Programming Research Group

PRG-TR-18-95

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Refinement of Concurrent Object Oriented Programs

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

November 1995

Abstract

FOOPS is a concurrent object oriented language. Based on FOOPS operational semantics, we introduce a notion of refinement for states of FOOPS systems together with a proof technique for proving refinement. Using this notion, we define refinement of FOOPS (method) expressions and programs. Although we focus on FOOPS, our definition of refinement is independent of this language.

We also illustrate the use of refinement for stepwise formal development of programs in FOOPS. Based on that, we give an overall evaluation of our approach and comment on alternative approaches for the refinement of concurrent object oriented programs, concluding that none of them provide both a general definition of refinement and an effective proof technique such as the one presented here.

1 Introduction

FOOPS is a concurrent object oriented specification language having an executable subset [Goguen and Meseguer, 1987, Rapanotti and Socorro, 1992]. It includes a functional language derived from OBJ [Goguen and Winkler, 1988], which is a first order, purely functional language providing an algebraic style for the specification, rapid prototyping, and implementation of abstract data types. OBJ also supports

*Supported by CAPES, Brazil, Grant 2184-91-8, and the CEC under ESPRIT-2 BRA Working Group 6071, IS-CORE (Information Systems COrrectness and REusability). 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.

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 [Goguen, 1989], and many additional features. Here we consider an extension of FOOPS with features for specifying systems of distributed and autonomous objects. Essentially, this extension allows the definition of non terminating (autonomous) methods, and provides constructors (called method combiners) for expressing concurrency, nondeterminism, and atomic execution.

An operational semantics [Plotkin, 1981] for this extension of FOOPS is given in [Borba and Goguen, 1994, Borba, 1995]; it describes the behaviour of FOOPS *method expressions* (commands). Based on this semantics, here we define a notion of refinement that indicates whether the behaviour of a FOOPS system is simulated by another system, so that we can replace the latter by the former without noticing any difference; this extends to (method) expressions, programs, and specifications as well. Although we focus on FOOPS, our definition of refinement is quite independent of this language; it essentially depends on minimal assumptions about the operational semantics, which are satisfied by the FOOPS operational semantics given in [Borba and Goguen, 1994, Borba, 1995].

We also illustrate the use of refinement for stepwise formal development of programs in FOOPS; we use examples involving memory cells and different implementations of buffers. Furthermore we show how the definition of refinement can be used to derive laws of FOOPS method combiners. We include proofs of refinement in order to give a better insight on our approach, by showing how it works in detail, what aspects it considers, how complex the proofs are, and by analysing to what extent the proofs can be mechanized.

Similarly to [Milner, 1989], [Mosses, 1992], and [Hennessy, 1988], we use operational semantics' framework to define refinement and to reason about specifications. So, in order to develop notions of refinement for FOOPS, we adapt ideas from [Milner, 1989], [Milner, 1971], and [Park, 1981], where a general approach for defining refinement (and equivalence) of transition systems is developed and justified; this approach is quite simple and provides an effective proof technique for verifying refinement. We also consider aspects of data refinement [Hoare, 1972, Jifeng *et al.*, 1986, Goguen and Malcolm, 1994].

This chapter is organized in the following way. First, we give an overview of FOOPS. Then we introduce and justify the notion of refinement of states and its associated proof technique. After that we derive basic properties of refinement of states. Then we introduce the notions of refinement of expressions and specifications, deriving basic properties of those relations as well. Following that, the proof technique is used for verifying refinement of several examples. Based on that, we identify different aspects considered by the definition of refinement, evaluate our approach, and comment on alternative approaches for refinement of concurrent object oriented programs.

2 Overview of FOOPS

FOOPS extends OBJ with some concepts from object oriented programming. This motivates two central design decisions (see [Goguen and Wolfram, 1991, Goguen and Meseguer, 1987, Rapanotti and Socorro, 1992, Socorro, 1993] for more details about the design of FOOPS): 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.

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 .
```

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 .
```

The meaning of these functions is given by axioms (equations).

```
  ⋮
  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`.

2.2.1 Axioms

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`.

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.

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 .
```

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. This is called the *frame assumption*; it avoids writing equations indicating that some attributes are not updated.

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.

2.2.2 Method Combiners

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

2.2.3 Object Creation and Deletion

Dynamic object creation and deletion are respectively provided in FOOPS 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 choosen 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. Contrasting to **new**, the argument of **remove** might be an arbitrary expression, it does not have to be an object identifier; however, it is supposed to yield an identifier.

Objects can also be introduced together with the definition of their associated class, where values for their stored attributes are specified. These are called specified objects and are particularly useful when defining classes of recursive data structures such as stacks and linked lists. Specified objects have the same status as objects created at runtime; this means that they can be modified and removed.

3 Refinement for FOOPS

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 behaviour 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.

Here we want to investigate whether a state is simulated (refined) by another state. 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 thought, the experiments done with P might let us make more observations than if Q were used.

We have to define the notions of experiment and observation that will be of interest for us. The only kind of experiment that we can make with a FOOPS system is to invoke *visible* operations 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;
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.

Note that the relation of simulation between states depends 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.

Now we make more precise what experiments are of interest for us. First we consider that only the following are **operations**: methods, attributes, and object creation and deletion routines. In addition to *operations*, an user of a FOOPS system can require the evaluation of functions. However, the evaluation of functions does not directly change the system database, nor depends on the system state; that's why they are not considered for experiments. We use $Ops(\mathsf{P})$ to denote the family of *operation* symbols associated to a specification P; this family is indexed by the rank of the operations.

Second, we assume that the arguments of visible operations are object identifiers or elements of the ADTs corresponding to the sorts in the arity of those operations. Although complex expressions formed by identifiers, visible operations, and functions could be used as arguments, this would have the same effect as invoking a

sequence of simpler expressions as experiments; so it is redundant to consider that possibility here.

Hereafter the family of sets of **experiments** (expressions formed by the application of a visible operation to object identifiers and elements of ADTs) associated to a specification P is denoted by $\mathcal{E}(\mathsf{P})$. This family is indexed by the sorts or classes associated to experiments; so we use $\mathcal{E}_u(\mathsf{P})$ to denote the element of the family $\mathcal{E}(\mathsf{P})$ corresponding to type u.

Note that $\mathcal{E}(\mathsf{P}) \subseteq \mathcal{E}(\mathsf{Q})$ also indicates that the states of systems specified by Q have the same visible attributes as the states associated to P, since we consider that attributes are operations. Furthermore, it is easy to see that in order to check whether $\mathcal{E}(\mathsf{P}) \subseteq \mathcal{E}(\mathsf{Q})$, it is enough to check whether

1. the visible operations of P are also visible operations of Q, having the same name and rank;

2. the object identifiers associated to P are also associated to Q; and

3. the ADTs of P corresponding to sorts of the results yielded by experiments, or to sorts having elements used to make experiments, should also be ADTs of Q.

The last condition also guarantees that states associated to P may possibly be simulated by states associated to a specification Q. In general, this is not implied by $\mathcal{E}(\mathsf{P}) \subseteq \mathcal{E}(\mathsf{Q})$, since the functional terms used to form experiments of P might have a different meaning in Q.

As can be concluded from the discussion above, we assume that functional specifications included in FOOPS specifications have an initial semantics (i.e., they denote ADTs). In this way, it is easier to concentrate on the issues related to refinement at the object level; that's our main focus here. Also it is only a small extension to consider refinement of FOOPS specifications containing functional specifications with initial and loose semantics.

Condition 3 introduced above can be expressed by the following predicate: $\mathsf{P} \leq_{PS(\mathsf{P})} \mathsf{Q}$, which holds if the ADTs specified by P when restricted to $PS(\mathsf{P})$ are also ADTs of Q, where $PS(\mathsf{P})$ denotes the set of **primary** sorts of P; that is, the set formed by the sorts of P having terms yielded by experiments or used to make experiments.

According to the notion of simulation discussed in this section (see Item 3 from the list of aspects that we can observe from states), 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 behaviour associated to Q is a particular case of the observable behaviour associated to P; that's the essence of simulation.

3.1 Simulations

Now we start to formalize the finer of the two notions of simulation discussed so far. In order to represent states and reason about state transitions due to evaluating expressions, we use the framework provided by operational semantics. States are represented by configurations of the operational semantics, that is, 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". We use $Conf(\mathsf{P})$ to denote the set of all configurations associated to a specification P, and $I(\mathsf{P})$ to denote the family of sets of object identifiers associated to P (each set is formed by the symbols that can be used to identify objects of a given class).

State transitions are specified by the transition relation

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

which defines the operational semantics associated to P; it indicates how an expression is evaluated in a database state. When not confusing, we write $\rightarrow$ instead of $\rightarrow_{\mathsf{P}}$. Similarly, the relation $\rightarrow^*$ denotes the transitive, reflexive closure of $\rightarrow$; also, we write $P \not\rightarrow$ if there is no P' such that $P \rightarrow P'$ (this means that the expression in P cannot be further evaluated). Lastly, if $P \rightarrow^* P'$ then we say that P' is a $\rightarrow^*$-**derivative** of P; a corresponding terminology is used for $\rightarrow$.

We use the term "fully evaluated expression" to refer to object identifiers and *evaluated* functional expressions (elements of ADTs). The notation $\mathcal{FE}(e)$ indicates 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 $\mathcal{FE}_{\mathsf{P}}(e)$. For fully evaluated terms p and q respectively related to specifications P and Q, equality is denoted by $p =_{(\mathsf{P},\mathsf{Q})} q$; it holds if p and q are the same object identifiers and $I(\mathsf{P}) \subseteq I(\mathsf{Q})$, or if p and q are the same element of an ADT that is defined by both P and Q.

