Mechanizing FOOPS Refinement Proofs

Paulo Borba* and Joseph A. Goguen†
Programming Research Group
Oxford University Computing Laboratory

February 15, 1996

Abstract

FOOPS is a concurrent object oriented specification language with an executable subset. It has an structural operational semantics and associated notions of refinement, which support formal development of concurrent object oriented programs. In this article we present tools and techniques for mechanizing FOOPS refinement proofs. We concentrate on the use of the tools, rather than on their implementation. In particular, we describe a symbolic simulator and a proof assistant, discussing how they can be integrated and used in order to simplify refinement proofs and make them efficient and partly automatic. This is essential for supporting formal development of software in practice.

1 Introduction

The importance of formal methods for software development is significantly recognized. This is mainly justified by the high level of reliability and robustness of complex systems developed using languages having a clear mathematical semantics [19, 17, 11] and allowing formal proofs that design steps satisfy (refine) specifications [15, 14, 5].

However, among other factors, the industrial uptake of formal methods depends crucially on tools to support the use of those methods [4]. In particular, tools for automating tedious (refinement) proof tasks are essential for speeding up formal verification and making proofs more reliable and partly automatic.

In this article we describe tools and techniques for supporting formal development of software in FOOPS, a concurrent object oriented specification language having an executable subset [9, 18]. In particular, we present tools for mechanizing FOOPS refinement proofs. A *symbolic simulator* is used to automate most of the routine work of checking the transitions from a given state, according to FOOPS operational semantics presented in [2]. A *proof assistant* supports mechanical formal reasoning based on FOOPS refinement theory defined in [3], which provides notions of refinement of FOOPS states, expressions, and programs. We also discuss techniques for integrating those tools and using them for simplifying refinement proofs and making them efficient and partly automatic. Those aspects are essential for supporting formal development of software in practice.

Those tools were developed using the OBJ language [12] and the 2OBJ meta-logical framework [10, 21]. In fact, the symbolic simulator is an OBJ program, whereas the proof assistant is implemented by an encoding of FOOPS refinement theory in 2OBJ. So refinement proofs can be realized using OBJ's implementation of equational reasoning by term rewriting, and 2OBJ's facilities for applying general tactics and inference rules.

*Currently at the Computer Science Department of the Federal University of Pernambuco, Recife, Brazil. Supported in part by CAPES, Grant 2184-91-8, and CNPq, Grant 301021/95-3. Electronic mail: `phmb@di.ufpe.br`.

†Supported in part by the Science and Engineering Research Council, the CEC under ESPRIT-2 BRA Working Groups 6071, IS-CORE (Information Systems COrrectness and REusability) and 6112, COMPASS (COMPrehensive Algebraic Approach to System Specification and development), Fujitsu Laboratories Limited, and a contract under the management of the Information Technology Promotion Agency (IPA), Japan, as part of the Industrial Science and Technology Frontier Program "New Models for Software Architectures," sponsored by NEDO (New Energy and Industrial Technology Development Organization). Electronic mail: `Joseph.Goguen@prg.oxford.ac.uk`.

This article is structured in the following way. First, we give an overview of FOOPS, including details about its operational semantics and notions of refinement. After that we describe FOOPS symbolic simulator followed by the proof assistant, concentrating on the aspects necessary for using the tools, rather than on their implementation. Lastly we illustrate how the tools can be used for proving refinement and present general conclusions.

2 Overview of FOOPS

FOOPS is a concurrent object oriented specification language having an executable subset [9, 18]. It includes a functional language derived from OBJ [12], which is a first order, purely functional language providing an algebraic style for the specification, rapid prototyping, and implementation of abstract data types. OBJ supports the declaration of "loose" theory specifications as well as "tight" executable specifications.

FOOPS extends OBJ by providing a simple "declarative" style for object oriented programming and specification. It supports classes of objects with associated methods and attributes, object identity, dynamic object creation and deletion, overloading, polymorphism, inheritance with overriding, dynamic binding, parameterized programming [7], and many additional features. In this article we consider an extension of FOOPS with features for specifying systems of distributed and autonomous objects [2, 1]. Essentially, this extension allows the definition of non terminating (autonomous) methods, and provides constructors (called method combiners) for expressing concurrency, nondeterminism, and atomic execution.

As FOOPS extends OBJ with some concepts from object oriented programming, two central design decisions[1] were motivated: data elements are not objects, and classes are not modules. In this way, FOOPS provides different constructs for defining abstract data types (ADTs) for values and classes for objects. Consequently, there are two constructs for specifying inheritance.

The clear distinction between data elements and objects divides the language in two parts: the functional level and the object level. At each level, there are two kinds of modules: one is used to define executable code, and is simply called a *module*; the other, called a *theory*, is used to specify properties of the operations of an abstract data type or a class.

In this section we briefly describe the functional and object levels of FOOPS. We also introduce general aspects of FOOPS operational semantics [2] and notion of refinement [3].

2.1 Functional Level

The functional level of FOOPS is a syntactical variant of OBJ. At this level it is possible to define abstract data types, where the keywords `sort` and `fn` respectively introduce the name of the set of data elements, and the associated operation (function) symbols. A very simple functional theory is

```
fth TRIV is
  sort Elt .
endfth
```

It introduces the sort `Elt`. The models of `TRIV` are just sets.

As an example of a parameterized functional level module, we consider (piece by piece) a module `LIST`, which defines lists of elements of a given sort. This module is parameterized by the sort of the elements in a list:

```
fmod LIST[E :: TRIV] is
  pr NAT .
```

The following declaration introduces a sort for nonempty lists and another for lists,

```
  sorts NeList List .
  subsorts Elt < NeList < List .
```

[1] See [13, 9, 18, 20] for more details about the design of FOOPS.

where elements are considered singleton (nonempty) lists and nonempty lists are, of course, lists, as indicated by the subsort relationship (<). This is what specifies inheritance at the functional level; for example, as all elements of `Elt` are elements of `List`, all functions associated to `List` can also be used for the elements of `Elt`.

The empty list is represented by the constant `nil`, and `_._` denotes the function that concatenates two lists.

```
  fn nil : -> List .
  fn _._ : List List -> List [assoc id: nil] .
  fn _._ : NeList List -> NeList .
  fn _._ : NeList NeList -> NeList .
```

The underscores in `_._` serve as placeholders for the arguments of this function. As indicated by the attributes, this function is associative and has `nil` as identity.

Some other functions are `head`, which gives the first element of a non empty list; `tail`, which maps a non empty list to one obtained by removing its first element; and `#_`, which gives the number of elements in a list.

```
  fn head : NeList -> Elt .
  fn tail : NeList -> List .
  fn #_ : List -> Nat .
```

Properties establishing the meaning of these functions are given by axioms (equations).

```
  ax head(E . L) = E .
  ax tail(E . L) = L .
  ax # nil = 0 .
  ax # (L . E) = # L + 1 .
endfmod
```

The keyword `ax` precedes an axiom, and `endfmod` indicates the end of a functional module.

2.2 Object Level

At the object level, it is possible to define classes, which are collections of (potential) objects with same attributes and methods. FOOPS has a general computational model where objects are naturally distributed and (internally) concurrent. Objects are dynamically created and deleted, and there are special operations for performing these actions. Furthermore, each object has an unique identifier, which is used by other objects for access. In this way, methods and attributes have at least one object identifier as argument, indicating which method is going to execute the corresponding operation.

Let us consider an object module `BUFFER` defining a class of bounded buffers. This module is parameterized by the capacity of buffers (a positive natural number), specified by the functional requirements theory `MAX`:

```
fth MAX is
  pr NAT .
  fn max : -> NzNat .
endfth
```

The sort of positive natural numbers is represented by `NzNat` (it is defined in `NAT`). The module `BUFFER` is also parameterized by the sort of the elements to be stored in buffers:

```
omod BUFFER[E :: TRIV, M :: MAX] is
  pr LIST[E] .
```

Here the elements of a buffer are stored in a list; so, it is necessary to import the functional module `LIST`, instantiating it with the argument module giving the sort of elements.

The class `Buffer` of bounded buffers is introduced by the declaration

```
class Buffer .
```

Attributes are defined as operations from an object identifier to a value that denotes a current property of the related object. Objects of `Buffer` have the attribute `elems`, corresponding to the list of elements in a buffer.

```
at elems_ : Buffer -> List [hidden] .
```

As indicated by the declaration `[hidden]`, the attribute `elems` is only visible inside `BUFFER`; so, clients of the objects of `Buffer` cannot directly look at the elements stored in buffers. In addition to `elems`, two more attributes are associated to `Buffer`:

```
at empty?_ : Buffer -> Bool .
at full?_ : Buffer -> Bool .
```

The attribute `empty?` indicates whether the buffer is empty, whereas `full?` indicates whether the buffer contains the maximum number of elements.

Like attributes, methods are defined as operations having an object of its class as parameter. They might also have some extra parameters. Methods either evaluate to a special result or to the identifier of the object that performs it. For objects of `Buffer`, the available methods are the following: `reset`, which removes all elements from a buffer; `get`, which removes the first element of a non-empty buffer and gives it as result; `put`, which inserts an element at the end of a buffer, if it is not full; and `del`, which removes the first element of a non empty buffer.

```
me reset : Buffer -> Buffer .
me get : Buffer -> Elt .
me put : Buffer Elt -> Buffer .
me del : Buffer -> Buffer [hidden] .
```

The last one is hidden because we do not allow clients to remove an element from a buffer unless it is going to be used, which can be done using `get`.

Attributes can be classified as *stored* or *derived*. The value of a stored attribute is kept as part of the local state of an object. On the other hand, the value of a derived attribute is not stored by an object, but can be computed from the values of other attributes. Hence, one must specify how this is done; in FOOPS, we use equations for that. If no equation is given for an attribute, it is considered a stored attribute.

For `Buffer`, we define `elems` as a stored attribute. The others are derived; so, we introduce the following equations:

```
ax empty? B = (elems B) == nil .
ax full? B = #(elems B) == max .
```

This indicates that the buffer is empty if the list of the elements stored in it is empty; also, the buffer is full if the size of its associated list is `max`. Note that every module in FOOPS automatically imports a built-in module of booleans containing the usual operations, and the overloaded equality[2] (`_==_`) and inequality operations (`_=/=_`).

Equations defining attributes can only contain functions, attributes, and object identifiers. This kind of equation is interpreted as left-to-right rewrite rules, but attributes are atomically evaluated, without interference from the execution (evaluation) of methods. Of course, an attribute expression can only be evaluated in a database state if the corresponding attribute is defined in that state.

The behavior of methods can be specified by two different kinds of axioms. A *direct method axiom* (DMA) specifies how a stored attribute is updated by a given method. In fact, a DMA is an equation such that its left hand side (LHS) indicates its associated attribute and method, whereas its right hand side (RHS) is an expression specifying the new value for the attribute to be updated. For instance, the behavior of `reset` is given by the DMA

```
ax elems(reset(B)) = nil .
```

[2] Two expressions are equal if their evaluation yield the same values.

which specifies that after the execution of `reset` by an object `B`, the value of `elems`, for `B`, is `nil`.

Further examples of DMAs are

```
cx elems(put(B,E)) = (elems B) . E if not(full? B) .
cx elems(del(B)) = tail(elems B) if not(empty? B) .
```

where the methods are only executed if the (enabling) conditions are satisfied; otherwise, the evaluation is suspended. The new value for the specified attribute is computed in terms of the method arguments and the current attribute values. If there is no axiom specifying the new value for a stored attribute after the execution of a given method, this method does not update that attribute.

The evaluation of methods specified by DMAs is atomic and yields the identifier of the object which executes the method; only this objects is modified, and its attributes are updated as specified. As for attribute equations, the axiom's RHS and condition must be formed by functions, attributes, and object identifiers.

Alternatively to DMAs, *indirect method axioms* (IMAs) may be used for defining methods. IMAs are equations that specify how a method is defined in terms of other operations; this is indicated by a method expression, i.e., an expression formed by methods, attributes, functions, object identifiers, and method combiners (operators on method expressions). For example, the method `get` is specified by the IMA

```
acx get(B) = result head(elems B) ; del(B)
if not(empty? B) .
```

where `result_;_` is a method combiner which evaluates its first argument (from left to right) and then evaluates the second one, yielding the value resulting from the evaluation of the first argument.

