ==> (i at s = cs)        [(i at s' = cs)]
                  ==> (i at s = cs & k at s = cs) [augment]
                ==> (i = k)                       [IH]
                ==> (i = k & i =/= k)             [augment]
                ==> false                         [prop-taut]]))
        assume case2 := (k at s =/= cs)
         (!direct-ih hyp s case2 IH exit-axioms))
}
```

References

1. Diaconescu, R., Futatsugi, K., Ogata, K.: CafeOBJ: Logical foundations and methodologies. Comput. Inform. **22**, 257–283 (2003)
2. Clavel, M., Durn, F., Eker, S., Lincoln, P., Mart-Oliet, N., Meseguer, J., Quesada, J.: Maude: specification and programming in rewriting logic. Maude System documentation (1999)
3. Mossakowski, T., Haxthausen, A.E., Sannella, D., Tarlecki, A.: Casl the Common Algebraic Specification Language. In: Logics of Specification Languages. Part of the series Monographs in Theoretical Computer Science pp. 241–298 (2008)
4. Nipkow, T.: Programming and Proving in Isabelle/HOL. Technical Report (2014)
5. Autexier, S., Mossakowski, T.: Integrating HOL-CASL into the development graph manager MAYA. Frontiers of combining systems. Lect. Notes Comput. Sci. **2309**, 2–17 (2002)
6. Codescu, M., Horozal, F., Kohlhase, M., Mossakowski, T., Rabe, F., Sojakova, K.: Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In Till Mossakowski, Hans-Jrg Kreowski (eds.), Recent Trends in Algebraic Development Techniques, 20th International Workshop, WADT 2010, vol. 7137, 139–159, Lecture Notes in Computer Science. Springer, Berlin (2010)
7. Arkoudas, K.: Athena, proofcentral.org, (2004)
8. CafeOBJ Algrebraic Specification and Verification. https://cafeobj.org/
9. CafeOBJ@NTUA blog. https://cafeobjntua.wordpress.com/2016/05/26/on-combining-algebraic-specications-with-first-order-logic-via-athena/
10. Smith, M., Klarlund, N.: Verification of a Sliding Window Protocol Using IOA and MONA. Research Report RR-3959, INRIA (2000)
11. Goguen, J., Malcolm, G.: A hidden agenda. Technical Report No. CS97-538, Ed.: University of California at San Diego (1997)
12. Ogata, K., Futatsugi, K.: Compositionally writing proof scores of invariants in the OTS/-CafeOBJ method. J. Univers. Comput. Sci. **19**(6), 771–804 (2013)
13. Futatsugi, K., Gaina, D., Ogata, K.: Principles of proof scores in CafeOBJ. Theor. Comput. Sci. **464**, 90112 (2012)
14. Gaina, D., Lucano, D., Ogata, K., Futatsugi, K.: On automation of OTS/CafeOBJ method. In: Specification, Algebra, and Software, LNCS **8373**, 578–602 (2014)
15. Ogata, K., Futatsugi, K.: Proof scores in the OTS/cafeOBJ method. In Proceedings of the Conference on Formal Methods for Open Object-Based Distributed Systems, vol. 2884 170–184 (2003)
16. Arkoudas, K., Musser, D.: Fundamental Proof Methods in Computer Science. MIT Press (2017)
17. Musser, D.: Understanding Athena Proofs
18. Vampire, Web page. www.vprover.org/
19. Spass, Web page. www.spass-prover.org/
20. Algebraic modeling of topological and computational structures (AlModTopCom). http://www.math.ntua.gr/~sofia/ThalisSite/publications.html
21. Ouranos, I., Ogata, K., Stefaneas, P.: TESLA Source Authentication Protocol Verification Experiment in the Timed OTS/CafeOBJ Method: Experiences and Lessons Learned. IEICE Trans. **97**(5), 1160–1170 (2014)