The notion of simulation between states is defined as the union of all relations having the following special property: if the pair (P, Q) is in the relation then whatever can be observed by performing experiments with Q can also be observed by performing experiments with P. Note that this property is exactly what guarantees that one state simulates another.

Relations having this special property are called simulations. Let us formalize this concept now. Hereafter we use P and Q for configurations, $\mathcal{P}$ and $\mathcal{Q}$ for their respective database states, and p and q for their respective expressions.

Definition 3.1 Let P and Q be specifications such that $\mathcal{E}(\mathsf{P}) \subseteq \mathcal{E}(\mathsf{Q})$, $I(\mathsf{P}) \subseteq I(\mathsf{Q})$, and $\mathsf{P} \leq_{PS(\mathsf{P})} \mathsf{Q}$. A relation $\mathcal{S} \subseteq Conf(\mathsf{P}) \times Conf(\mathsf{Q})$ is a (P, Q)-**simulation** if $(P, Q) \in \mathcal{S}$ implies the following:

1. Whenever $Q \rightarrow Q'$ then, for some P', $P \rightarrow^* P'$ and $(P', Q') \in \mathcal{S}$.
2. If $Q \not\rightarrow$ then, for some P',
   (a) $P \rightarrow^* P' \not\rightarrow$;

(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 $\langle exp,\ \mathcal{Q}\rangle \rightarrow^* Q' \not\rightarrow$ then, for some P',

(a) $\langle exp,\ \mathcal{P}\rangle \rightarrow^* P' \not\rightarrow$;

(b) $(P',\ Q') \in \mathcal{S}$; and

(c) $(\langle p,\ \mathcal{P}'\rangle,\ \langle q,\ \mathcal{Q}'\rangle) \in \mathcal{S}$.

□

Those conditions reflect the ideas that we have introduced about simulation of states. For example, 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.

Lastly, Item 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.

Also, it may seem that the formal definition does not require visible attributes to have the same value in related states. However, this is indirectly done by condition 3, since we consider that attributes are used to form experiments.

3.1.1 General Assumptions

Our notion of simulation is based on some assumptions which have not been made explicit so far. Basically, we assume that operations used as experiments are atomic and terminating; that's why the formal definition of simulation does not consider the intermediate states reached during the evaluation of experiments, and assumes that the evaluation always terminates (see condition 3 of the definition of simulation). However, note that the evaluation of experiments might take more than one transition; the extra transitions correspond to the replacement of methods for their definition, as long as the transitions do not change the database state nor depend on it.

Without those assumptions, it is not possible to analyse the effect of experiments in isolation; it would be necessary to consider the effect of experiments consisting of a group of operations being concurrently executed, including an arbitrary number

of instances of the same operations. Indeed, it would be very difficult to reason about that.

Furthermore, note that the assumptions introduced above do not imply that all available methods have to be atomic and terminating; indeed, nonatomic and nonterminating methods can be defined as hidden operations. As discussed in [Borba, 1995], visible methods can initiate nonterminating and nonatomic computations by invoking hidden methods as expressions to be evaluated in the background. As usual, nonatomic computations are subjected to interference from the evaluation of other expressions being concurrently evaluated.

Those assumptions suggest a programming methodology: visible methods should be atomic and terminating (in general, it is simple to check that). In fact, it might be very difficult to reason about and predict the behaviour of programs allowing the parallel execution of many nonatomic methods (it is necessary to reason about all possible interleaved interferences caused by those methods). So we suggest that nonatomic methods should only be used in a controlled way, being invoked via visible atomic operations, as discussed above.

3.1.2 Properties

Simulations have interesting properties; some of them we will describe here. Those properties are particularly useful when using simulations to reason about refinement, as we will confirm later. For instance, the next theorem reveals a special (and obvious) simulation.

Theorem 3.1 For a specification P, the identity relation on $Conf(\mathsf{P})$, denoted $\mathcal{ID}_{\mathsf{P}}$, is a (P, P)-simulation.

Proof: It easily follows from the reflexivity of $\subseteq$, $\leq_{PS(\mathsf{P})}$, $\mathcal{ID}_{\mathsf{P}}$, and $=_{(\mathsf{P}, \mathsf{P})}$, and by observing that

$$Q \rightarrow Q' \text{ implies } Q \rightarrow^* Q'$$

and

$$Q \not\rightarrow \text{ implies } Q \rightarrow^* Q \not\rightarrow.$$

□

As indicated by the following theorem, union of relations preserve simulations.

Theorem 3.2 Let P and Q be specifications. For (P, Q)-simulations $\mathcal{S}$ and $\mathcal{S}'$, $\mathcal{S} \cup \mathcal{S}'$ is a (P, Q)-simulation.

Proof: Let us assume that

$$(P, Q) \in \mathcal{S} \cup \mathcal{S}'.$$

Therefore

$$(P, Q) \in \mathcal{S} \text{ or } (P, Q) \in \mathcal{S}'.$$

So we split the proof in two cases. First assume that $(P, Q) \in \mathcal{S}$. Now let $Q \rightarrow Q'$. As $(P, Q) \in \mathcal{S}$, for some P', we have

$$P \rightarrow^* P' \text{ and } (P', Q') \in \mathcal{S}.$$

This trivially implies that $(P',\ Q') \in \mathcal{S} \cup \mathcal{S}'$.

Now let $Q \not\rightarrow$. As $(P,\ Q) \in \mathcal{S}$, for some P', we have

$$P \rightarrow^* P' \not\rightarrow, \qquad \mathcal{FE}_{\mathsf{P}}(p') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(q),$$
$$\mathcal{FE}_{\mathsf{P}}(p') \Rightarrow p' =_{(\mathsf{P},\,\mathsf{Q})} q, \qquad \text{and } (P',\ Q) \in \mathcal{S}.$$

Hence $(P',\ Q) \in \mathcal{S} \cup \mathcal{S}'$.

Finally let $\langle exp,\ \mathcal{Q} \rangle \rightarrow^* Q' \not\rightarrow$, for any $exp \in \mathcal{E}(\mathsf{P})$. As $(P,\ Q) \in \mathcal{S}$, for some P', we have

$$\langle exp,\ \mathcal{P} \rangle \rightarrow^* P' \not\rightarrow,$$
$$(P',\ Q') \in \mathcal{S}, \text{ and}$$
$$(\langle p,\ \mathcal{P}' \rangle,\ \langle q,\ \mathcal{Q}' \rangle) \in \mathcal{S}.$$

Therefore $(P',\ Q') \in \mathcal{S} \cup \mathcal{S}'$ and $(\langle p,\ \mathcal{P}' \rangle,\ \langle q,\ \mathcal{Q}' \rangle) \in \mathcal{S} \cup \mathcal{S}'$.

The proof follows in a similar way for the second case: $(P,\ Q) \in \mathcal{S}'$. □

Relational composition also preserves simulations. This result is established by the following theorem, where $\mathcal{S}\ ;\ \mathcal{S}'$, defined as

$$\{(P,\ Q) \mid \text{for some } O,\ (P,\ O) \in \mathcal{S} \text{ and } (O,\ Q) \in \mathcal{S}'\},$$

denotes the composition of $\mathcal{S}$ with $\mathcal{S}'$.

Theorem 3.3 Let O, P, and Q be specifications. For a $(\mathsf{P},\ \mathsf{O})$-simulation $\mathcal{S}$ and a $(\mathsf{O},\ \mathsf{Q})$-simulation $\mathcal{S}'$, we have that $\mathcal{S}\ ;\ \mathcal{S}'$ is a $(\mathsf{P},\ \mathsf{Q})$-simulation. □

The proof of this theorem is as easy as the proof of Theorem 3.2; it can be found in [Borba, 1995].

3.2 Refinement of States

Indeed, many relations are simulations. For example, the identity relation on $Conf(\mathsf{P})$ is a $(\mathsf{P},\ \mathsf{P})$-simulation, as shown by Theorem 3.1. For fixed specifications satisfying the properties required by the definition of simulation, the least simulation is the empty relation, whereas the largest simulation is obtained as the union of all simulations. We take this relation as our definition of simulation (refinement) of states.

Definition 3.2 Let P and Q be specifications. A state $Q \in Conf(\mathsf{Q})$ **simulates** (or **refines**) a state $P \in Conf(\mathsf{P})$, denoted $P \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} Q$, if there is a $(\mathsf{P},\ \mathsf{Q})$-simulation containing the pair $(P,\ Q)$. Formally,

$$\sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} = \bigcup \{\mathcal{S} \mid \mathcal{S} \text{ is a } (\mathsf{P},\ \mathsf{Q})\text{-simulation}\}.$$

We usually write $P \sqsubseteq Q$ for $P \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} Q$ when P and Q are clear from the context. □

Observe that $\sqsubseteq_{(\mathsf{P},\,\mathsf{Q})}$ is the empty relation when there is no associated simulation. From this definition, we conclude that 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.

Now we present some basic properties of the refinement relation on states; those properties will be extensively used for reasoning about refinement. The first theorem confirms a claim made at the beginning of this Section.

Theorem 3.4 Let P and Q be specifications such that $\mathcal{E}(\mathsf{P}) \subseteq \mathcal{E}(\mathsf{Q})$, $I(\mathsf{P}) \subseteq I(\mathsf{Q})$, and $\mathsf{P} \leq_{PS(\mathsf{P})} \mathsf{Q}$. The refinement relation on states $\sqsubseteq_{(\mathsf{P},\,\mathsf{Q})}$ is the largest $(\mathsf{P},\,\mathsf{Q})$-simulation.