Similarly to DMAs, no method symbol or method combiner is allowed in an IMA's condition. IMAs are interpreted as left-to-right rewrite rules. Whereas the evaluation of the IMA's condition is atomic, the evaluation of the IMA's RHS is not atomic and may be interfered by the execution of other methods. However, atomicity can be achieved by using the atomic evaluation operator `[_]`, which atomically executes its argument, without interference from the execution of other methods. We assume that IMAs introduced by the keyword `acx` (or `aax`) have their condition and RHS atomically evaluated. In fact, an IMA like

```
acx m(O) = e if c .
```

stands for the following declarations:

```
me m' : C -> C' .
ax m'(O) = e if c .
ax m(O) = [m'(O)] .
```

where `m'` is any method symbol not used in the specification where the original axiom appears.

In addition to `result_;_` and `[_]`, FOOPS provides other method combiners: sequential composition, `_;_`; (interleaving) parallel composition, `_||_`; (external) nondeterministic choice, `_[]_`; and conditional, `if_then_else_fi`.

Dynamic object creation and deletion are respectively provided by the following operators:

- `new.C() : -> C`,
- `new : C -> C`, and
- `remove : C -> C`,

for each class `C`.

For a given class `C`, the operator `new.C()` creates an object of `C` with a nondeterministically chosen identifier that is not already being used for another object. This identifier is given as the result of the evaluation of the operator.

The operator `new` creates an object of the same class as the object identifier given as argument, if this identifier is not associated to another object (otherwise, the operation cannot be executed). This identifier is used for the created object and yielded by the operator.

The operator `remove` receives an object identifier as argument, removes its associated object and yields this identifier. If this identifier does not correspond to an object in the state, the operation is not evaluated.

2.3 Operational Semantics

A FOOPS specification (a non-parameterized module) defines a FOOPS system; it describes the structure of the objects that the system may have, and the behavior of the methods and attributes associated to those objects. Object creation, object deletion, and method execution change the state of a FOOPS system. Such a state consists of information about existing objects in the system (database state), and expressions being concurrently evaluated.

FOOPS operational semantics [2] formalizes the notions of states and state transitions due to evaluating expressions. States are represented by pairs formed by an expression and a database state, represented as $\langle e, \mathcal{P} \rangle$, for an expression e and a database state $\mathcal{P}$. The first corresponds to the expressions being evaluated in the state, and the second contains information about the objects in the state. When not confusing, database states are just called "states". The initial database state associated to P is represented by $\emptyset_{\mathsf{P}}$, which contains only objects introduced by P and information about their respective attributes. Also, we use $Conf(\mathsf{P})$ to denote the set of all configurations associated to a specification P.

State transitions are specified by the transition relation

$$\longrightarrow_{\mathsf{P}} \subseteq Conf(\mathsf{P}) \times Conf(\mathsf{P}),$$

which actually defines the operational semantics of P; it indicates how an expression is evaluated in a database state. When not confusing, we write $\longrightarrow$ instead of $\longrightarrow_{\mathsf{P}}$. Similarly, the relation $\longrightarrow^*$ denotes the transitive, reflexive closure of $\longrightarrow$; also, we write $P \not\longrightarrow$ if there is no P' such that $P \longrightarrow P'$ (this means that the expression in P cannot be further evaluated).

This transition relation is inductively defined over the syntax of method expressions by inference rules which indicate how we can infer that two configurations are related (i.e., there is a transition from one to the other), assuming that some others are related. For instance, the following two rules define the semantics of the atomic evaluation operator:

$$\frac{\langle e, \mathcal{D} \rangle \longrightarrow^* \langle v, \mathcal{D}' \rangle}{\langle \texttt{[}e\texttt{]}, \mathcal{D} \rangle \longrightarrow \langle v, \mathcal{D}' \rangle} \qquad\qquad \frac{\neg\mathbf{terminating}(\langle e, \mathcal{D} \rangle)}{\langle \texttt{[}e\texttt{]}, \mathcal{D} \rangle \longrightarrow \langle \texttt{[}e\texttt{]}, \mathcal{D} \rangle}$$

where v is a "fully evaluated expression"—an object identifier or an *evaluated* functional expression—and we say that a configuration is **terminating** if there is no infinite sequence of $\longrightarrow$-transitions from it. Intuitively, the atomic evaluation of an expression corresponds to its full evaluation in only one step (transition). So if the evaluation of the expression to be atomically evaluated does not terminate, the atomic expression does not terminate as well. In fact, it behaves as a divergent process that does not modify the state.

2.4 Refinement

FOOPS refinement notions [3] consider refinement of states, expressions, and programs. In this article we only discuss refinement (simulation) of states. Basically, a state P is simulated by a state Q if whatever can be observed by performing experiments with Q can also be observed by performing the same experiments with P. In this case, if experiments are done with Q, we cannot detect whether Q or P is being used, even though the experiments done with P might let us make more observations than if Q were used.

The only kind of experiment that we can make with a FOOPS system is to invoke *visible* operations (i.e., methods, attributes, and object creation and deletion routines) with arbitrary arguments. This results in a new state having the invoked operation as one of the expressions to be evaluated. Here are the aspects that we can observe from a state P:

1. the values of *visible* attributes of the objects in P;
2. the results yielded by the expressions being evaluated in P; and

3. whatever can be observed by performing experiments with the states that can be immediately reached from P due to the execution of the expressions in P.

According to the last item and the notion of simulation we can conclude that if P is simulated by Q then states immediately reached from Q simulate states immediately reached from P. However, a better definition of simulation can be obtained if we relax this strong correspondence between state transitions. In fact, it is only important that states *immediately* reached from Q simulate states *eventually* reached from P. This is enough because in this case the observations that we can make from Q are equivalent to observations that we can make from P when we fail to observe some intermediate states. This still implies that the observable behavior associated to Q is a particular case of the observable behavior associated to P; that is the essence of simulation.

Note that both notions of simulation discussed so far depend only on the notions of experiment and observation. Indeed, one state might simulate another even if they have different (internal) structures, and are associated to systems specified by different specifications. However, the experiments and observations should be meaningful for both systems. In general, given states P and Q respectively related to specifications P and Q, Q may only simulate P if the observations and experiments introduced by P are also introduced by Q. Only in this way we can compare the effects of performing the *same* experiment with P and Q.

Based on FOOPS operational semantics, now we formalize the finer notion of simulation, which was originally presented in [3]. First let $\mathcal{E}(\mathsf{P})$ denote the set of **experiments** associated to a specification P. Also let $\mathcal{FE}(e)$ indicate that the expression e is fully evaluated, according to a specification P that should be understood from the context where $\mathcal{FE}(e)$ is used; otherwise, we write $\mathcal{FE}_{\mathsf{P}}(e)$. Furthermore, for fully evaluated terms p and q respectively related to specifications P and Q, assume that equality is denoted by $p =_{(\mathsf{P},\,\mathsf{Q})} q$; it holds if p and q are the same object identifiers or if they are the same element of an ADT that is defined by both P and Q.

Assuming that the observations and experiments introduced by a specification P are also introduced by a specification Q, the relation of simulation between states of P and Q is defined as the union of all relations $\mathcal{S} \subseteq \mathit{Conf}(\mathsf{P}) \times \mathit{Conf}(\mathsf{Q})$ such that $(P,\ Q) \in \mathcal{S}$ implies the following:

1. Whenever $Q \longrightarrow Q'$ then, for some P', $P \longrightarrow^{*} P'$ and $(P',\ Q') \in \mathcal{S}$.
2. If $Q \not\longrightarrow$ then, for some P',
   (a) $P \longrightarrow^{*} P' \not\longrightarrow$;
   (b) $\mathcal{FE}_{\mathsf{P}}(p') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(q)$;
   (c) if $\mathcal{FE}_{\mathsf{P}}(p')$ then $p' =_{(\mathsf{P},\,\mathsf{Q})} q$; and
   (d) $(P',\ Q) \in \mathcal{S}$.
3. For any experiment $exp \in \mathcal{E}(\mathsf{P})$, whenever $\langle exp,\ \mathcal{Q}\rangle \longrightarrow^{*} Q' \not\longrightarrow$ then, for some P',
   (a) $\langle exp,\ \mathcal{P}\rangle \longrightarrow^{*} P' \not\longrightarrow$;
   (b) $(P',\ Q') \in \mathcal{S}$; and
   (c) $(\langle p,\ \mathcal{P}'\rangle,\ \langle q,\ \mathcal{Q}'\rangle) \in \mathcal{S}$.

where P and Q denote configurations, $\mathcal{P}$ and $\mathcal{Q}$ denote their respective database states, and p and q denote their respective expressions.

Item 1 above says that any state immediately reached from Q is related to some state that might eventually be reached from P.

Condition 2 indicates that if the expression in Q cannot be further evaluated then the expression in P might eventually reach the same situation; when this happens, the resulting states will be related by $\mathcal{S}$ and the results yielded by the evaluation of the expression in Q will also be yielded by the evaluation of the expression in P. In other words, the results of the evaluation of expressions in Q might eventually be observed from P.

Condition 3 says that performing the same experiment with Q and P leads to states related by $\mathcal{S}$; by condition 2, this implies that the experiments yield the same results when performed in both states.

Furthermore, if an experiment cannot be performed with Q (its corresponding operation is not enabled, because it cannot be executed in a particular state), then performing the same experiment with P results in a state related to Q by $\mathcal{S}$; this usually means that the experiment cannot be performed with P as well.

Those conditions reflect the ideas that we have introduced about simulation of states. However, they are based on a central assumption: operations used as experiments are atomic and terminating. Note that this does not imply that all available methods have to be atomic and terminating; indeed, nonatomic and nonterminating methods can be defined as hidden operations. In [3], this assumption is analyzed and justified.

Relations having those special properties are called (P, Q)-**simulations**. Note that if the pair (P, Q) is in such a relation then whatever can be observed by performing experiments with Q can also be observed by performing experiments with P; this guarantees that one state simulates another. So in order to prove that a state P is simulated by a state Q, it is enough to find a simulation containing the pair (P, Q). In fact, this is a very effective proof technique which makes proofs easier.

In order to find a candidate simulation relation it is necessary to have some knowledge about the behavior of expressions and the relation between different representations for object states. After a candidate is found, it remains to check the conditions from the definition of simulation; that is a routine task which requires only the application of the transition rules of the operational semantics and checking whether the resulting states are related.

3 A Symbolic Simulator for FOOPS

The main functionality of the FOOPS symbolic simulator (hereafter FOOPS simulator, or simply simulator) is to indicate state transitions resulting from the evaluation of an expression in a database state.

The simulator was derived from the inference rules defining FOOPS operational semantics in such a way that any transition indicated by the simulator can be justified by these rules. The converse is not valid in general since no simulator can decide upon termination of programs and this would be necessary in order to automatically justify some of the transitions of the operational semantics. Nevertheless, under certain circumstances, the simulator can be used without loss of generality.

The simulator manipulates database states having objects whose attributes are instantiated with (correspond to) either actual values or *symbolic* values—that is, symbols representing a set of actual values having common properties. If a (database) state has some objects whose attributes are instantiated with symbolic values, we say it is a *symbolic* state; otherwise, we say it is an actual state. A symbolic state denotes a (possibly infinite) set of actual states having common properties. Moreover, any transition from a symbolic state represents a transition from any of the actual states denoted by this symbolic state. Thus the simulator can be very useful for animating FOOPS specifications and checking possible transitions from a set of states (configurations) having certain properties.

However, transitions depending on arbitrary properties of symbolic values (e.g., transitions corresponding to the evaluation of conditional axioms) cannot be automatically justified. In general, this would require automatic theorem proving involving non trivial theories. Hence the simulator has to indicate possible transitions together with the conditions that have to be valid in order to justify those transitions.

