Semantics and Refinement for a Concurrent Object Oriented Language

Paulo Henrique Monteiro Borba

Wolfson College

Oxford University Computing Laboratory

Programming Research Group

Thesis submitted for the Degree of Doctor of Philosophy
at the University of Oxford

Trinity Term, 1995

Abstract

FOOPS is a concurrent object oriented specification language with an executable subset. In this thesis we propose an extension of FOOPS with features for specifying systems of distributed and autonomous objects. This extension supports most features of concurrent object oriented programming, including classes of objects with associated methods and attributes, object identity, dynamic object creation and deletion, overloading, polymorphism, inheritance with overriding, dynamic binding, concurrency, nondeterminism, atomic execution, evaluation of method expressions as background processes, and object protection.

The main contribution of this thesis is to develop a framework for supporting formal development of software in the extension of FOOPS mentioned above. In particular, we introduce a structural operational semantics for FOOPS, a notion of refinement for concurrent object oriented programs, congruence properties of refinement of FOOPS programs, and tools for mechanizing refinement proofs.

The operational semantics is the core of the formal definition of FOOPS. It is used to define notions of refinement for FOOPS states, programs, and specifications. Those notions and associated proof techniques for proving refinement are used to illustrate stepwise formal development of programs in FOOPS. The congruence properties of refinement (with respect to some of FOOPS operators) justify compositional development of software in FOOPS. The tools help to validate the framework introduced in this thesis and motivate its use in practice.

Acknowledgements

I wish to thank Professor Joseph Goguen for his supervision, encouragement, discussions, and interesting suggestions to the work described in this thesis. Most of this work was inspired by his principles and ideas.

I am grateful to Professor Hans-Dieter Ehrich and Mike Reed for examining this thesis and making suggestions to improve it.

Special thanks to Steve Schneider and Grant Malcolm for giving many suggestions which helped to simplify the operational semantics and notions of refinement described in this thesis. I am grateful to Steve Schneider, Adolfo Socorro, and Simone Veglioni for carefully reading and correcting parts of this thesis.

Grit Denker, Professor Hans-Dieter Ehrich, Professor C. A. R. Hoare, He Jifeng, Carroll Morgan, Lucia Rapanotti, Augusto Sampaio, Jeff Sanders, and Ignacio Trejos-Zelaya deserve my gratitude for listening and making comments to the work presented in this thesis.

I wish to thank Adolfo Socorro and Răzvan Diaconescu for helpful discussions about OBJ, FOOPS, and order sorted algebra. Ignacio Trejos-Zelaya also offered support by indicating related bibliography.

Thanks to Silvio Meira, Augusto Sampaio, and André Santos for providing me with inspiration, motivation, and interesting discussions about different aspects of Computer Science.

The work described in this thesis was financially supported by the Brazilian government (CAPES).

Aos meus pais.

Contents

Chapter 1

Introduction

Object oriented analysis, design, and programming methods are being successfully used for development of industrial applications [67, 86, 6]. This success is mainly achieved by providing efficient techniques for reuse, maintenance, and incremental development of software. Those techniques are supported by concepts such as class of objects with unique identifiers, methods, attributes, overloading, inheritance with overriding, dynamic binding, and generic classes [67].

Meanwhile, the importance of formal methods for software development is being gradually recognized [50, 66, 56, 47]. This importance is mainly justified by the high level of reliability of systems developed using languages having a clear mathematical semantics [44, 81, 78] and allowing formal proofs that design steps satisfy (refine) specifications [71, 56, 51, 69, 17].

Indeed, the integration of formal methods with object oriented techniques can be quite fruitful [57, 86]. In this thesis we develop a framework for supporting formal development of software in the concurrent object oriented specification language FOOPS [39]. In this chapter we present an overview of FOOPS' features, followed by an outline of the thesis.

1.1 FOOPS

The acronym FOOPS stands for Functional and Object Oriented Programming System, but we usually use it for the language provided by the system. FOOPS was originally presented in [39]. It was designed with the intention of combining functional and object oriented programming with a logical semantics. Its design was mainly based on ideas from the theories of institutions [33] and order sorted algebra [40], and the languages OBJ [45, 43], Clear [9, 10], and LIL [24, 25].

FOOPS is a concurrent object oriented specification language having an executable subset [39, 79]. It includes a functional language derived from OBJ [45], 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 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 using (conditional) equations. 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 [23, 27], and many additional features.

In this thesis we introduce 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, atomic execution, and evaluation of method expressions as background processes. Furthermore, those constructors may be used to define more complex ones, by using facilities for method combiner definition.

The extension of FOOPS considered here also provides a mechanism for object protection; in general terms, it is possible to specify which objects are allowed to directly invoke methods of a protected object. In fact, this can be used for defining private references (object identifiers) in the sense of [58]; that is, references that can only be used by a particular object. Object protection seems to be essential for specifying and reasoning in a practical way about systems consisting of arbitrary object graphs [58].

1.2 Outline of the Thesis

The main contribution of this thesis is to introduce a framework for supporting formal development of software in the extension of FOOPS mentioned in Section 1.1. In particular, we introduce an operational semantics for FOOPS, a notion of refinement for concurrent object oriented programs, congruence properties of refinement of FOOPS programs, and tools for mechanizing refinement proofs.

Now we describe the structure of the thesis.

In Chapter 2 we give an overview of the mathematical theory of order sorted algebra [40], which is used to define FOOPS operational semantics. We basically summarize the work presented in [40, 34, 21] and introduce some additional notation.

In Chapter 3 we introduce the extension of FOOPS mentioned in Section 1.1, informally describing the operational semantics of most constructs of the language. This introduction is based on some work which was first presented in [7].

In Chapter 4 we present a structural operational semantics [78] for the extension of FOOPS proposed in Chapter 3. We consider most features of concurrent object oriented programming. This operational semantics is the core of FOOPS' formal definition. It was originally presented in [7].

Based on the operational semantics described in Chapter 4, in Chapter 5 we define notions of refinement for FOOPS states, programs, and specifications. We also present a proof technique for proving refinement, and illustrate the use of refinement for stepwise formal development of programs in FOOPS. This is based on some work which was originally presented in [8].

In Chapter 6 we prove that the notion of refinement of FOOPS programs (introduced in Chapter 5) is a congruence with respect to most FOOPS predefined method combiners. We also derive weaker results for some of the combiners that do not preserve refinement. The congruence results justify compositional development of software in FOOPS.

In Chapter 7 we describe tools for mechanizing proofs of refinement of FOOPS states, programs, and specifications. In particular, we describe a symbolic simulator and a proof assistant for FOOPS. We concentrate on the use of the tools, rather than on their implementation. The tools are quite useful for partially validating the framework presented in this thesis.

Lastly, in Chapter 8 we summarize the main results and conclusions of this thesis. We also compare our work with related approaches, and indicate directions for further research.

In Appendix A we present some proofs that were omitted from Chapter 6.

Chapter 2

Overview of Order sorted Algebra

Order sorted algebra (OSA) is a mathematical theory supporting multiple inheritance, overloading, polymorphism, error handling, partial functions, and multiple representation in an algebraic framework [40]. The main idea to solve these problems is the definition of a partial order on the set of sorts of a given specification. This is interpreted as subset inclusion in the algebras that are models of this specification.

Besides being the mathematical theory supporting FOOPS functional level, OSA is used to define FOOPS operational semantics. Here we introduce some notation, definitions, and basic results of OSA, as originally presented in [40, 34, 21] with some additional notation. More motivation and the history of OSA can be found in [40, 34].

2.1 Signatures

Signatures indicate the sorts and operations in a specification. In this section we formalize this notion. First we introduce the notation of sorted (also called "indexed") sets. Given a "sort set" ("index set") S, an S-**sorted set** A is a family $\{A_s \mid s \in S\}$ of sets A_s for each "sort" ("index") $s \in S$. Similarly, given S-sorted sets A and B, an S-**sorted function** $f : A \to B$ is an S-sorted family $\{f_s : A_s \to B_s \mid s \in S\}$. For a fixed S, operations on S-sorted sets are defined componentwise. For example, given S-sorted sets A and B, $A \cup B$ is defined as $(A \cup B)_s = A_s \cup B_s$, for each $s \in S$. We write $|A|$ for the distributed union of all sets in A; that is, $|A| = \bigcup_{s \in S} A_s$. Also, $e \in A$ is an abbreviation for $e \in |A|$.

Here, S is a **partially ordered set** (**poset**), i.e., there is a binary relation $\leq$ on S that is reflexive, transitive, and antisymmetric. We use the extension of the ordering on S to strings of equal length in S^*; that is, $s_1 \ldots s_n \leq s'_1 \ldots s'_n$ iff $s_i \leq s'_i$ for $1 \leq i \leq n$. Similarly, $\leq$ extends to pairs $\langle w, s\rangle$ in $S^* \times S$: $\langle w, s\rangle \leq \langle w', s'\rangle$ iff $w \leq w'$ and $s \leq s'$.

Definition 2.1.1 An **order sorted signature** is a triple $(S, \leq, \Sigma)$ where $(S, \leq)$ is a poset and Σ is an $S^* \times S$-sorted family $\{\Sigma_{w,s} \mid w \in S^* \text{ and } s \in S\}$. Elements of (the sets in) Σ are called operation (or function) symbols, or for short, **operations**.

An order sorted signature is **monotone** iff the operations satisfy the following **monotonicity condition**:

$$\sigma \in \Sigma_{w1,s1} \cap \Sigma_{w2,s2} \text{ and } w1 \leq w2 \text{ imply } s1 \leq s2.$$

When the the poset (S,$\leq$) is clear, we write Σ for $(S, \leq, \Sigma)$. When $\sigma \in \Sigma_{w,s}$ we say that σ has **rank** $\langle w, s\rangle$, **arity** w, and (value, result, or coarity) **sort** s. A special case is $w = \lambda$, the empty string; then $\sigma \in \Sigma_{\lambda,s}$ is a **constant symbol** of sort s. Because of the monotonicity condition, overloaded constants are not allowed, since $\lambda = w1 = w2$ implies $s1 \leq s2$. □

An important observation is that the theory of OSA can be developed without the monotonicity condition for signatures [34]. In fact, it is necessary to avoid this condition in order to model FOOPS database states, as we will see later (see [34] for more motivation for not enforcing the monotonicity condition).

Given a signature $(S, \leq, X)$, we say that X is a **ground signature** iff it is formed only by distinct constant symbols; that is, $X_{\lambda,s} \cap X_{\lambda,s'} = \emptyset$ whenever $s \neq s'$, and $X_{w,s} = \emptyset$ unless $w = \lambda$. For a signature Σ, $\Sigma(X)$ is an abbreviation for $\Sigma \cup X$, if X is a ground signature disjoint from Σ (i.e., $X \cap \Sigma = \emptyset$). In this case, we may call X a Σ-**variable family**.

2.2 Algebras

Algebras are the models that provide actual functions to interpret the operation symbols in a signature.

Definition 2.2.1 Let $(S, \leq, \Sigma)$ be an order sorted signature. An $(S, \leq, \Sigma)$**-algebra** A is a family $\{A_s \mid s \in S\}$ of sets called the **carriers** of A, together with a function $A_\sigma : A_w \to A_s$ for each σ in $\Sigma_{w,s}$, where $s \leq s'$ implies $A_s \subseteq A_{s'}$, $A_w = A_{s1} \times \cdots \times A_{sn}$ when $w = s1 \ldots sn$, and A_w is an one point set when $w = \lambda$. When $(S, \leq)$ is clear, $(S, \leq, \Sigma)$-algebras may be called **order sorted** Σ**-algebras**.

We say that Ω is a **signature of non-monotonicities** for Σ if $\Omega \subseteq \Sigma$. Then, an order sorted Σ-algebra A is **monotone except on** Ω iff

> $\sigma \in \Sigma_{w1,s1} \cap \Sigma_{w2,s2}$ and $w1 \leq w2$ and $s1 \leq s2$ imply $A_\sigma : A_{w1} \to A_{s1}$ equals $A_\sigma : A_{w2} \to A_{s2}$ on A_{w1}, unless $\sigma \in \Omega_{w1,s1}$.

An order sorted Σ-algebra A is **monotone** iff it is monotone except on the empty signature. □

Thus we say that an operation is non-monotonic if its meaning in a subsort is not the same as its meaning in a corresponding supersort, when restricted to elements of the subsort.

We will be interested in certain functions between algebras; essentially, those functions preserve the structure of algebras. Functions having this property are defined below.

Definition 2.2.2 Let $(S, \leq, \Sigma)$ be an order sorted signature, and let A and B be order sorted $(S, \leq, \Sigma)$-algebras. Then an $(S, \leq, \Sigma)$-**homomorphism** $h : A \to B$ is an S-sorted function $h = \{h_s : A_s \to B_s \mid s \in S\}$ satisfying the following **homomorphism condition**:

$$(1)\ h_s(A^{w,s}_\sigma(a)) = B^{w,s}_\sigma(h_w(a)) \text{ for each } \sigma \in \Sigma_{w,s} \text{ and } a \in A_w$$

where $h_w(a) = \langle h_{s1}(a1), \ldots, h_{sn}(an)\rangle$ when $w = s1 \ldots sn$ and $a = \langle a1, \ldots, an\rangle$ with $ai \in A_{si}$ for $i = 1, \ldots, n$ when $w \neq \lambda$. If $w = \lambda$, condition (1) specializes to

$$(1')\ h_s(A^{\lambda,s}_\sigma) = B^{\lambda,s}_\sigma.$$

When the poset $(S, \leq)$ is clear, $(S, \leq, \Sigma)$-homomorphisms are also called (**order sorted**) Σ-**homomorphisms**.

If A and B are monotone, a **monotone** $(S, \leq, \Sigma)$-homomorphism $h : A \to B$ is an $(S, \leq, \Sigma)$-homomorphism satisfying the following **restriction condition**:

(2) $s \leq s'$ and $a \in A_s$ imply $h_s(a) = h_{s'}(a)$.

□

2.3 Terms

The algebra whose carrier sets are formed by the terms we can construct from a given signature Σ is called the **term algebra**; it is denoted by $\mathcal{T}_\Sigma$. In this section we describe an inductive construction defining the term algebra. For an order sorted signature $(S, \leq, \Sigma)$, the term algebra is the least family $\{\mathcal{T}_{\Sigma,s} \mid s \in S\}$ of sets satisfying the following conditions:

- $\Sigma_{\lambda,s} \subseteq \mathcal{T}_{\Sigma,s}$ for $s \in S$;
- $\mathcal{T}_{\Sigma,s'} \subseteq \mathcal{T}_{\Sigma,s}$ if $s' \leq s$;
- if $\sigma \in \Sigma_{w,s}$ and if $ti \in \mathcal{T}_{\Sigma,si}$ where $w = s1 \ldots sn \neq \lambda$, then (the string) $\sigma\underline{(}t1 \ldots tn\underline{)} \in \mathcal{T}_{\Sigma,s}$.

Also,

- for $\sigma \in \Sigma_{w,s}$ let $\mathcal{T}_\sigma : \mathcal{T}_w \to \mathcal{T}_s$ send $t1, \ldots, tn$ to (the string) $\sigma\underline{(}t1 \ldots tn\underline{)}$.

Then we can write $\sigma(t1, \ldots, tn)$ for $\sigma\underline{(}t1 \ldots tn\underline{)}$. It is now easy to check that $\mathcal{T}_\Sigma$ is an order sorted Σ-algebra.

The terms considered above are **ground terms**, in the sense that they involve no variables. In fact, terms with variables can be seen as a special case of ground terms, by enlarging the signature with new constants that correspond to variable symbols. Given an order sorted signature $(S, \leq, \Sigma)$ and a Σ-variable family X, we can obtain a new order sorted signature $(S, \leq, \Sigma(X))$ and form $\mathcal{T}_{\Sigma(X)}$. This can be viewed as an order sorted Σ-algebra, by forgetting the constants in X; let us denote this algebra $\mathcal{T}_\Sigma(X)$. This gives the algebra of Σ-terms with variables in X.

A term may have many different sorts. In particular, if $t \in \mathcal{T}_\Sigma$ has sort s then it also has sort s', for any s' such that $s \leq s'$. A condition on signatures called **regularity** guarantees that every term has a well defined least sort [40]. Here is the formal definition:

Definition 2.3.1 An order sorted signature $(S, \leq, \Sigma)$ is **regular** iff it is monotone, and given $\sigma \in \Sigma_{w1,s1}$ and $w0 \leq w1$, there is a least rank $\langle w, s\rangle$ such that $w0 \leq w$ and $\sigma \in \Sigma_{w,s}$. □

So, given a regular order sorted signature $(S, \leq, \Sigma)$, for any $t \in \mathcal{T}_\Sigma$ there is a least $s \in S$ such that $t \in \mathcal{T}_{\Sigma,s}$; this is called the **least sort** of t and it is denoted by $LS(t)$. In practice, regularity is not a strong restriction since non regular signatures can be translated into regular ones where the rank of an operation is considered to be part of its name.

Considering Σ-variable families X and Y, given an S-sorted map $a : X \to \mathcal{T}_\Sigma(Y)$, there is a unique Σ-homomorphism $\overline{a} : \mathcal{T}_\Sigma(X) \to \mathcal{T}_\Sigma(Y)$ which **substitutes** a term $a(x)$ for each variable $x \in X$ into each term t in $\mathcal{T}_\Sigma(X)$, yielding a term $\overline{a}(t)$ in $\mathcal{T}_\Sigma(Y)$ (see [40]). The map a is called a **substitution** and $\overline{a}(t)$ denotes the application of this substitution to a term t. Usually, we use the alternative notations

$$t(x_1 \leftarrow t_1, x_2 \leftarrow t_2, \ldots, x_n \leftarrow t_n) \text{ and } a(t)$$

instead of $\overline{a}(t)$, where $|X| = \{x_1, \ldots, x_n\}$ and $a(x_i) = t_i$, for $i = 1, \ldots, n$. Moreover, we omit the pair $x_i \leftarrow t_i$ whenever $t_i = x_i$.

The key for developing OSA without the monotonicity condition for signatures is to consider typed (parsed) terms; that is, terms together with their sort information [34]. We introduce the term algebra P_Σ of fully parsed terms associated to Σ. We let P_Σ be the least S-sorted set such that $\sigma \in \Sigma_{w,s}$ and $ti \in P_{\Sigma,si}$, for $i = 1, \ldots, n$, where $w = s1 \ldots sn$, and $s \leq s'$ imply $\sigma.ws(t1, \ldots, tn) \in \mathcal{P}_{\Sigma,s'}$. The definitions introduced so far and the ones to come can be extended in an obvious way for parsed terms, but for simplicity we only consider unparsed terms. Moreover, parsed terms have least sorts even if the related signature is non monotonic or not regular.

Now, given a regular signature Σ, we can define a parsing function $\rho_\Sigma : |\mathcal{T}_\Sigma| \to |\mathcal{P}_\Sigma|$ which transforms an untyped term t into a fully typed term t' such that the sort of t' is the least sort of t. When not confusing, we drop the subscript from ρ. Here is the formal definition:

$$\rho(\sigma(e_1, \ldots, e_n)) = \sigma.wu(\rho(e_1, \ldots, e_n)),$$

where $\rho(e_1, \ldots, e_n) = \rho(e_1), \ldots, \rho(e_n)$, $u = LS(\sigma(e_1, \ldots, e_n))$, and w is the least sequence of sorts of size n such that $\sigma \in \Sigma_{w,u}$ and $LS(e_1), \ldots, LS(e_n) \leq w$. It is easy to extend ρ to equations (see Section 2.4), set of equations and variable families. Here we omit the details. Also, for simplicity, we let unparsed terms be used in places where parsed terms are expected, if the associated signature is regular. In those cases, we assume that an unparsed term t abbreviates $\rho(t)$. In the same way, an unparsed equation might be used when a parsed equation is expected.