Proof: By Theorem 3.2, $\sqsubseteq_{(\mathsf{P},\,\mathsf{Q})}$ is a $(\mathsf{P},\,\mathsf{Q})$-simulation and includes any other $(\mathsf{P},\,\mathsf{Q})$-simulation. □

The refinement relation on states has some reflexive and transitive properties, as indicated by the following theorem.

Theorem 3.5 For specifications O, P, and Q,

1. $P \sqsubseteq_{(\mathsf{P},\,\mathsf{P})} P$; and
2. $P \sqsubseteq_{(\mathsf{P},\,\mathsf{O})} O$ and $O \sqsubseteq_{(\mathsf{O},\,\mathsf{Q})} Q$ imply $P \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} Q$.

Proof: By Theorem 3.1, $\mathcal{ID}_{\mathsf{P}}$ is a $(\mathsf{P},\,\mathsf{P})$-simulation and $(P,\ P) \in \mathcal{ID}_{\mathsf{P}}$. This proves 1. Now we prove 2. Let us assume that

$$P \sqsubseteq_{(\mathsf{P},\,\mathsf{O})} O \text{ and } O \sqsubseteq_{(\mathsf{O},\,\mathsf{Q})} Q.$$

Hence there is a $(\mathsf{P},\,\mathsf{O})$-simulation $\mathcal{S}$ and a $(\mathsf{O},\,\mathsf{Q})$-simulation $\mathcal{S}'$ such that

$$(P,\ O) \in \mathcal{S} \text{ and } (O,\ Q) \in \mathcal{S}'.$$

Thus we have

$$(P,\ Q) \in \mathcal{S}\ ;\ \mathcal{S}'.$$

By Theorem 3.3, $\mathcal{S}\ ;\ \mathcal{S}'$ is a $(\mathsf{P},\,\mathsf{Q})$-simulation. □

3.3 Refinement of Expressions

Based on the notion of refinement of states, we can define refinement of (method) expressions. First observe that the FOOPS system described by a specification P has an associated **initial (database) state**, denoted $\emptyset_{\mathsf{P}}$, which contains only objects introduced by P and information about their respective attributes. States reached by performing an arbitrary (possibly empty) sequence of experiments with an initial state are called **reachable (database) states**.

For expressions p and q respectively related to specifications P and Q, we say that p is refined by q when the evaluation of p in any reachable state $\mathcal{P}$ of P is simulated by the evaluation of q in a state $\mathcal{Q}$ reached from $\emptyset_{\mathsf{Q}}$ by performing the same sequence of experiments used to reach $\mathcal{P}$. (By the definition of simulation,

this already considers that the evaluation of the expressions might be interfered by experiments.) Hence, when P and Q are the same specification, p is refined by q if the evaluation of p in any reachable state is simulated by the evaluation of q in the same state.

By analysing the definition of simulation of states, it is easy to check that an alternative way of expressing the ideas above is saying that p is refined by q if the evaluation of q in the initial state of Q simulates the evaluation of p in the initial state of P. This is formalized below.

Definition 3.3 Let P and Q be specifications. An expression $q \in \mathcal{T}_{\mathsf{Q}}$ **simulates** (or **refines**) $p \in \mathcal{T}_{\mathsf{P}}$, denoted $p \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q$, if

$$\langle p, \emptyset_{\mathsf{P}} \rangle \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \langle q, \emptyset_{\mathsf{Q}} \rangle,$$

where $\mathcal{T}_{\mathsf{P}}$ denotes the family of terms associated to P. □

Reflecting the notion of refinement of states, the notion of refinement of expressions requires the results yielded by related expressions to be the same. Also, due to the direct relationship with refinement of states, the relation of refinement of expressions has reflexive and transitive properties. This is stated by the following theorem.

Theorem 3.6 For specifications O, P, and Q,

1. $p \sqsubseteq_{(\mathsf{P},\,\mathsf{P})} p$; and
2. $p \sqsubseteq_{(\mathsf{P},\,\mathsf{O})} o$ and $o \sqsubseteq_{(\mathsf{O},\,\mathsf{Q})} q$ imply $p \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q$.

□

The proof of this theorem is straightforward; it can be found in [Borba, 1995].

3.4 Refinement of Specifications

Based on the notion of refinement of expressions, we can introduce a simple definition of refinement of specifications. Basically, a specification P is refined by a specification Q if any experiment of P is refined by its corresponding experiment of Q. This is formalized below.

Definition 3.4 A specification P is **refined** by a specification Q, denoted $\mathsf{P} \sqsubseteq \mathsf{Q}$, if

$$exp \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} exp,$$

for any $exp \in \mathcal{E}(\mathsf{P})$. □

Note that in order to prove $\mathsf{P} \sqsubseteq \mathsf{Q}$ we need only to prove $\emptyset_{\mathsf{P}} \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \emptyset_{\mathsf{Q}}$.

Similarly to refinement of states and expressions, refinement of specifications has reflexive and transitive properties. This is stated by the following theorem.

Theorem 3.7 For specifications O, P, and Q,

1. $\mathsf{P} \sqsubseteq \mathsf{P}$; and

2. $\mathsf{P} \sqsubseteq \mathsf{O}$ and $\mathsf{O} \sqsubseteq \mathsf{Q}$ imply $\mathsf{P} \sqsubseteq \mathsf{Q}$.

□

We omit the proof of this theorem since it is trivial; the details can be found in [Borba, 1995].

4 Proving Refinement

In order to explore our notion of refinement, we introduce some FOOPS specifications giving examples of simulation relations and refinement, together with the necessary proofs.

4.1 Memory Cells

Let us consider a specification defining a class of memory cells for storing natural numbers:

```
omod CELL is
   pr NAT .
   class Cell .
   at val_ : Cell -> Nat .
   me _:=_ : Cell Nat -> Cell .
   me _:=_+_ : Cell Cell Nat -> Cell .
   vars C C' : Cell .
   var N : Nat .
   ax val(C := N) = N .
   ax val(C := C' + N) = val(C') + N .
endomod
```

Here objects of `Cell` have the attribute `val` (for *val*ue), which indicates the natural number stored in a cell, and two methods for assigning values to a cell. The method `_:=_` assigns a given natural number to a cell, whereas `_:=_+_` assigns to a cell the value stored in another cell plus a natural number (the cells and the natural number are given as arguments). Both methods are atomic, since they are defined by DMAs.

In order to keep it simple, we first study the simulation of states related to the same specification. Based on the specification `CELL`, let's examine the relationship between the expressions

```
x := 0 ; x := x + 1
```
(1)

and

```
x := 1,
```
(2)

where `x` is the identifier of an object of `Cell`.

Note that the expressions above may be seen as programs of an imperative programming language. Indeed, by providing a class of memory cells, we can use

our notion of simulation for reasoning about (concurrent) imperative programs as well; they are just a particular case of the object oriented programs that we can write using FOOPS.

Both expressions have the effect of assigning `1` to `x`, independently of the state of the system (as long as `x` exists). In a sequential environment, this is enough to guarantee that they are equivalent; we can just consider the input/output behaviour of programs, since the evaluation of expressions cannot be interfered with.

However, interference usually occurs in a concurrent environment; so in order to assure equivalence, we also have to consider if expressions behave in an equivalent way when subjected to the same interference (that's why our notion of refinement considers that experiments can be made with the intermediate states reached during the evaluation of an expression). Indeed, the expressions (1) and (2) are not equivalent in a concurrent environment: just consider that the evaluation of (1) might be interfered—after the execution of `x := 0` and before `x := x + 1`—whereas (2) is atomic. For example, if the expressions (1) and (2) are interfered with by `x := 1`, then they will behave differently. This can be justified by checking that, independently of the system state, the expression

```
x := 1 || x := 1
```

is deterministic: it assigns `1` to `x`. On the other hand,

```
x := 1 || x := 0 ; x := x + 1
```

is nondeterministic: it can either assign `1` to `x`, or assign `2` to `x`, depending on whether `x := 0` is evaluated after or before `x := 1`.

The discussion above also tells us that (1) is not a refinement of (2), since the behaviour of the first is not a particular case of the behaviour of the second when both are subjected to the same interference. However, we cannot conclude that (2) is not a refinement of (1). Let us investigate that here, by checking whether

$$\langle \texttt{x := 0 ; x := x + 1}, \mathcal{P} \rangle \tag{3}$$

is refined by

$$\langle \texttt{x := 1}, \mathcal{P} \rangle, \tag{4}$$

for any reachable database state $\mathcal{P}$.

As discussed in Section 3.2, this can be verified by finding a (`CELL`, `CELL`)-simulation relating the configurations (states) described above. As proved by Theorem 3.1, such a simulation may exist because the restrictions on related specifications are trivially valid, since the specifications are the same.

4.1.1 A Simulation

Let us now check whether the relation $\mathcal{S}$, consisting of pairs in the forms

$$(\langle \texttt{x := 0 ; x := x + 1}, \mathcal{P} \rangle \quad , \quad \langle \texttt{x := 1}, \mathcal{P} \rangle) \tag{5}$$

$$(\langle e, \mathcal{P} \rangle \quad , \quad \langle e, \mathcal{P} \rangle), \tag{6}$$

for any reachable database state $\mathcal{P}$ and any expression e, is a simulation. Pairs of the first form should be in $\mathcal{S}$ because we are interested in proving that (3) is refined by (4), whereas pairs of the second form represent states reached from the states in the first pair, and states resulting from the execution of experiments.