The underlying system supporting the simulator is an extension of a FOOPS interpreter [18, 20], whereas the core of the simulator is implemented by a FOOPS functional level module: `SIMULATOR`. In fact, the main function of the simulator is `simulate`, where, for a group of sentences `H`, a specification `SP`, and an associated state `P`,

```
simulate(SP,P,H)
```

denotes a set of pairs in the form `[ H' , P' ]` such that `P` $\rightarrow_{\texttt{SP}}$ `P'` if the sentences in `H'` are valid[3]. The sentences in `H` correspond to properties that the symbolic values of `P` should have, whereas `H'` contains `H` and possibly some additional constraints on the symbolic values of `P'`.

In this section we show how the specifications that we are interested in simulating are loaded into the simulator. Following that we explain how expressions, states, and sentences are represented. Later we

[3]For simplicity, we use objects of the simulator in place of objects of the operational semantics; in fact, there is a way of mapping the former into the latter, and vice versa.

also illustrate the use of the simulator and discuss in which circumstances it conforms to the operational semantics.

3.1 Specifications

In order to simulate a specification SP, it should be first loaded into the extension of the FOOPS interpreter mentioned in the previous section and then integrated to the core of the simulator in the following way:

```
omod SIM-TH is
  ex SP .
  ex SIMULATOR .
endomod
```

As we will show later, this module can be used to simulate the specification SP, by using the ADTs provided by SIMULATOR, and the extended FOOPS interpreter for carrying on computations; that is, rewriting terms for checking state transitions.

For instance, a specification ready for simulation can be obtained by instantiating BUFFER (defined on Section 2.2) in order to obtain a specification of buffers that are able to store six natural numbers:

```
omod SP is
  pr BUFFER[view to NAT is sort Elt to Nat . endv,
            view to NAT is fn max to 6 . endv] .
endomod
```

Hereafter we will use SP to denote this specification.

In fact, the current implementation of the simulator only accepts specifications in a small extension of the FOOPS syntax informally described in Section 2. However, the extra information available in those specifications can be automatically derived by preprocessing FOOPS specifications [1], and the simulator could be easily extended in that direction. So, for simplicity, in this article we assume that the simulator accepts specifications in the original FOOPS syntax.

3.2 Expressions

The types (sorts and classes) of a FOOPS specification are automatically declared a subtype of Universal when the corresponding module is loaded into the simulator. So Universal is a kind of universal type corresponding to the expressions (terms) that can be simulated.

Considering the module SIM-TH (introduced in the previous section), any term formed with the operations of SP can be simulated, since any type of SP is a subtype of Universal. But in order to simulate more useful expressions, we have to indicate the object identifiers associated to the class Buffer. This can be easily done by extending SIM-TH[4] with the following declaration:

```
mes b b' b1 b2 : -> Buffer .
```

This indicates that b, b', b1, and b2 are object identifiers associated to Buffer. We can then simulate expressions such as put(b,0), denoting the insertion of 0 into the buffer b.

3.3 Database States

In order to represent database sates (for short, just states), the simulator provides an special ADT. In this article we shall be interested in three operations from this ADT: an operation for creating the initial state associated to a specification; an operation for adding an object to a state; and an operation for updating the value of an attribute. Those operations are respectively declared as follows:

```
fn initial-state_ : Specification -> State .
fn add-object_to_ : Universal State -> State .
fn update-attribute_with_in_ : Universal Universal State -> State .
```

[4] The interpreter extends the last loaded module with any declaration preceded by openr and followed by close.

The sorts Specification and State respectively represent the ADTs of specifications and database, both defined by the simulator. The operations above are enough to represent all database states that we are interested in for simulation.

In order to use the function initial-state to create the initial state associated to SP, we have to obtain a representation of the module SP as an element of the sort Specification. This can be done by extending SIM-TH in the following way:

```
fn SP : -> Specification [memo] .
ax SP = module-to-specification("SP") .
```

The function module-to-specification is provided to yield the element of Specification denoted by the FOOPS module whose name is given as argument[5]. Note that we are using the symbol SP both for the FOOPS module and its representation as an element of Specification. The declaration [memo] indicates that the value associated to SP is computed once and then memorized; this is used for efficiency reasons.

We can now extend SIM-TH by introducing some database states:

```
fns st st1 : -> State [memo] .
ax st = initial-state SP .
ax st1 = update-attribute (elems b) with nil in (add-object b to st) .
```

Thus the state st has no objects, whereas st1 has the object b with its attribute elems set to nil.

Finally, configurations are represented in the simulator by elements of the sort Configuration, which denotes an ADT of configurations. This ADT has an operation that will be of interest for us in this article:

```
fn <_,_> : Universal State -> Configuration .
```

It creates a configuration, given an expression and a database state. For example, consider the following configuration:

```
fn conf : -> Configuration .
ax conf = < put(b,s(s(0))) , st1 > .
```

We assume that SIM-TH is extended by those declarations.

3.4 Sentences

Properties of the symbolic values used for simulation must be expressed as equations. Equations are also used by the simulator to express the conditions justifying certain transitions. In the simulator, equations are represented by using the following operator:

```
fn _=_ : Universal Universal -> Sentence .
```

The sort Sentence is defined by the simulator; it represents an ADT of sentences (equations). For instance, the expressions

    not(0 == 6) = true   and   ((s(0) : nil) == nil) = true

are typical equations associated to the specification SP. Note that '==' is FOOPS built-in equality operation, whereas '=' is simply a constructor of the sort Sentence.

Sentences can be grouped by using the operations associated to the sort Sentences, also provided by the simulator:

```
fn * : -> Sentences .
fn __ : Sentence Sentences -> Sentences .
```

The constant * denotes the empty group of sentences. The inclusion of a sentence in a group of sentences is represented by juxtaposition. For example, the group denoted by the expression

[5]This module should have been previously loaded into the interpreter.

```
(not(0 == 6) = true)  (not(s(0) == 6) = true)  *
```

is formed by the sentences 'not(0 == 6) = true' and 'not(s(0) == 6) = true'.

Lastly, we can pair groups of sentences with configurations using the following operation:

```
fn [_,_] : Sentences Configuration -> SentConf .
```

The sort SentConf represents an ADT of pairs of groups of sentences and configurations. For example, consider that SIM-TH is extended by the following pair:

```
fn sentconf : -> SentConf .
ax sentconf = [ * , conf ] .
```

where conf is the configuration defined in the previous section.

3.5 Simulations

As previously discussed, the main function provided by the simulator is simulate, which is formally declared as follows:

```
fn simulate : Specification Configuration Sentences -> SentConfSet .
```

The sort SentConfSet corresponds to an ADT of sets of elements of SentConf (introduced in the previous section). The simulator also provides another function that is quite useful for simulation and animation of specifications; it is declared as follows:

```
fn final-states : Specification SentConf -> SentConfSet .
```

For a specification SP, a group of sentences H, and a configuration P, if [H' , P'] is in

```
final-states(SP,[ H , P ])
```

then P $\rightarrow^*_{\mathtt{SP}}$ P' $\not\rightarrow_{\mathtt{SP}}$ unless one of the sentences in H' is not valid.

Now we illustrate the application of those functions[6] using the module SIM-TH (defined on Section 3.1) and the declarations introduced in previous sections. Let us consider a database state stP having only an uninitialized object b':

```
fn stP : -> State [memo] .
ax stP = add-object b' to st .
```

The evaluation of stP yields b', representing the state having only the uninitialized object b'. We can check the effect of executing reset(b') in stP by evaluating

```
simulate(SP,< reset(b') , stP >,*)
```

This yields the following:

```
[ true = true * , < b' , b' elems_: nil > ] + {}
```

where 'b' elems_: nil' represents a database state containing the object b' with its attribute elems set to nil, {} denotes the empty set of pairs of sentences and configurations, and e + s denotes the set containing e and all elements in the set s. So there is only one transition from the simulated state; the execution of reset(b') is deterministic. Also, note that the unique transition depends only on the condition expressed by the tautology true = true[7]; hence it is always possible.

The expression simulate(SP,< empty?(b) , stP >,*) evaluates to {}. In fact, empty?(b) cannot be executed in stP since b is not in stP. Thus let us try to evaluate empty?(b) in a database containing the object b:

[6] For legibility, we omit details from the output of the simulator (e.g., module names, parentheses, etc.).

[7] The simulator generates such a condition because an unconditional axiom e = f is actually an abbreviation for the conditional axiom e = f if true.

```
fn stP1 : -> State [memo] .
ax stP1 = add-object b to st .
```

But the evaluation of `simulate(SP,< empty?(b) , stP1 >,*)` also yields `{}`, since `b` is not initialized in `stP1`. So let us try to execute `empty?(b)` in a state having `b` with its attribute `elems` set to the arbitrary symbolic list `l` (this should be defined in `SIM-TH` or `SP`):

```
fn l : -> List .
fn stP2 : -> State [memo] .
ax stP2 = update-attribute (elems b) with l in stP1 .
```

Then the expression `simulate(SP,< empty?(b) , stP2 >,*)` evaluates to

```
[ * , < [| l == nil |]-{SP} , b elems_: l > ] + {}
```

indicating that there is only one transition from the simulated state, and this transition does not depend on any condition. The expression `[| l == nil |]-{SP}` denotes the evaluated (normal) form of `l == nil` with respect to `SP`.

The execution of `put(b,e)`, for an arbitrary symbolic element `e`, in `stP2` is illustrated by evaluating

```
simulate(SP,< put(b,e) , stP2 >,*)
```

which results in

```
[ not(# l == 6) = true * , < b , b elems_: l : e > ] + {}
```

This indicates that there is only one state transition resulting from the execution. Moreover, this transition can only be justified if the sentence `not(# l == 6) = true` is valid. So `put(b,e)` can only be executed in `stP2` if the buffer `b` is not full; that is, the length of the symbolic list `l` is not six.

Let us now illustrate the execution of a nondeterministic expression. The evaluation of

```
simulate(SP,< put(b,e) [] put(b,0) , stP2 >,*)
```

yields

```
[ not(# l == 6) = true * , < b , b elems_: l : e > ] +
[ not(# l == 6) = true * , < b , b elems_: l : 0 > ] + {}
```

indicating that two different resulting states may be reached, both depending on the same conditions. Similarly, the evaluation of `simulate(SP,< put(b,e) || put(b,0) , stP2 >,*)` yields

```
[ not(# l == 6) = true * , < b || put(b,0) , b elems_: l : e > ] +
[ not(# l == 6) = true * , < put(b,e) || b , b elems_: l : 0 >] + {}
```

corresponding to the alternatives generated by the interleaving of the expressions.

Lastly, let us simulate the execution of the object creation operation. The expression

```
simulate(SP,< reset(new.Buffer()) , stP2 >,*)
```

evaluates to

```
[ * , < reset(BufferId34) , BufferId34 b elems_: l > ] + {}
```

where `BufferId34` is an object identifier automatically created by the simulator, and the resulting database state contains the objects `BufferId34` and `b`; the former is not initialized, whereas the attribute `elems` of the latter is set to `l`. The states resulting from the full execution (i.e., not just one state transition) of `reset(new.Buffer())` in `stP2` can be checked by evaluating

```
final-states(SP,[ * , < reset(new.Buffer()) , stP2 > ])
```

which results in the following:

```
[true = true * , <BufferId34 , BufferId34 elems_: nil b elems_: l>] + {}
```

where the attribute `elems` of `BufferId34` and `b` are respectively set to `nil` and `l`.

3.6 Correctness

Ideally, the transitions indicated by the simulator should exactly be the transitions defined by FOOPS operational semantics; that is, the simulator should be correct with respect to the semantics, it should conform to the semantics. This is not the case of the FOOPS simulator since the definition of the operational semantics depends on issues such as termination of programs, as discussed on Section 2.3. In fact, any transition indicated by the simulator is a transition specified by the operational semantics. However, the reverse only holds under certain circumstances, which we will describe now.

Obviously, the simulator cannot decide upon termination of programs. So it cannot automatically justify transitions specified by the rule of the atomic evaluation operator (see Section 2.3) that depends on termination; in fact, those transitions are not indicated by the simulator. Therefore the simulator can only conform to the semantics when expressions to be atomically evaluated terminate in the contexts where they are used, since only in those cases the transitions discussed above cannot happen. Actually, the simulator could indicate those transitions provided that they were conditional on termination assumptions. But, as shown in Section 3.4, assumptions indicated by the simulator should be equations relating expressions of the specification being simulated. Moreover, extending the language of assumptions to include termination predicates would not solve other problems related to termination issues, as we will describe next.