2.4 Equations

In this section we give a formal definition of equation.

Definition 2.4.1 For a regular order sorted signature $(S, \leq, \Sigma)$, a Σ-**equation** is a triple $\langle X, t, t' \rangle$ where X is a Σ-variable family and t, t' are in $\mathcal{T}_\Sigma(X)$ with $LS(t)$ and $LS(t')$ in the same connected component of $(S, \leq)$[1]. We will use the notation $(\forall X)\ t = t'$. When the variable set X can be deduced from the context (for example, if X contains just the variables that occur in t and t', with sorts that are uniquely determined or have been previously declared) we allow it to be omitted; that is, we allow unquantified notation for equations. We also say that an equation is **unquantified** if $X = \emptyset$.

Order sorted conditional equations generalize order sorted equations in the usual way, i.e., they are expressions of the form $(\forall X)\ t = t'$ `if` C, where the **condition** C is a finite set of unquantified Σ-equations involving only variables in X (when $C = \emptyset$, conditional Σ-equations are regarded as ordinary Σ-equations). $\square$

For conciseness, sometimes we use variations on the notation for equations: (X, l, r, C) stands for $(\forall X)\ l = r$ `if` C; (l, r, C) is used when $X = \emptyset$; and we write $l = r$ if $X = C = \emptyset$.

[1] Given a poset $(S, \leq)$, let $\cong$ denote the transitive and symmetric closure of $\leq$. Then $\cong$ is an equivalence relation whose equivalence classes are called the **connected components** of $(S, \leq)$.

2.5 Order sorted Equational Deduction

Here we introduce the rules of deduction for OSA with conditional equations given in [40]. This yields a construction for initial and free order sorted algebras as quotients of term algebras by the congruence generated by the rules of deduction from given equations. The details can be found in [40].

Given an order sorted signature $(S, \leq, \Sigma)$ and a set Γ of conditional Σ-equations, we consider each unconditional equation in Γ to be **derivable**. The following rules allow deriving further (unconditional) equations:

(1) *Reflexivity.* Each equation of the form
$$(\forall X)\ t = t$$
is derivable.

(2) *Symmetry.* If
$$(\forall X)\ t = t'$$
is derivable, then so is
$$(\forall X)\ t' = t.$$

(3) *Transitivity.* If the equations
$$(\forall X)\ t = t',\ (\forall X)\ t' = t''$$
are derivable, then so is
$$(\forall X)\ t = t''.$$

(4) *Congruence.* If $\theta, \theta' : X \to \mathcal{T}_\Sigma(Y)$ are substitutions such that for each $x \in X$, the equation
$$(\forall Y)\ \theta(x) = \theta'(x)$$
is derivable, then given $t \in \mathcal{T}_\Sigma(X)$, the equation
$$(\forall Y)\ \theta(t) = \theta'(t)$$
is also derivable.

(5) *Substitutivity.* If
$$(\forall X)\ t = t'\ \texttt{if}\ C$$
is in Γ, and if $\theta : X \to \mathcal{T}_\Sigma(Y)$ is a substitution such that for each $u = v$ in C, the equation
$$(\forall Y)\ \theta(u) = \theta(v)$$
is derivable, then so is
$$(\forall Y)\ \theta(t) = \theta(t').$$

Given a set of equations Γ, there is a congruence $=_\Gamma$ relating two terms iff we can prove that they are equal from the equations in Γ and applications of the rules above. Furthermore, this congruence splits the term algebra $\mathcal{T}_\Sigma$ into equivalence classes of terms modulo Γ, resulting in another Σ-algebra represented by $\mathcal{T}_{\Sigma,\Gamma}$. Hence, given a term t, $[t]_\Gamma$ denotes its equivalence class under Γ, and $[\![t]\!]_\Gamma$ denotes the representative of this class (as shown in [44], this can always be freely chosen without problems so we do not give any more details of its definition).

Lastly, note that the concepts introduced here can be easily extended to consider parsed terms.

2.6 Theory Presentations

Specifications are modelled by the concept of theory presentation.

Definition 2.6.1 An **order sorted theory presentation** (hereafter, presentation) is an ordered 4-tuple, $(S, \leq, \Sigma, \Gamma)$, where $(S, \leq, \Sigma)$ is an order sorted signature and Γ is a set of Σ-equations. $\square$

For a presentation $P = (S, \leq, \Sigma, \Gamma)$, we let $\mathcal{T}_P$, $t =_P t'$, $[t]_P$, and $[\![t]\!]_P$ respectively mean $\mathcal{T}_{\Sigma,\Gamma}$, $t =_\Gamma t'$, $[t]_\Gamma$, and $[\![t]\!]_\Gamma$. Also, given a signature $(S, \leq, \Sigma')$, we use $P \cup \Sigma'$ for the presentation $(S, \leq, \Sigma \cup \Sigma', \Gamma)$.

Now, we extend the definition of presentation to allow non-monotonic operations.

Definition 2.6.2 An **order sorted presentation with a signature of non-monotonicities** is an ordered 5-tuple, $(S, \leq, \Sigma, \Omega, \Gamma)$, where $(S, \leq, \Sigma)$ is an order sorted signature, Γ is a set of *parsed* Σ-equations, and Ω is a signature of non-monotonicities such that $\Sigma - \Omega$ is monotone. $\square$

For reasoning about this kind of presentation, we assume default equations relating monotonic operations having the same name and related ranks. This is necessary because parsed equations are used, so related operations do not necessarily agree on the intersection of their arities. The default equations are in the form: $\sigma.ws(\tilde{x}) = \sigma.w's'(\tilde{x})$, for any $\sigma \in \Sigma_{w,s} \cap \Sigma_{w',s'}$ such that $w \leq w'$ and $\sigma \notin \Omega_{w,s}$, where X is a Σ-variable family, $x_i \in X$, for $i = 1 \,..\, k$, $\tilde{x}$ stands for $x_1, \ldots, x_k$, and $w = LS(x_1), \ldots, LS(x_k)$. We let $\Gamma^\star$ be the union of Γ with default equations.

Hence, for a presentation $P = (S, \leq, \Sigma, \Omega, \Gamma)$, we let $\mathcal{T}_P$, $t =_P t'$, $[t]_P$, and $[\![t]\!]_P$ respectively mean $\mathcal{T}_{\Sigma,\Gamma^\star}$, $t =_{\Gamma^\star} t'$, $[t]_{\Gamma^\star}$, and $[\![t]\!]_{\Gamma^\star}$. Lastly, given a signature $(S, \leq, \Sigma')$, we use $P \cup \Sigma'$ for the presentation $(S, \leq, \Sigma \cup \Sigma', \Omega, \Gamma)$.

Chapter 3

Overview of FOOPS

FOOPS was first presented in [39], whereas [79] and [82] detailed describe a revised design of the language. Based on ideas from [73] and [29], here we introduce an extension of FOOPS for specifying systems of concurrent and autonomous objects.

FOOPS extends OBJ with some concepts from object oriented programming. This motivates two central design decisions (see [46, 39, 79, 82] for more details about the design of FOOPS): data elements are not objects, and classes are not modules. The first distinction is based on the fact that data elements (typically natural numbers) are stateless, but objects (typically buffers) have an internal state that can change with time. In this way, FOOPS provides different constructs for defining abstract data types for values and classes for objects. Consequently, there are two constructs for specifying inheritance. In fact, overloading, polymorphism, and inheritance are also available for the specification of ADTs, by the definition of subsorts (subtypes).

The second design decision recognizes the necessity to have a construction where related classes and abstract data types can be defined together. In FOOPS, this is provided by modules, which are the main programming unit of the language. This is one of the main aspects of FOOPS (also derived from OBJ); it includes a powerful module interconnection language, supporting parameterized modules with semantic interface requirements, which allows the programming style known as "Parameterized Programming" [23].

Further justification for both decisions is given in [46], where this approach is compared with others.

This 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 module: 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. Essentially, programs are written in modules and specifications are written in theories. Furthermore, theories are also used to specify the syntactic and semantic restrictions that must be satisfied by the actual arguments of a parameterized module. In order to specify how a theory is interpreted (satisfied) by another theory or module—necessary, for example, when instantiating a parameterized module—the language provides views, which are bindings indicating how the classes, sorts, and operations symbols of a theory are interpreted in another theory or module.

Here we briefly describe the functional level of FOOPS and some of its parameterized programming features. Following this, we give a detailed description of the object level of FOOPS and intuitions about its operational semantics.

3.1 Functional Level

The functional level of FOOPS is a syntactical variant of OBJ. At this level it is possible to define abstract data types, which are sets of data elements together with associated operations. A FOOPS functional module defines one or more 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`, but it has no constraints about the operations associated to it. Hence, the only requirement that actual arguments to a module parameterized by `TRIV` must satisfy is to have a defined sort. 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:

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

This module is parameterized by the sort of the elements in a list (parameter `E`). In order to define an operation giving the number of elements in a list, we use a built-in module defining natural numbers; the keyword `pr` indicates that the module `NAT` is imported and we do not add or identify data elements of the sorts defined in the module.

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. Hence,

```
nil . nil
```

is a well formed term; that is, the application of `_._` to `nil` and `nil`. Note that `_._` is overloaded, and concatenation of nonempty lists results in a nonempty list. As indicated by the attributes, this function is associative (`assoc`) and has `nil` as identity (`id: nil`).

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. These are introduced by the following declarations:

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

The functions `head` and `tail` are only defined for nonempty lists. This gives the effect of partial functions, by defining them as total on specific subsorts restricting their domain. The meaning of these functions is given by axioms (equations). In a module defining code, equations are interpreted as left-to-right rewrite rules. For the example being discussed, the following equations are necessary:

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

The keyword `var` introduces variables of a given sort, whereas `ax` precedes an axiom, and `endfmod` indicates the end of a functional module.

Instead of writing the first axiom for `#_`, we could have written the equivalent conditional axiom (`cx` indicates that the axiom is conditional):

```
    cx # L = 0 if L == nil .
```

where the condition for which the axiom is valid (or may be applied) follows `if`. Note that every module in FOOPS automatically imports a built-in module of booleans containing the usual operations, and the overloaded equality[1] (`_==_`) and inequality operations (`_=/=_`).

3.1.1 Retracts

An interesting point of FOOPS is how expressions (terms) such as `head(tail(l))`, for a given term `l` of sort `NeList` (written `l:NeList`), are parsed. In fact, `tail(l):List`, but `head` requires an argument of sort `NeList`. Thus, we should conclude that `head(tail(l))` is not a well formed expression. However, as `tail(l)` may be equal to an element of `NeList` (when `l` has more than one element), FOOPS is flexible enough to allow us write this kind of expression which is actually parsed as

```
    head(r:List>NeList(tail(l)))
```

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

where `r:List>NeList` is a special function, called retract, which lower the sort of an expression of `List` to the required subsort `NeList`. It is defined by

```
fn r:List>NeList : List -> NeList .
var NL : NeList .
ax r:List>NeList(NL) = NL .
```

In this way, an expression formed by the application of a retract is only reduced if the argument of the retract has the required subsort. Otherwise, the retract remains as an error message, indicating that the expression is not well parsed. In FOOPS, retracts are automatically defined between related sorts, and inserted in expressions whenever necessary. In a similar way, retracts are also available at the object level.

3.2 Object Level

At the object level, it is possible to define classes, which are collections of objects with same attributes and methods. Attributes correspond to properties of objects; they represent the internal state of objects. Methods are operations that objects can perform; they modify the state of objects. In addition to modifying states, methods may also yield results.

Attributes are atomically evaluated. Methods may be atomically evaluated or not; invocation of methods is synchronous and can be understood as remote procedure calls. An object can be evaluating many nonatomic methods at the same time, including different instances of the same method. Naturally, there are operators for controlling the interference of methods executing in parallel.

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.

An object level module defines one or more classes and related abstract data types. In addition to that, abstract data types defined in functional modules can be imported by object modules. This is how the two levels are integrated. 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
```

where 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. In this instantiation, no view is specified since there is a trivial interpretation—the identity—from the theory constraining the arguments of `LIST` (i.e., `TRIV`) to the theory constraining `E`. The class `Buffer` of bounded buffers is introduced by the declaration

```
class Buffer .
```

Inheritance could also be defined at the object level, by a `subclass` declaration (similar to `subsort`). This implies that any attribute or method associated to a class is also available to its subclasses, since objects of a subclass are also objects of an associated superclass.

Attributes are defined as operations from an object identifier to a value that denotes a current property of the related object. Multi-argument attributes have other arguments in addition to an identifier; this means that this attribute's associated property depends on the extra arguments. 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. Alternatively, we could have added the declaration:

```
hidden elems_ : Buffer -> List .
```

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. The following declarations introduce those methods:

```
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`.

3.2.1 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:

```
var B : Buffer .
var E : Elt .
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 in the form

```
aax m(O) = e .
```

for `m : C -> C´`, is an abbreviation for

```
ax m(O) = [e] .
```

and 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. This is necessary if an expression has to be evaluated without interference from others. Sometimes, nonatomic methods are useful, mainly when efficiency is essential; but they should not be arbitrarily used because it is very difficult to reason about programs consisting of the parallel execution of many nonatomic methods (it is necessary to reason about all possible interleaved interferences caused by those methods).

Lastly, we introduce a (weak) class invariant to `Buffer`; that is, a condition that should be satisfied for all objects of the class, independently of their state. In order to express that all bounded buffers can have at most `max` elements, we introduce the declaration

```
  inv #(elems B) <= max .
endomod
```

In fact, this kind of invariant is also considered valid if all attributes used in the boolean expression are not defined. For example, immediately after an object of `Buffer` is created, `elems` has no associated value (the built-in object creation operation does not initialize attributes, see Section 3.2.3); even so, we consider that the invariant is valid in this initial state.

After creating a buffer, the only possible operation is `reset` because `elems` is not defined; the method `reset` clearly enforces the invariant, since `# nil` is `0`. It is also easy to check that the other operations related to `Buffer` preserve this invariant.

A stronger kind of invariant requires all attributes used in the predicate to be defined. (This can be introduced by the keyword `str-inv`, instead of `inv`.) For example, the boolean expression

```
#(elems B) <= max
```

is not a *strong* invariant for `Buffer`, since the object creation operation does not respect it. In this case, we would have to hide this creation operation and introduce a customized operation that enforces that invariant. Note that in order to check whether a strong invariant is preserved for an object `o`, we should be aware of the effect caused by the deletion of other objects in the system, since some attributes of `o` might be undefined after that.

Constraints like class invariants are just annotations, they have no effect for the semantics of a FOOPS module. In fact, they just document properties of a given specification. They can be seen as proof obligations which, if discharged, might help a lot to reason about specifications.

3.2.2 Method Combiners

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

The semantics of these combiners is given later. Here we informally describe some of them; we suppose the reader has a general intuition about the others, since they are usually available in other programming languages.

Result

The result method combiner (`result_;_`) fully evaluates its first argument (from left to right) and then evaluates the second. When both arguments are fully evaluated, the first one is given as result.

This operator is mainly useful when an expression should yield a specific value, but this value has to be evaluated before other operations are executed. For example, consider the method `get`, defined in Section 3.2.1. Its behaviour could not be easily expressed without `result_;_`.

Indeed, `result_;_` can be used to simulate some of the behaviour provided by the `return` statement in languages such as C and C++, and special conventions for variables names in Pascal and Eiffel for indicating the value to be returned by a function. For instance, the C++ code corresponding to the "FOOPS like" method definition

```
m(O) = e ; X := f ; g ; return X .
```

and the Eiffel code corresponding to

```
m(O) = e ; Result := f ; g .
```

could be represented in FOOPS by

```
m(O) = e ; result f ; g .
```

where `e`, `f`, and `g` are method expressions.

Evaluation in the Background

Here we introduce a method combiner that resembles the UNIX operator `&` for evaluation of a program in the *background*. This means that the operator starts the evaluation of an expression but does not wait until it terminates. Instead, expressions following this operator are evaluated concurrently to the expression in the background. Also, the result generated by the expression in the background is discarded; this expression is only executed for its effects.

The FOOPS method combiner `_&_` starts the evaluation of its second argument (from left to right) in the background, and then yields its first argument. In fact, the UNIX unary postfix operator `_&` may be defined by

```
ax P & = skip & P .
```

In FOOPS, `P &` is a method expression (not a command or program, in UNIX terminology [13]), so it must yield a value; that is the role of the dummy constant `skip` in the axiom above.

The main application of this operator is to start the execution of nonterminating methods. For example, if `m` is non terminating, invoking `m` like in `m(o) ; n(o)` is not very useful because `n(o)` will never be evaluated. Instead, we can use `m(o) & n(o)`. In this way, the evaluation of `m(o)` starts and `n(o)` is concurrently evaluated.

Method Combiner Definition

New method combiners may be introduced as abbreviations for complex method expressions. This can be done by equations. For example, the internal nondeterministic choice operator `_Or_` is defined in terms of external choice by the axiom

```
ax P Or Q = (skip ; P) [] (skip ; Q) .
```

where `P` and `Q` are variables, and `skip` is any functional constant. In this axiom, the arguments of `_[]_` may be immediately evaluated; so, the external choice will be nondeterministic. As desired, this implies that the internal choice does not depend whether its arguments are ready for evaluation or not.

3.2.3 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 from the database state, 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.

The operators for object creation do not assign initial values for attributes. Hence attributes should be explicitly set by special methods, since a non initialized attribute cannot be evaluated. Automatic initialization is not provided here because it can be easily simulated by the operators introduced above together with method combiners and methods for setting attributes. For example, suppose that a class `C` has two stored attributes `a` and `a´`, and methods `set-a` and `set-a´` for assigning values to those attributes. A creation operation for `C` that also initializes those attributes is given by the method combiner `create`, defined by

```
ax create(O) = [new(O) ; set-a(O,v) ; set-a´(O,v´)] .
```

where `O` is a variable of class `C`, and `v` and `v´` are chosen initialization values for the respective attributes. The atomic evaluation operator guarantees that the created object can only be accessed after its attributes are initialized. Similarly to `new`, the operator `create` is not executed by an object; in fact, it should be executed even if its argument is an object identifier that is not in the state. So, it is modelled as a method combiner. Also, in order to behave properly, `create` should only be invoked with an object identifier as argument.

Contrasting to the simplicity of the approach used above, it might be problematic to initialize objects of recursive classes, multi-argument attributes, and to find default values for attributes in general. In fact, it might be the case that some attributes cannot be automatically initialized. That is the main reason for not trying to automatically initialize attributes.

We can also easily simulate creation operations having attribute initialization values as arguments. For instance, the method combiner

```
mc create(_,a = _,a´ = _) : C S S´ -> C .
```

can be used to create objects of class `C`, assigning the values received as arguments to the attributes `a` and `a´` (respectively assumed to be of types `S` and `S´`). This is formalized by the following axiom:

```
ax create(O,a = V,a´ = V´) =
     [new(O) ; set-a(O,V) ; set-a´(O,V´)] .
```

where `O:C`, `V:S`, and `V´:S´`.

Auto-methods

Auto-methods are autonomously invoked in the background when objects of their corresponding classes are created. They may be used to define initialization operations for objects, but their main application is the specification of autonomous (active) objects; that is, objects that automatically perform some operations, instead of waiting for a request from another object. In fact, autonomous objects can simulate (non terminating) processes in an object oriented framework.

Auto-methods can be modeled by standard methods and the operator for background evaluation. For this, we have to provide a customized operation for creation of objects, based on the pre-defined operation `new`. Basically, this customized operation should invoke `new` with the related auto-methods as expressions to be evaluated in the background.

For example, suppose that we want to define `m`, associated to class `C`, as an auto-method. The customized creation operation could be defined by