In order to check the conditions in the definition of simulation, we must have some knowledge about the transition relation $\rightarrow$. Although we have not defined it here, the informal description of FOOPS concepts in Section 2 gives an informal overview of FOOPS operational semantics and justifies the state transitions discussed in this chapter.

By Definition 3.1, $\mathcal{S}$ is a simulation if its pairs satisfy some properties. Now we consider whether those properties are valid for pairs of $\mathcal{S}$ in the first form. First we assume that `x` is in $\mathcal{P}$. So the unique possible transition from configuration 4 is

$$\langle \texttt{x := 1}, \mathcal{P} \rangle \rightarrow \langle \texttt{x}, \mathcal{P} \oplus \texttt{val x = 1} \rangle, \tag{7}$$

where $\mathcal{P} \oplus$ `val x = 1` denotes the database state derived from $\mathcal{P}$ by updating the attribute `val x` with the value `1`; that's the expected effect of an assignment operation.

From configuration 3, the unique possible sequence of transitions is

$$\begin{aligned} &\langle \texttt{x := 0 ; x := x + 1}, \mathcal{P} \rangle \\ &\quad \rightarrow \langle \texttt{x ; x := x + 1}, \mathcal{P} \oplus \texttt{val x = 0} \rangle \\ &\quad \rightarrow \langle \texttt{x := x + 1}, \mathcal{P} \oplus \texttt{val x = 0} \rangle \\ &\quad \rightarrow \langle \texttt{x}, \mathcal{P} \oplus \texttt{val x = 0} \oplus \texttt{val x = 1} \rangle. \end{aligned}$$

Note that `x := x + 1` is atomically executed (since the behaviour of `_:=_+_` is specified by a DMA), and the second transition is necessary for eliminating the result yielded by `x := 0`. Also, observe that

$$\mathcal{P} \oplus \texttt{val x = 0} \oplus \texttt{val x = 1}$$

is equal to

$$\mathcal{P} \oplus \texttt{val x = 1},$$

since the update specified by

$$\texttt{val x = 0}$$

is overwritten by the update indicated by `val x = 1`. From this result and the sequence of transitions introduced above, we can say that

$$\langle \texttt{x := 0 ; x := x + 1}, \mathcal{P} \rangle \rightarrow^* \langle \texttt{x}, \mathcal{P} \oplus \texttt{val x = 1} \rangle. \tag{8}$$

After analysing the possible transitions from states, it is straightforward to verify that condition 1 (of Definition 3.1) is valid: just consider the unique transitions (7) and (8), and that the pair

$$(\langle \texttt{x}, \mathcal{P} \oplus \texttt{val x = 1} \rangle, \langle \texttt{x}, \mathcal{P} \oplus \texttt{val x = 1} \rangle)$$

is a particular instance of the pairs of $\mathcal{S}$ in the second form.

Condition 2 is valid for pairs of $\mathcal{S}$ in the first form, since

$$\langle \texttt{x := 1}, \mathcal{P} \rangle \not\rightarrow$$

does not hold, as shown by transition (7).

In general, it is trivial to check condition 3 when the database states in the related configurations are the same and the relation in investigation contains the identity relation; that's certainly the case of $\mathcal{S}$. In fact, we do not even have to consider what are the experiments associated to `CELL`: let's assume an arbitrary experiment *exp*. So if

$$\langle exp, \mathcal{P} \rangle \rightarrow^* Q' \not\rightarrow$$

for some state Q', then condition 3a holds by taking P' as Q', and considering the transition above; that's possible because only one specification is being considered, so we are also dealing with only one transition relation: $\rightarrow_{\texttt{CELL}}$. Also, condition 3b is valid, since $(Q',\ Q')$ is clearly in $\mathcal{S}$ ($\mathcal{S}$ contains the identity). Lastly, we can verify that condition 3c is valid by checking that

$$(\langle \texttt{x := 0 ; x := x + 1}, \mathcal{Q}' \rangle, \langle \texttt{x := 1}, \mathcal{Q}' \rangle)$$

is a particular instance of the pairs of $\mathcal{S}$ in the first form.

This concludes the proof that, when `x` is in $\mathcal{P}$, the pairs of $\mathcal{S}$ in the first form satisfy the properties of pairs in a simulation relation. Now let's assume that `x` is not in $\mathcal{P}$. Condition 1 becomes vacuously true since

$$\langle \texttt{x := 1}, \mathcal{P} \rangle \not\rightarrow \tag{9}$$

because `x` is not in $\mathcal{P}$ (an attribute or a method specified by DMAs can only be executed if its associated object exists). For the same reason,

$$\langle \texttt{x := 0 ; x := x + 1}, \mathcal{P} \rangle \not\rightarrow . \tag{10}$$

The sequential composition cannot be executed because the first expression cannot be executed.

Based on (9) and (10), we can verify condition 2a, by observing that

$$\langle \texttt{x := 0 ; x := x + 1}, \mathcal{P} \rangle \rightarrow^* \langle \texttt{x := 0 ; x := x + 1}, \mathcal{P} \rangle \not\rightarrow$$

since $\rightarrow^*$ is reflexive. As the expressions in the states of (9) and (10) are not fully evaluated, conditions 2b and 2c are valid. Lastly, condition 2d is satisfied because the resulting pairs are instances of the pairs of $\mathcal{S}$ in the first form.

It remains only to verify condition 3. However, this follows exactly in the same way as when `x` is in $\mathcal{P}$.

Verifying that pairs of $\mathcal{S}$ in the second form satisfy the conditions in Definition 3.1 is trivial; in fact, this is a particular case of what was done to prove Theorem 3.1.

Finally, we can concluded that $\mathcal{S}$ is a simulation. From this and Definition 3.2, it follows that

$$\langle \texttt{x := 0 ; x := x + 1}, \mathcal{P} \rangle \sqsubseteq \langle \texttt{x := 1}, \mathcal{P} \rangle,$$

for any reachable database state $\mathcal{P}$. In fact, this is enough to guarantee that

$$\texttt{x := 0 ; x := x + 1}$$

is refined by

$$\texttt{x := 1}.$$

4.1.2 Failing to be a Simulation

In the beginning of this section we claimed that

$$\texttt{x := 1},$$

is not refined by

$$\texttt{x := 0 ; x := x + 1};$$

We have informally shown that the behaviour of the last is not in general a particular case of the behaviour of the former when both are subjected to the same interference. For further exploring our approach, we investigate what our definition of simulation says about that.

Similarly to what has been done before, let's check if

$$\langle \texttt{x := 1}, \mathcal{P} \rangle \tag{11}$$

is refined by

$$\langle \texttt{x := 0 ; x := x + 1}, \mathcal{P} \rangle, \tag{12}$$

for any reachable database state $\mathcal{P}$. Basically, we should find a (CELL, CELL)-simulation containing pairs in the form

$$(\langle \texttt{x := 1}, \mathcal{P} \rangle, \langle \texttt{x := 0 ; x := x + 1}, \mathcal{P} \rangle),$$

for any reachable database state $\mathcal{P}$. By condition 1 from the definition of simulation, and observing that

$$\langle \texttt{x := 0 ; x := x + 1}, \mathcal{P} \rangle \rightarrow \langle \texttt{x ; x := x + 1}, \mathcal{P} \oplus \texttt{val x = 0} \rangle,$$

a simulation containing the pair above must relate the state

$$\langle \texttt{x ; x := x + 1}, \mathcal{P} \oplus \texttt{val x = 0} \rangle \tag{13}$$

to some $\rightarrow^*$-derivative of

$$\langle \texttt{x := 1}, \mathcal{P} \rangle.$$

The only $\rightarrow^*$-derivatives of this state are itself and

$$\langle \texttt{x}, \mathcal{P} \oplus \texttt{val x = 1} \rangle \tag{14}$$

(see transition (7)).

The first $\rightarrow^*$-derivative cannot be related to state (13) by any simulation: just consider performing the experiment `val(x)` with both states. The experiment performed with (13) yields the value `0`, whereas the experiment performed with $\langle$`x := 1`, $\mathcal{P}\rangle$ does not necessarily yield the same value (only if the value associated to `x` in $\mathcal{P}$ is `0`), and might even be suspended (if `val x` is not defined in $\mathcal{P}$). Hence, condition 3 in the definition of simulation would not be valid.

Also, trying to relate (14) and (13) does not work, for a similar reason as discussed above. Again, this can be checked by trying the experiment `val(x)` with both states. Essentially, the problem is with the intermediate state (13) reached from (12), since it is not a simulation of any state reached from (11); this is why the behaviour of (12) is not a particular case of the behaviour of (11). As there is no simulation relating the pairs of states analysed here, we confirm the claim made at the beginning of the section.

4.1.3 Further Examples of Simulation

Following the same line as discussed in the previous sections, we can prove that

```
x := x + 1 ; x := x - 1                                   (15)
```

is refined by

```
skip(x),                                                  (16)
```

where

```
me skip : Cell -> Cell .
ax skip(C) = C .
```

(note that the method `skip` does not change the state of objects). In fact, a simulation relation very similar to $\mathcal{S}$ can be chosen as the basis for this proof. As in Section 4.1.2, we can also verify that (16) is not refined by (15).