In order to indicate transitions specified by the other rule of the atomic evaluation operator, the simulator essentially has to compute the transitive closure of $\rightarrow_{\texttt{SP}}$ for a given state. But this computation only terminates if there is no infinite sequence of transitions from that state. In fact, the evaluation of `simulate` might not terminate. Again, we can conclude that the simulator can only conform to the semantics when expressions to be atomically evaluated terminate.

4 A Proof Assistant for FOOPS

As shown in Section 2.4, in order to prove refinement we have to reason about many different concepts (e.g., states and state transitions, simulation relation, and fully evaluated terms). Hereafter we use the name "refinement theory" for the collection of those concepts and associated properties. The FOOPS proof assistant (for short, proof assistant) supports mechanical formal reasoning based on the refinement theory.

The proof assistant is based on the 2OBJ system, which is a theorem prover that can be tailored to prove theorems in any logical system. By loading the encoding of a logic into 2OBJ, we automatically obtain a theorem prover for that logic. Once 2OBJ is tailored for a particular logic, we can define theories based on this logic and build proofs over these theories. Usually, proofs are interactively built through 2OBJ's graphical interface. Here we simply indicate the proof steps necessary for completing a proof, instead of showing how those can be given as commands to 2OBJ.

In particular, the proof assistant is based on a version of 2OBJ tailored for first order predicate logic with equality, which is suitable for formalizing the refinement theory. In fact, the refinement theory is encoded as a first order predicate logic theory. This encoding is the core of the proof assistant. In order to prove refinement, we reason about an "user-defined" extension of this encoding with axioms specific for the FOOPS specifications of interest.

Here we present aspects of the encodings of the refinement theory and first order predicate logic. We discuss only the aspects that we have to know in order to build refinement proofs. First we consider the encoding of the logic, as originally described in [21]; this should give a general overview of the basic machinery available for constructing refinement proofs. Then we consider the aspects of the encoding of the refinement theory that are necessary for proving condition 1 of the definition of simulation (see Section 2.4); the other conditions can be encoded by using essentially the same techniques, so we omit the details in this article.

4.1 First Order Predicate Logic

In 2OBJ, the encoding of a logic is given by ADTs specified in OBJ (a syntactic variant of the functional level of FOOPS), where sentences and proofs are represented as terms of this ADT, and operations on sentences and proofs are defined as operations of the ADT. In particular, the first order predicate logic with equality is encoded by the OBJ module FOPC. Here we introduce some of the definitions in this module. A more detailed description can be found in [21]. First we explain how sentences and sequents are encoded. Then we describe some of the inference rules provided by FOPC. Lastly, we show how to define tactics in 2OBJ.

4.1.1 Sentences

In the encoding, the usual connectives of first order logic are represented by the following OBJ operations:

```
fn |A| _ . _ : Var Sentence -> Sentence .
fn |E| _ . _ : Var Sentence -> Sentence .
fn _ ^ _ : Sentence Sentence -> Sentence .
fn _ v _ : Sentence Sentence -> Sentence .
fn _ -> _ : Sentence Sentence -> Sentence .
fn ~ _ : Sentence -> Sentence .
fn _ = _ : Term Term -> Sentence .
fn False : -> Prop .
fn True : -> Prop .
```

where the sorts Var, Term, Sentence, and Prop respectively denote the ADTs of first order predicate logic variables, terms, sentences, and propositions. For example, the sentence $(\forall x)\ x = x$ is encoded by the OBJ term `|A| x . x = x`, assuming that x is a constant of sort Var, which is a subsort of Term.

4.1.2 Sequents

In FOPC, first order predicate logic with equality is encoded by a variant of Gentzen's sequent calculus. In order to prove a theorem T, we should start from the sequent (goal) '$\vdash T$' and repeatedly apply inference rules in a backwards way. This generates a tree of sequents where each node in the tree is derivable by some inference rule from its children. Then the proof is complete when all leaf nodes correspond to the conclusions of inference rules having no premises.

In the encoding, sequents are represented by the following operation:

```
fn _ |- _ : SentenceList Sentence -> Sequent .
```

where the sorts Sequent and SentenceList respectively denote ADTs of sequents and lists of sentences. Considering the term '`H |- C`', C is a sentence corresponding to the conclusion to be derived from the list of hypothesis represented by the list of sentences H. Elements of a list of sentences are separated by ';'. The list with no sentences is denoted by the constant '*'.

4.1.3 Inference Rules

In the encoding of a logic, inference rules are represented by OBJ operations. For example, the following sequent calculus inference rules

$$\frac{H \vdash A \qquad H \vdash B}{H \vdash A \wedge B}\ \wedge\text{-introduction} \qquad \frac{H_1, \ldots, H_n, A, B \vdash C}{H_1, \ldots, H_n \vdash C}\ \wedge\text{-elimination} \quad (H_i = A \wedge B)$$

are respectively represented by the operations andi and ande, which are introduced by the following declarations:

```
fn andi : -> Rule .
fn ande : Nat -> Rule .
```

where the sort Rule corresponds to an ADT of inference rules. In order to use the first rule for justifying proofs, we simply write andi, whereas we write ande(i) in order to use the particular instance of the $\wedge$-elimination rule where the related conjunction is in position i of the hypothesis list.

An inference rule is defined by a set of equations specifying the subgoals that have to be proved in order to prove a given goal by applying an instance of the rule. For instance, the rule andi is defined by the following equation:

```
ax andi ( H |- X ^ Y ) = ( H |- X ) , ( H |- Y ) .
```

Note that the subgoals are separated by ‘`,`’. This is just to give a general idea of how inference rules are defined in `FOPC`; we do not actually have to know that for proving refinement.

Besides `andi` and `ande`, the following inference rules from `FOPC` are useful for proving refinement:

Disjunction:

$$\frac{H_1;\ldots;H_n;A\vdash C \quad H_1;\ldots;H_n;B\vdash C}{H_1;\ldots;H_n\vdash C}\texttt{ore}(i) \quad (H_i = A\vee B)$$

Implication and Trivial truth:

$$\frac{H;A\vdash B}{H\vdash A\rightarrow B}\texttt{impi} \qquad\qquad \frac{}{H\vdash True}\texttt{Truei}$$

Universal Quantification:

$$\frac{H\vdash A}{H\vdash \forall V.A}\texttt{Ai}$$

where all instances of V in A are replaced by a completely new variable if V is in H.

$$\frac{H_1;\ldots;H_n;A[t/V]\vdash C}{H_1;\ldots;H_n\vdash C}\texttt{Ae}(i,t) \quad (H_i = \forall V.A)$$

where $A[t/V]$ is the same as A except that all instances of V are replaced by the term t.

Existential Quantification:

$$\frac{H_1;\ldots;H_n;A\vdash C}{H_1;\ldots;H_n\vdash C}\texttt{Ee}(i) \quad (H_i = \exists V.A)$$

where all instances of V in A are replaced by a completely new variable if V is in H or C.

Miscellaneous:

$$\frac{H\vdash X \qquad\qquad H;X\vdash C}{H\vdash C}\texttt{cut}(X) \qquad\qquad \frac{}{H\vdash A\vee\neg A}\texttt{exmid}$$

The module `FOPC` has much more rules, including some rules that provide access to equational reasoning using OBJ's term rewriting mechanism; those will be introduced later.

4.1.4 Tactics

Tactics can be seen as programs for applying inference rules. So a group of common proof steps can naturally be encoded as a tactic. In 2OBJ, new tactics can be programmed using OBJ and the built-in tactics and tacticals (i.e., tactics combiners) described below:

- The application of the tactic `t THEN t'` to a goal corresponds to the application of the tactic `t` to the same goal and, when `t` succeeds, `t'` is applied to the subgoals resulting from the application of `t`. If `t` or `t'` fails then `t THEN t'` fails as well.
- The application of the tactic `t THENL` $(\texttt{t}_1,\ldots,\texttt{t}_n)$ to a goal corresponds to the application of the `t` to the same goal and, when `t` succeeds, the tactics $\texttt{t}_1,\ldots$, and $\texttt{t}_n$ are applied to the corresponding subgoals created by the application of `t`. Note that the number of subgoals must be `n`; otherwise, `t THENL` $(\texttt{t}_1,\ldots,\texttt{t}_n)$ fails. It also fails if `t` or any of the tactics $\texttt{t}_1$, …, or $\texttt{t}_n$ fails.
- The application of the tactic `t ELSE t'` to a goal corresponds to the application of `t` to the same goal and, if `t` fails, `t'` is applied to that goal. If `t'` fails then `t ELSE t'` fails as well.
- The application of the tactic `idtac` to a goal yields the same goal as a subgoal. This tactic is very useful in tactic programming; it always succeeds.

Using those operations, a tactic can be defined by an equation specifying the effect of applying the tactic to a specific goal. For example, a tactic for introducing case analysis in a proof can be easily defined by using the inference rules of `FOPC` in the following way:

```
fn CASE : Sentence -> Tactic .
ax CASE(Se) SQ = cut(Se v ~ Se) THENL (exmid,ore(-1)) .
```

where `SQ` is a variable of sort `Sequent`. By applying the tactic `CASE(sent)`, we split the proof in two parts: in one of them, it is assumed that `sent` is valid; in the other, it is assumed that `~sent` is valid. More elaborated examples of how new tactics can be programmed are given in [21].

4.2 Refinement Theory

Based on the encoding of first order predicate logic described in the previous section, we can define first order theories, which are encoded by extending the module `FOPC` with new sorts, operations, and equations corresponding to the encoding of the sorts, functions, and axioms of the theories. In this section we partly describe the OBJ module `REF-TH`, which is an extension of `FOPC` with the encoding of the refinement theory. We consider only the details necessary for proving Condition 1 of the definition of simulation relation. This illustrates the basic machinery provided by 2OBJ and the proof techniques necessary for proving refinement. The remaining conditions and the definitions of refinement are formalized in a similar way, using essentially the same techniques.

Observe that most objects of the refinement theory are also objects of the symbolic simulator described in Section 3. In particular, the simulator provides OBJ implementations of ADTs of terms, specifications, database states, and configurations, which have an essential role in the refinement theory. So the encoding of those objects in the assistant can be essentially given by importing the module `SIMULATOR`, which defines the simulator, into the module `REF-TH`. In addition to that, extra elements should be added to sorts such as `State`, `Specification`, etc; those extra elements correspond to first order predicate logic variables (elements of sort `Var`, not OBJ variables) ranging over the elements of the corresponding ADTs of states, specifications, etc.

In that way, the simulator and the proof assistant share the same encoding of the ADTs mentioned above. Alternatively, we could have defined similar ADTs for each of the tools. For example, we could have provided an ADT of "semantic states" (i.e., those defined in `SIMULATOR` and manipulated by the simulator) and an ADT of "syntactic states" (i.e., those manipulated by the assistant, including logic variables). It would then be necessary to define functions for mapping states from one ADT to states of the other, so that the simulator and the assistant could communicate. Indeed, this approach is quite inefficient and implies a lot of code duplication. On the other hand, it stresses the conceptual distinction between "semantic" and "syntactic" states. As we will show next, in the current implementation of the assistant, this distinction is given by applying a function that indicates whether an element of the unique ADT of states is a "semantic" state, instead of a variable.

4.2.1 Database States

In the encoding of the refinement theory we refer to the operations on states introduced in Section 3.3 and the following ones:

```
fn objects_ : State -> UniversalSet .
fn eval_in_ : Universal State -> Universal .
fn is-semantic-state?_ : State -> Bool .
```

where the sort `UniversalSet` corresponds to an ADT of sets of elements of `Universal`. The first function gives the set of identifiers of the objects in a database state. The second function yields the value of an attribute expression in a particular state. The last function indicates whether an element of `State` is an actual (semantic) state. This distinction is important since only "semantic states" can be simulated.