```
ax create(O) = new(O) & m(O) .
```

where `O` is a variable of class `C`. This operation creates an object of `C` with the identifier given as argument and then invokes `m`. This method may be an initialization operation or a nonterminating method like

```
cx m(O) = n(O) ; m(O) if cond(O) .
```

In this case, the object behaves as a process which is executing `n` whenever the condition `cond(O)` is valid.

3.2.4 Other Aspects of FOOPS

Here we briefly describe some other aspects of FOOPS which are formally specified in other sections of this text. Details about these aspects can be found in [79].

First, FOOPS uses the convention that method and attribute applications are evaluated bottom-up. This means that a method or attribute can only be executed if its arguments are fully evaluated, i.e., the arguments cannot contain any attribute, method, retract, or method combiner symbol. They must be real values: evaluated functional terms or object identifiers. Other evaluation strategies are not appropriate because symbolic (lazy) method or attribute evaluation does not make sense for objects: a method or attribute can only be executed when it has real arguments. However, the order in which the arguments of a method or attribute are evaluated is not fixed. Observe that this is a source of nondeterminism, since the arguments may be evaluated in different contexts.

A flexible way of inheriting properties of a superclass is by redefining some of its methods and attributes. In FOOPS, this is indicated by writing `[redef]` after the declaration of the new operation symbol and rank (i.e., arguments and result type). As FOOPS provides dynamic binding, objects of the subclass use the new version of the operation, unless explicitly stated that the original version is desired; this can be done using the qualified notation `op.C`, where `op` is the operation name and `C` is the name of the superclass having the original version. FOOPS adopts the variant syntactic rule for redefinitions; this means that the arity of the specialized version must be smaller or equal to the arity of the original version, and the result of the first must be greater or equal to the result of the second (see [82] for details). A specialized version of a redefinition of an operation is considered to be a redefinition as well.

Lastly, objects may 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 are in the initial database state of the system, but they have the same status as objects created at runtime; this means that they can be modified and removed.

3.2.5 Protected Objects

In FOOPS, we can create objects that are protected from some other objects, in the sense that a protected object only executes methods directly invoked by a specific and selected group of objects. Roughly, this corresponds to the behaviour provided by hiding and abstraction mechanisms in process algebras, where a process might not be allowed to access some protected channels.

Object protection facilitates programming and reasoning with references (like object identifiers, and pointers in procedural languages) by reducing possible interferences to objects; this is done by restricting the objects that are allowed to request the execution of methods of a protected object. Also, by having arbitrary interference usually one cannot provide full encapsulation of complex objects nor the desired system behaviour; so, the system specification should include explicit, artificial code for avoiding undesirable interferences. However, it seems more appropriate to directly support a mechanism for object protection.

For instance, object protection is quite useful for defining linked lists of cells representing a sequence, because the intermediate cells in the list should only be accessed by their respective previous cell [58]. In fact, only the first cell in the list should accept arbitrary interference. The intermediate cells should be protected. Another example is a simple communication protocol, consisting of two agents and a channel used for communication between them. In this case, the channel should only be accessed by the two agents; it should be protected from other objects, which could disrupt the communication. The channels are truly encapsulated only if they are protected; only in this case the protocol can be reused without restrictions about the environment where it is going to be used.

In particular, one application of object protection is the definition of constant objects; that is, objects that are always in the same state. This can be obtained by creating an object that is protected from any other object. In this way, constant objects cannot be removed. This might be useful for the definition of recursive data structures like linked lists and stacks, where a constant object representing the empty list or stack is usually necessary.

In order to use the mechanism for object protection, we should indicate which objects are allowed to directly request the execution of methods associated to a protected object. This is done at object creation time, by giving a set of object identifiers as argument to `new`. The empty set means that the created object cannot execute any method. Alternatively, `any` may be given as argument, meaning that any object can directly invoke methods of the created object.

The object protection status of an specified object is indicated by a declaration in the form `[pr: status]`, which should appear after the declaration of the object.

For supporting object protection, there are special object creation operations:

- `new.C : Universal -> C`, and
- `new : C Universal -> C`,

for a class `C`, where `Universal` is the type associated to the sets of object identifiers given as argument to the creation operations. In fact, the operators for object creation introduced in Section 3.2.3 can be seen as abbreviations for the operators introduced in this section. Indeed, `new.C()` corresponds to `new.C(any)`, and `new(o)` is the equivalent of `new(o,any)`.

For indicating the desired protection, the following syntactic constructors are available: `_++_`, `{}`, and `any`, where the first one may be used for adding an element to a set, and the second one denotes the empty set. For example, the expression `o ++ o´ ++ {}` denotes the set formed by the identifiers `o` and `o´`.

New objects may be dynamically added to the group of objects that is allowed to invoke methods of a protected object, if the object that requests this operation is part of this group. There is a special operation for doing that:

```
addpr : C Universal -> C,
```

for any class `C`, which adds the objects specified by its second (from left to right) argument to the collection of objects that can invoke methods of the object identified by its first argument. The second argument to the operation above should be constructed with the syntax constructors used for indicating the desired object protection for the creation operation.

A special case of the mechanism for object protection introduced here is provided by languages supporting composite objects (i.e., objects that incorporate others objects, instead of having references to them). Composite objects can be modelled in FOOPS by indicating that the incorporated objects can only be accessed by the object that incorporates them. Also, the notation introduced in [58] supports private references, which can be used by only one object, giving a similar effect to composite objects. (In particular, [58] emphasizes the essential role played by private references for asserting invariants about object graphs and reasoning about them.) Our mechanism for object protection is clearly more general than the mechanism for private references.

3.2.6 Operational Semantics

Here we informally describe some aspects of FOOPS operational semantics. Operationally, a system implemented in FOOPS consists of a database containing information about the current objects in the system. This information can be retrieved by the evaluation of attributes, and modified by the execution of methods or by the deletion and creation of objects. Modifying this information changes the database state.

As in [39], we can represent a state of the FOOPS database, for a specification `P`, by an order sorted presentation with a signature of non-monotonicities. This is necessary in order to indicate non-monotonic operations; that is, operations whose meaning in a subsort is not the same as their meaning in a corresponding supersort, when restricted to elements of the subsort (see Section 2.2).

Presentations representing states are formed by the following components: the definition of the abstract data types of `P`; functions and sorts corresponding to attributes and classes defined in `P`; constants of the sorts representing classes, denoting objects; axioms of `P` specifying the meaning of derived attributes; and equations establishing the values of stored attributes for objects in the database. Also, the subsort relationships in these theories reflect the subclass and subsort relationships in `P`. Using this abstract representation for states, typical operations on states are defined in a natural and simple way.

In order to illustrate the content of a presentation representing a database state, let us consider the specification defined by the module `BUFFER[NAT,SIX]`, where `SIX` is a functional module defining a constant `max` equal to `6`. For this specification, part of a possible database state looks like

```
fth STATE1 is
  ex LIST[NAT] .
```

```
    sort Buffer .
    fn elems_ : Buffer -> List .
    fn empty?_ : Buffer -> Bool .
    ⋮
    var B : Buffer .
    ax empty? B = (elems B) == nil .
    ⋮
```

where we represent a presentation with a signature of non-monotonicities with the same syntax as a FOOPS module (assuming that non-monotonic operations are indicated by the tag `[redef]`, and unparsed equations are used when there is no ambiguity). This state contains the functional part of the specification (i.e., `LIST[NAT]`), a sort corresponding to the class `Buffer`, functions representing attributes, and their associated axioms.

In addition to that, the following declarations indicate that the class `Buffer` has three objects in this state (identified by `b1`, `b2`, and `b3`):

```
    fns b1 b2 b3 : -> Buffer .
    ax elems b1 = nil .
    ax elems b2 = (1 . 2) .
  endfth
```

where the equations specify values for their stored attributes. Note that `b3` has not been initialized.

Using this representation, typical operations on states can be easily defined. For instance, given a database state, attributes are evaluated by reducing the corresponding expression in the module representing the database. For example, considering `STATE1`, the evaluation of

`elems(b2)` and `empty?(b1)`,

respectively result in `1 . 2` and `true`; this can be deduced by equational reasoning, from the equations in `STATE1`. Method execution changes the state of the database. For example, executing `put(b1,5)` in `STATE1` changes the database to the state represented by a presentation in the form

```
  fth STATE2 is
    ⋮
    ax elems b1 = 5 .
    ax elems b2 = (1 . 2) .
  endfth
```

containing the same information as `STATE1`, except that the equation `elems b1 = nil` is replaced by `elems b1 = 5`.

Also, adding the object `b4` to `STATE2` results in a state with one more constant of sort `Buffer`:

```
  fth STATE3 is
    ⋮
    fns b1 b2 b3 b4 : -> Buffer .
    ax elems b1 = 5 .
    ax elems b2 = (1 . 2) .
  endfth
```

On the other hand, removing the object `b2` from `STATE3` yields a state in the form

```
fth STATE4 is
  ⋮
  fns b1 b3 b4 : -> Buffer .
  ax elems b1 = 5 .
endfth
```

The object and its related equations were removed from the database.

Remember that the value of a redefined attribute usually does not agree with the value of its original version for objects of the subclass. So, the same should be allowed for the functions modelling those attributes in database states. That is why we use order sorted presentations with a signature of non-monotonicities to model database states; just order sorted presentations are not adequate for doing that in an elegant way. For example, a specification containing

```
pr NAT .
subclass C < C´ .
at a : C´ -> Nat .
at a : C -> Nat [redef] .
var X´ : C´ .
var X : C .
ax a(X´) = 0 .
ax a(X) = 1 .
```

should have database states in the form

```
ex NAT .
subsort C < C´ .
fn a : C´ -> Nat .
fn a : C -> Nat [redef] .
var X´ : C´ .
var X : C .
ax a.C´Nat(X´.C´) = 0.Nat .
ax a.CNat(X.C) = 1.Nat .
```

where parsed equations are used to avoid ambiguity. This allows both versions of `a` to have different definitions without generating any inconsistency. On the other hand, if states were represented by order sorted presentations without a signature of non-monotonicities, unparsed equations would be used, being possible to prove that `0` is equal to `1` using the equations defining `a`. This is obviously not desirable; it would also mean that the semantics of the functional level is affected by the semantics of the object level.

Chapter 4

An Operational Semantics for FOOPS

Based on the informal description of FOOPS given in the previous chapter, we now define a structural operational semantics [78] for FOOPS, considering features such as classes of objects with associated methods and attributes, object identity, dynamic object creation and deletion, overloading, polymorphism, inheritance with overriding, dynamic binding, concurrency, nondeterminism, atomic execution, definition of method combiners, evaluation of expressions as background processes, and object protection. However, we concentrate on the object level of FOOPS. An operational semantics for the functional level may be found in [60].

We adopt a special approach for modelling states of the operational semantics. Adapting some ideas from [39], we use order sorted theory presentations [40, 34] to represent states. This has the advantage of using the power of the theory of ADTs for defining operations on states and reasoning about them. As states are represented by an abstract structure, the semantics is defined in a simple way. In fact, lots of complications are avoided and a concise semantic definition can be obtained, although the language supports many features. Also, the use of this approach clearly facilitates the definition of the semantics, since the original FOOPS design used concepts from order sorted algebra.

Along with the semantic description, we give comments that clarify many concepts and phenomena related to object oriented languages. In particular, we show how the semantics suggests an appropriate programming style for FOOPS and indicates how inconsistencies in FOOPS specifications may be avoided. We also justify the semantics adopted for some constructs, relating it to alternatives. In special, we compare an interleaving parallel composition operator with a parallel composition operator which allows simultaneous evaluation of expressions. It turns out that these operators are equivalent in the context of FOOPS, given some mild assumptions on the notion of equivalence of programs used for the language.

This chapter is structured in the following way. First we give the basis for the semantic definition; we introduce formal definitions and operations for FOOPS signatures, specifications, and (runtime) database states. Then we gradually describe the semantics; we give rules for function, method, and attribute evaluation, followed by rules for method combiners, dynamic creation and deletion of objects, and a mechanism for object protection.

4.1 Signatures and Specifications

A FOOPS module defines a signature and a specification. A FOOPS signature contains a sort and a class hierarchy, and names (together with typing and overriding information) of functions, methods, and attributes. A FOOPS specification is formed by a signature and some axioms (equations) that specify properties of the elements of the related signature.

Later, we show that signatures should also provide information about method combiners and other features supported by FOOPS. Now, we just give a simplified definition which will be extended when necessary.

Definition 4.1.1 A FOOPS **signature** consists of

1. A "sort set" $U = S \cup C$, where S has sort names and C has class names. The sets S and C are disjoint because a sort and a class cannot have the same name. The sort `Bool` (for boolean) is in S.

2. A partial order $\leq$ on U, which establishes the sort and class hierarchy. Classes and sorts are not related: $u \leq t \Rightarrow u \in S \Leftrightarrow t \in S$, for any $t, u \in S \cup C$.

3. A $U^* \times U$-sorted family $\Sigma = F \cup A \cup M$, where F, A, and M respectively contain names for functions, attributes, and methods. Functions are related to sorts: $F_{w,u} = \emptyset$ if $wu \notin S^+$; F has the standard boolean operations; attributes have one class parameter at least: $A_{w,u} = \emptyset$, if $w \in S^*$; specified objects are related to classes: $M_{\lambda,u} = \emptyset$, if $u \in S$; methods have a class parameter: $M_{w,u} = \emptyset$, if $w \in S^+$; and there are retracts

$$\texttt{r:A>B : A -> B} \in \mathit{Retr}_{\texttt{A},\texttt{B}},$$

if $\texttt{A} \cong \texttt{B}$, where $\mathit{Retr} \subseteq (F \cup A)$. Lastly, a method and an attribute with related ranks cannot have the same name, in order to avoid mixing methods up with attributes (i.e., for any $\sigma \in A_{w,u}$, there are no w' and u' such that $wu \leq w'u'$ or $w'u' \leq wu$, and $\sigma \in M_{w',u'}$).

4. A family $R \subseteq A \cup M$ formed by names of redefined methods and attributes. As specialized versions of redefined operations are considered to be redefined, if $\sigma \in R_{w,u} \cap \Sigma_{w',u'}$ and $w'u' \leq wu$ then $\sigma \in R_{w',u'}$.

□

Sometimes, we use Σ to denote the signature $(U, \leq, \Sigma, R)$. Furthermore, we rely on the fact that a FOOPS signature Σ can be seen as the order sorted signature $(U, \leq, \Sigma)$. In this way, the notation and concepts related to order sorted signatures (e.g., terms, least sort, equations, etc.) are available for FOOPS signatures as well.

The constraints imposed on the components of a signature correspond to some of the restrictions enforced on FOOPS modules. For instance, as a module defining the abstract data type of booleans is automatically included in any FOOPS module (for allowing conditional equations), signatures must have a sort `Bool` with its associated operations. Also, as discussed in Section 3.2.4, the restriction on R is enforced on the operations of FOOPS modules.

In some cases, a more general approach is used, by not imposing restrictions on signatures components. So, the semantics of some constructs is indirectly given, by translation to more general constructs. As long as this approach does not complicate the semantic definition, we use it and indicate how the translation can be done. For example, if booleans were not assumed to be in signatures, we would only be able to give the semantics of equations having a set of pairs of terms as a condition (following OSA), instead of a boolean expression (as actually supported by FOOPS). In this case, we would have to specify how the second kind of equation can be seen as a particular case of the first.

The generalization mentioned in the example above would slightly complicate the semantics. Hence, we do not use it. However, the semantics for qualification notation for redefined operations (i.e., `m.C` where `m` is redefined and `C` is a class name) is indirectly given and does not affect the semantics. Basically, we do not assume that signatures include a special (qualified) operation name for each redefined operation. Instead, we consider that the FOOPS signature corresponding to a module containing

```
subclass A < A´ .
subclass B < B´ .
subclass C < C´ .
me m : A´ B´ -> C´ .
me m : A B -> C [redef] .
```

is the same as the signature associated to a module with the declarations above plus the following one:

```
me m.A´ : A´ B´ -> C´ .
```

which provides a qualified notation for `m`. We assume that symbols containing "." cannot be used as operation names in FOOPS modules (unless it corresponds to the qualified notation of some operation). Hence, this additional declaration does not introduce any conflict and `m.A´` can be used to access the original version of `m`.

The same technique can be used to support qualified notation for attributes. However, note that it is meaningless to ask for the original version of a redefined stored attribute for an object of the subclass, since it has no associated value (it is neither directly stored in the state nor necessarily equal to the specialized version). The same happens for original versions of derived attributes defined in terms of redefined stored attributes. Hence, we do not need to provide a qualification notation for this kind of attribute.

An obvious motivation against allowing methods and attributes to have the same name and related ranks is that two operations cannot be distinguished if they have the same name and rank. A less obvious motivation is illustrated by the following signature:

```
pr NAT .
subclass A < A´ .
at a : A´ -> Nat .
me a : A -> Nat .
```

In this way, `a` may be interpreted as a method or as an attribute, depending on its argument. Indeed, in some expressions, it might be the case that `a` is parsed as an attribute and, after some argument evaluation, it is parsed as a method. Besides being confusing, this has bad consequences, mainly for expressions in the RHS of DMAs and conditions of equations, where no methods are allowed. For instance, for a method `m`, the following