Perhaps more interesting is verifying that

```
skip(x) ; x := 0
```

is refined by

```
x := 0,
```

and vice versa. We just have to check that the relations denoted by

$$(\langle \texttt{skip(x) ; x} := 0, \mathcal{P}\rangle \quad , \quad \langle \texttt{x := 0}, \mathcal{P}\rangle) \tag{17}$$

$$(\langle e, \mathcal{P}\rangle \quad , \quad \langle e, \mathcal{P}\rangle), \tag{18}$$

and

$$(\langle \texttt{x := 0}, \mathcal{P}\rangle \quad , \quad \langle \texttt{skip(x) ; x := 0}, \mathcal{P}\rangle) \tag{19}$$

$$(\langle \texttt{x := 0}, \mathcal{P}\rangle \quad , \quad \langle \texttt{x ; x := 0}, \mathcal{P}\rangle) \tag{20}$$

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

for any reachable database state $\mathcal{P}$ and any expression e, are (`CELL`, `CELL`)-simulations. We omit the details here.

Also, we can prove that the nondeterministic choice

`x := 0 [] x := 1`

is refined by the assignment

`x := 1`,

by proving that the relation formed by pairs in the form

$$(\langle \texttt{x := 0 [] x := 1}, \mathcal{P}\rangle \quad , \quad \langle \texttt{x := 1}, \mathcal{P}\rangle) \tag{21}$$

$$(\langle e, \mathcal{P}\rangle \quad , \quad \langle e, \mathcal{P}\rangle), \tag{22}$$

for any reachable database state $\mathcal{P}$ and any expression e, is a simulation. It is also easy to check that the nondeterministic choice is not a refinement of the assignment.

4.2 Buffers

Here we exemplify the use of the simulation technique for proving refinement of states related to different specifications. We consider the specification (implementation) of buffers represented by lists, introduced in Section 2.2, and a simplified specification of buffers represented by array and pointers.

First, we introduce the following functional level module defining arrays of elements of an arbitrary sort having a constant (denoted by `err`, as specified by `TRIVE`, which is an extension of `TRIV` with the constant `err`) that is used as the value yielded by the array indexing operation when the array is not initialized for a particular index. 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? .
  var A : Array .
  vars I J : Nat .
  var E : 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 Buffer .
at array_ : Buffer -> Array [hidden] .
at fst_ : Buffer -> Nat [hidden] .
at next_ : Buffer -> Nat [hidden] .
at empty?_ : Buffer -> Bool .
at full?_ : Buffer -> 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 : Buffer -> Buffer .
me get : Buffer -> Elt .
me put : Buffer Elt -> Buffer .
me del : Buffer -> Buffer [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. This enable us to use our refinement technique.

4.2.1 Refinement of Parameterized Modules

In order to further explore our approach, we should now try to prove that the operations specified in BUFFERA simulate the corresponding operations specified in BUFFER. As we have seen in previous sections, this can be done by verifying simulation of states. However, our notion of simulation assumes that states are associated to specifications (unparameterized FOOPS modules), whereas BUFFER and BUFFERA are parameterized specifications (parameterized FOOPS modules). Using our notion, we can only verify simulation of states associated to particular instantiations of parameterized modules.

It would be desirable if we could check whether states associated to an arbitrary instantiation of BUFFER with arguments satisfying TRIVE and MAX are simulated by states associated to the instantiation of BUFFERA with the same arguments. This is necessary for ensuring that if we instantiate both modules in the same way (as long as the arguments satisfy TRIVE and MAX) then the operations resulting from the instantiation of the first are simulated by the operations resulting from the instantiation of the second.

In fact, we can use our definition of simulation in order to achieve the general result mentioned above. We just have to consider the (unparameterized) specifications obtained from BUFFER and BUFFERA by assuming the minimum information about their instantiation. This information is given by the requirements theories TRIVE and MAX. Intuitively, if we can prove refinement for the "minimum instantiation" of the parameterized modules, we should be able to prove refinement for any other instantiation.

However, the "minimum instantiation" of the parameterized modules gives a minimum family $\mathcal{E}$ of experiments, since comparing with other instantiations, it has the least number of functions. Fortunately, we need only a small extension of our definition of simulation to consider the general form of (expressions denoting) experiments, rather than the experiments themselves. Contrasting with $\mathcal{E}$, the general form of experiments does not depend on a particular instantiation. So instead of analysing the effects of particular experiments (see condition 3 of Definition 3.1), we should analyse the effect of the invocation of visible operations without assuming any information about their arguments (except that they are identifiers or elements of ADTs). That's the approach that we are going to use now, in order to reason about the modules defining buffers. Hereafter we also use the names BUFFER and BUFFERA when we want to refer to the "minimum instantiation" of the corresponding parameterized modules.

First, we have to check whether the specifications of buffers can be related by simulations; according to Definition 3.1, $\mathcal{E}$(BUFFER) should be contained in $\mathcal{E}$(BUFFERA), and

$$\texttt{BUFFER} \leq_{PS(\texttt{BUFFER})} \texttt{BUFFERA} \tag{23}$$

should hold. The set of primary sorts of `BUFFER` is formed by `Elt` and `Bool`. In both specifications, the definition of the functions associated to the primary sorts is basically given by `TRIVE` and the standard definition of an ADT of booleans (included in any specification). Also, some extra functions are introduced: `BUFFER` introduces `head`, whereas `BUFFERA` introduces `_[_]`. However, the equations defining those functions indicate that they yield an existing element of `Elt`; they do not introduce any extra element to `Elt`. Hence, we can conclude that (23) is valid, since the difference between the two specifications does not affect their meaning; the denoted ADTs are the same.

Also note that both specifications introduce the same visible operations; this and (23) guarantees that the experiments we can make with states of `BUFFER` can also be made with states of `BUFFERA`, if we assume that both specifications have the same family of object identifiers (in practice, we can assume that an element of the family I is formed by names prefixed by the name of the class associated to this element).

From what discussed above, we can also conclude that the requirements on specifications are satisfied for any instantiation of the parameterized modules defining buffers, assuming that they are instantiated with the same arguments satisfying the minimum requirements. Just observe that

- all instantiations of a module have the same family of visible operations, if the arguments are functional modules (they do not define methods or attributes);
- the ADTs of a particular instantiation of `BUFFER` and `BUFFERA` are the same, since they are essentially defined by their arguments, which are the same as well; and
- we assume that all instantiations of a module have the same related set of object identifiers.

4.2.2 A Simulation

Now we can verify the simulation of states associated to the specifications of buffers that we have previously introduced. 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

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

So let's check whether such a buffer represented by array and pointers simulates a buffer represented by a list `l` whenever `i <= j` and `l` is equal to `i < a > j`, with respect to the functional theory `EB` obtained by extending `LIST` with `ARRAY` and the definition above.

Following Section 3.2, we can check if our intuitions are correct by trying to prove that a relation $\mathcal{S} \subseteq Conf(\texttt{BUFFER}) \times Conf(\texttt{BUFFERA})$ containing states in the following form is a simulation:

$$(\langle e, \mathcal{P}\rangle \quad , \quad \langle e, \mathcal{Q}\rangle), \tag{24}$$

where e is either a fully evaluated expression of a primary sort of `BUFFER` or an experiment or object identifier of `BUFFER`, and $\mathcal{P}$ and $\mathcal{Q}$ are reachable database states, respectively associated to `BUFFER` and `BUFFERA`, such that

- the objects in $\mathcal{P}$ and $\mathcal{Q}$ are the same;
- 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, with respect to `EB`, to the value denoted by

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

  in $\mathcal{Q}$,

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

Before verifying that $\mathcal{S}$ is a simulation, note that the experiments associated to `BUFFER` have one of the following general forms: `empty?(b)`, `full?(b)`, `reset(b)`, `put(b,e)`, `get(b)`, `new.Buffer()`, `new(b)`, and `remove(b)`, for an arbitrary buffer `b` and an element `e` of sort `Elt`.

Let us now check whether the pairs of $\mathcal{S}$ satisfy the conditions in Definition 3.1. We first assume that e has the general form `reset(b´)`, for an arbitrary buffer `b´`, and that `b´` is in $\mathcal{P}$ (this implies that `b´` is also in $\mathcal{Q}$, since both states are related by $\mathcal{S}$). Thus, the unique transition from

$$\langle \texttt{reset(b´)}, \mathcal{Q}\rangle \tag{25}$$

leads to

$$\langle \texttt{b´}, \mathcal{Q} \oplus \texttt{fst b´ = 0} \oplus \texttt{next b´ = 0} \oplus \texttt{array b´ = null}\rangle, \tag{26}$$

whereas the unique transition from

$$\langle \texttt{reset(b´)}, \mathcal{P}\rangle \tag{27}$$

leads to

$$\langle \texttt{b´}, \mathcal{P} \oplus \texttt{elems b´ = nil}\rangle. \tag{28}$$

This clearly implies that

$$\langle \texttt{reset(b´)}, \mathcal{P}\rangle \rightarrow^* \langle \texttt{b´}, \mathcal{P} \oplus \texttt{elems b´ = nil}\rangle.$$

So in order to check condition 1, we just have to prove that the states (28) and (26) are related by $\mathcal{S}$. First, note that those resulting states have the same fully evaluated expression (`b'`), as required by $\mathcal{S}$. Second, let $\mathcal{P}'$ and $\mathcal{Q}'$ be the (reachable) database states respectively in (28) and (26). Thus observe the following:

- the objects in $\mathcal{P}'$ and $\mathcal{Q}'$ are the same, since $\mathcal{P}$ and $\mathcal{Q}$ have the same objects, and the execution of `reset(b')` does not create any objects;
- the attribute `elems b'` has a defined value in $\mathcal{P}'$, and `array b'`, `fst b'`, and `next b'` also have defined values in $\mathcal{Q}'$;
- in $\mathcal{Q}'$, the condition `(fst b') <= (next b')` is equal to `0 <= 0`, which is trivially valid; and
- the value of `elems b'` in $\mathcal{P}'$ is `nil`, whereas the value of

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

  in $\mathcal{Q}'$ is `0 < null > 0`, which is the same as `nil`, by the definition of `_ < _ > _`.