Besides the definition of the functions mentioned above, the module `REF-TH` also has the definition of a function for merging "closed" database states; we say that a database state is **closed** when it contains only references to objects in itself. The merging of `st` with `st'` is denoted by `st ++ st'`, where `st` and `st'` are closed states, and the merging function is declared as follows:

```
fn _++_ : State State -> State .
```

The objects in `st` are overwritten by the objects in `st'` having the same identifier.

The equations defining those functions on states are indeed encodings of axioms of the refinement theory. As we will show later, those equations can be used for equational reasoning using `FOPC`'s inference rules that give access to OBJ's term rewriting mechanism. In fact, when proving refinement, those equations are only used for reducing terms. This can be transparently done using `FOPC`'s rules, without having any knowledge about the equations; so we omit the equations here.

4.2.2 Configurations

Configurations are formed by an expression and a database state. In the encoding of the refinement theory, this basic property of configurations is stated by the axiom `conf-inst-ax`, which is declared as follows:

```
fn conf-inst-ax : -> Axiom .
ax axiom conf-inst-ax = (* |- |A| p . |E| pp . |E| dp . p = < pp , dp >) .
```

where `p`, `pp`, and `dp` are elements of sorts `Configuration-Var`, `Universal-Var`, and `State-Var`, respectively; that is, they are first order predicate logic variables respectively ranging over configurations, expressions, and database states. As described in Section 3.3, the operation `< _ , _ >` creates configurations.

The declaration above illustrates how to encode properties of a theory into 2OBJ. Some properties can alternatively be encoded by equations. Also, for pragmatical reasons, properties of the refinement theory sometimes are encoded as inference rules. For example, considering that `freevars(t)` yields the set of free variables in a term `t`, the inference rule

```
fn conf-inst : Configuration Universal State -> Rule .
cx conf-inst(P,pP,dP) (H |- G) = H ; (P = < pP , dP >) |- G
if not(pP in freevars(H |- G)) and
   not(dP in freevars(H |- G)) .
```

essentially express the same property as the axiom `conf-inst-ax`, but it also encapsulates common proof steps. In particular, applying the rule `conf-inst(p,pp,dp)`—for variables `p`, `pp`, and `dp`—to a goal is equivalent to applying the tactic

```
Ae(i,p) THEN Ee(i + 1) THEN Ee(i + 2)
```

to the same goal, assuming that the axiom `conf-inst-ax` is at position `i` in the hypothesis list of that goal. In fact, using inference rules for expressing properties is not only more efficient from the computational point of view, but it also makes proofs shorter and easier to understand.

4.2.3 Relations on Configurations

In `REF-TH`, pairs of configurations are constructed with the following operation:

```
fn [_,_] : Configuration Configuration -> ConfPair .
```

where the sort `ConfPair` corresponds to an ADT of pairs of configurations. The ADT of relations on configurations is defined by the operations associated to the sort `Relation`. Here we are interested in proving that a pair of configurations is in a given relation. This can be indicated by sentences formed by the operation

```
fn _[-_ : ConfPair Relation -> Sentence .
```

where `[ p , q ] [- s` indicates that the pair of configurations `[ p , q ]` is in the relation `s`.

Arbitrary relations can be defined by equations. For example, the identity relation on configurations associated to a specification `sp`, denoted `id(sp)`, is defined as follows:

```
fn id : Specification -> Relation .
ax [axiom id] [ P , Q ] [- id(SP) =
        ((expression P) =~= (expression Q))
      ^ ((state P) =~= (state Q))
      ^ (is P compatible-with SP ? = true)
      ^ (is Q compatible-with SP ? = true) .
```

where syntactical equality is represented by the operation

```
fn _=~=_ : Universal Universal -> Sentence .
```

and the functions `expression` and `state` respectively give the expression and database state associated to a configuration. Also, the predicate `is p compatible-with sp ?` indicates that `p` is a configuration associated to `sp`; that is: $\mathtt{p} \in \mathit{Conf}(\mathtt{sp})$. The equation above is tagged `id`, by the declaration `[axiom id]`, so that we can name it when performing equational reasoning.

4.2.4 Equational Reasoning

As mentioned before, the module `FOPC` has inference rules that provide access to equational reasoning using OBJ's term rewriting mechanism with the equations encoding properties of the refinement theory, such as the equation `id` introduced in the previous section. Here we describe the rules that are necessary for proving refinement.

First we consider an inference rule for applying exhaustive rewriting to a specific subterm of a goal's conclusion. It is declared by

```
fn red : Id Selection -> Rule .
```

and `red(id,sel)` applies rewriting using the equations in the module identified by `id`, where the selected subterm of the goal is identified by `sel`, which should be in one of the following forms:

- `top`, which selects the whole of a term; or
- $(\mathtt{n}_1,\ldots,\mathtt{n}_n)$, which selects the $\mathtt{n}_n$th subterm of the ... of the $\mathtt{n}_1$th subterm of a term. For example, `( 2 )` selects the subterm `3 + 1` from `2 + (3 + 1)`, whereas `( 2 1 )` selects the subterm `3` from `2 + (3 + 1)`.

The module identifier `id` should be in the form `'m`, where `m` is the name of the OBJ module having the equations that we want to use for rewriting; when proving refinement, that should be `REF-TH`.

Equations in the hypothesis list of a goal can be applied to a specific subterm of the conclusion of the same goal. This can be done by using the following `FOPC` rule:

```
fn app : Nat Direction Search Substitution Selection -> Rule .
```

where `app(i,dir,search,subs,sel)` applies the `i`th equation of the hypothesis list of a goal. The equation is applied to the subterm selected by `sel`, in the direction specified by `dir`, which is either `fwd`, for left to right (forward) application, or `bwd` for right to left (backward) application. The argument `search` indicates whether the equation is to be applied exhaustively (`within`) or just once (`at`), and `subs` is a '`,`'-separated list of substitutions for variables in the equation. Terms in the form `'var <- term` specify that the variable `var` is to be replaced by `term`. The empty list of substitutions is represented by the constant `nosub`.

The next rule can be used to apply a named equation to a specific subterm of a goal's conclusion:

```
fn app : Id Id Direction Search Substitution Selection -> Rule .
```

Its second argument is the name of the equation that should be applied, whereas the first argument is the identifier of the module containing this equation. The other arguments are specified as in the previous rule.

In `REF-TH`, we introduce additional inference rules for supporting equational reasoning. In particular, the following three rules respectively are similar to the three rules described so far:

```
fn red-hyp : Id Selection Nat -> Rule .
fn app : Nat Direction Substitution Search Selection Nat -> Rule .
fn app : Id Id Direction Search Substitution Selection Nat -> Rule .
```

However, instead of applying equations to the conclusion of a goal, those rules apply equations to one of the sentences in the hypothesis list of a goal. The position of this sentence in the list is indicated by the last argument of each one of the rules.

In the encoding, there is also a rule for substituting syntactically equal expressions in a specific subterm of a goal's conclusion. This rule is declared by

```
fn synt-eq-app : Nat Direction Search Selection -> Rule .
```

and it is similar to the second rule described in this section, except that the selected element of the hypothesis list should be a term in the form `t =~= t'`, rather than an equation. Lastly, the rule `synt-eq` can be used to derive a sentence in the form `t =~= t'`, provided that the terms `t` and `t'` are syntactically equal.

4.2.5 State Transitions

Sentences indicating state transitions can be created using the following operations:

```
fn _-->{_}_ : Configuration Specification Configuration -> Sentence .
fn _-->*{_}_ : Configuration Specification Configuration -> Sentence .
```

where `p -->{sp} q` is the encoding of $\mathtt{p} \rightarrow_{\mathtt{sp}} \mathtt{q}$, and `p -->*{sp} q` is the encoding of $\mathtt{p} \rightarrow^{*}_{\mathtt{sp}} \mathtt{q}$. The axioms and inference rules expressing properties of sentences formed with the operations above will be presented later.

4.2.6 Simulation Relations

We can indicate that a relation on configurations is a simulation by using the following operation:

```
fn simulation? : Relation Specification Specification -> Sentence .
```

where the second and third arguments should be the specifications respectively associated to the configurations in the source and target of the relation given as the first argument.

As in this article we are considering only the details necessary for proving condition 1 of the definition of simulation, we assume that the following inference rule can be used to prove that a relation is a simulation:

```
fn simulation? : -> Rule .
ax simulation? (H |- simulation?(S,spP,spQ)) =
   (H |- |A| p . |A| q .
         ([ p , q ] [- S) ->
         (|A| q' .   (q -->{spQ} q')
                -> (|E| p' .  (p -->*{spP} p') ^ ([ p' , q' ] [- S)))) .
```

where `p`, `q`, `p'`, and `q'` are first order predicate logic variables ranging over configurations. This rule corresponds to a simplified definition of simulation obtained by omitting several constraints (see [3] for the original definition).

Now observe that a predicate in the form

```
(|A| q' .   (q -->{spq} q')
         -> (|E| p' .  (p -->*{spp} p') ^ ([ p' , q' ] [- s)))
```

for a relation `s` and specifications `spp` and `spq`, is trivially valid if there is no transition from `q`. This definitely happens when the expression of `q` is fully evaluated, which is indicated by the boolean expression `FE(expression q,spq)`. Thus we can justify the following inference rule:

```
fn simulation-fe : Nat -> Rule .
cx simulation-fe(N)
   (H |- (|A| Qv' .   (Q -->{spQ} Qv')
                  -> (|E| Pv' .  (P -->*{spP} Pv')
                                 ^ ([ Pv' , Qv' ] [- S)))) = []
if (FE(expression Q,spQ) = true) == hyp(N,H) .
```

where `Qv'` and `Pv'` are OBJ variables of sort `Configuration-Var`, and the constant '`[]`' denotes the empty list of goals. So no goals remain to be proved after applying this rule. But note that the rule can only be applied to a goal in the general form described above. Moreover, the `N`th element of the hypothesis list of this goal, denoted by `hyp(N,H)`, has to indicate that the expression of `Q` is fully evaluated.

Sometimes we actually have to check state transitions in order to prove that a relation is a simulation. As discussed in Section 3, we can use the FOOPS symbolic simulator in order to automatically check state transitions under certain circumstances. In fact, we can invoke the simulator from the proof assistant since the module `SIMULATOR` is imported into the module `REF-TH`. For that end, we provide an inference rule for automatically justifying state transitions based on the simulator's output.

Recall that the simulator can only be used to check a transition in the form `p -->{spp} p'` if the expressions to be atomically evaluated in `p` terminate. In the encoding, this is formalized by the following sentence

```
atomic-exps-terminate?(expression p,spp)
```

However, the refinement theory does not provide enough information for proving this kind of sentence. So it is mainly used as an annotation. An extended version of the encoding should provide rules for reasoning about termination as well.

Also, the simulator can only work with "semantic" specifications and states, instead of variables of sorts `Specification-Var` and `State-Var`. Hence the values of the expressions `is-semantic-spec?(spp)` (see Section 4.2.1) and `is-semantic-state?(state p)` should be true if we want to automatically check a transition in the form `p -->{spp} p'`, where the function `is-semantic-spec?` is the corresponding of `is-semantic-state?` for specifications.

Besides indirectly indicating that the simulator can be used to check state transitions, the following inference rule also reveals that the evaluation of an expression in a database state `st` depends only on any closed database state contained in `st` and having the objects corresponding to the identifiers in that expression:

```
fn simulation-db : -> Rule .
cx simulation-db
   (H |- (|A| Qv' .   (< qQ , (dQ ++ dQ') > -->{spQ} Qv')
                  -> (|E| Pv' .  (< pP , (dP ++ dP') > -->*{spP} Pv')
                                 ^ ([ Pv' , Qv' ] [- S)))) =
       (H |- atomic-exps-terminate?(pP,spP)) ,
       (H |- atomic-exps-terminate?(qQ,spQ)) ,
       (H |- is < pP , dP' > related-to < qQ , dQ' > by(S,spP,spQ,dP,dQ))
 if is-semantic-spec?(spP) and is-semantic-spec?(spQ) and
    is-semantic-state?(dP') and is-semantic-state?(dQ') and
    object-ids(pP,signature spP) subset (objects dP') and
    object-ids(qQ,signature spQ) subset (objects dQ') .
```

In fact, the evaluation of `qQ` in the state '`dQ ++ dQ'`' is equivalent to the evaluation of `qQ` in `dQ'`, provided that the identifiers referenced in `qQ` are in `dQ'`. The function `object-ids` yields the set of object identifiers in a given expression.