```
var  X : A´ .
cx m(X) = ... if 0 <= a(X) .
```

is a valid equation because `a` is parsed as an attribute in `a(X)`. However, if `m` is not redefined, the evaluation of `m(o)`, for `o` of class `A`, requires the evaluation of the condition `0 <= a(o)` which requires one method invocation, since `a` is parsed as a method in `a(o)`. But this is not supported by FOOPS—there is no reasonable semantics for that. The same problem happens if this confusion occurs in the RHS of a DMA, or if an attribute updated by a DMA is sometimes parsed as a method.

In fact, this problem can also happen with pathological non regular signatures satisfying the restriction discussed above, but not with regular signatures (the ones that are of interest for specifications, as we will see later). Also, observe that there is no confusion between functions and methods (or attributes) because functions ranks are not related (by $\leq$) to method (or attribute) ranks, since they only have sorts.

Lastly, we could have added one more constraint on signatures: an operation should only be in R if it is overloaded by another with a greater rank. However, by analysing the semantic rules given here, we conclude that this constraint does not affect the semantics; that is, the meaning of a non overloaded operation is the same whether it is considered redefined or not. Hence, we do not introduce this restriction.

Now, we give a definition of FOOPS specification.

Definition 4.1.2 A FOOPS **specification** is formed by a signature Σ and a set E of Σ-equations containing standard equations for the boolean operations (as given in [45], for example) and retract equations `r:A>B(X) = X`, for any types `A` and `B` in the same connected component of U (with respect to $\leq$), and a variable `X` of least type `B`. □

Hereafter, we use A to refer to the family of attributes in a specification P, when P is clear from the context; otherwise, we use the notation $A(\mathsf{P})$. The same convention is used for the other components: E, S, C, F, etc. Also, FE and AE respectively denote the set of functional and attribute equations in E. The set of functional equations is defined as the largest set of F-equations included in E, whereas AE is defined as the largest set of $F \cup A$-equations included in E and disjoint from FE.

Implicitly, the last definition requires Σ to be regular, since E cannot be empty and (unparsed) equations only make sense for regular signatures (see Section 2.4). As we have seen in Section 2.3, regularity is not a big restriction. Furthermore, this guarantees the least type (parse) for method expressions; so, we can work with untyped terms without ambiguity, leading to a conciser and simpler semantic description.

Observe that the definition above does not restrict the form of axioms. However, by the operational semantics that is described here (as we will see later), only DMAs and IMAs determine the behaviour of methods; other kinds of method axioms are irrelevant for the semantics.

It is also important to note that equations may be non sort (or class) decreasing. This flexibility is usually desired for specifications in general. It affects the semantics because the evaluation of an expression might then lead to an expression of a greater sort. If this happens and the expression is the argument of an operation that cannot be applied to an argument of greater sort, the evaluation will result in a non well formed term. The solution for this problem is given in later sections, but it basically consists of inserting retracts (in order to lower the sort of expressions) during the evaluation of arguments and some pre-defined method combiners.

Even if *method* and *attribute* equations were sort decreasing, retracts would have to be included in every specification, in order to avoid the problem discussed above. This happens because attribute evaluation is specified in terms of an operation which gives the representative of an equivalence class (see Section 2.5); for specifications in general, we cannot guarantee that an equivalence class has a term with a smaller type than all other terms. Hence, attribute evaluation might yield a term of greater type, or even of a non related type.

On the other hand, retracts could be avoided if *all* equations were sort decreasing. For this to work, representatives should be of a least type. This is feasible assuming that functional equations are sort decreasing as well.

Sometimes we use specification (signature) when we refer to a FOOPS module. In those cases, we actually mean the specification (signature) corresponding to that module.

4.2 Database States

As discussed in Section 3.2.6, a FOOPS database state can be represented by a presentation with a signature of non-monotonicities. Here we make more precise what are the contents of such a presentation.

First, we assume that for any specification P there is an associated C-sorted family $I(\mathsf{P})$ (just I, when not confusing) of disjoint components; that is, $I_u \cap I_{u'} = \emptyset$, if $u \not\equiv u'$. Each component is formed by symbols which can be used as identifiers of objects of a given class. This fixed connection between identifiers and classes is necessary because we are representing those concepts in the framework of OSA; so, each symbol should have a fixed, pre-defined rank. From an implementation point of view, this is essential for static type checking of expressions. Here we assume that I is provided by the FOOPS system.

In order to ensure least parse of terms, we assume that identifiers cannot have the same name as functional constants (formally, $|I| \cap F_{\lambda,u} = \emptyset$, for any $u \in S$). Lastly, the family I must contain the identifiers of the objects specified in P: $M_{\lambda,u} \subseteq I_u$, for any $u \in C$. Hereafter I may alternatively be seen as an $U^* \times U$-sorted family, by considering that $I_{\lambda,u} = I_u$, for $u \in C$; and $I_{w,u} = \emptyset$, for $w \neq \lambda$ or $u \notin C$.

Definition 4.2.1 For a specification P, a **database state** is a presentation with a signature of non-monotonicities, consisting of the following components:

1. A signature $(U, \leq, D)$, where $D = F \cup A \cup \mathit{Id}$, for some $\mathit{Id} \subseteq I$ containing the identifiers of the objects in this state.

2. A signature $\Omega = R \cap A$ of non-monotonicities, containing redefined attributes.

3. A set $DE = FE \cup AE \cup \mathit{IdE}$ of parsed D-equations, for some finite set IdE of equations establishing the values for some of the stored attributes of objects in Id. Actually, (D, DE) has to be a conservative extension of $(F \cup \mathit{Id}, FE)$, in the sense that the equations in DE should not relate functional expressions nor object identifiers that cannot be related by the

equations in FE[1].

□

As in the convention for specifications, we use D, DE, and Id for the corresponding components of a database state $\mathcal{D}$, when it is clear from the context; otherwise, we use $D(\mathcal{D})$, etc.

Observe that Id and IdE are the components of the database that can change from one state to another, by the execution of expressions which may create, remove, or change the state of objects.

Strictly speaking, signatures of presentations representing database states should have a universal type, in order to guarantee that equational satisfaction is closed under isomorphism [40]. However, any useful subset of FOOPS should include method combiners, which require the existence of a universal type (see Section 4.4). Hence, we assume that this is provided for any specification. Otherwise, it could be simply added to the signature of the definition above.

The restriction on the equations of database states is important to guarantee that constructs from the object level do not interfere with the semantics of the functional level. Otherwise, the functional theory associated to states would not be related to the specified functional theory. This would mean that the results yielded by expressions evaluated in states would not have the same meaning as the corresponding elements of the specified functional theory. If those restrictions on equations cannot be satisfied, the specification will not have any associated database state. This might happen if the equations in AE are contradictory. For instance, a specification including

```
pr NAT .
class C .
at a : C -> Nat .
var X : C .
ax a(X) = 0 .
ax a(X) = 1 .
```

where `a` is not redefined, violates the restriction because the attribute equations relate two functional expressions (`0` and `1`) which are not related by `NAT` (assuming this is a specification of the abstract data type of natural numbers). In fact, this specification has no associated database state.

Observe that some attributes may have no associated value in a particular database state (this implies that they cannot be evaluated). The restrictions on database states do not prevent this. An advantage of this flexibility is that `new` does not have to initialize attributes; as briefly discussed before, default values for attributes might not even be available for specifications such as

```
pr NAT .
class List .
at val_ : List -> Nat .
at next_ : List -> List .
⋮
```

which specifies a recursive class of linked lists. Also, the operation `remove` does not need to assign an *ad hoc* `nil` or `void` value to attributes containing the identifier of the object to be removed, in

[1]Formally, for a given (D, DE)-algebra Alg, monotone except on Ω, there exists a monotone $(F \cup Id, FE)$-algebra Alg' and an injective $(F \cup Id)$-homomorphism from Alg' to $Alg \restriction F \cup Id$.

order to avoid dangling identifiers. Instead, after the execution of `remove`, those attributes have no associated value. For example, the deletion of `l2` from a state in the form

```
⋮
fns l1 l2 : -> List .
ax val l1 = 4 .
ax next l1 = l2 .
```

simply yields a state in the form

```
⋮
fn l1 : -> List .
ax val l1 = 4 .
```

where the attribute `next l1` is simply not defined.

We let D_{P} be the family of all database states for a given specification P; that is, the family of all presentations (with a signature of non-monotonicities) that satisfy the requirements in Definition 4.2.1, for a fixed P. Note that D_{P} is not necessarily the family of all database states reachable from the initial one by the execution of method expressions. Naturally, this family is contained in D_{P}.

Lastly, for a database state $\mathcal{D}$ and some $t \in \mathcal{T}_D$, we assume that the choice of the representative $[\![t]\!]_{\mathcal{D}}$ of the equivalence class $[t]_{\mathcal{D}}$ is a functional term or an object identifier whenever this is possible (i.e., $[\![t]\!]_{\mathcal{D}}$ is in $\mathcal{T}_{F \cup Id}$ if $|\mathcal{T}_{F \cup Id}| \cap [t]_{\mathcal{D}} \neq \emptyset$). We do not give any more details on the definition of representatives. Instead, we let it be defined when needed.

4.2.1 Operations on Database States

In addition to the usual operations associated to presentations (e.g., $[\![\, e \,]\!]_{\mathcal{D}}$), some specific operations on database states (presentations with a signature of non-monotonicities) are necessary for defining the operational semantics. Here we introduce them. First, we define an operation that updates databases. Later, we give operations for adding and removing objects from databases.

Updating Databases

The update of a database $\mathcal{D}$ with equations Γ is denoted by $\mathcal{D} \oplus \Gamma$. Basically, this operation adds and removes some equations from a database. The added equations, denoted by Γ, establish "new" values for attributes. The removed equations are the ones that specify "old" values for the updated attributes.

First, we define the operation $\oplus$ for overwriting a set of equations by an unquantified, unconditional equation. Informally, for a set of equations Γ and an equation e, $\Gamma \oplus e$ is a set consisting of e and all equations in Γ whose LHS or RHS is not (syntactically) the same as the LHS of e. Note that we may refer to the term "equation" when we actually mean "parsed equation".

Definition 4.2.2 The **overwriting** of a finite set of Σ-equations by an unquantified, unconditional Σ-equation is defined by the following equations:

- $\emptyset \oplus (l, r) = \{(l, r)\}$;

- $(\Gamma \cup \{(X, l', r', C)\}) \oplus (l, r) = \Gamma \oplus (l, r)$, if $l \equiv l'$ or $l \equiv r'$; otherwise,
- $(\Gamma \cup \{(X, l', r', C)\}) \oplus (l, r) = (\Gamma \oplus (l, r)) \cup \{(X, l', r', C)\}$,

for any set of equations Γ, and any Σ-equations (X, l', r', C) and (l, r). $\Box$

Assuming that l is the application of an attribute to arguments, $\Gamma \oplus (l, r)$ gives a set of equations derived from Γ by adding the equation (l, r), and deleting all equations specifying the value of the attribute denoted by l, .

We need an auxiliary concept in order to extend the definition of overwriting for a set of equations. A set of unquantified, unconditional equations is called contradictory if it has two different equations composed by the same term. The following definition formalizes this.

Definition 4.2.3 A set Γ of unquantified, unconditional Σ-equations is **contradictory** if it contains two different equations (l, r) and (l', r') such that $l \equiv l'$ or $l \equiv r'$. $\Box$

Notice that this is a syntactical definition in the sense that a set containing two equations with the same LHS but different RHS is considered contradictory, even if the RHS are equivalent (modulo some equations). The definition of overwriting is also syntactical in a similar sense. This is appropriate for our purposes in this text.

Now, we can define overwriting for a set of equations.

Definition 4.2.4 Given two finite sets of Σ-equations Γ and $\Gamma' = \{e_1, \ldots, e_k\}$, for $k \geq 1$, if Γ' is a non contradictory set of unquantified, unconditional equations then the **overwriting** of Γ by Γ', denoted $\Gamma \oplus \Gamma'$, is defined as $\Gamma \oplus e_1 \oplus \cdots \oplus e_k$. Also, $\Gamma \oplus \emptyset$ is defined as Γ. $\Box$

Note that this uniquely defines the overwriting operation since Γ' is non contradictory, so $\Gamma \oplus e_i \oplus e_j$ is the same as $\Gamma \oplus e_j \oplus e_i$, for any $i, j \leq k$.

Lastly, we introduce the definition that can be used to update database states.

Definition 4.2.5 The **overwriting** of a presentation (with a signature of non-monotonicities) $P = (S, \leq, \Sigma, \Omega, \Gamma)$ by a non contradictory finite set of unquantified, unconditional Σ-equations Γ', denoted $P \oplus \Gamma'$, is the presentation $(S, \leq, \Sigma, \Omega, \Gamma \oplus \Gamma')$. $\Box$

Adding Objects to Databases

The operation $\cup$ adds some operation symbols to the signature of a presentation (see Section 2.6). So, it can be used to add objects to a database, without any initialization, if the symbols represent object identifiers. In this way, $\mathcal{D} \cup Id$, for a $U^* \times U$-sorted family Id of object identifiers, adds the identifiers in Id to the database $\mathcal{D}$ of a specification P.

Removing Objects from Databases

The operation for removing objects from databases deletes object identifiers from the signature of a given presentation. Moreover, the equations formed by terms containing these symbols are removed as well. This means that all references to an object are removed after this object is deleted; that is, the attributes containing these references do not have any associated value in the resulting database. Here is the formal definition:

Definition 4.2.6 The **deletion** of a $S^* \times S$-sorted family Id of operation symbols from a presentation (with a signature of non-monotonicities) $P = (S, \leq, \Sigma, \Omega, \Gamma)$, represented by $P \ominus Id$, is the presentation $(S, \leq, \Sigma - Id, \Omega - Id, \Gamma')$, where Γ' is the set of all $(\Sigma - Id)$-equations in Γ. □

4.3 Methods, Attributes, and Functional Expressions

Now, we start to describe a structural operational semantics for the object level of FOOPS. In this section, we concentrate on the semantics of functional expressions, methods, and attributes. In the following sections, we progressively give the semantics of other language features.

Here we use the approach for operational semantics introduced in [78]. We assume some familiarity with that. The semantics is given by a relation that indicates how an expression is evaluated in a database state. A pair formed by a method expression and a database state is called a configuration. The relation specifies the transitions from one configuration to another, according to how an expression is evaluated and how it changes the database; each transition corresponds to a computational step during the evaluation of an expression.

Let us formalize those concepts. First, for a specification P, we define the family $\mathcal{T}_{\mathsf{P}} = \mathcal{T}_{\Sigma \cup I}$ of method expressions (without variables). By the definition of I, $\Sigma \cup I$ is regular whenever Σ is. Regularity of Σ is guaranteed by the definition of specification. This implies the existence of a least type (parse) for method expressions. That is why we can use untyped (unparsed) terms to define the semantics. The typing functions LS and ρ can be used whenever some type information is necessary.

Second, for a specification P, the semantics is given by a transition relation

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

on configurations, where $Conf(\mathsf{P}) = \mathcal{T}_{\mathsf{P}} \times D_{\mathsf{P}}$. This relation is inductively defined over the syntax of method expressions by inference (transition) rules which indicate how we can infer that two configurations are related (i.e., there is a transition from one to the other), assuming that some others are related. Only transitions that can be deduced from the inference rules that we will give in this text are allowed. In other words, $\longrightarrow_{\mathsf{P}}$ is the least relation satisfying these inference rules.

Note that there is no fixed relation between the object identifiers used in a method expression and the ones in the database state where the expression is going to be evaluated. So, even expressions with identifiers of non-existing objects may be evaluated. Naturally, this will only be successful if these identifiers are not necessary for the evaluation of the expression.

For conciseness, we use $\longrightarrow$ for $\longrightarrow_{\mathsf{P}}$, when P is clear from the context. Also, $P \longrightarrow P'$ stands for $(P, P') \in \longrightarrow$, where P and P' are configurations.

4.3.1 Functional Expressions

The semantics of the functional level of FOOPS is basically given in [22] and [60]. Following these, we introduce one rule for evaluating functional expressions. But first, we give the following notation, for a specification P:

- database states $\mathcal{D}, \mathcal{D}' \in D_{\mathsf{P}}$; and

- functional terms: $v_i \in \mathcal{T}_F$, for $i = 1..k$, for some natural number k. Also, $\tilde{v}$ is an abbreviation for $v_1, \ldots, v_k$.

The evaluation of a functional expression is performed in one transition and yields the evaluated form of this expression. This is only done if the expression is not already in its evaluated form. Here we consider that the evaluated form of a functional expression is the representative of its equivalence class with respect to its related functional theory. The Rule **Fun** (for *fun*ctional) formalizes those aspects.

Rule 4.3.1 (Fun) For any $f \in F$,

$$\frac{}{\langle f(\tilde{v}), \mathcal{D}\rangle \rightarrow \langle [\![f(\tilde{v})]\!]_{FE}, \mathcal{D}\rangle}$$

if $f(\tilde{v}) \not\equiv [\![f(\tilde{v})]\!]_{FE}$. □

From the rule above, we can observe that a function cannot be evaluated unless its arguments are functional terms. This restriction is essential to ensure that the operational semantics motivates a reasonable equality on method expressions—one that preserves functional equality. In order to illustrate the need for this restriction, consider the following specification:

```
pr NAT .
fn f : Nat -> Nat .
fn g : Nat -> Nat .
var X : Nat .
ax f(X) = X + X .
ax g(X) = 2 * X .
```

By equational reasoning, we can prove that `f(n)` is equal to `g(n)`, for any term `n` of sort `Nat`. Now, suppose that we extend the above functional specification with

```
class C .
me m : C -> Nat .
```

and assume that `m` changes the state. If functions can be evaluated with non functional arguments, usually `f(m(o))` will not be equal to `g(m(o))`, for some `o:C`, contradicting what we proved before about `f` and `g`; so, functional equality is not preserved. This happens because `f(m(o))` may be evaluated to `m(o) + m(o)`, whereas `g(m(o))` may be evaluated to `2 * m(o)`. Clearly, the evaluation of the first resulting expression invokes `m` twice, whereas the evaluation of the second invokes `m` just once. Moreover, the state changes and the generated results may be different, for each invocation.

4.3.2 Attributes

Before giving the semantics of attribute evaluation, we introduce some notation. For a specification P, hereafter consider arbitrary

- sort and class symbols $u, u', u_i, u'_i \in U$, for $i = 1..k$; and

- object identifiers and evaluated functional terms not having retracts: $v_i \in \mathcal{T}_{(F-Retr)\cup I}$, for $i = 1..k$, where $LS(v_i)$ is u_i, and $v_i \in \mathcal{T}_F$ implies $v_i \equiv [\![v_i]\!]_{FE}$. Also, we let v and v' be in $\{v_1, \ldots, v_k\}$, where $LS(v) = u$.

We write "fully evaluated term" to refer to object identifiers and *evaluated* functional terms not having retracts.

For simplicity, we consider that the first argument (from left to right) of an attribute or method is the identifier of the object that will perform the associated operation. As we have not imposed a corresponding restriction on signatures, we should show how the semantics of the more general case is derived from the semantics which assumes that simplification. But this can be easily done (by changing the position of parameters, for instance), so we omit the details.

Attributes can only be evaluated if its associated object exists; so the first argument of the attribute should be the identifier of an object in the database state being used for evaluation. Also, the other arguments must be fully evaluated before evaluation. This evaluation is atomic, only reads the database used for evaluation (so, the state does not change), and yields the value of that attribute in this particular state. This value is determined by the equations in that state. The evaluation is only possible if the attribute has an associated value in that state; otherwise, the evaluation is suspended. For example, non initialized attributes cannot be evaluated. The Rule **Att** (for *att*ribute) formalizes those aspects.

Rule 4.3.2 (Att) For any $a \in A$,

$$\frac{}{\langle a(\tilde{v}),\, \mathcal{D}\rangle \rightarrow \langle [\![a(\tilde{v})]\!]_{\mathcal{D}\cup I},\, \mathcal{D}\rangle}$$

if $a(\tilde{v}) \in \mathcal{T}_{F\cup A\cup I}$, $v_1 \in Id(\mathcal{D})$, and $[\![a(\tilde{v})]\!]_{\mathcal{D}\cup I}$ is in $\mathcal{P}_{F\cup Id(\mathcal{D})}$. $\Box$

The first condition guarantees that $a(\tilde{v})$ is an attribute application; note that $a \in A$ is not enough to guarantee that, since there might be a method named a as well (i.e., a might belong to M as well). The second condition assures that the attribute's associated object exists. The last condition checks if the attribute has an associated value in the database. As we use $\mathcal{D} \cup I$ (instead of $\mathcal{D}$) in $[\![a(\tilde{v})]\!]_{\mathcal{D}\cup I}$, we can evaluate the attribute expression even if it contains identifiers of objects not in the database. However, if this is the case, the evaluation is only performed if these identifiers are irrelevant to determine the value associated to that attribute.

By the convention introduced in Section 2.3, $[\![a(\tilde{v})]\!]_{\mathcal{D}\cup I}$ stands for $[\![\rho(a(\tilde{v}))]\!]_{\mathcal{D}\cup I}$, since $\mathcal{D}$ is a presentation with a signature of non-monotonicities. As ρ gives the least parse of an expression and the equations in $\mathcal{D}$ are parsed, only the most specific equations (the ones associated to a particular type of an operation) of $\mathcal{D}$ are used to define the value of $a(\tilde{v})$. This means that this attribute is dynamically bound to the specialized version of its associated operation; on the other hand, attributes used in attribute equations are statically bound. In fact, using the theory of OSA in this way, we can only obtain a partial form of dynamic binding for derived attributes, whereas we can get a full form of dynamic binding for stored attributes.

Qualified Notation for Attributes

In Section 4.1, we have discussed a technique to support qualified notation for redefined attributes. Now, we give its semantics, by showing what is the specification corresponding to a FOOPS module having redefined attributes. Basically, the specification should contain all declarations from the module plus the operations providing the qualified notation, and one equation for each redefined attribute. These equations stipulate that a given qualified attribute is equal to the related original one. For instance, a module with the following declarations