Instead of considering the attributes of all objects in $\mathcal{P}'$ and $\mathcal{Q}'$, we consider only properties of the stored attributes of `b'`, since those are the only attributes that change due to the execution of `reset(b')`; we can conclude that the corresponding properties for the other attributes are valid from what is assumed about $\mathcal{P}$ and $\mathcal{Q}$ (they are related by $\mathcal{S}$).

Condition 2 is vacuously valid, since there is a transition from state (25).

Now it remains to check condition 3. Considering experiments in the form `empty?(b)`, we observe that the unique transition from

$$\langle \texttt{empty?(b)}, \mathcal{Q} \rangle \tag{29}$$

leads to

$$\langle [\![\texttt{i == j}]\!]_{\texttt{BA}}, \mathcal{Q} \rangle,$$

where `BA` is an abbreviation for $FE(\texttt{BUFFERA})$ (the set of functional equations of `BUFFERA`, and `i` and `j` are respectively the values of `fst b` and `next b` in $\mathcal{Q}$ (this implies that `array b` has value `a` in $\mathcal{Q}$, for some array `a`). If those attributes are not defined in $\mathcal{Q}$, no transition is possible from (29). Similarly, no transition is possible from

$$\langle \texttt{empty?(b)}, \mathcal{P} \rangle, \tag{30}$$

if `elems b` is not defined in $\mathcal{P}$. However, if this is not the case, there is a unique transition from (30); it leads to

$$\langle [\![\texttt{l == nil}]\!]_{\texttt{B}}, \mathcal{P} \rangle,$$

where `B` is an abbreviation for $FE(\texttt{BUFFER})$ and `l` is the value of `elems b` in $\mathcal{P}$.

As $\mathcal{P}$ and $\mathcal{Q}$ are related by $\mathcal{S}$, we can deduce that `l` is equal, with respect to `EB`, to `i < a > j`. So, using `EB`, by the definition of `_ < _ > _`, we can prove that `l == nil` is equal to `i == j`. It is also easy to see that `EB` is a conservative

(protecting) extension of the functional theory of LIST. Hence we can assume that $[\![\, e \,]\!]_{\mathtt{B}}$ is the same as $[\![\, e \,]\!]_{\mathtt{EB}}$ for any e of BUFFER. Therefore we have that $[\![\, \mathtt{l == nil} \,]\!]_{\mathtt{B}}$ is the same as $[\![\, \mathtt{i == j} \,]\!]_{\mathtt{B}}$. It is then easy to check that the resulting states are related by $\mathcal{S}$, since i == j is a term of Bool, a primary sort of BUFFER. We have omitted the details about the functional level reasoning above because here we concentrate on the object level of FOOPS (see [Goguen, 1994] for the details and justification).

Also, as $\mathcal{P}$ and $\mathcal{Q}$ are related by $\mathcal{S}$, the attributes of b are defined either in both states or in none; hence, either both transitions are possible or none is possible. Thus, it follows that condition 3 is valid considering experiments in the form empty?(b).

Similarly, we can check that condition 3 is valid for experiments in the form full?(b). Also, the proof that the condition is valid for experiments in the form reset(b) follows the same lines of our proof that pairs of $\mathcal{S}$ having reset(b′) as e satisfy condition 1. We omit the details here.

For experiments in the form put(b,e), let's first assume that the stored attributes of b are defined in $\mathcal{Q}$ (this implies that elems b is defined as l in $\mathcal{P}$, for some list l), and that the value of full? b in $\mathcal{Q}$ is false. So by the definition of full?, j - i == max is equal to false, where i and j are respectively the values of fst b and next b in $\mathcal{Q}$. Hence, as l is equal to i < a > j, we can infer that # l == max is equal (with respect to EB) to false; that is, the value of full? b in $\mathcal{P}$ is false as well.

In this case, the unique transitions caused by put(b,e) are

$$\begin{aligned} &\langle \mathtt{put(b,e)},\, \mathcal{Q} \rangle \rightarrow \\ &\quad \langle \mathtt{b},\, \mathcal{Q} \oplus \mathtt{next\ b = j + 1} \oplus \mathtt{array\ b = a[j] := e} \rangle, \end{aligned} \tag{31}$$

where a is the value of array b in $\mathcal{Q}$, and

$$\langle \mathtt{put(b,e)},\, \mathcal{P} \rangle \rightarrow \langle \mathtt{b},\, \mathcal{P} \oplus \mathtt{elems\ b = l\ .\ e} \rangle. \tag{32}$$

Observe that these resulting states are related by $\mathcal{S}$, because the resulting expressions are the same fully evaluated term; the database states $\mathcal{P}$ and $\mathcal{Q}$ are related by $\mathcal{S}$; no new objects were created; the updated attributes were already defined before the transitions; the condition i <= j + 1 trivially follows from i <= j; and we can prove that l . e is equal, with respect to EB, to

$$\mathtt{i < a[j] := e > j + 1},$$

since l is equal to i < a > j.

Now we verify that condition 3 also follows when the stored attributes of b are not defined in $\mathcal{Q}$, or when the value of full? b in $\mathcal{Q}$ is true. In those cases, we can prove that either elems b is not defined in $\mathcal{P}$, or that and the value of full? b in $\mathcal{P}$ is true. Hence, no transition is possible; the experiments cannot be performed with $\mathcal{P}$ and $\mathcal{Q}$. This implies condition 3.

As proved when considering experiments in the form empty? b, the value of empty? b in $\mathcal{Q}$ is the same as its value in $\mathcal{P}$. So when analysing the effect of experiments in the form get(b), we assume that the value of empty? b in both $\mathcal{P}$

and $\mathcal{Q}$ is `false`. Otherwise, the experiment cannot be performed neither in $\mathcal{P}$ nor in $\mathcal{Q}$, and it is then trivial to check condition 3.

Assuming that the buffers are not empty, the unique transitions are the following:

$$\langle \texttt{get(b)},\, \mathcal{Q}\rangle \;\rightarrow\; \langle [\![\, \texttt{a[i]} \,]\!]_{\texttt{BA}},\, \mathcal{Q} \oplus \texttt{fst b = i + 1}\rangle, \tag{33}$$

and

$$\langle \texttt{get(b)},\, \mathcal{P}\rangle \;\rightarrow\; \langle [\![\, \texttt{head(l)} \,]\!]_{\texttt{B}},\, \mathcal{P} \oplus \texttt{elems b = tail(l)}\rangle. \tag{34}$$

Note that the evaluation of `get` is atomic, and that `a[i]` is equal (with respect to `EB`) to `head(l)`, since `l` is equal to `i < a > j`, and `i < j` (the buffer is not empty); so it also follows that `i <= j + 1`. It is then easy to check that the resulting states are related by $\mathcal{S}$, and that condition 3 is valid. We omit the details here.

The object creation operations `new.Buffer()` and `new(b)` just add an object to the database state; they do not initialize attributes. In fact, the behaviour of those operations just depends on the family of identifiers associated to a specification, and the objects in the state; the first determines what identifiers may be chosen by `new.Buffer()`, whereas the other determines when `new(_)` can be evaluated (it might be suspended, if its argument is the identifier of an object in the state). As `BUFFER` and `BUFFERA` have the same associated family of identifiers, and the objects in $\mathcal{P}$ and $\mathcal{Q}$ are the same, the creation operations have the same effect in $\mathcal{P}$ and $\mathcal{Q}$. It follows that condition 3 holds for this kind of experiment. We omit the details again.

Similarly to `new(b)`, the experiment `remove(b)` only depends on what objects are in the state. So it will behave in the same way when performed with $\mathcal{P}$ and $\mathcal{Q}$. If `b` is an object in the state, this operation removes all its stored attributes from the state. Hence, it follows that the resulting states are related by $\mathcal{S}$, and that condition 3 is satisfied. This concludes the proof that pairs of $\mathcal{S}$ relating `reset(b')` satisfy the conditions of a simulation relation whenever `b'` is in $\mathcal{P}$.

Now we assume that `b'` is not in $\mathcal{P}$ (this implies that `b'` is not in $\mathcal{Q}$ as well). Observe that checking the effect of experiments for this case is the same as checking the effect of experiments when `b'` is in $\mathcal{P}$, as previously discussed. Hence, we just need to verify conditions (1) and (2). The first is vacuously true; there is no transition from

$$\langle \texttt{reset(b')},\, \mathcal{Q}\rangle,$$

since `b'` is not in $\mathcal{Q}$. The second condition can be verified by checking that `reset(b')` cannot be evaluated in $\mathcal{P}$ as well, since `b'` is not in $\mathcal{P}$; so we may conclude that

$$\langle \texttt{reset(b')},\, \mathcal{P}\rangle \;\rightarrow^{*}\; \langle \texttt{reset(b')},\, \mathcal{P}\rangle \not\rightarrow,$$

the resulting states are related by $\mathcal{S}$, and the expression `reset(b')` is not fully evaluated.

In order to complete the proof that $\mathcal{S}$ is a (`BUFFER`, `BUFFERA`)-simulation, we should analyse the pairs of $\mathcal{S}$ having expressions in other forms different from `reset(b')`. However, that's essentially what we have proved when considering the effects of performing experiments with the database states related by $\mathcal{S}$; we have just used `b` instead of `b'`.

4.3 Laws of FOOPS Method Combiners

The notion of refinement of expressions introduced here can also be used for proving general properties of FOOPS method combiners, if those properties are expressed as refinement inequations. Besides providing a better insight on the semantics of method combiners, those properties may be used for proving refinement, or deriving correct implementations from specifications.