As specified by the axiom `is-related-to` (omitted in this article), a sentence in the form

```
is P related-to Q by(S,spP,spQ,dP,dQ)
```

is an abbreviation for

```
are simulate(spQ,Q,*) derived-from P by(S,spP,dP,dQ)
```

This is what really provides access to the simulator, by indicating that all transitions from `Q`—as given by the simulator—can be derived from `P`, since, as specified by the axioms `are-derived-from-e` and `are-derived-from` (omitted in this article),

```
are (SC₁ + ··· + SCₙ + {}) derived-from P by(S,spP,dP,dQ)
```

stands for

```
  is SC₁ derived-from [ * , P ] by(S,spP,dP,dQ)
^ ⋮
^ is SCₙ derived-from [ * , P ] by(S,spP,dP,dQ)
^ True
```

where a sentence in the form '`is SC derived-from [ SSL , P ] by(S,spP,dP,dQ)`' abbreviates

```
  is SC together-with [ SSL , P ] in(S,dP,dQ)
v is SC derived-from simulate(spP,P,SSL) by(S,spP,dP,dQ)
```

Note that this also provides access to the simulator.

A sentence in the form

```
is SC derived-from (SC₁ + ··· + SCₙ + {}) by(S,spP,dP,dQ)
```

abbreviates the sentence

```
  is SC derived-from SC₁ by(S,spP,dP,dQ)
v ⋮
v is SC derived-from SCₙ by(S,spP,dP,dQ)
v False
```

whereas a sentence in the form '`is [ SSL , Q ] together-with [ SSL' , P ] in(S,dP,dQ)`' is equivalent to

```
  ((^^ SSL) -> (^^ SSL'))
^ ([ < (expression Q) , (dQ ++ (state Q)) > ,
     < (expression P) , (dP ++ (state P)) > ] [- S)
```

where `^^(S₁...Sₙ)` stands for `S₁^···^Sₙ`; the first part of the sentence basically specifies that the state transition that lead to `P` can happen whenever the state transition that lead to `Q` happens.

5 Proving Refinement

In order to illustrate how the proof assistant can be used for proving refinement, we present parts of the proof that the specification of bounded buffers represented by lists is refined by a simplified specification of buffers represented by array and pointers.

First we introduce the specification of buffers based on array and pointers and explain how the FOOPS specifications that we want to reason about are integrated to the proof assistant. Later we show how to define relations on configurations associated to those specifications and then prove that the relations are simulations. We consider just one of the conditions of the definition of simulation; the others can be proved in a similar way using essentially the same techniques.

5.1 Specifications

Let us introduce a functional level module defining arrays of elements of an arbitrary sort having a constant that is used as the value yielded by the array indexing operation when the array is not initialized for a particular index. This constant is specified by `TRIVE`, which is an extension of `TRIV` with the sort `Elt?`, supersort of `Elt`, and its constant `err`. We assume that arrays are indexed by natural numbers.

```
fmod ARRAY[E :: TRIVE] is
  pr NAT .
  sort Array .
  fn null : -> Array .
  fn _[_] := _ : Array Nat Elt -> Array .
  fn _[_] : Array Nat -> Elt? .
  ⋮
  ax null[I] = err .
  ax (A[I] := E)[I] = E .
  cx (A[I] := E)[J] = A[J] if I =/= J .
endfmod
```

The empty array is represented by `null`; the expression `a[i] := e` denotes an array identical to `a` except that `e` is stored in position `i`; and `a[i]` is the element of `a` stored in position `i` (if there is no element in this position, `a[i]` yields `err`).

The implementation of buffers using arrays is given by the object level module `BUFFERA`; similarly to `ARRAY`, the sort of the elements to be stored in buffers must have a constant:

```
omod BUFFERA[E :: TRIVE, M :: MAX] is
  pr ARRAY[E] .
  class BufferA .
  at array_ : BufferA -> Array [hidden] .
  at fst_ : BufferA -> Nat [hidden] .
  at next_ : BufferA -> Nat [hidden] .
  at empty?_ : BufferA -> Bool .
  at full?_ : BufferA -> Bool .
```

This module introduces the same visible attributes as `BUFFER`, but has different hidden attributes: `array`, which corresponds to the array containing the elements of a buffer; and the pointers `fst` and `next`, which contain indexes of the array respectively indicating where the first element of the buffer is stored, and where the next element will be inserted. This module also has the same methods as `BUFFER`:

```
  me reset : BufferA -> BufferA .
  me get : BufferA -> Elt .
  me put : BufferA Elt -> BufferA .
  me del : BufferA -> BufferA [hidden] .
```

As the elements of the array between the two indexes (including `fst`, but not `next`) are supposed to be the elements of the buffer, we can say that the buffer is empty when the pointers point at the same position; it is full when the difference between the pointers `next` and `fst` is `max`:

```
  ax empty? B = (fst B) == (next B) .
  ax full? B = (next B) - (fst B) == max .
```

where '`-`' denotes the operation of subtraction on natural numbers. The operation `reset` is implemented by setting both pointers to `0` and the array to `null`:

```
  ax next(reset(B)) = 0 .
  ax fst(reset(B)) = 0 .
  ax array(reset(B)) = null .
```

Deleting the first element of a buffer is implemented by incrementing the `fst` pointer by one; the operation `get` is specified as in `BUFFER`, considering that the first element in a buffer `B` is denoted by `(array B)[fst B]`. Note that `del` and `get` can only be performed if the buffer is not empty:

```
  ax fst(del(B)) = (fst B) + 1 if not(empty? B) .
  acx get(B) = result (array B)[fst B] ; del(B) if not(empty? B) .
```

Lastly, inserting an element in a buffer involves storing it in the next available position of the array, and increasing the `next` pointer by one:

```
    cx next(put(B,E)) = (next B) + 1 if not(full? B) .
    cx array(put(B,E)) = (array B)[next B] := E if not(full? B) .
endomod
```

However, this can only be done if the buffer is not full. Note that all visible operations of `BUFFER` and `BUFFERA` are atomic. As discussed on Section 2.4, this enable us to use our notion of refinement.

The module `BUFFERA` can be instantiated in the same way as `BUFFER` was instantiated in order to obtain `SP` in Section 3.1 (hereafter, let `SP-LIST` denote `SP`):

```
omod SP-ARRAY is
  pr BUFFERA[view to NAT is sort Elt to Nat . sort Elt? to Nat .
                            fn err to 0 . endv,
             view to NAT is fn max to 6 . endv] .
endomod
```

except that we have also to specify that `0` will be the value returned by the indexing operation when the array is not initialized for a particular index.

By integrating `SP-LIST` and `SP-ARRAY` with the core of the proof assistant we can prove refinement involving those two different specifications of buffers. This version of the proof assistant is defined by the following FOOPS object level module:

```
omod ASS-TH is
  ex SP-LIST .
  ex SP-ARRAY .
  ex REF-TH .
endomod
```

which can be loaded into a version of 2OBJ built on top of the extended FOOPS interpreter that supports the simulator, rather than on top of an OBJ interpreter.

It is important to note that for each type (sort or class) of a FOOPS specification, the proof assistant automatically defines a subtype representing the set of first order predicate logic variables ranging over elements of that type[8]. Moreover, all types corresponding to a set of variables are declared subtypes of `Universal-Var`, which is a subtype of `Universal`. For example, the sorts `Buffer-Var` and `BufferA-Var` are automatically defined as subtypes of `Buffer` and `BufferA` when `ASS-TH` is loaded into the proof assistant.

5.2 A Simulation

The definition of `BUFFERA` is based on our intuitions that a buffer can be represented by an array `a`, and indexes `i` and `j`, considering that the elements of the buffer are the elements of `a` from index `i` to index `j`, denoted by `i < a > j`, where

```
    fn _ < _ > _ : Nat Array Nat -> List .
    ax I < A > J = if I >= J then nil else A[I] . (I + 1) < A > J fi .
```

So let us check whether such a buffer represented by array and pointers simulates (refines) a buffer represented by a list `l` whenever `i <= j` and `l` is equal to `i < a > j`.

Following Section 2.4, a first step for checking whether our intuitions are correct is trying to prove that a relation $\mathtt{sim1} \subseteq \mathit{Conf}(\mathtt{BUFFER}) \times \mathit{Conf}(\mathtt{BUFFERA})$ containing states in the following form is a simulation:

$$(\langle e, \mathcal{P}\rangle \quad , \quad \langle e, \mathcal{Q}\rangle),$$

where e is a fully evaluated expression and $\mathcal{P}$ and $\mathcal{Q}$ are database states, respectively associated to `BUFFER` and `BUFFERA`, such that

[8] This is not actually supported by the current implementation of the proof assistant; however, it could be easily extended for that.

- the objects in $\mathcal{P}$ and $\mathcal{Q}$ have the same identifiers;
- either `array b`, `fst b`, and `next b` are defined in $\mathcal{Q}$, or none is defined; also, the attribute `elems b` has a defined value in $\mathcal{P}$ iff the attributes `array b`, `fst b`, and `next b` have defined values in $\mathcal{Q}$;
- the condition `fst b <= next b` is satisfied in $\mathcal{Q}$; and
- the value denoted by `elems b` in $\mathcal{P}$ is equal to the value denoted by

  ```
  (fst b) < (array b) > (next b)
  ```

  in $\mathcal{Q}$,

for any buffer `b` in $\mathcal{P}$.

By extending the module `ASS-TH`, introduced in the previous section, we can define `sim1` in the assistant and check whether it is a simulation. But first we have to represent the modules `SP-LIST` and `SP-ARRAY` as elements of the sort `Specification`:

```
fn SP-LIST : -> Specification [memo] .
ax SP = module-to-specification("SP-LIST") .
fn SP-ARRAY : -> Specification [memo] .
ax SP = module-to-specification("SP-ARRAY") .
```

The definition of `sim1` consists of three parts: constraints on expressions, configurations, and database states:

```
fn sim1 : -> Relation .
[axiom sim1] ax [ P , Q ] [- sim1 =
     ((expression P) expr-sim1 (expression Q))
   ^ (P conf-sim1 Q)
   ^ ((state P) state-sim1 (state Q)) .
```

As formalized below, `sim1` relates only configurations of `SP-LIST` to configurations of `SP-ARRAY`:

```
fn _conf-sim1_ : Configuration Configuration -> Sentence .
[axiom conf-sim1] ax P conf-sim1 Q =
     (is P compatible-with SP-LIST ? = true)
   ^ (is Q compatible-with SP-ARRAY ? = true) .
```

Moreover, only fully evaluated, syntactically equal expressions are related by `sim1`:

```
fn _expr-sim1_ : Universal Universal -> Sentence .
[axiom expr-sim1] ax pP expr-sim1 qQ =
     (pP =~= qQ)
   ^ FE(pP,SP-LIST) = true
   ^ FE(qQ,SP-ARRAY) = true .
```

The following sentence formalizes the constraints on states related by `sim1`, as previously discussed:

```
fn _state-sim1_ : State State -> Sentence .
[axiom state-sim1] ax dP state-sim1 dQ =
     (objects(dP) === objects(dQ))
   ^ (|A| (bv).Buffer . |A| (bv).BufferA .
          (((bv).Buffer =~= (bv).BufferA) ^ ((bv).Buffer In? objects(dP)))
       -> ( ((bv).BufferA In? objects(dQ))
          ^ ((eval (fst(bv) <= next(bv)) in dQ) = true)
          ^ ((eval elems(bv) in dP) =
             (eval (fst(bv) < array(bv) > next(bv)) in dQ))
          ^ (dP defn-atts-sim1(bv,bv) dQ))) .
```

where the sorts `Buffer-Var` and `BufferA-Var` have a constant `bv`, and sentences formed by the operation

```
fn _defn-atts-sim1(_,_)_ : State Buffer BufferA State -> Sentence .
```

are defined as follows:

```
[axiom defn-atts-sim1] ax dP defn-atts-sim1(BL,BA) dQ =
     ((((eval elems(BL) in dP) == elems(BL)) = true) ^
      (((eval array(BA) in dQ) == array(BA)) = true) ^
      (((eval fst(BA) in dQ) == fst(BA)) = true) ^
      (((eval next(BA) in dQ) == next(BA)) = true))
   v ((~ (((eval elems(BL) in dP) == elems(BL)) = true)) ^
      (~ ((((eval array(BA) in dQ) == array(BA)) = true))) ^
      (~ (((eval fst(BA) in dQ) == fst(BA)) = true)) ^
      (~ (((eval next(BA) in dQ) == next(BA)) = true))) .
```

Note that in order to check whether an attribute `at(e)` is not defined in a state `db` it is enough to check whether the expression '`eval at(e) in db`' yields `at(e)`.