```
pr NAT .
subclass A < A´ .
at a : A´ -> Nat .
at a : A -> Nat [redef] .
```

is translated to a specification containing the declarations above plus the following:

```
at a.A´ : A´ -> Nat .
var X : A´ .
ax a.A´(X) = a(X) .
```

Because the equation above is parsed as

```
ax a.A´.A´Nat(X.A´) = a.A´Nat(X.A´) .
```

we have that `a.A´` is actually equal to the original version of `a`, as desired.

4.3.3 Methods Specified by DMAs

In order to define the semantics of method evaluation, we introduce some new notation. Given a specification P, we assume arbitrary

- Σ-variable family X such that $|X| = \{x_1, \ldots, x_k\}$, $u_i \leq LS(x_i)$, and $x_i \equiv x_j$ implies $v_i \equiv v_j$, for $i, j = 1 \,..\, k$;
- expressions formed by functions, attributes, variables, and identifiers of specified objects[2]: $g, h, c \in \mathcal{T}_{F \cup A \cup (M \cap I)(X)}$; and
- method symbol $m' \in \{m.LS(x_1), m\}$, for some $m \in M$ not in the form *symbol.class-name*.

As any specification contains the sort `Bool`, hereafter we write $(\forall X)\ l = r\ \texttt{if}\ c$ instead of

$$(\forall X)\ l = r\ \texttt{if}\ \{(c, \texttt{true})\},$$

for any term c (as above) of sort `Bool`. Also, $\tilde{x}$ abbreviates $x_1, \ldots, x_k$, and $\tilde{y}$ is a sequence of variables from X. Lastly, we write $(\tilde{x} \leftarrow \tilde{v})$ for $(x_1 \leftarrow v_1, \ldots, x_k \leftarrow v_k)$.

Now, we give a formal definition of DMA.

Definition 4.3.1 For a specification P, a **DMA** is a Σ-equation in the form

$$(\forall X)\ a(m(\tilde{x}), \tilde{y}) = g\ \texttt{if}\ c$$

where m is a method: $m(\tilde{x}) \in \mathcal{T}_{M(X)}$; the result of m belongs to its associated class: $LS(m(\tilde{x})) = LS(x_1)$; and a is an attribute: $a(x_1, \tilde{y}) \in \mathcal{T}_{A(X)}$. If $\tilde{y}$ is the empty sequence then $a(m(\tilde{x}), \tilde{y})$ stands for $a(m(\tilde{x}))$. $\Box$

[2] Non constant method symbols are neither allowed in conditions nor in the RHS of DMAs.

The evaluation of a method specified by DMAs is atomic and yields the identifier of the object that executes this method; this object must be in the state where the method is evaluated. Some of the attributes of this object are updated. The resulting database state is the overwriting of the previous state by equations specifying those updates. The LHS and RHS of these equations respectively correspond to the updated attribute and its new value. These equations are derived from the DMAs related to the method (as required by the frame assumption).

However, only the related DMAs with the following properties are considered for evaluation: the DMA's condition, when instantiated with the method arguments, is valid in the current state; the DMA's instantiated RHS must be defined in the current state; and the DMA is either associated to the class indicated by the qualified notation, or to a class greater or equal to u and smaller or equal to u', where u is the class associated to the version of the method being evaluated, and u' is the least class (greater or equal to u) redefining this method (if there is no such u', any related DMA may be used). Lastly, the method is only evaluated if there are some attributes to update and the updates are not contradictory; otherwise, it is suspended.

All the aspects discussed above are considered by the following rule.

Rule 4.3.3 (DMA) For any $m \in M$, let Γ be the set of all equations in the form

$$a(v_1, \tilde{y})(\tilde{x} \leftarrow \tilde{v}) = [\![g(\tilde{x} \leftarrow \tilde{v})]\!]_{\mathcal{D}\cup I}$$

such that E has a DMA in the form

$$(\forall X)\ a(m(\tilde{x}), \tilde{y}) = g \ \texttt{if}\ c;$$

`true` belongs to $[c(\tilde{x} \leftarrow \tilde{v})]_{\mathcal{D}\cup I}$; $[\![g(\tilde{x} \leftarrow \tilde{v})]\!]_{\mathcal{D}\cup I}$ is in $\mathcal{P}_{F\cup Id(\mathcal{D})}$; and $m' = m$ and $m \in R_{w',u'}$, for some $u_1, \ldots, u_n \leq w'$, imply that $LS(x_1)$ is smaller than or equal to the least class greater than or equal to u_1 such that m is redefined.

If $v_1 \in Id(\mathcal{D})$ and Γ is a non empty and non contradictory set, then

$$\frac{}{\langle m'(\tilde{v}),\ \mathcal{D}\rangle \to \langle v_1,\ \mathcal{D} \oplus \Gamma\rangle}$$

□

Remember that the operation $\oplus$ overwrites a database state (see Section 4.2.1).

DMAs can only specify a deterministic behaviour for methods. So, note that if there are contradictory (or even redundant) DMAs in E, their associated method cannot be executed because its specified behaviour is considered inconsistent; the evaluation is suspended. Obviously, this should be avoided. In the rule above, this is reflected by requiring Γ to be non contradictory. Similarly, suppose that two versions of a method are defined, but the specialized version is not considered a redefinition. In this case, following the last rule, equations from both definitions may be used to evaluate the specialized version. So, unless both versions specify the same updates, they are considered inconsistent; again, Γ will be contradictory and consequently the method won't be executed.

The equations defining the versions of a redefined method associated to a class and a corresponding subclass do not necessarily have to agree on the behaviour that they specify. Indeed, usually they do not. Thus, only the most specialized method equations can be used for execution.

This corresponds to the semantics of dynamic binding. That is why the restriction on $LS(x_1)$ is necessary in Rule **DMA**. Also, if a qualified notation is used, observe that the indicated class must be the same as the least class of x_1, by the restrictions on m'; so, only equations associated to the indicated class can be used for evaluation.

Rule **DMA** makes clear that the adoption of the variant syntactic rule (which is implied by regularity) for redefinitions may cause some anomalies which require attention and should be avoided. For instance, consider the following specification

```
pr NAT .
subclass A < A´ .
subclass B < B´ .
me m : A´ B´ -> A´ .
me m : A B -> A [redef] .
at a : A´ -> Nat .
var X´ : A´ .
var Y´ : B´ .
var X : A .
var Y : B .
ax a(m(X´,Y´)) = ··· .
ax a(m(X,Y)) = ··· .
```

where `m` is redefined in a valid way. Observe that the expression `m(a,b´)`, for `a:A` and `b´:B´`, cannot be evaluated using the DMA associated to the specific version of `m` because the class of `b´` is not a subclass of the class of `Y`; that is, `m(a,b´)` does not match with `m(X,Y)`. Also, the DMA related to the original version cannot be used for evaluation because it is not the most specialized one (i.e., the class of `X´` is not the same as the class of `a`). This situation should be avoided because although `m(a,b´)` is a valid expression, it cannot be evaluated; it will be suspended forever.

From the definition of $\oplus$, we conclude that this operation might not preserve the properties of database states if its arguments do not satisfy some conditions. If this is the case in the use of $\oplus$ in the last rule, no transition is possible (because the result of $\oplus$ is not a valid database state). This means that the method cannot be executed. So, this circumstance should also be avoided. In general, this might happen if the state has an equation specifying an attribute value and this equation is not removed when that attribute is updated. For instance, this happens if a DMA specifies an update for a derived attribute. In this case, the state resulting from the update is a presentation containing two equations determining two values for the same attribute (one corresponding to the original derived attribute equation and another associated to the update). If these values are not equal (with respect to the related functional theory), the resulting presentation is not a valid database state, since it relates two expressions which are not related by the associated functional theory.

A similar problem occurs when a stored attribute is redefined by a derived one. In fact, a method associated to the superclass might try to update the redefined, derived attribute. But we have already discussed that derived attributes should not be updated. For example, consider the specification

```
pr NAT .
```

```
subclass C < C´ .
me m : C´ -> C´ .
at a : C´ -> Nat .
at a : C -> Nat [redef] .
var X : C´ .
var Y : C .
ax a(m(X)) = 1 .
ax a(Y) = 0 .
```

where m is not redefined. Note that the original version of a is a stored attribute, whereas the specialized version of a is a derived attribute. Thus, the execution of m(o), for some o:C, adds the equation

```
a.CNat(o.C) = 1.Nat
```

to the database. But this conflicts with the equation

```
a.CNat(Y.C) = 0.Nat
```

which must be in any state (it belongs to AE). This would violate the restriction on states because we would then be able to prove that 0 equals 1 from the equations in the resulting state.

Here we do not give a complete semantics for updating of multi-argument stored attributes. In fact, this cannot be done in a simple and abstract way if presentations are used to model database states. So, we just give the semantics of DMAs specifying the update of only one attribute, as in

```
ax a(m(O,X),X) = e,
```

where evaluating m(o,x) only updates the attribute a(o,x). That is why in the last rule we require the variables in $\tilde{y}$ to be in $\tilde{x}$. This implies that after instantiation, the LHS, RHS, and condition do not have any variables. So, the resulting RHS and condition can be evaluated, and the instantiated LHS specifies the value of only one attribute.

It is difficult to give the semantics of DMAs such as

```
ax a(m(O),X) = e,
```

where the execution of m(o) should update all attributes a(o,x), for any x:X. In order to consider this kind of DMAs, states would have to keep a history of updates for each multi-argument stored attribute. This is needed because such updates might not completely invalidate the previous ones, since each update may determine values for an arbitrary range of attributes. The value associated to a specific attribute could then be computed by checking what is the last update that determines it. This approach could be represented in our model for states. However, it turns out to be very detailed.

4.3.4 Methods Specified by IMAs

In this section we give the semantics of methods specified by IMAs. First, we give a formal definition for IMA.

Definition 4.3.2 For a specification P, an **IMA** is a Σ-equation in the form

$$(\forall X)\ m(\tilde{x}) = \mathit{expr}\ \texttt{if}\ c$$

where m is a method (i.e., $m(\tilde{x}) \in \mathcal{T}_{M(X)}$) and expr is a method expression with variables from X: $\mathit{expr} \in \mathcal{T}_{\Sigma(X)}$. □

The execution of a method specified by an IMA corresponds to the evaluation of this IMA's RHS (instantiated with the method arguments), even if the method's associated object is not in the database used for evaluation (methods specified in this way are seen as abbreviations for complex expressions). Of course, the evaluation can only happen if the IMA's condition (instantiated with arguments) is satisfied in the state where the method is going to be evaluated; otherwise the evaluation is suspended. Also, similarly to DMAs, only specialized IMAs should be used for evaluation. The following rule considers those aspects.

Rule 4.3.4 (IMA) For any $m \in M$, if the IMA

$$(\forall X)\ m(\tilde{x}) = \mathit{expr}\ \texttt{if}\ c$$

is in E; $\texttt{true} \in [c(\tilde{x} \leftarrow \tilde{v})]_{\mathcal{D}\cup I}$; and $m' = m$ and $m \in R_{w',u'}$, for some $u_1, \ldots, u_n \leq w'$, imply that $LS(x_1)$ is smaller than or equal to the least class greater than or equal to u_1 such that m is redefined, then

$$\frac{}{\langle m'(\tilde{v}),\, \mathcal{D}\rangle \rightarrow \langle \mathit{expr}(\tilde{x} \leftarrow \tilde{v}),\, \mathcal{D}\rangle}$$

□

Because of dynamic binding, only the most specialized IMAs can be used for evaluation. Contrasting to DMAs (see comments following Rule 4.3.3), if two or more IMAs that do not agree on the specified behaviour are used to define the same method, this method has a nondeterministic behaviour. Similarly, if two versions of a method are defined, but the specialized version is not considered a redefinition, the IMAs related to both versions may be used to evaluate the specialized version, which will probably be nondeterministic.

Lastly, observe that the same anomalies associated to methods defined by DMAs, due to the adoption of the variant syntactic rule for redefinitions, might also happen for methods specified by IMAs (see Section 4.3.3 for details).

4.3.5 Arguments

From the transition rules given so far, it can be observed that the least sort of an expression is always in the same connected component of U (with respect to $\leq$) as the least sort of the result yielded by the evaluation of this expression. However, the least sort of this result might be greater, smaller or even not related (by $\leq$) to the least sort of that expression. This is due to the flexibility of the FOOPS type system. For instance, this happens because axioms are not necessarily sort decreasing, like in

```
sorts A B C .
subsort A < C .
```

```
subsort B < C .
class D .
at a : D -> A .
var X : D .
ax a(X) = e .
```

where `e` is a constant of sort `B`. So, the evaluation of `a(o)`, for `o:D`, gives `e`. But this resulting expression has least sort `B`, whereas `a(o)` has least sort `A`, which is not related to `B`.

Another example where this may happen is

```
subclass C < C´ .
subsorts A < A´ .
at a : C´ -> A´ .
at a : C -> A .
var X : C´ .
ax a(X) = e .
```

where `e` is a constant of sort `A´` and `a` is not redefined (there are no special equations for the specialized version of `a`). Hence, `a(o)`, for `o:C`, has least sort `A`, but evaluates to `e` which has least sort `A´`, greater than `A`.

This fact has to be considered when evaluating arguments because an operation may not be defined for the type of the result of the evaluation of one of its arguments. In this case, a retract should be introduced to give the right type to the result.

In order to produce results of interest, retracts should be eventually eliminated (evaluated). Otherwise, operations might block or return exceptional retracted values. This can be avoided if the retracts in axioms can be eventually eliminated (a non sort decreasing axiom may be considered a sort decreasing axiom, by inserting an adequate retract to its RHS). A functional retract can be eliminated if its argument is evaluated to an element that is equal (with respect to the associated functional theory) to an element of the desired sort. Similarly, an object level retract is eliminated if its argument evaluates to an object identifier of the desired class.

Now we introduce the Rule **Arg** for *arg*ument evaluation. Hereafter, we consider arbitrary method expressions without variables: $e, f, e_i, e'_i \in \mathcal{T}_{\mathsf{P}}$, for $i = 1 \mathinner{..} k$.

Rule 4.3.5 (Arg) For any $i \in \{1, \ldots, k\}$ and $op \in (F \cup A \cup M)_{w,s}$ such that $LS(e_1) \ldots LS(e_k) \leq w$,

$$\frac{\langle e_i,\, \mathcal{D}\rangle \rightarrow \langle e'_i,\, \mathcal{D}'\rangle}{\langle op(e_1, \ldots, e_i, \ldots, e_k),\, \mathcal{D}\rangle \rightarrow \langle op(e'_1, \ldots, \tau(e'_i), \ldots, e'_k),\, \mathcal{D}'\rangle}$$

where $e'_j = e_j$ if $j \neq i$, for $j = 1 \mathinner{..} k$; and $\tau(e'_i) = e'_i$ if $LS(e'_i) \leq LS(e_i)$, otherwise $\tau(e'_i) =$ `r:u´>u(`e'_i`)`, where `u´` $= LS(e'_i)$ and `u` $= LS(e_i)$. □

Notice that there is no fixed order to evaluate arguments; the order is nondeterministically chosen. In fact, the result of the evaluation of the expression may be nondeterministic, if some arguments change the state. Furthermore, the evaluation of one argument might be interleaved with the evaluation of the others. However, from the semantic point of view, a step (transition) in the evaluation of one argument cannot happen at the same time as a step in the evaluation of another argument. In fact, in Section 4.4.2, we argue that trying to support the simultaneous execution of expressions turn out to be equivalent to the interleaving semantics which we adopt here.

4.4 Method Combiners

Now, we show how the semantics given in the previous section can be modified and extended to support method combiners. It is simpler to directly give the semantics of each FOOPS predefined method combiner independently, instead of trying to specify some of them in terms of others. Here we introduce transition rules giving the semantics of each combiner. Lastly, we give one rule which specifies how new combiners, defined in terms of the predefined ones, are evaluated.

Method combiners are conceptually different from methods and attributes. Indeed, they do not correspond to operations related to objects. Hence, they have a special semantics. Among other particularities, they offer some control over the order of evaluation of their arguments, and they yield results depending whether some particular arguments are fully evaluated. Similarly to methods, method combiners are neither allowed in the RHS of DMAs nor in conditions.

In order to give the semantics of method combiners, we assume that signatures contain one more component: $MC \subseteq \Sigma$, formed by method combiners names. Moreover, we consider that method combiners are not mixed up (in the sense of Definition 4.1.1) with functions, methods, or attributes. The following operations, which are in MC, represent the predefined method combiners:

- `_;_ : U T -> T`;
- `_||_, _[]_ : U U -> U`;
- `result_;_ : U T -> U`;
- `if_then_else_fi : T U U -> U`; and
- `[_] : U -> U`;

for any types `T, U` $\in U$.

For supporting the parallel composition and choice of method expressions having unrelated types, we consider that a universal type `Universal` is in U. This type includes any class or sort. That is, $u \leq$ `Universal`, for any $u \in U$. However, `Universal` is neither a class nor a sort. In this way, an expression `e || f` is well formed, even if the sorts of `e` and `f` are not related. This is possible because `_||_` can be parsed with the type "`Universal Universal -> Universal`".

Now, we proceed to give the semantics of method combiners.

4.4.1 Sequential Composition

The argument on the left of the sequential composition operator (`_;_`) has to be fully evaluated before the evaluation of the other argument starts. Rule **Seq** (for *seq*uential) is used for evaluation of the left argument and it indicates that transitions from this argument provokes transitions from a sequential composition:

Rule 4.4.1 (Seq)

$$\frac{\langle e,\, \mathcal{D}\rangle \to \langle e',\, \mathcal{D}'\rangle}{\langle e\ ;\ f,\, \mathcal{D}\rangle \to \langle e'\ ;\ f,\, \mathcal{D}'\rangle}$$

□

When the left argument is fully evaluated, there is a transition to start the evaluation of the argument on the right, as indicated by Rule **SeqE** (for *seq*uential composition *e*limination):

Rule 4.4.2 (SeqE)

$$\frac{}{\langle v\ ;\ e,\, \mathcal{D}\rangle \to \langle e,\, \mathcal{D}\rangle}$$

□

Here we adopt a "waiting semantics" for method expressions; that is, if a method cannot be executed in a database state (because no axiom specifying its behavior has a valid condition) then the object requiring the corresponding service (the client) has to "wait" until the service can be provided. However, notice that this does not necessarily mean that the client will be blocked, since it may be executing other tasks concurrently.

In the last rule, the adoption of the "waiting semantics" is reflected by ensuring that the right argument of the sequential composition operator is only executed when the left one is fully evaluated. This is also reflected in the rules for other method combiners in the following sections. In order to capture a "non waiting semantics", the evaluation of f should start as soon as e cannot be evaluated. Thus, if a service is not available, the client does not wait and proceeds to the execution of the next service.

The first alternative was chosen because it gives a useful synchronization mechanism between clients and servers. This would have to be simulated by some form of "busy waiting", if the non waiting semantics were used. Usually, this simulation complicates the code and it is quite inefficient. On the other hand, the "non waiting" behaviour can always be naturally and efficiently simulated in terms of the "waiting" behaviour. For example, suppose that the operation `put` inserts an element in a buffer only when the buffer is not full. Thus, adding the axiom

```
ax put(B,N) = B if full?(B) .
```

releases the client if the buffer is full; the client does not need to wait for a place in the buffer.

Furthermore, the "non waiting semantics" approach is not uniform, since in this case the left argument of `_;_` may be discarded (when its corresponding service cannot be provided), but the argument of an operation has to be eventually evaluated. For example,