For example, we can assert that the sequential composition method combiner is associative by verifying that

$$p \ ; \ (q \ ; \ o) \sqsubseteq_{(\mathsf{P},\mathsf{P})} (p \ ; \ q) \ ; \ o \tag{35}$$

and

$$(p \ ; \ q) \ ; \ o \sqsubseteq_{(\mathsf{P},\mathsf{P})} p \ ; \ (q \ ; \ o), \tag{36}$$

for any specification P and method expressions $p, q, o \in \mathcal{T}_{\mathsf{P}}$.

In order to verify 35, we have to prove that

$$\langle p \ ; \ (q \ ; \ o), \emptyset_{\mathsf{P}} \rangle \sqsubseteq_{(\mathsf{P},\mathsf{P})} \langle (p \ ; \ q) \ ; \ o, \emptyset_{\mathsf{P}} \rangle.$$

This can be directly verified by checking that the relation containing $\mathcal{ID}_{\mathsf{P}}$ and all pairs in the form

$$(\langle p \ ; \ (q \ ; \ o), \mathcal{D} \rangle \quad , \quad \langle (p \ ; \ q) \ ; \ o, \mathcal{D} \rangle), \tag{37}$$

for any $\mathcal{D} \in Db(-\mathsf{P})$, p, q, and o, is a (P, P)-simulation. This is straightforward, by using Theorem 3.1 and the rules of the operational semantics for sequential composition:

$$\langle e \ ; \ f, \mathcal{D} \rangle \rightarrow \langle e' \ ; \ f, \mathcal{D}' \rangle \ \textit{if} \ \langle e, \mathcal{D} \rangle \rightarrow \langle e', \mathcal{D}' \rangle \tag{38}$$

and

$$\langle v \ ; \ e, \mathcal{D} \rangle \rightarrow \langle e, \mathcal{D} \rangle \tag{39}$$

where v is a fully evaluated expression. Basically, we just have to observe the following facts. First let p be a fully evaluated expression. By Rule 39, we have

$$\langle p \ ; \ q, \mathcal{D} \rangle \rightarrow \langle q, \mathcal{D} \rangle.$$

Therefore, by Rule 38, we have

$$\langle (p \ ; \ q) \ ; \ o, \mathcal{D} \rangle \rightarrow \langle q \ ; \ o, \mathcal{D} \rangle.$$

But, by Rule 39, we also have

$$\langle p \ ; \ (q \ ; \ o), \mathcal{D} \rangle \rightarrow \langle q \ ; \ o, \mathcal{D} \rangle.$$

Second let's assume that p is not a fully evaluated expression and $\langle p, \mathcal{D} \rangle \not\rightarrow$. Hence

$$\langle p \ ; \ q, \mathcal{D} \rangle \not\rightarrow .$$

Therefore

$$\langle (p \ ; \ q) \ ; \ o, \mathcal{D} \rangle \not\rightarrow,$$

since $p \ ; \ q$ is clearly not fully evaluated. But we also have

$$\langle p \ ; \ (q \ ; \ o), \mathcal{D} \rangle \not\rightarrow .$$

Finally assume that p is not a fully evaluated expression and $\langle p, \mathcal{D} \rangle \rightarrow \langle p', \mathcal{P} \rangle$, for some p' and $\mathcal{P}$. By Rule 38, we have

$$\langle p \ ; \ q, \mathcal{D} \rangle \rightarrow \langle p' \ ; \ q, \mathcal{P} \rangle.$$

Hence, by Rule 38, we have

$$\langle (p \ ; \ q) \ ; \ o, \mathcal{D} \rangle \rightarrow \langle (p' \ ; \ q) \ ; \ o, \mathcal{P} \rangle.$$

But then we also have

$$\langle p \ ; \ (q \ ; \ o), \mathcal{D} \rangle \rightarrow \langle p' \ ; \ (q \ ; \ o), \mathcal{P} \rangle.$$

The Inequation 36 and many other properties of method combiners can be proved in a similar way.

5 Aspects Considered by Refinement

Now we analyse the aspects that are considered by our definition of refinement. We discuss what $P \sqsubseteq Q$ tell us about the states P and Q, in addition to the fact that whatever can be observed by performing experiments with Q can also be observed doing the same experiments with P. This clarifies the aspects that have to be considered when defining refinement for a concurrent object oriented programming language. Hereafter let's assume that $P \sqsubseteq Q$.

It is well known in other frameworks that refinement can be achieved by reducing nondeterminism [Milner, 1989, Hoare, 1985, Morgan, 1994]. Here we follow the same approach. At most, Q should be as nondeterministic as P; in other words, Q should be more predictable than P. This is what guarantees that the behaviour of Q is a particular case of the behaviour of P, as expressed by condition 1 from the definition of simulation, and exemplified in Section 4.1.3, where we check that a nondeterministic choice is refined by an assignment.

Data refinement [Hoare, 1972, Jifeng *et al.*, 1986] is also guaranteed by our definition of refinement. In fact, we abstract from the particular representation of object states, as long as they are observation equivalent (i.e., whatever can be observed by means of experiments from one can also be observed from the other). In the definition of simulation, this is reflected by condition 3; in the examples, this is illustrated in Section 4.2, where we prove refinement of programs that use different state structures.

Although we have not given any example of refinement of parallel expressions, concurrency is considered by our definition of refinement. In particular, we consider the effects of interference on the intermediate states reached during the execution of an expression. As illustrated in Section 4.1, the expression

```
x := 0 ; x := x + 1
```

is not a refinement of `x := 1` exactly because we consider possible interaction.

In fact, refinement can also be achieved by reducing the possibilities of interferences (from the execution of other operations) during the execution of an expression. This guarantees that the execution of Q in parallel with other expressions is at least as deterministic as the execution of P in parallel with the same expressions. In general, this means that the behaviour we can observe from Q can also be observed from P.

One obvious way of reducing the possibilities for interferences is to reduce the number of intermediate states reached during the evaluation of an expression. For instance, this can be done by providing an atomic implementation of a nonatomic expression, as illustrated in Section 4.1.1, where we proved that

```
x := 0 ; x := x + 1
```

is refined by `x := 1`. In the definition of simulation, this is reflected by condition 1, where a $\rightarrow$-transition from Q can be matched by many transitions from P.

However, note that Q can lead to even more intermediate states than P, as long as those extra states are simulations of neighbouring states, which in turn should be simulations of corresponding intermediate states reached from P. In this way, no extra interference is possible in Q. In other words, refinement can also be obtained by introducing and removing *stuttering* steps (i.e., idle steps modulo observation equivalence of states). In the examples, this was illustrated by the equivalence (defined as refinement in both ways) of

```
skip(x) ; x := 0
```

and `x := 0`. In the definition of simulation, this is also reflected by condition 1, where a $\rightarrow$-transition from Q can be matched by none transition from P.

This is what supports the refinement of atomic operations, since an atomic operation may be refined by a nonatomic operation as long as the additional transitions in the evaluation of the latter correspond to stuttering steps. In this way the sequence of intermediate states reached during the execution of the nonatomic operation can be split in two parts such that all the states in one part are equivalent to the state prior to the evaluation of the operation, and all states in the other part are equivalent to the state after the execution of the operation. Hence this kind of refinement increases the level of parallelism without increasing the level of nondeterminism.

6 Conclusions

We introduced a notion of refinement for FOOPS together with an effective proof technique. This establishes a basis for formal development of software in FOOPS. We described definitions of refinement for FOOPS system states, expressions, and specifications. We also derived basic properties of those relations. In [Borba, 1995], it is proved that the relation of refinement of expressions introduced here is a congruence with respect to the following FOOPS method combiners: sequential composition, parallel composition, result combiner (`result_;_`), conditional, and the

internal choice method combiner (which can be defined in terms of the *external* choice combiner _[]_). Choosing the parallel composition combiner as example, it is proved that

$$p \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q$$

implies

$$p \mid\mid o \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q \mid\mid o,$$

for any expression o formed by experiments of P and the method combiners mentioned above. In [Borba, 1995], it is also derived a weaker compositionality result for the atomic evaluation operator.

We concentrated on the object level of FOOPS and only considered refinement of "flat" specifications. Definitions of refinement and equivalence for the functional level are described in [Goguen, 1994]. Although we focused on FOOPS, our definition of refinement only depends on minimal assumptions about FOOPS and its operational semantics; those assumptions were described here. This suggests that our approach could be adapted for other object oriented concurrent languages having remote procedure call as a mechanism for interaction and synchronization.

In order to illustrate the use of our notion of refinement for stepwise formal development of programs in FOOPS, we considered examples involving memory cells and different implementations of buffers. We also explained how to derive laws of FOOPS method combiners from the definition of refinement. By analysing those examples we observed which aspects are relevant for refinement of concurrent object oriented programs: data refinement (including dynamic data structures), nondeterminism, concurrency, interference, and refinement of atomic operations. We also concluded that the definition of refinement presented here is divergence insensitive; that is, it may relate an expression not having a divergent behavior to an expression having a divergent behavior. Just observe that, by the definition of simulation, a $\rightarrow$-transition from a state can be matched by no transitions from a related state.

We included proofs of refinement together with detailed explanation in order to give a better insight on our approach, by showing how it works in detail. In particular, we can conclude that proofs of refinement using the simulation technique are not complicated. The major task involved is to find a candidate simulation relation. This requires some knowledge about the behaviour of expressions, and the relation between different representations for object states. Checking the conditions from the definition of simulation 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. All those points show that the framework of operational semantics and simulations can be effectively used for defining refinement of concurrent object oriented programs.