In order to prove that `sim1` is a simulation, we have to discharge the following goal:

```
* |- simulation?(sim1,SP-LIST,SP-ARRAY)
```

By applying the inference rule `simulation?` (see Section 4.2.6) to the goal above, we obtain the following subgoal:

```
* |- |A| p . |A| q .
         ([ p , q ] [- sim1) ->
         |A| q' . (q -->{SP-ARRAY} q') ->
                  |E| p' . (p -->*{SP-LIST} p') ^ ([ p' , q' ] [- sim1)
```

Then we can apply the rule `Ai` twice and then `impi`, generating the subgoal

```
([ p , q ] [- sim1) |-
     |A| q' . (q -->{SP-ARRAY} q') ->
              |E| p' . (p -->*{SP-LIST} p') ^ ([ p' , q' ] [- sim1)
```

In order to unfold the definition of `sim1`, we use equational reasoning by applying the rule

```
app('ASS-TH,'sim1,fwd,at,nosub,top,1).
```

Thus we obtain the subgoal

```
([ p , q ] [- sim1) ;
(expression(p) expr-sim1 expression(q)) ^ (p conf-sim1 q) ^
(state(p) state-sim1 state(q)) |-
     |A| q' . (q -->{SP-ARRAY} q') ->
              |E| p' . (p -->*{SP-LIST} p') ^ ([ p' , q' ] [- sim1)
```

which, after applying `ande(2)`, results in the following subgoal:

```
... ; expression(p) expr-sim1 expression(q) ; ... |-
     |A| q' . (q -->{SP-ARRAY} q') ->
              |E| p' . (p -->*{SP-LIST} p') ^ ([ p' , q' ] [- sim1)
```

Now we can unfold the definition of `expr-sim1` by applying the rule

```
app('ASS-TH,'expr-sim1,fwd,at,nosub,top,3)
```

yielding the subgoal

```
... ; (expression(p) =~= expression(q)) = true ^
FE(expression p,SP-LIST) = true ^ FE(expression q,SP-ARRAY) = true |-
     |A| q' . (q -->{SP-ARRAY} q') ->
              |E| p' . (p -->*{SP-LIST} p') ^ ([ p' , q' ] [- sim1)
```

which, by applying `ande(5)` and then `ande(7)`, can be reduced to the subgoal

```
... ; FE(expression p,SP-LIST) = true ^ FE(expression q,SP-ARRAY) = true ;
... ; FE(expression p,SP-LIST) = true ; FE(expression q,SP-ARRAY) = true |-
     |A| q' . (q -->{SP-ARRAY} q') ->
              |E| p' . (p -->*{SP-LIST} p') ^ ([ p' , q' ] [- sim1)
```

Then, as the expression of `q` is fully evaluated, we can simply apply the rule `simulation-fe(10)` (see Section 4.2.6) in order to complete the proof.

We have presented those proof steps in order to illustrate how the proof assistan works. However, in practice, the kind of proof presented in this section can be automatically performed using simple user-defined tactics encapsulating the proof steps presented above.

5.3 Another Simulation

Now we consider the proof that a simple extension of the relation `sim1` is also a simulation. First we define the relation, in the same style used to define `sim1`:

```
fn sim2 : -> Relation .
[axiom sim2] ax [ P , Q ] [- sim2 =
      ((expression P) expr-sim2 (expression Q))
    ^ (P conf-sim1 Q)
    ^ ((state P) state-sim1 (state Q)) .
```

Note that the definitions of the relations `sim1` and `sim2` only differ with respect to the constraints on expressions. Expressions related by `sim2` must be syntactically equal and either fully evaluated or equal to `put(b,e)`, where `e` is a fully evaluated term of sort `Nat` and `b` is an identifier of class `Buffer` or `BufferA`. In the encoding, this is formalized by the following declarations:

```
fn _expr-sim2_ : Universal Universal -> Sentence .
[axiom expr-sim2] ax pP expr-sim2 qQ =
      (pP expr-sim1 qQ) v (pP expr-sim2-aux qQ) .

fn _expr-sim2-aux_ : Universal Universal -> Sentence .
[axiom expr-sim2-aux] ax pP expr-sim2-aux qQ =
    (pP =~= put((b).Buffer,e)) ^ (qQ =~= put((b).BufferA,e)) .
```

As `b` and `e` are arbitrary, the proof that `sim2` is also a proof that the extension of `sim2` that relates expressions in the form `put(b,e)`, for any `b` and `e`, is also a simulation.

In order to prove that `sim2` is a simulation, we use properties of the operation `++` on database states. As those properties are actually specific to particular FOOPS specifications, they were not introduced with the definition of `++` on Section 4.2.1. In fact, they should be defined by the persons who instantiate the proof assistant for reasoning about particular specifications; they are responsible for the correctness of those properties. For instance, the following rule indicates that a database state `st` related to `SP-LIST` is equal to the merge of itself with the state of `SP-LIST` containing only an uninitialized object `b`, provided that `b` is in `st` and the attribute `elems` of `b` is not defined in `st`:

```
fn db-inst-bl : State Buffer Nat Nat Nat -> Rule .
cx db-inst-bl(DB,BL,M,N,I) (H |- G) =
   H ; DB = (DB ++ add-object BL to initial-state(SP-LIST)) |- G
if hyp(N,H) == (BL In? objects(DB)) and
   hyp(M,H) == (((eval elems(BL) in DB) == elems(BL)) = true) and
   matches((is < pP , DB > compatible-with SP-LIST ?) = true,hyp(I,H)) .
```

Similarly, the following inference rule formalizes the corresponding property for SP-ARRAY:

```
fn db-inst-ba : State BufferA Nat Nat Nat -> Rule .
cx db-inst-ba(DB,BA,M,N,I) (H |- G) =
   H ; DB = (DB ++ add-object BA to initial-state(SP-ARRAY)) |- G
if hyp(N,H) == (BA In? objects(DB)) and
   hyp(M,H) == (((((eval array(BA) in DB) == array(BA)) = true)) ^
               (((eval fst(BA) in DB) == fst(BA)) = true) ^
               (((eval next(BA) in DB) == next(BA)) = true)) and
   matches((is < pP , DB > compatible-with SP-ARRAY ?) = true,hyp(I,H)) .
```

The function matches checks whether its second argument is an instance of the first.

Now we consider that a database state st related to SP-LIST has an object b with its attribute elems defined. So we can conclude that st is equal to the merge of itself with the state of SP-LIST containing only the object b having its attribute elems set to an arbitrary symbolic value, provided that we assume that this symbolic value is equal to the value of elems(b) in st. This property is formalized as follows:

```
fn db-inst-bl : State Buffer List Nat Nat Nat -> Rule .
cx db-inst-bl(DB,BL,L,M,N,I) (H |- G) =
   H ; L = (eval elems(BL) in DB) ;
       DB = (DB ++ update-attribute elems(BL) with L in
                    (add-object BL to initial-state(SP-LIST))) |- G
if hyp(N,H) == (BL In? objects(DB)) and
   hyp(M,H) == (~ (((eval elems(BL) in DB) == elems(BL)) = true)) and
   matches((is < pP , DB > compatible-with SP-LIST ?) = true,hyp(I,H)) and
   not(L in freevars(H |- G)) .
```

Similarly, we can formalize the corresponding property for SP-ARRAY:

```
 fn db-inst-ba : State BufferA Nat Nat Array Nat Nat Nat -> Rule .
 cx db-inst-ba(DB,BA,‘M,‘N,A,M,N,I) (H |- G) =
    H ; A = (eval array(BA) in DB) ; ‘M = (eval fst(BA) in DB) ;
        ‘N = (eval next(BA) in DB) ;
        DB = (DB ++ update-attribute fst(BA) with ‘M in
                    (update-attribute next(BA) with ‘N in
                      (update-attribute array(BA) with A in
                        (add-object BA to initial-state(SP-ARRAY))))) |- G
if hyp(N,H) == (BA In? objects(DB)) and
   hyp(M,H) == ((~ ((((eval array(BA) in DB) == array(BA)) = true))) ^
               (~ (((eval fst(BA) in DB) == fst(BA))  = true)) ^
               (~ (((eval next(BA) in DB) == next(BA))  = true))) and
   matches((is < pP , DB > compatible-with SP-ARRAY ?) = true,hyp(I,H)) and
   not(A in freevars(H |- G)) and
   not(‘M in freevars(H |- G)) and not(‘N in freevars(H |- G)) .
```

We can now sketch the proof that the relation sim2 is a simulation. We start with the goal

```
* |- simulation?(sim2,SP-LIST,SP-ARRAY)
```

and the proof follows as the proof that sim1 is a simulation, except that after applying the rule Ai twice, we should apply the tactic

```
     conf-inst(p,pp,dp) THEN app(1,fwd,within,nosub,top)
THEN conf-inst(q,qq,dq) THEN app(2,fwd,within,nosub,top)
```

which respectively instantiates p and q with < pp , dp > and < qq , dq >, where qq and dq respectively are elements of sorts Universal-Var and State-Var. Then the proof follows by applying impi, other basic proof steps, and unfolding the definitions of sim2 and expr-sim2, resulting in the following subgoal:

```
p = < pp , dp > ; ... ;
  (expression(< pp , dp >) expr-sim1 expression(< qq , dq >))
v (expression(< pp , dp >) expr-sim2-aux expression(< qq , dq >)) |-
  |A| q' . (< qq , dq > -->{SP-ARRAY} q') ->
     |E| p' . (< pp , dp > -->*{SP-LIST} p') ^ ([ p' , q' ] [- sim2)
```

Then by using the rule `ore` we split the proof in two parts, according to the disjunction shown in the previous goal. We can complete the proof of one of the parts in the same way as we have completed the proof that `sim1` is a relation. In the other part, we can proceed with the proof by unfolding the definition of `expr-sim2-aux` and reducing some terms until we have the sentences `qq =~= put(b,e)` and `pp =~= put(b,e)` in the hypothesis list. Thus we can apply the rule `synt-eq-app` (see Section 4.2.4) twice and obtain the following subgoal:

```
... ; pp =~= put(b,e) ; ... |-
  |A| q' . (< put(b,e) , dq > -->{SP-ARRAY} q') ->
     |E| p' . (< put(b,e) , dp > -->*{SP-LIST} p') ^ ([ p' , q' ] [- sim2)
```

At this point we split the proof in two parts by applying the case analysis tactic

```
CASE(b In? objects(dp))
```

The case where '`~ b In? objects(dp)`' is trivial since, by the definition of `state-sim1`, this implies that '`~ b In? objects(dq)`'; therefore there is no transition from `< put(b,e) , dq >`. We omit the details in this article. In the case where '`b In? objects(dp)`', we introduce another case analysis by applying the following tactic:

```
CASE(((eval elems(b) in dp) == elems(b)) = true)
```

Here we just consider the case where `b` is not initialized in the database state `dp`. The proof of the other case follows in a similar way, except that the alternative version of the inference rules `db-inst-bl` and `db-inst-ba` would be used. We proceed by unfolding the definition of `conf-sim1` and applying the rule `db-inst-bl(dp,SP-LIST,b,m,n,o)`, where `m`, `n`, and `o` are natural numbers corresponding to specific positions in the hypothesis lisp of the current goal. Then we generate the following subgoal:

```
... ; dp = (dp ++ (add-object b to initial-state(SP-LIST))) |-
    |A| q' . (< put(b,e) , dq > -->{SP-ARRAY} q') ->
       |E| p' . (< put(b,e) , dp > -->*{SP-LIST} p') ^ ([ p' , q' ] [- sim2)
```

We can now use the rule `app(k,fwd,within,nosub,top)`, where `k` is the length of the hypothesis list of the goal above. After that and some basic proof steps, we can unfold the definitions of `state-sim1` and `defn-atts-sim1`. Then we are able to apply the rule

```
db-inst-ba(dq,SP-ARRAY,b,x,y,z)
```

for some specific natural numbers `x`, `y`, and `z`, resulting in the following subgoal:

```
... ; dq = (dq ++ (add-object b to initial-state(SP-ARRAY))) |-
  |A| q' . (< put(b,e) , dq > -->{SP-ARRAY} q') ->
     |E| p' .  (< put(b,e) , dp ++ (add-object b to initial-state(SP-LIST)) >
                -->*{SP-LIST} p')
              ^ ([ p' , q' ] [- sim2)
```

We then use the rule `app(w,fwd,within,nosub,top)`, where `w` is the length of the hypothesis list of the goal above. After reducing the database expressions merged with `dp` and `dq`, we can apply the rule `simulation-db`. This generates three subgoals:

```
* |- atomic-exps-terminate?(put(b,e),SP-LIST)
* |- atomic-exps-terminate?(put(b,e),SP-ARRAY)
* |- is < put(b,e) , b > related-to < put(b,e), b >
     by(sim2,SP-LIST,SP-ARRAY,dp,dq)
```

where the state having only the uninitialized object `b'` is represented by '`b'`'.