```
put(b,5) ; put(b,4)
```

evaluates to `put(b,4)` in a state where the buffer `b` is full. On the other hand, no transition is possible from `put(put(b,5),4)` in the same state.

4.4.2 Parallel Composition

Here we give an interleaving semantics for parallel composition. So, transitions from the arguments of a parallel composition operator are interleaved and they cause transitions from the parallel composition, as shown by the Rule **ParL** (for *par*allel composition *l*eft argument evaluation)

Rule 4.4.3 (ParL)

$$\frac{\langle e,\, \mathcal{D}\rangle \rightarrow \langle e',\, \mathcal{D}'\rangle}{\langle e \mid\mid f,\, \mathcal{D}\rangle \rightarrow \langle e' \mid\mid f,\, \mathcal{D}'\rangle}$$

□

and the symmetric **ParR** (for *par*allel composition *r*ight argument evaluation):

Rule 4.4.4 (ParR)

$$\frac{\langle f,\, \mathcal{D}\rangle \rightarrow \langle f',\, \mathcal{D}'\rangle}{\langle e \mid\mid f,\, \mathcal{D}\rangle \rightarrow \langle e \mid\mid f',\, \mathcal{D}'\rangle}$$

□

Also, the arguments of a parallel composition operator can be eliminated when they are fully evaluated. This is specified by the Rules **ParLE** (for *par*allel composition *l*eft argument *e*limination)

Rule 4.4.5 (ParLE)

$$\frac{}{\langle v \mid\mid e,\, \mathcal{D}\rangle \rightarrow \langle e,\, \mathcal{D}\rangle}$$

□

and **ParRE** (for *par*allel composition *r*ight argument *e*limination):

Rule 4.4.6 (ParRE)

$$\frac{}{\langle e \mid\mid v,\, \mathcal{D}\rangle \rightarrow \langle e,\, \mathcal{D}\rangle}$$

□

Simultaneous Transitions

The interleaving semantics does not consider the behaviour caused by simultaneous transitions from the arguments of the parallel composition operator. In fact, considering simultaneous transitions seem more natural than interleaving, since objects might be part of a distributed system (where expressions might be simultaneously evaluated). Hence, let us consider the introduction of a parallel composition combiner which allows simultaneous transitions: `_|||_ : U U -> U`, for any `U` $\in U$.

First, remember that an attribute cannot be both read and written at the same time because of physical limitations. So, simultaneous transitions are only possible if the attributes accessed (i.e, read and/or written) in one transition are different from the attributes written in the other. Also, if an object is removed or created in one transition, it cannot be accessed, removed, or created by a simultaneous transition.

Some of those constraints cannot be elegantly expressed here because they rely on information that is abstracted by our framework. (For example, there's no simple procedure for determining what attributes are read during a given transition, since the evaluation of attributes is specified in terms of the representative of an equivalence class modulo equations.) Hence, instead of using the constraints above for defining a rule considering simultaneous evaluation of arguments, we use a weaker condition: the updates made by one transition do not interfere with the updates made by a simultaneous transition. Formally, if

$$\langle e,\, \mathcal{D}\rangle \;\rightarrow\; \langle e',\, \mathcal{D}_1\rangle \;(1) \text{ and } \langle f,\, \mathcal{D}\rangle \;\rightarrow\; \langle f',\, \mathcal{D}_2\rangle \;(2)$$

then

$$\langle e,\, \mathcal{D}_2\rangle \rightarrow \langle e',\, \mathcal{D}'\rangle \text{ and } \langle f,\, \mathcal{D}_1\rangle \rightarrow \langle f',\, \mathcal{D}'\rangle.$$

It is easy to verify that this is implied by the constraints mentioned at the beginning of this section. First, observe that $\mathcal{D}_1$ and $\mathcal{D}_2$ can be respectively represented in the forms

$$\mathcal{D} \ominus RO \cup NO \oplus \Gamma \text{ and } \mathcal{D} \ominus RO' \cup NO' \oplus \Gamma',$$

for some RO, NO, Γ, RO', NO' and Γ', indicating the changes made by e and f, where $RO \cap NO = \emptyset$ and $RO' \cap NO' = \emptyset$. So, if the transitions 1 and 2 are possible then $\langle f,\, \mathcal{D}_1\rangle$ leads to

$$\langle f',\, \mathcal{D}_1 \ominus RO' \cup NO' \oplus \Gamma'\rangle,$$

since the stronger condition assures that the first step in the evaluation of f does not access the changes made by the first step in the evaluation of e. A similar reasoning can be used to show that $\langle e,\, \mathcal{D}_2\rangle$ leads to

$$\langle e',\, \mathcal{D}_2 \ominus RO \cup NO \oplus \Gamma\rangle.$$

Lastly, it remains to check that the stronger condition implies that $\mathcal{D}_1 \ominus RO' \cup NO' \oplus \Gamma'$ is the same as $\mathcal{D}_2 \ominus RO \cup NO \oplus \Gamma$. This can be easily done; we omit the details.

Now we give the semantics of $\_ \,|||\, \_$ using the weaker condition. Basically, $\_ \,|||\, \_$ is defined by the rules for interleaving, replacing $\_ \,||\, \_$ by $\_ \,|||\, \_$, plus Rule **TCPar** (for *t*ruly *c*oncurrent *par*allel composition):

Rule 4.4.7 (TCPar)

$$\frac{\begin{array}{c}\langle e,\, \mathcal{D}\rangle \rightarrow \langle e',\, \mathcal{D}_1\rangle, \langle f,\, \mathcal{D}_1\rangle \rightarrow \langle f',\, \mathcal{D}'\rangle,\\ \langle f,\, \mathcal{D}\rangle \rightarrow \langle f',\, \mathcal{D}_2\rangle, \langle e,\, \mathcal{D}_2\rangle \rightarrow \langle e',\, \mathcal{D}'\rangle\end{array}}{\langle e \,|||\, f,\, \mathcal{D}\rangle \rightarrow \langle e' \,|||\, f',\, \mathcal{D}'\rangle}$$

□

In fact, the weaker condition used above allows more transitions than expected. However, this turns out to be enough for our purposes here: we are interested in proving the equivalence (with respect to some mild notion of observation equivalence) of two operators for parallel composition; basically, we want to show that the extra rule for simultaneous evaluation of arguments is redundant. So, if we prove that this is the case considering the rule above, it follows that this is also the case if we consider a rule with a stronger premise. This equivalence is what should be expected since it is desirable to specify systems of distributed objects without worrying whether computations are being carried out simultaneously.

Here we suppose that the notion of equivalence that we are interested does not distinguish a configuration C having the transitions

$$C \longrightarrow C_1 \longrightarrow C_2$$

from a configuration C' having the transitions

$$\begin{tikzcd} C' \arrow[r] \arrow[rr, bend left] & C'_1 \arrow[r] & C'_2, \end{tikzcd}$$

where C'_1 and C'_2 are respectively (observation) equivalent to C_1 and C_2; and any other transition from C' is matched, in a similar way, by some transitions from C, and vice versa.

This is a mild assumption on equivalences over configurations. Roughly, it says that a configuration that can lead to a resulting configuration in either one or two transitions is equivalent to a configuration which can reach an equivalent resulting configuration in two equivalent transitions. This should be valid for most reasonable and interesting observation equivalences because the configurations reached from C and C' are equivalent, and any sequence of observations that can be made on the intermediate states reached by one of the configurations corresponds to a possible sequence of observations from the other.

We let $\approx$ denote the equivalence on configurations that we are interested. Thus the following theorem establishes the equivalence of the two operators for parallel composition.

Theorem 4.4.1 For any database state $\mathcal{D}$ and method expressions e and f,

$$\langle e \mid\mid f,\, \mathcal{D}\rangle \approx \langle e \mid\mid\mid f,\, \mathcal{D}\rangle. \qquad (\omega)$$

Proof: Here we informally justify the theorem. A formal proof depends on the formal definition of $\approx$; so it is only given in Section 5.2.4.

We split the proof in three cases. First, we consider that both e and f are fully evaluated. In this case, the possible transitions from $\langle e \mid\mid f,\, \mathcal{D}\rangle$ and $\langle e \mid\mid\mid f,\, \mathcal{D}\rangle$ are justified by Rules 4.4.5, 4.4.6, and the corresponding ones to $\_\mid\mid\mid\_$:

$$\begin{tikzcd} & \langle e \mid\mid f,\, \mathcal{D}\rangle \arrow[dl, "1"'] \arrow[dr, "2"] & & & \langle e \mid\mid\mid f,\, \mathcal{D}\rangle \arrow[dl, "3"'] \arrow[dr, "4"] & \\ \langle e,\, \mathcal{D}\rangle & & \langle f,\, \mathcal{D}\rangle & \langle e,\, \mathcal{D}\rangle & & \langle f,\, \mathcal{D}\rangle \end{tikzcd}$$

From these diagrams, we can conclude that $\langle e \mid\mid f,\, \mathcal{D}\rangle$ is equivalent to $\langle e \mid\mid\mid f,\, \mathcal{D}\rangle$, since the transitions from the first (1 and 2) are matched by the transitions from the second (3 and 4), in the sense that they lead to equivalent configurations (we assume that $\approx$ is reflexive).

Now we assume that f is fully evaluated but e is not. Thus, we have the following possible transitions:

$$\begin{tikzcd}
& \langle e \mid\mid f,\, \mathcal{D}\rangle \arrow[dl, "1"'] \arrow[dr, "2"] & & & \langle e \mid\mid\mid f,\, \mathcal{D}\rangle \arrow[dl, "3"'] \arrow[dr, "4"] & \\
\langle e,\, \mathcal{D}\rangle & & \langle e' \mid\mid f,\, \mathcal{D}_1\rangle & \langle e,\, \mathcal{D}\rangle & & \langle e' \mid\mid\mid f,\, \mathcal{D}_1\rangle
\end{tikzcd}$$

whenever $\langle e,\, \mathcal{D}\rangle \to \langle e',\, \mathcal{D}_1\rangle$. This is justified by Rules 4.4.3, 4.4.6, and the corresponding ones to $\mid\mid\mid$. In the diagrams above, transitions 1 and 3 clearly match. In the meantime, let us assume that

$$\langle e' \mid\mid f,\, \mathcal{D}_1\rangle \approx \langle e' \mid\mid\mid f,\, \mathcal{D}_1\rangle. \tag{ξ}$$

So we can infer that transitions 2 and 4 match, proving ω for this case. We can use a similar reasoning if e is fully evaluated but f is not.

Lastly, if neither e nor f is fully evaluated, the possible transitions from both configurations (as defined by Rules 4.4.3, 4.4.4, the corresponding ones to $\_\mid\mid\mid\_$, and Rule 4.4.7) are the following:

$$\begin{tikzcd}
& \langle e \mid\mid f,\, \mathcal{D}\rangle \arrow[dl, "1"'] \arrow[dr, "2"] & \\
\langle e' \mid\mid f,\, \mathcal{D}_1\rangle \arrow[dr, dotted, "3"'] & & \langle e \mid\mid f',\, \mathcal{D}_2\rangle \arrow[dl, dotted, "4"] \\
& \langle e' \mid\mid f',\, \mathcal{D}'\rangle &
\end{tikzcd}$$

and

$$\begin{tikzcd}
& \langle e \mid\mid\mid f,\, \mathcal{D}\rangle \arrow[dl, "5"'] \arrow[dr, "6"] \arrow[dd, dotted, "7"] & \\
\langle e' \mid\mid\mid f,\, \mathcal{D}_1\rangle \arrow[dr, dotted, "8"'] & & \langle e \mid\mid\mid f',\, \mathcal{D}_2\rangle \arrow[dl, dotted, "9"] \\
& \langle e' \mid\mid\mid f',\, \mathcal{D}'\rangle &
\end{tikzcd}$$

whenever $\langle e,\, \mathcal{D}\rangle \to \langle e',\, \mathcal{D}_1\rangle$ and $\langle f,\, \mathcal{D}\rangle \to \langle f',\, \mathcal{D}_2\rangle$. By Rule 4.4.7, transition 7 is only possible if $\langle f,\, \mathcal{D}_1\rangle \to \langle f',\, \mathcal{D}'\rangle$ and $\langle e,\, \mathcal{D}_2\rangle \to \langle e',\, \mathcal{D}'\rangle$; this justifies transitions 3, 4, 8, and 9. Transitions 1 and 2 respectively match 5 and 6 if we assume that

$$\langle e' \mid\mid f,\, \mathcal{D}_1\rangle \approx \langle e' \mid\mid\mid f,\, \mathcal{D}_1\rangle \tag{η}$$

and

$$\langle e \mid\mid f',\, \mathcal{D}_2\rangle \approx \langle e \mid\mid\mid f',\, \mathcal{D}_2\rangle. \tag{κ}$$

Also, by the assumption we made about $\approx$, transition 7 (together with transition 5 followed by 8) is matched by transition 1 followed by 3. This proves ω for this case.

Now, we have only to check propositions ξ, η, and κ. They can be informally justified by a similar reasoning as the one used for ω. $\square$

From this theorem and assuming that $\approx$ is preserved by `_||_`, we can guarantee that an expression not containing `_|||_` is equivalent to an expression obtained from the first by substituting `_|||_` for `_||_`. Similarly, `_|||_` can be replaced by `_||_`. Hence, we conclude that there is no need to introduce the operator `_|||_`; it is semantically equivalent to `_||_` and has a more complicated semantic definition. The extra rule for `_|||_` is redundant.

Furthermore, the theorem above indicates that the implementation of interleaving may simultaneously evaluate two expressions, given some mild conditions. Fortunately, as we have shown, this does not generate any behaviour that cannot be observed from the interleaving of the evaluation of the two expressions.

As mentioned before, simultaneous transitions are only possible if the attributes accessed in one transition are different from the attributes written in the other. This is a realistic restriction if attributes are directly implemented in terms of memory cells and transitions correspond to the execution of atomic transactions, which imply that attributes might be blocked. However, this restriction could be relaxed if the implementation provides a copy of each attribute; that is, one copy for reading (access) and another for writing (access). The reading copy could be used by many clients at the same time, whereas the write copy could be blocked by only one client at a given time. In this case, a complex mechanism is necessary to keep the consistency between the two copies. On the other hand, the reading copy may be read at the same time that the writing copy is being updated. Also, considering that an atomic transaction would only block the attributes that might be updated, a transaction could write to an attribute being read by a simultaneous transaction.

Clearly, this approach does not seem to be practical. It is likely that the efficiency gain obtained with the simultaneous execution is not worth because of the burden related to the management of copies and extra memory space necessary to keep a copy of each attribute. Despite this, disregarding implementation issues, we superficially explore the consequences of this approach, from the semantic point of view.

First, let us assume that the operator `_//_ : U U -> U`, for any `U` $\in U$, is defined by rules like the ones related to `_||_` plus one rule that allows simultaneous transitions whenever the attributes written in one transition are different from the attributes written in the other. Also, an object removed or created in one transition cannot be accessed, removed, or created in a simultaneous transition. (So, the only difference between `_|||_` and `_//_` is that the second allows attributes read in one transition to be written in a simultaneous transition.) As in the definition of `_|||_`, some of the conditions necessary to formalize a rule considering simultaneous transition cannot be elegantly expressed in our framework. However, we can still argue that `_//_` is not equivalent to interleaving. This means that the extra behaviour associated to `_//_` cannot be expressed by the interleaving of two transitions.

First, consider the following specification defining a class of memory cells for storing natural numbers

```
pr NAT .
```

```
class Cell .
at v : Cell -> Nat .
me _:=_ : Cell Nat -> Cell .
me _:=_ : Cell Cell -> Cell .
vars C C´ : Cell .
var N : Nat .
ax v(C := N) = N .
ax v(C := C´) = v(C´) .
```

where each object of `Cell` has an attribute `v` (for *v*alue) which stores a natural number, and two methods for changing the contents of a cell.

$$\begin{tikzcd}[column sep=small]
 & & \langle \texttt{X := Y // Y := X}, \mathcal{D}\rangle \arrow[dl, "1"'] \arrow[dr, "2"] \arrow[ddd, "5"] & & \\
 & \langle \texttt{X // Y := X}, \mathcal{D}_1\rangle \arrow[dl, "3"'] & & \langle \texttt{X := Y // Y }, \mathcal{D}_2\rangle \arrow[dr, "4"] & \\
\langle \texttt{X // Y }, \mathcal{D}_1 \oplus \texttt{v(Y) = 1}\rangle & & & & \langle \texttt{X // Y }, \mathcal{D}_2 \oplus \texttt{v(X) = 0}\rangle \\
 & & \langle \texttt{X // Y }, \mathcal{D} \oplus \texttt{v(X) = 1} \oplus \texttt{v(Y) = 0}\rangle & &
\end{tikzcd}$$

Figure 4.1: Transitions generated by `X := Y // Y := X`.

Now suppose that `X` and `Y` are identifiers of cells; the evaluation of the expression