Well established approaches for refinement and equivalence of sequential programs [Morgan, 1994, Hoare *et al.*, 1987, Bauer *et al.*, 1989] are of limited use for refinement of concurrent systems, since they are based on semantic frameworks that consider only the relational (input/output) behaviour of programs. By contrast, in order to define a notion of refinement for FOOPS, we used the frameworks of structural operational semantics [Plotkin, 1981] and bisimulations [Park, 1981].

Based on those frameworks, notions of equivalence have also been developed for process algebras [Milner, 1989] and a general notation for giving the semantics of programming languages [Mosses, 1992].

An approach for refinement of object *based* concurrent programs is described in [Jones, 1992], where many examples of formal development of programs are presented. Most examples are simple and elegant, and use a few refinement preserving transformation rules. Assertional "Hoare style" inference rules for reasoning about rely and guarantee conditions [Jones, 1983] are also used for proving refinement. However, no general definition of refinement is proposed; although the semantics of the language presented in [Jones, 1992] is given in terms of a process algebra [Milner *et al.*, 1989], the notions of equivalence and refinement of the algebra cannot be directly used to define the notion of refinement of the language [Jones, 1993].

Another approach for refinement of object oriented concurrent programs is described in [Lano and Goldsack, 1994], which gives an overview of a general notion of refinement for a concurrent object oriented specification language. However, no properties of refinement or associated proof techniques are discussed.

Comparing with those alternative approaches for the refinement of concurrent object oriented programs, our approach is unique in the sense of providing both a general definition of refinement and an effective proof technique.

Acknowledgements

Thanks to Steve Schneider and Grant Malcolm for giving many suggestions which helped to simplify the work described here. The first author also had the opportunity to discuss in detail most of the aspects presented here with Grit Denker and Prof. Hans Dieter Ehrich, during an enjoyable visit to Braunschweig, supported by the IS-CORE project.

References

[Bauer *et al.*, 1989] F. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations—intuition-guided programming. *IEEE Transactions on Software Engineering*, 15(2), February 1989.

[Borba and Goguen, 1994] Paulo Borba and Joseph Goguen. An operational semantics for FOOPS. In Roel Wieringa and Remco Feenstra, editors, *Working papers of the International Workshop on Information Systems—Correctness and Reusability, IS-CORE'94*, Technical Report IR-357. Vrije Universiteit, Amsterdam, September 1994. A longer version appeared as Technical Monograph PRG-115, Oxford University, Computing Laboratory, Programming Research Group, November, 1994.

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

[Goguen and Malcolm, 1994] Joseph Goguen and Grant Malcolm. Proof of correctness of object representations. In A.W. Roscoe, editor, *A Classical Mind: essays dedicated to C.A.R. Hoare*. Prentice Hall, 1994.

[Goguen and Meseguer, 1987] 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.

[Goguen and Winkler, 1988] 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.

[Goguen and Wolfram, 1991] 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.

[Goguen, 1989] 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.

[Goguen, 1994] Joseph Goguen. *Theorem Proving and Algebra*. October 1994. Lecture Notes, Oxford University Computing Laboratory.

[Hennessy, 1988] Matthew Hennessy. *Algebraic Theory of Processes*. The MIT Press, 1988.

[Hoare *et al.*, 1987] C.A.R. Hoare, J. Michael Spivey, Ian Hayes, Jifeng He, Carroll Morgan, A. William Roscoe, Jeff Sanders, Ib Sorenson, and Bernard Sufrin. Laws of programming. *Communications of the Association for Computing Machinery*, 30(8):672–686, 1987.

[Hoare, 1972] C.A.R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[Hoare, 1985] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Jifeng *et al.*, 1986] He Jifeng, C.A.R. Hoare, and Jeff Sanders. Data refinement refined. volume 213 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[Jones, 1983] Cliff Jones. Specification and design of (parallel) programs. In R.E.A. Mason, editor, *Information Processing'83*, pages 321–332. North-Holland, 1983.

[Jones, 1992] Cliff Jones. An object-based design method for concurrent programs. Technical Report UMCS-92-12-1, Department of Computer Science, University of Manchester, 1992.

[Jones, 1993] Cliff Jones. Process-algebraic foundations for an object-based design notation. Technical Report UMCS-93-10-1, Department of Computer Science, University of Manchester, 1993.

[Lano and Goldsack, 1994] Kevin Lano and Stephen Goldsack. Refinement and subtyping in formal object-oriented development. In Roel Wieringa and Remco Feenstra, editors, *Working papers of the International Workshop on Information Systems—Correctness and Reusability, IS-CORE'94*, number IR-357. Vrije Universiteit, Amsterdam, September 1994.

[Milner *et al.*, 1989] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. Technical Report ECS-LFCS-89-85, 86, Laboratory for Foundations of Computer Science, Edinburgh University, 1989.

[Milner, 1971] Robin Milner. An algebraic definition of simulation between programs. Technical Report CS-205, Stanford University, Computer Science Department, 1971.

[Milner, 1989] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

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

[Mosses, 1992] Peter Mosses. *Action Semantics.* Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[Park, 1981] David Park. Concurrency and automata on infinite sequences. In *Proceedings of 5th GI Conference*, volume 104 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

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

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

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

Index

`[_]`, 7
`#_`, 4
`_:=_+_`, 16
`_:=_`, 16
`_[_] := _`, 22
`_[_]`, 22
`_._`, 3
`_;_`, 7
`_[]_`, 7

`aax`, 7
`acx`, 7
ADT, 3, 8–10
`ARRAY`, 22
`Array`, 22
`array`, 22
atomic evaluation, 2, 6, 7, 11, 12
attribute, 4, 8
    definition, 5
    derived, 5
    evaluation, 6
    stored, 5
    visible, 5, 9
`ax`, 4
axioms, 4, 5
    DMA, 6
    IMA, 6

`BUFFER`, 4
`Buffer`, 4
`BUFFERA`, 22

`CELL`, 16
`Cell`, 16
class, 3, 4
`class`, 4
compositionality, *see* refinement
conditional, 7
configuration, 10

data refinement, *see* refinement
database state, 8, 10
    initial, 14
    reachable, 14

`del`, 5, 23
derivative, 10
DMA, *see* axioms

`elems`, 5
`empty?`, 5, 22
`endfmod`, 4
equality, 10
`err`, 22
experiment, 8, 9, 11
    example of, 20, 26
expression
    fully evaluated, 10

`fn`, 3
FOOPS, 1, 2
    method expressions, 2
    refinement for, 2, 8
    semantics, 2
    system, 8, 14
`fst`, 22
`full?`, 5, 22
function, 3
    definition, 3, 4
    evaluation, 8

`get`, 5, 23

`head`, 4
`hidden`, 5

identifier, *see* object
`if_then_else_fi`, 7
IMA, *see* axioms
inheritance, 2, 3

`LIST`, 3
`List`, 3

`MAX`, 4
`max`, 4
method, 2, 4, 8
    atomicity, 12
    definition, 5, 6
    evaluation, 6, 7

visible, 12
method combiner, 2, 6, 7, 29
module, 3, 8
minimum instantiation of, 24

`NAT`, 4
`NeList`, 3
`new`, 7
`next`, 22
`nil`, 3
nondeterministic choice, 7
`null`, 22
`NzNat`, 4

OBJ, 1, 3
object, 3–5
creation, 7, 8
deletion, 7, 8
identifier, 4–6, 8–10
specified, 7
observation, 8, 9
operation, 8
visible, 5, 8

parallel composition, 7
primary sort, *see* sort
`put`, 5, 23

refinement, 2, 8, 16
aspects of, 31
compositionality of, 32
congruence, 32
data refinement, 31
divergence, 33
of expressions, 14, 15, 29
reflexivity of, 15
transitivity of, 15
of specifications, 15
reflexivity of, 15
transitivity of, 15
of states, 8, 13
identity, 14
reflexivity of, 14
transitivity of, 14
proof technique for, 2, 14, 33
proofs, 16, 33
`remove`, 7
`reset`, 5, 23
result, 6
`result_;_`, 6

sequential composition, 7
associativity of, 30
simulation, 8, 10, 14
composition of, 13
definition, 10
example of, 17, 21, 25, 30
properties of, 12, 13
union of, 12
sort, 3
primary, 9, 24
`sort`, 3
specified object, *see* object
stuttering, 32
subsort, 3
`subsort`, 3

`tail`, 4
theory, 3, 4
transition relation, 10
`TRIV`, 3
`TRIVE`, 22

`val`, 16

Contents

1 Introduction **1**

2 Overview of FOOPS **2**
2.1 Functional Level . . . 3
2.2 Object Level . . . 4
2.2.1 Axioms . . . 5
2.2.2 Method Combiners . . . 7
2.2.3 Object Creation and Deletion . . . 7

3 Refinement for FOOPS **8**
3.1 Simulations . . . 10
3.1.1 General Assumptions . . . 11
3.1.2 Properties . . . 12
3.2 Refinement of States . . . 13
3.3 Refinement of Expressions . . . 14
3.4 Refinement of Specifications . . . 15

4 Proving Refinement **16**
4.1 Memory Cells . . . 16
4.1.1 A Simulation . . . 17
4.1.2 Failing to be a Simulation . . . 20
4.1.3 Further Examples of Simulation . . . 21
4.2 Buffers . . . 22
4.2.1 Refinement of Parameterized Modules . . . 23
4.2.2 A Simulation . . . 25
4.3 Laws of FOOPS Method Combiners . . . 29

5 Aspects Considered by Refinement **31**

6 Conclusions **32**