We do not consider the proof of the first two subgoals since the encoding of the refinement theory described in this article has not enough information about termination. In order to prove the third subgoal, we have to unfold the definition of `is-related-to` (see Section 4.2.6), generating the subgoal

```
* |- are simulate(SP-ARRAY,< put(b,e) , b >,*) derived-from < put(b,e) , b >
     by(sim2,SP-LIST,dp,dq)
```

which can be discharged by reducing the expression formed by `simulate`, unfolding the definition of `are-derived-from-e`, and applying the rule `Truei`. The definition of `are-derived-from-e` can be unfolded because the simulator yields `{}`, since `b` is not initialized in the database state used for evaluation.

6 Conclusions

In this article we presented tools for mechanizing FOOPS refinement proofs. In particular, we described the main features of a symbolic simulator and a proof assistant for FOOPS and illustrated how they can be used for mechanizing refinement proofs.

The symbolic simulator automatically indicates state transitions resulting from the evaluation of an expression in a database state, which may have attributes instantiated with either actual values or symbolic values. In fact, the simulator is very useful for checking possible transitions from a set of states having common properties, since this set can be represented by a state having attributes instantiate with symbolic values. As transitions depending on arbitrary properties of symbolic values cannot be automatically justified, the simulator has to indicate the conditions under which a particular transition is possible. We also described in which circumstances the simulator conforms to FOOPS operational semantics [2] and, therefore, can be used for proving refinement.

The proof assistant supports mechanical formal reasoning based on an encoding of FOOPS refinement theory [3]. This includes the encoding of first order predicate logic with equality presented in [21]. We described some of the aspects of the encoding that are useful for proving refinement. The remaining details of the refinement theory can be encoded in a similar way by using essentially the same techniques.

In addition to the encoding of the concepts of the refinement theory, the proof assistant also provides special inference rules corresponding to properties that are quite useful for mechanically proving refinement. For instance, a special rule establishes that the proof assistant can access the "model checking" capabilities of the symbolic simulator. Also, special "database instatiation" rules specify that a database state is equal to a merge of itself with any "closed" sub-database. Moreover, other rules specify that the evaluation of an expression in a database state formed by the merge of two others is equivalent to its evaluation in just one of the states, provided that the object identifiers referenced in the expressions are in that state.

We have also illustrated how the proof assistant can be used for proving refinement using different specifications of bounded buffers as example. We explained the steps necessary for mechanically proving refinement:

- load FOOPS theories of interest into the proof assistant;
- integrate those theories to the core of the proof assistant;
- encode candidate simulation relations using equations;
- use inference rules to encode "database instantiation" theorems for the theories of interest, if necessary; and
- prove that the candidate relations are simulations.

In the exaples presented in this article, we only indicated the steps necessary for interactively completing proofs. However, a lot of refinement proofs can be mostly automated by defining suitable tactics in 2OBJ. In fact, some proofs can be automatically done using the proof assistant. In most cases, the assistant helps to reduce object level theorems to functional level theorems, which result from proofs that a configuration pair belongs to a given relation, and proofs that transitions from one state can happen whenever transitions from another state can happen. In general, those functional level theorems are related to non trivial functional theories, so that they cannot be automatically proved.

The symbolic simulator is an OBJ [12] program, whereas the proof assistant is implemented by an encoding of FOOPS refinement theory in the 2OBJ meta-logical framework [10, 21]. The computational facilities supported by those tools have proved to be quite useful for mechanically proving refinement of concurrent object oriented programs: OBJ's term rewriting facilities, which efficiently implement equational reasoning; and 2OBJ's engine for applying inference rules and tactics.

In order to encode the refinement theory in 2OBJ, several factors were considered: ease of proving, automation degree, and computational efficiency. In particular, we briefly analyzed the use of inference rules, general axioms, and equations for encoding properties in 2OBJ. Observing the factors above, wherever adequate we used equations; otherwise, we used inference rules in most cases. We also presented alternatives for integrating the simulator with the proof assistant and justified our design decision of sharing part of the encoding of the objects of the refinement theory with the encoding of objects of the symbolic simulator. Whereas this approach is efficient and avoids a lot of code duplication, it blurs the conceptual distinction between "semantic" and "syntactic" objects.

The adequacy of OBJ for implementing prototypes of simulation tools has been discussed and confirmed in [22] and [16]. In this work we followed a similar approach. OBJ is also a quite effective tool for formal reasoning [6, 8]. Because of its simple semantics and efficient decision procedures for equational logic, we should ideally use OBJ for formally reasoning about refinement as well. However, it seems that some of the concepts of the refinement theory cannot be directly defined by equations as supported by OBJ. For instance, the definition of simulation cannot be easily encoded in equational logic. That is why we use the 2OBJ system for implementing the proof assistant.

The tools described in this article are prototypes. Nevertheless, they were quite useful for partially validating our approach for refinement and supporting its use in small projects. In order to be used for large software projects, those tools should be more efficient, provide more features, and offer graphical user interfaces. We believe that the description and use of the prototypes will be of great importance for producing industrial strength versions of the tools.

References

[1] Paulo Borba. *Semantics and Refinement for a Concurrent Object Oriented Language*. PhD thesis, Oxford University, Computing Laboratory, Programming Research Group, July 1995.

[2] Paulo Borba and Joseph Goguen. An operational semantics for FOOPS. In Roel Wieringa and Remco Feenstra, editors, *International Workshop on Information Systems—Correctness and Reusability, IS-CORE'94*. Vrije Universiteit, Amsterdam, September 1994. A longer version appeared as Technical Monograph PRG-115, Oxford University, Computing Laboratory, Programming Research Group, November 1994.

[3] Paulo Borba and Joseph Goguen. Refinement of concurrent object oriented programs. In Stephen Goldsack and Stuart Kent, editors, *Formal Methods and Object Technology,* Chapter 11. Springer-Verlag, To appear 1996. Also appeared as Technical Report TR-18-95, Oxford University, Computing Laboratory, Programming Research Group, November 1995.

[4] George Cleland and Donald MacKenzie. Inhibiting factors, market structure and the industrial uptake of formal methods. In *International Workshop on Industrial Strength Formal Specification Techniques*. IEEE Computer Society Press, Boco Ratan, Florida, April 1995.

[5] Edsger Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[6] Joseph Goguen. OBJ as a theorem prover, with application to hardware verification. In V.P. Subramanyan and Graham Birtwhistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 218–267. Springer, 1989. Also Technical Report SRI-CSL-88-4R2, SRI International, Computer Science Lab, August 1988.

[7] Joseph Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison Wesley, 1989.

[8] Joseph Goguen. Proving and rewriting. In Héléne Kirchner and Wolfgang Wechler, editors, *Proceedings, Second International Conference on Algebraic and Logic Programming*, pages 1–24. Springer, 1990. Lecture Notes in Computer Science, Volume 463.

[9] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987.

[10] Joseph Goguen, Andrew Stevens, Keith Hobley, and Hendrik Hilberdink. 2OBJ, a metalogical framework based on equational logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69–86, 1992. Also in *Mechanized Reasoning and Hardware Design*, edited by C.A.R. Hoare and M.J.C. Gordon, Prentice-Hall, 1992, pages 69–86.

[11] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Technical Report RC 6487, IBM T.J. Watson Research Center, October 1976. In *Current Trends in Programming Methodology, IV*, Raymond Yeh, editor, Prentice-Hall, 1978, pages 80–149.

[12] Joseph Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Lab, August 1988. Revised version to appear with additional authors José Meseguer, Kokichi Futatsugi and Jean-Pierre Jouannaud, in *Applications of Algebraic Specification using OBJ*, edited by Joseph Goguen.

[13] Joseph Goguen and David Wolfram. On types and FOOPS. In Robert Meersman, William Kent, and Samit Khosla, editors, *Object Oriented Databases: Analysis, Design and Construction*, pages 1–22. North Holland, 1991. Proceedings, IFIP TC2 Conference, Windermere, UK, 2–6 July 1990.

[14] Cliff Jones. *Systematic Software Development using VDM.* Prentice-Hall, second edition, 1986.

[15] Carroll Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.

[16] Kazuhito Ohmaki, Koichi Takahashi, and Kokichi Futatsugi. A LOTOS simulator in OBJ. In Juan Quemada, Jose Manas, and Enrique Vazquez, editors, *Proceedings of FORTE'90*, pages 653–656, Madrid, 1990. IFIP TC6.1, Ministerio de Industria y Energia.

[17] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, September 1981.

[18] Lucia Rapanotti and Adolfo Socorro. Introducing FOOPS. Technical Report PRG-TR-28-92, Oxford University, Computing Laboratory, Programming Research Group, November 1992.

[19] David Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

[20] Adolfo Socorro. *Design, Implementation, and Evaluation of a Declarative Object Oriented Language.* PhD thesis, Oxford University, Computing Laboratory, Programming Research Group, December 1993.

[21] Andrew Stevens, Keith Hobley, and Joseph Goguen. Mechanised theorem proving with 2OBJ: A tutorial introduction. To appear, 1996.

[22] Koichi Takahashi, Kazuhito Ohmaki, and Kokichi Futatsugi. Rapid prototyping of LOTOS's behavioral expressions in OBJ. In *Proceedings of an International Conference organized by the IPSJ to Commemorate the 30th Anniversary*, pages 1–7. Information Processing Society of Japan, 1990.

Contents

1 Introduction **1**

2 Overview of FOOPS **2**
2.1 Functional Level 2
2.2 Object Level 3
2.3 Operational Semantics 6
2.4 Refinement 6

3 A Symbolic Simulator for FOOPS **8**
3.1 Specifications 9
3.2 Expressions 9
3.3 Database States 9
3.4 Sentences 10
3.5 Simulations 11
3.6 Correctness 13

4 A Proof Assistant for FOOPS **13**
4.1 First Order Predicate Logic 13
4.1.1 Sentences 14
4.1.2 Sequents 14
4.1.3 Inference Rules 14
4.1.4 Tactics 15
4.2 Refinement Theory 16
4.2.1 Database States 16
4.2.2 Configurations 17
4.2.3 Relations on Configurations 17
4.2.4 Equational Reasoning 18
4.2.5 State Transitions 19
4.2.6 Simulation Relations 19

5 Proving Refinement **21**
5.1 Specifications 21
5.2 A Simulation 23
5.3 Another Simulation 26

6 Conclusions **29**