```
X := Y // Y := X
```

in a database state $\mathcal{D}$ where `v(X) = 0` and `v(Y) = 1` is illustrated by the diagram in Figure 4.1, where $\mathcal{D}_1$ and $\mathcal{D}_2$ are respectively the states

$$\mathcal{D} \oplus \texttt{v(X) = 1} \text{ and } \mathcal{D} \oplus \texttt{v(Y) = 0}.$$

Note that transition 5 would not be possible if `_|||_` (or `_||_`) were used instead of `_//_`. As can be seen, it is not equivalent to 1 followed by 3, or 2 followed by 4. This shows that the two operators are not equivalent, since the resulting states are clearly not (observation) equivalent.

The operators are not equivalent, but there are weaker relations among the different operators for parallelism. For example, the expression analysed above is equivalent to

`X := v(Y) || Y := v(X)` and `X := v(Y) ||| Y := v(X)`,

since `v(X)` and `v(Y)` have to be evaluated before the assignment can be executed (this is not atomically done). Also, we can say that `e // f` is refined by `e ||| f` (or `e || f`), for any expressions `e` and `f`, because any behaviour observed from the evaluation of the first can also be observed from the evaluation of the second (but not the other way around).

Lastly, it is important to observe that the new operator is sensitive to the group of attributes updated by its arguments. For example, consider the following observation equivalent expressions:

`[ X := Y ; Y := Y ]` and `X := Y`.

Also, we have that `Y := X` is equivalent to `Y := X`. Now note that the expression

(1) `Y := X // X := Y`

is not equivalent to

(2) `Y := X // [ X := Y ; Y := Y ]`

since the (sub)expressions of 1 can be executed at the same time (they write to different attributes), whereas this is not possible for 2 (both arguments update `Y`). In fact, expression 1 might behave in a way that cannot be simulated by 2.

From the previous example, we conclude that it is not possible to find a reasonable notion of equivalence that is preserved by this operator. This essentially means that `_//_` cannot be used for compositional software development. So, it is not very useful in practice. That is another reason for choosing an interleaving semantics for FOOPS parallel composition.

4.4.3 Nondeterministic Choice

The choice operator (`_[]_`) nondeterministically chooses one of its arguments for evaluation. Using process algebra's terminology (see [51, 69, 48]), here we opt for an *external* choice operator rather than an *internal* choice operator. In fact, the latter can be simply defined in terms of the former (see Section 3.2.2).

Essentially, the nondeterminism of an external choice is partly resolved by the environment where the choice is evaluated. In fact, an argument may only be chosen if it can be evaluated in the current environment, or if it is already fully evaluated. If both arguments may be chosen, the operator autonomously (internally) resolves the nondeterminism.

Transitions from the choice between two expressions correspond to transitions from one of the expressions. This is specified by the Rule **ChoiceL** (for *choice l*eft argument evaluation)

Rule 4.4.8 (ChoiceL)

$$\frac{\langle e, \mathcal{D}\rangle \rightarrow \langle e', \mathcal{D}'\rangle}{\langle e \;[]\; f, \mathcal{D}\rangle \rightarrow \langle e', \mathcal{D}'\rangle}$$

□

and the similar **ChoiceR** (for *choice r*ight argument evaluation)

Rule 4.4.9 (ChoiceR)

$$\frac{\langle f, \mathcal{D}\rangle \rightarrow \langle f', \mathcal{D}'\rangle}{\langle e \;[]\; f, \mathcal{D}\rangle \rightarrow \langle f', \mathcal{D}'\rangle}$$

□

Moreover, a fully evaluated argument may be chosen by the choice without changing the database. This is expressed by the Rule **ChoiceLE** (for *choice left* argument *elimination*)

Rule 4.4.10 (ChoiceLE)

$$\frac{}{\langle v \,[]\, e,\, \mathcal{D}\rangle \rightarrow \langle v,\, \mathcal{D}\rangle}$$

□

and the symmetric **ChoiceRE** (for *choice r*igth argument *e*limination)

Rule 4.4.11 (ChoiceRE)

$$\frac{}{\langle e \,[]\, v,\, \mathcal{D}\rangle \rightarrow \langle v,\, \mathcal{D}\rangle}$$

□

It might be difficult to efficiently implement the kind of choice discussed here because it tries to "guess" whether an argument can be evaluated or not. This contrasts with an internal choice operator, which can be simply and efficiently implemented. Also, the internal choice is likely to be more useful in FOOPS specifications. However, we have adopted an external choice for two main reasons. First, it is more fundamental and can be used to define internal choice in a very simple way. Second, choice is usually used in FOOPS as an abstraction tool for writing specifications (when it is desirable to abstract from the reasons which determine one behaviour or another), rather than as an operator to construct implementations (as usually necessary in process algebras).

4.4.4 Result

As probably expected, the rules giving the semantics of the result method combiner should be similar to the rules for sequential composition. This is confirmed by the Rule **ResultL** for evaluation of the left argument of the result combiner:

Rule 4.4.12 (ResultL)

$$\frac{\langle e,\, \mathcal{D}\rangle \rightarrow \langle e',\, \mathcal{D}'\rangle}{\langle \texttt{result}\ e\ ;\ f,\, \mathcal{D}\rangle \rightarrow \langle \texttt{result}\ e'\ ;\ f,\, \mathcal{D}'\rangle}$$

□

When the left argument is fully evaluated, the evaluation of the right one may start. But contrasting with sequential composition, the left argument is not eliminated. Those aspects are described by the Rule **ResultR** (for *result r*ight argument evaluation):

Rule 4.4.13 (ResultR)

$$\frac{\langle e,\, \mathcal{D}\rangle \rightarrow \langle e',\, \mathcal{D}'\rangle}{\langle \texttt{result}\ v\ ;\ e,\, \mathcal{D}\rangle \rightarrow \langle \texttt{result}\ v\ ;\ e',\, \mathcal{D}'\rangle}$$

□

Finally, when both arguments are fully evaluated, the left one is given as result, as indicated by Rule **ResultE** (for *result* *e*limination):

Rule 4.4.14 (ResultE)

$$\frac{}{\langle \texttt{result } v \texttt{ ; } v',\, \mathcal{D}\rangle \rightarrow \langle v,\, \mathcal{D}\rangle}$$

□

4.4.5 Conditional

Besides conditional axioms, FOOPS has a method combiner that may be used for specifying conditional behaviour. In fact, this combiner provides a more general mechanism than conditional axioms because its condition may be an arbitrary method expression, whereas the latter cannot have conditions containing method symbols or method combiners.

First, the conditional method combiner (`if_then_else_fi`) fully evaluates its condition. After that, based on the result, one of the alternatives is choosen. The Rule **IfCond** specifies the evaluation of the condition:

Rule 4.4.15 (IfCond) For any $c, c' \in \mathcal{T}_\Sigma$,

$$\frac{\langle c,\, \mathcal{D}\rangle \rightarrow \langle c',\, \mathcal{D}'\rangle}{\langle \texttt{if } c \texttt{ then } e \texttt{ else } f \texttt{ fi},\, \mathcal{D}\rangle \rightarrow \langle \texttt{if } c' \texttt{ then } e \texttt{ else } f \texttt{ fi},\, \mathcal{D}'\rangle}$$

□

When the condition is fully evaluated, the first alternative is given as result, if the condition is true, as indicated by Rule **IfCondT**:

Rule 4.4.16 (IfCondT) If $v =_{FE}$ `true` then

$$\frac{}{\langle \texttt{if } v \texttt{ then } e \texttt{ else } f \texttt{ fi},\, \mathcal{D}\rangle \rightarrow \langle e,\, \mathcal{D}\rangle}$$

□

If the condition is false, the conditional yields the second alternative:

Rule 4.4.17 (IfCondF) If $v =_{FE}$ `false` then

$$\frac{}{\langle \texttt{if } v \texttt{ then } e \texttt{ else } f \texttt{ fi},\, \mathcal{D}\rangle \rightarrow \langle f,\, \mathcal{D}\rangle}$$

□

It might seem that this method combiner could be equivalently defined as a function, in terms of equations at FOOPS functional level. However, remember that functions can only be evaluated if their arguments are functional terms or object identifiers. This implies that the alternatives of a conditional would have to be evaluated before one of them is choosen. In general, that is not the desired behaviour because the evaluation of the alternatives might change the state.

4.4.6 Atomic Evaluation

Intuitively, the atomic evaluation of an expression corresponds to its evaluation in only one step (transition). In fact, this means that the attributes accessed by an expression being atomically evaluated cannot be modified by other expressions being concurrently evaluated. Atomic evaluation might is necessary when an expression has to be evaluated without interference from others; that is why FOOPS provides the atomic evaluation operator (`[_]`). Following those intuitions, we introduce the next rule:

Rule 4.4.18 (Atomic)

$$\frac{\langle e, \mathcal{D}\rangle \rightarrow^* \langle v, \mathcal{D}'\rangle}{\langle \mathtt{[}e\mathtt{]}, \mathcal{D}\rangle \rightarrow \langle v, \mathcal{D}'\rangle}$$

where $\rightarrow^*$ denotes the transitive, reflexive closure of $\rightarrow$[3]. □

Observe that the atomic evaluation only succeeds if the related expression can be fully evaluated. This corresponds to the semantics of atomic transactions in database systems, where the fact that the expression cannot be fully evaluated is considered a failure. Usually, after a failure, transactions recover the state previous to the beginning of the transaction; that is why there is no transition if the expression cannot be fully evaluated.

This approach is really useful for database systems, but it might be quite inefficient from the implementation point of view. So, if efficiency is essential, the atomic evaluation operator should only be used for expressions that can be fully evaluated in the contexts where they are used.

If the evaluation of the expression to be atomically evaluated does not terminate, the atomic expression does not terminate as well. In fact, it behaves as a divergent process (infinite loop) that does not modify the state, since the updates made by an atomic expression are only visible after its evaluation (intermediate states reached during the evaluation of an atomic expression cannot be observed). The following rule reflects those comments:

Rule 4.4.19 (Diverge) If $\langle e, \mathcal{D}\rangle$ is nonterminating,

$$\frac{}{\langle \mathtt{[}e\mathtt{]}, \mathcal{D}\rangle \rightarrow \langle \mathtt{[}e\mathtt{]}, \mathcal{D}\rangle}$$

□

where

Definition 4.4.1 A configuration is **terminating** if there is no infinite sequence of $\rightarrow$-transitions from it. □

From the rule above, we conclude that practical applications should not use the atomic evaluation operator for expressions whose evaluation, in the contexts where they are used, might not terminate.

[3]We omit the obvious rules necessary for defining $\rightarrow^*$.

4.4.7 Method Combiner Definition

During evaluation, method combiners defined by the user are simply replaced by the expression that they abbreviate, as described by Rule **MCDef** (for *m*ethod *c*ombiner *def*inition):

Rule 4.4.20 (MCDef) For any $mc \in MC$ and $expr \in \mathcal{T}_{\Sigma(X)}$, if mc is not a FOOPS predefined method combiner,

$$(\forall X)\ mc(\tilde{x}) = expr$$

is in E, and $mc(\tilde{x}) \in \mathcal{T}_{MC(X)}$, then

$$\frac{}{\langle mc(\tilde{e}),\, \mathcal{D}\rangle \rightarrow \langle expr(\tilde{x} \leftarrow \tilde{e}),\, \mathcal{D}\rangle}$$

where $\tilde{e}$ stands for $e_1, \ldots, e_k$, and $LS(e_i) \leq LS(x_i)$, for $i = 1 \,..\, k$. □

As can be seen from the rule above, the arguments of a method combiner do not have to be fully evaluated before the combiner is replaced by the expression that it abbreviates. In particular, they might contain method and method combiner symbols. This is necessary for most applications. In this way, conditional axioms cannot be used to define method combiners, since arguments would have to be evaluated before the evaluation of the condition.

Observe that introducing more than one axiom for the same method combiner gives a non-deterministic behaviour for it, since all axioms can be applied. In particular, adding rules for a predefined method combiner probably changes its semantics. That is why this is not allowed here.

4.5 Object Creation and Deletion

Dynamic object creation and deletion are respectively provided in FOOPS by the operators `new` and `remove`. In this section, we describe the semantics of both operators. First, we consider object creation. Later, we introduce object deletion. These operators are modelled as method combiners because they are associated to classes, not to objects; that is, their corresponding operations are not performed by objects. Hereafter we consider that signatures contain the following special combiners:

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

for each class `C` $\in \mathcal{C}$. (This extends the definition of signature, in the same way as done in Section 4.4.) Observe that the class name is used to form the operation name of the first creation operation; this is important to indicate the class of the objects to be created. A class name is not necessary to distinguish the different versions of the other creation operation because this information is already provided by the class of its argument (identifiers have a fixed and pre-defined class; see Section 4.2).

4.5.1 Object Creation

For a given class C, the operator `new.C()` creates an object of C having an identifier nondeterministically choosen from $I_{\mathtt{C}}$, but that is not already being used for another object. This identifier is given as the result of the evaluation of the operator, as specified by the next rule:

Rule 4.5.1 (Creation) For any class $\mathtt{C} \in C$ and any identifier $v \in I_{\mathtt{C}}$ not associated to an object in $\mathcal{D}$ (i.e., $v \notin \mathit{Id}(\mathcal{D})$),

$$\frac{}{\langle \mathtt{new.C()},\, \mathcal{D}\rangle \rightarrow \langle v,\, \mathcal{D} \cup V\rangle}$$

where V is a $U^* \times U$-sorted family containing v only; that is, $V_{\lambda,\mathtt{C}} = \{v\}$ and $V_{w,u} = \emptyset$ if $w \neq \lambda$ or $u \not\equiv \mathtt{C}$. □

Remember that the operation $\cup$ adds objects to a database state, according to the identifiers given as arguments, without setting their attributes (see Section 4.2.1).

Note that the creation operation directly introduces unbounded nondeterminism to FOOPS, if the family I of object identifiers is formed by infinite sets. In this case, infinitely many identifiers may be choosen for creating an object.

The operator `new` creates an object of the same class as the identifier given as argument, if this identifier is not already associated to another object (otherwise, the operation cannot be executed). This identifier is used for the created object and yielded by the operator. These aspects are formalized by Rule **CreationId** (for object *creation* with *id*entifier):

Rule 4.5.2 (CreationId) For any class $\mathtt{C} \in C$ and any identifier $v \in I_{\mathtt{C}}$ not associated to an object in $\mathcal{D}$ (i.e., $v \notin \mathit{Id}(\mathcal{D})$),

$$\frac{}{\langle \mathtt{new}(v),\, \mathcal{D}\rangle \rightarrow \langle v,\, \mathcal{D} \cup V\rangle}$$

where V is a $U^* \times U$-sorted family containing v only, as defined in Rule 4.5.1. □

From this rule, we conclude that in order to create an object of a given class, we have to know what identifiers are related to this class. Only these identifiers may be used for creating and accessing objects of this class. In practice, this is not a big restriction. For example, each family in I can be choosen (by the FOOPS system) to contain only names prefixed by the name of the class associated to the family. So, we can easily know what are the identifiers associated to a given class.

Observe that the operators for object creation do not assign initial values for attributes.

4.5.2 Object Deletion

The operator `remove` receives an object identifier as argument, removes its associated object from the database state, and yields this identifier. If this identifier does not correspond to an object in the state, the operation is suspended. The following rule formalize those aspects:

Rule 4.5.3 (Deletion) For any class $\mathtt{C} \in C$ and any identifier $v \in I_{\mathtt{C}}$ associated to an object in $\mathcal{D}$ (i.e., $v \in \mathit{Id}(\mathcal{D})$),

$$\frac{}{\langle \mathtt{remove}(v),\, \mathcal{D}\rangle \rightarrow \langle v,\, \mathcal{D} \ominus V\rangle}$$

where V is a $U^* \times U$-sorted family containing v only, as in Rule 4.5.1. □

Remember that the operation $\ominus$ removes from a given database state all references to a particular object (see Section 4.2.1). This is necessary to avoid dangling identifiers. This operation might seem extremely centralized and inefficient, contrasting with the notion of distributed objects; but, in fact, it can be implemented in an efficient and decentralized way.

As the argument of `remove` does not have to be an object identifier, we need one more rule (**DeletionArg**, for evaluation of the *arg*ument of the *deletion* operation) indicating how the argument should be evaluated.

Rule 4.5.4 (DeletionArg)

$$\frac{\langle e,\, \mathcal{D}\rangle \to \langle e',\, \mathcal{D}'\rangle}{\langle \texttt{remove}(e),\, \mathcal{D}\rangle \to \langle \texttt{remove}(e'),\, \mathcal{D}'\rangle}$$

□

4.6 Protected Objects

In this section we give the semantics of object protection. An informal description of this mechanism was given in Section 3.2.5. First, we assume that signatures have the combiner `addpr` for changing the protection status of objects, and new object creation operations:

- `addpr : C Universal -> C`,
- `new.C : Universal -> C`, and
- `new : C Universal -> C`,

for each class $\texttt{C} \in C$. Also, the following combiners should be in signatures, for representing the object permission given as argument to `new`:

- `_++_ : Universal Universal -> Universal`,
- `{} : -> Universal`, and
- `any : -> Universal`.

Using those combiners we can create terms in the following forms: `any`, `{}`, and `o ++` s, which respectively denote the set of all identifiers, the empty set, and the set containing the identifier `o` and the identifiers in the set denoted by s, where s is a term in one of the forms shown above. Only terms in those forms denote an object permission.

Second, we have to modify the structure of configurations to incorporate information about object permission. For a specification P, this information is represented by a finite mapping (referred as the *permission mapping*), belonging to $\mathit{Perm}(\mathsf{P}) = |I(\mathsf{P})| \mapsto \mathbb{P}|I(\mathsf{P})|$, which maps an object identifier o to the set of identifiers of the objects that can directly invoke methods associated to o. So, we let configurations be represented by the elements of

$$\mathit{PrConf}(\mathsf{P}) = \mathcal{T}_{\mathsf{P}} \times \mathit{Perm}(\mathsf{P}) \times D_{\mathsf{P}}.$$

We drop the subscripts when not confusing, and we write $\langle e,\ (\varrho, \mathcal{D})\rangle$ for $(e, \varrho, \mathcal{D}) \in \mathit{PrConf}(\mathsf{P})$. For convenience, the elements of $\mathit{Perm}(\mathsf{P}) \times D_{\mathsf{P}}$ may also be called database states (containing object permission information).

We define the semantics of FOOPS with object protection following the same approach used in Section 4.3. Basically, for a specification P, we introduce the relation $\rightarrow_{\mathsf{P}} \subseteq \mathit{PrConf}(\mathsf{P}) \times \mathit{PrConf}(\mathsf{P})$. It is defined by the rules given in the previous sections—except Rules 4.3.3, 4.3.4, 4.5.1, 4.5.2, and 4.5.3—and some extra rules to follow, assuming that

- database states have some extra information; that is, the variables $\mathcal{D}$ and $\mathcal{D}'$ range over $\mathit{Perm}(\mathsf{P}) \times D_{\mathsf{P}}$, rather than over D_{P};

- the operations on D_{P} are composed with the projections and constructors associated to $\mathit{Perm}(\mathsf{P}) \times D_{\mathsf{P}}$ (let ϱ and db respectively give permission mappings and database states). For example, for $\mathcal{D} \in \mathit{Perm}(\mathsf{P}) \times D_{\mathsf{P}}$, we now assume that $\mathcal{D} \oplus \Gamma$ stands for

$$(\varrho(\mathcal{D}), db(\mathcal{D}) \oplus \Gamma)$$

  (similarly for $\cup$ and $\ominus$), and $[\![e]\!]_{\mathcal{D}}$ is an abbreviation for $[\![e]\!]_{db(\mathcal{D})}$ (similarly for $[e]_{\mathcal{D}}$ and $\mathit{Id}(\mathcal{D})$).

Note that the rules introduced in previous sections are still meaningful after the modification on the structure of states, since we have also modified the operations used to access states to consider the new structure.

In order to indicate the object requesting a particular service, we introduce the method combiner `_!_ : T U -> T`, for any types `T, U` $\in U$. The second argument of this operator is the identifier of the object requesting the evaluation of the first argument. The second argument of `_!_` and a permission mapping are enough to give the semantics of object protection, by modifying some of the transition rules introduced in previous sections (Rules 4.3.3, 4.3.4, 4.5.1, 4.5.2, and 4.5.3) and including new ones.

First we specify how `_!_` is propagated to subexpressions. This is necessary because a method can only be evaluated if there is an indication of the object which invoked it. For this, we have to add some new rules. Hereafter, we consider an arbitrary object identifier $o \in I$.

4.6.1 Attributes, Functions, and Identifiers

The evaluation of attributes, functions, and object identifiers does not update objects; hence, it does not depend on the mechanism for object protection, as formalized by the following rule:

Rule 4.6.1 (PrE) For any $e \in \mathcal{T}_{F \cup A \cup I}$,

$$\frac{}{\langle e \ \texttt{!}\ o,\ \mathcal{D}\rangle \rightarrow \langle e,\ \mathcal{D}\rangle}$$

□

4.6.2 Arguments

The next rule shows how the mechanism for object protection is propagated to the arguments of methods, if they do not already have it. In the following rules assume that $k \geq 1$.

Rule 4.6.2 (PrM) For any $m \in M_{w,u}$ such that $LS(e_1), \ldots, LS(e_k) \leq w$, if e_i, for some $1 \leq i \leq k$, is not in the form e ! f then

$$\frac{}{\langle m(\tilde{e}) \texttt{ ! } o,\, \mathcal{D}\rangle \rightarrow \langle m(\tilde{e} \texttt{ ! } o) \texttt{ ! } o,\, \mathcal{D}\rangle}$$

where $\tilde{e}$! o stands for e_1 ! $o, \ldots, e_k$! o. □

The evaluation of method combiners is independent of the mechanism for object protection. Combiners just propagate this mechanism, if necessary, as specified by the next two rules:

Rule 4.6.3 (PrMCP) For any $mc \in MC_{w,u}$ such that $LS(e_1), \ldots, LS(e_k) \leq w$, if e_i, for some $1 \leq i \leq k$, is not in the form e ! f then

$$\frac{}{\langle mc(\tilde{e}) \texttt{ ! } o,\, \mathcal{D}\rangle \rightarrow \langle mc(\tilde{e} \texttt{ ! } o),\, \mathcal{D}\rangle}$$

□

However, if the arguments already have the mechanism, it is not propagated, as formalized by the following rule.

Rule 4.6.4 (PrMC) For any $mc \in MC_{w,u}$ such that $LS(e_1), \ldots, LS(e_k) \leq w$, if all e_i, for $i = 1 \,..\, k$, are in the form e ! f then

$$\frac{}{\langle mc(\tilde{e}) \texttt{ ! } o,\, \mathcal{D}\rangle \rightarrow \langle mc(\tilde{e}),\, \mathcal{D}\rangle}$$

□

Now we give a rule for argument evaluation of expressions formed by the method combiner _!_.

Rule 4.6.5 (PrArg) For any i in $1, .., k$ and op in $(F \cup A \cup M)_{w,s}$ such that $LS(e_1) \ldots LS(e_k) \leq w$,

$$\frac{\langle e_i,\, \mathcal{D}\rangle \rightarrow \langle e'_i,\, \mathcal{D}'\rangle}{\langle op(e_1, \ldots, e_i, \ldots, e_k) \texttt{ ! } o,\, \mathcal{D}\rangle \rightarrow \langle op(e'_1, \ldots, \tau(e'_i), \ldots, e'_k) \texttt{ ! } o,\, \mathcal{D}'\rangle}$$

where $e'_j = e_j$, if $j \neq i$, for $j = 1 \,..\, k$; and $\tau(e'_i) = e'_i$, if $LS(e'_i) \leq LS(e_i)$; otherwise, $\tau(e'_i) =$ `r:u´>u`(e'_i), where `u´` $= LS(e'_i)$ and `u` $= LS(e_i)$. □

Note that the original rule for argument evaluation is not replaced by the above rule, since it is still useful for evaluation of the arguments of attributes, for example.

4.6.3 Methods

Now we introduce new rules for method evaluation. Those are small modifications of the rules in Sections 4.3.3, 4.3.4, and 4.3.5. Here the behaviour of a method depends on the object that requested it. The method might be executed or suspended, depending whether the invocation is allowed by the object protection mechanism. For a given state $\mathcal{D}$, only the objects in $\varrho(\mathcal{D})(v)$ can directly invoke methods of v (when v is not in the domain of $\varrho(\mathcal{D})$, $\varrho(\mathcal{D})(v)$ denotes the empty set).

Methods Specified by DMAs

First we consider the evaluation of methods specified by DMAs.

Rule 4.6.6 (PrDMA) For any $m \in M$ let Γ be the set of all equations

$$a(v_1, \tilde{y})(\tilde{x} \leftarrow \tilde{v}) = [\![g(\tilde{x} \leftarrow \tilde{v})]\!]_{\mathcal{D} \cup I}$$

such that E has a DMA in the form

$$(\forall X)\ a(m(\tilde{x}), \tilde{y}) = g \texttt{ if } c;$$

`true` belongs to $[c(\tilde{x} \leftarrow \tilde{v})]_{\mathcal{D} \cup I}$; $[\![g(\tilde{x} \leftarrow \tilde{v})]\!]_{\mathcal{D} \cup I}$ is in $\mathcal{P}_{F \cup Id(\mathcal{D})}$; and $m' = m$ and $m \in R_{w',u'}$, for some $u_1, \ldots, u_n \leq w'$, imply that $LS(x_1)$ is smaller than or equal to the least class greater than or equal to u_1 such that m is redefined.

If $v_1 \in Id(\mathcal{D})$, $o \in \varrho(\mathcal{D})(v_1)$, and Γ is a non empty and non contradictory set then

$$\frac{}{\langle m'(\tilde{v}) \texttt{ ! } o,\, \mathcal{D}\rangle \rightarrow \langle v_1,\, \mathcal{D} \oplus \Gamma\rangle}$$

□

Methods Specified by IMAs

Similarly to what was done in the last section, we modify the rule for evaluation of methods specified by IMAs. Note that the execution of a method m specified by an IMA sometimes requires the invocation of methods associated to other objects; those methods are directly invoked by the object that performs m.

Rule 4.6.7 (PrIMA) For any $m \in M$, if the IMA

$$(\forall X)\ m(\tilde{x}) = expr \texttt{ if } c$$

is in E, $\texttt{true} \in [c(\tilde{x} \leftarrow \tilde{v})]_{\mathcal{D} \cup I}$; $m' = m$ and $m \in R_{w',u'}$, for some $u_1, \ldots, u_n \leq w'$, imply that $LS(x_1)$ is smaller than or equal to the least class greater than or equal to u_1 such that m is redefined; and $o \in \varrho(\mathcal{D})(v_1)$ then

$$\frac{}{\langle m'(\tilde{v}) \texttt{ ! } o,\, \mathcal{D}\rangle \rightarrow \langle expr(\tilde{x} \leftarrow \tilde{v}) \texttt{ ! } v_1,\, \mathcal{D}\rangle}$$

□

Contrasting to the original rule for evaluation of methods specified by IMAs, the rule above introduces a transition that is dependent on the state; that is, the transition is only possible if the object protection is not violated.

4.6.4 Object Creation and Deletion

Lastly, we modify the rules for object creation and deletion. Basically, the only difference from the creation operations introduced in this section and the operations introduced in Section 4.5 is that the former set the object permission.

First we have to define the function *set* for mapping the extra argument of `new` to the set of identifiers that it represents:

$$\begin{aligned} set(\texttt{\{\}}) &= \emptyset \\ set(\texttt{any}) &= |I| \\ set(e \texttt{ ++ } f) &= \{e\} \cup set(f), \text{ if } e \in I \\ &= set(f), \text{ if } e \notin I \end{aligned}$$

Note that the arguments of `_++_` that are not object identifiers are discarded by *set*.

The following rules describe the semantics of the operations for object creation, where

$$\mathcal{D} \oplus \{v \mapsto s\} \text{ denotes } (\varrho(\mathcal{D}) \oplus \{v \mapsto s\}, db(\mathcal{D})),$$

for any database state $\mathcal{D}$, $v \in I$, and $s \subseteq |I|$; the operation for mapping overwriting is represented by $\oplus$. Also, we assume that e is a term specifying an object permission, as described above. Any object is allowed to invoke the creation operations; that is, those operations do not depend on the mechanism for object protection, as indicated by the following rule:

Rule 4.6.8 (PrCreation) For any class $\texttt{C} \in C$, and any identifier $v \in I_{\texttt{C}}$ not associated to an object in $\mathcal{D}$ (i.e., $v \notin Id(\mathcal{D})$),

$$\frac{}{\langle \texttt{new.C(}e\texttt{) ! } o,\, \mathcal{D}\rangle \rightarrow \langle v,\, \mathcal{D} \cup V \oplus \{v \mapsto set(e)\}\rangle}$$

where V is a $U^* \times U$-sorted family containing v only; that is, $V_{\lambda,\texttt{C}} = \{v\}$, and $V_{w,u} = \emptyset$ if $w \neq \lambda$ or $u \not\equiv \texttt{C}$. □

and

Rule 4.6.9 (PrCreationId) For any class $\texttt{C} \in C$ and any identifier $v \in I_{\texttt{C}}$ not associated to an object in $\mathcal{D}$ (i.e., $v \notin Id(\mathcal{D})$),

$$\frac{}{\langle \texttt{new(}v\texttt{,}e\texttt{) ! } o,\, \mathcal{D}\rangle \rightarrow \langle v,\, \mathcal{D} \cup V \oplus \{v \mapsto set(e)\}\rangle}$$

where V is a family containing v only, as in Rule 4.6.8. □

From those rules, we can see that by default an object does not have permission to invoke its own methods. If this is desired, as usually the case, this has to be specified at object creation time, by making sure that v is in $set(e)$.

We modify the semantics of `remove` in such a way that a deletion operation can only be performed if the object which asked for it is able to invoke methods of the object to be removed. Here is the new rule:

Rule 4.6.10 (PrDeletion) For any class $\mathtt{C} \in C$ and any identifier $v \in I_{\mathtt{C}}$ associated to an object in $\mathcal{D}$ (i.e., $v \in Id(\mathcal{D})$), if $o \in \varrho(\mathcal{D})(v)$ then

$$\frac{}{\langle \mathtt{remove(}v\mathtt{)\ !\ }o,\ \mathcal{D}\rangle \to \langle v,\ \mathcal{D} \ominus V \oplus \{v \mapsto \emptyset\}\rangle}$$

where V is a family containing v only, as in Rule 4.6.8. □

Note that the object protection information associated to an object is reset after its deletion.

Lastly, we define the semantics of the `addpr(_,_)` operator, which was informally described in Section 3.2.5.

Rule 4.6.11 (PrAdd) For any object v in $\mathcal{D}$ (i.e., $v \in Id(\mathcal{D})$), if $o \in \varrho(\mathcal{D})(v)$ then

$$\frac{}{\langle \mathtt{addpr(}v\mathtt{,}e\mathtt{)\ !\ }o,\ \mathcal{D}\rangle \to \langle v,\ \mathcal{D} \oplus \{v \mapsto set(e) \cup \varrho(\mathcal{D})(v)\}\rangle}$$

□

4.7 Evaluation in the Background

In order to give the semantics of `_&_` we use another structure for representing configurations. Now we consider that configurations also contain some information about the expressions being evaluated in the background. This reflects the conceptual distinction between both kinds of evaluation. Hereafter, for a specification P, configurations are represented by elements of

$$BgConf(\mathsf{P}) = \mathcal{T}_{\mathsf{P}} \times \mathcal{T}_{\mathsf{P}} \times Perm(\mathsf{P}) \times D_{\mathsf{P}},$$

where the first component of such a tuple is the "main" expression, the second is the expression in the background, and the last two correspond to the database state with object permission. We write $\langle e,\ f,\ (\varrho, \mathcal{D})\rangle$ or $\langle e,\ (f, \varrho, \mathcal{D})\rangle$ for $(e, f, \varrho, \mathcal{D}) \in BgConf(\mathsf{P})$.

We follow a similar approach to the one used in Section 4.6. First we define the relation $\hookrightarrow_{\mathsf{P}} \subseteq BgConf(\mathsf{P}) \times BgConf(\mathsf{P})$ by the rules used for the definition of $\to_{\mathsf{P}}$ in the last section, assuming that

- $\to$ is replaced by $\hookrightarrow$, in all rules;
- database states also contain information about expressions being evaluated in the background; that is, the variables $\mathcal{D}$ and $\mathcal{D}'$ range over pairs of method expressions and database states with permission information (i.e., $\mathcal{D}, \mathcal{D}' \in \mathcal{T}_{\mathsf{P}} \times Perm(\mathsf{P}) \times D_{\mathsf{P}}$); and
- the operations on database states are composed with the projections and constructors associated to $\mathcal{T}_{\mathsf{P}} \times Perm(\mathsf{P}) \times D_{\mathsf{P}}$ (let ϱ, bg and db respectively give permission mappings, background expressions, and database states). For example, for $\mathcal{D} \in \mathcal{T}_{\mathsf{P}} \times Perm(\mathsf{P}) \times D_{\mathsf{P}}$, we now assume that $\mathcal{D} \oplus \Gamma$ stands for

$$(bg(\mathcal{D}), \varrho(\mathcal{D}), db(\mathcal{D}) \oplus \Gamma)$$

(similarly for $\cup$ and $\ominus$), and $[\![e]\!]_{\mathcal{D}}$ abbreviates $[\![e]\!]_{db(\mathcal{D})}$ (similarly for $[e]_{\mathcal{D}}$ and $Id(\mathcal{D})$). Lastly, $\mathcal{D} \oplus \{v \mapsto s\}$ means $(bg(\mathcal{D}), \varrho(\mathcal{D}) \oplus \{v \mapsto s\}, db(\mathcal{D}))$.

But we also need to introduce one rule indicating how `_&_` is evaluated. Essentially, `_&_` starts the evaluation of its second argument in parallel with the current background expression. The first argument is then given as result, as indicated by Rule **BgOp** (for *b*ack*g*round *op*erator):

Rule 4.7.1 (BgOp) For any $b \in \mathcal{T}_\Sigma$ and $\mathcal{P} \in \mathit{Perm}(\mathsf{P}) \times D_{\mathsf{P}}$,

$$\frac{}{\langle e \;\texttt{\&}\; f,\, b,\, \mathcal{P}\rangle \hookrightarrow \langle e,\, f \;\texttt{||}\; b,\, \mathcal{P}\rangle}$$

□

Naturally, we assume that `_&_ : U T -> U` is in MC, for any `T, U` $\in \mathcal{U}$.

The rule above finally defines $\hookrightarrow$. However, note that this relation only considers the evaluation of the main expression. We still have to specify how the expression in the background is evaluated; this is done by the relation $\rightarrowtail_{\mathsf{P}} \subseteq \mathit{BgConf}(\mathsf{P}) \times \mathit{BgConf}(\mathsf{P})$, defined by the following rule (**BgEval**, for *b*ack*g*round *eval*uation). A background expression is evaluated just as if it were a main expression, and any additional background expression resulting from this evaluation will also be evaluated in the background.

Rule 4.7.2 (BgEval) For any $b \in \mathcal{T}_\Sigma$, $\mathcal{P} \in \mathit{Perm}(\mathsf{P}) \times D_{\mathsf{P}}$, and a fully evaluated expression `skip`,

$$\frac{\langle b,\, \texttt{skip},\, \mathcal{P}\rangle \hookrightarrow \langle f,\, f',\, \mathcal{P}'\rangle}{\langle e,\, b,\, \mathcal{P}\rangle \rightarrowtail \langle e,\, b',\, \mathcal{P}'\rangle}$$

where b' is f, if $f' \equiv \texttt{skip}$; otherwise, b' is $f \;\texttt{||}\; f'$. □

The transitions specified by $\rightarrowtail$ are called "internal transitions". They change the state without evaluating the main expression, which is what can be observed by an (external) user of a FOOPS system.

Finally, the semantics of FOOPS with evaluation of expressions in the background is given by the union of the relations $\hookrightarrow_{\mathsf{P}}$ and $\rightarrowtail_{\mathsf{P}}$. We use $\rightarrow_{\mathsf{P}} \subseteq \mathit{BgConf}(\mathsf{P}) \times \mathit{BgConf}(\mathsf{P})$ to denote this union; we omit the obvious rules defining it[4].

Using two different relations to define $\rightarrow_{\mathsf{P}}$ might seem unnecessarily complicated. But this clearly separates transitions caused by the main expression from transitions generated by the expression in the background. This simplifies the definition of certain operators that should not consider internal transitions. For example, this is the case of `[_]` and `_[]_`. Using only one relation complicates the semantic definition. In fact, contrasting to what has been done here, it would not be possible to define $\rightarrow$ using some of the rules introduced in the previous sections.

[4]Note that $\rightarrow_{\mathsf{P}}$ is overloaded in this text.

Chapter 5

Refinement of Concurrent Object Oriented Programs

Based on the operational semantics described in the last chapter, we introduce a notion of refinement for states of FOOPS systems together with a proof technique for proving refinement. Basically, a state P is refined by a state Q if the behaviour associated to P is simulated by Q, so that we can replace the latter by the former without noticing any difference. Using this notion, we define refinement of (method) expressions and specifications. But we concentrate on the object level of FOOPS. Definitions of refinement (and equivalence) for the functional level may be found elsewhere [21]. Also, the definition of refinement introduced here is based on the semantics of FOOPS without object protection and evaluation in the background. Notions of refinement considering those aspects of FOOPS can be easily obtained by extending the definitions that will be presented here.

Here 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, and to verify the equivalence between two parallel composition operators. 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 [69], [72], and [48], 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 [69], [68], and [77], 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 [49, 54, 36].

Although we focus on FOOPS, our definition of refinement is independent of this language; it essentially depends on minimal assumptions about the operational semantics, which will be described and are satisfied by the FOOPS operational semantics given in the last chapter.

This chapter is organized in the following way. First we introduce and justify the notion of refinement of states and its associated proof technique. This depends on minimal assumptions about FOOPS operational semantics, which we describe. 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. Following that, the proof technique is used for verifying refinement of several examples. Lastly, we identify the different aspects considered by the definitions of refinement.

5.1 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 though, 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 is 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.

Each specification has an associated family of experiments. 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 $\mathcal{E}(\mathsf{P}) \subseteq \mathcal{E}(\mathsf{Q})$[1] 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 is 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. Formally, considering that $Fun(\mathsf{P})$ denotes the functional specification associated to P (i.e., $(S(\mathsf{P}), \leq(\mathsf{P}), F(\mathsf{P}), FE(\mathsf{P})))$, $\mathsf{P} \leq_S \mathsf{Q}$ if $\mathcal{T}_{Fun(\mathsf{P})} \restriction PS(\mathsf{P})$ is isomorphic to $\mathcal{T}_{Fun(\mathsf{Q})} \restriction PS(\mathsf{P})$, where $\mathcal{T}_{Fun(\mathsf{P})} \restriction S$ denotes the algebra obtained from $\mathcal{T}_{Fun(\mathsf{P})}$ by forgetting the carriers and associated operations of the sorts not in S.

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

[1]The relation $\subseteq$ is only defined when the related families have the same index set (see Section 2.1); however it is just a small extension to consider that the argument (family) on the right of $\subseteq$ is restricted to the index set of the family on the left whenever the family on the right has a larger index set.

still implies that the observable behaviour associated to Q is a particular case of the observable behaviour associated to P; that is the essence of simulation.

5.1.1 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 FOOPS operational semantics (see Chapter 4, but here we briefly repeat the necessary definitions and notation). 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 5.1.1 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.

An obvious consequence of the definition of simulation is that the sorts and classes in the rank of visible operations must be introduced by related specifications, since the experiments of one specification must also be experiments of the other. Also, observe that if the pair $(P,\, Q)$ is in a simulation relation, then usually the objects in state P are also in Q, and vice versa; otherwise, some experiments can be performed in one state and not in the other, indicating that one state is not simulated by the other (see Item 3).

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 is 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 simple experiments in isolation; it would be necessary to consider the effect of complex experiments consisting of a group of simple experiments being concurrently executed, including an arbitrary number of instances of the same simple experiments. 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. Visible methods can initiate nonterminating and nonatomic computations by invoking hidden methods as expressions to be evaluated in the background (see Section 3.2.2). 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.

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 5.1.1 For a specification P, the identity relation on $Conf(\mathsf{P})$, denoted $\mathcal{ID}_{\mathsf{P}}$, is a $(\mathsf{P},\ \mathsf{P})$-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 5.1.2 Let P and Q be specifications. For $(\mathsf{P},\ \mathsf{Q})$-simulations $\mathcal{S}$ and $\mathcal{S}'$, $\mathcal{S} \cup \mathcal{S}'$ is a $(\mathsf{P},\ \mathsf{Q})$-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

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

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

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

Proof: As $\mathcal{S}$ and $\mathcal{S}'$ are respectively a (P, O)-simulation and a (O, Q)-simulation, we have

$$\begin{array}{rr} \mathcal{E}(\mathsf{P}) \subseteq \mathcal{E}(\mathsf{O}), & \mathcal{E}(\mathsf{O}) \subseteq \mathcal{E}(\mathsf{Q}), \\ I(\mathsf{P}) \subseteq I(\mathsf{O}), & I(\mathsf{O}) \subseteq I(\mathsf{Q}), \\ \mathsf{P} \leq_{PS(\mathsf{P})} \mathsf{O}, & \text{and } \mathsf{O} \leq_{PS(\mathsf{O})} \mathsf{Q}. \end{array}$$

So, by the transitivity of $\subseteq$, we have

$$\mathcal{E}(\mathsf{P}) \subseteq \mathcal{E}(\mathsf{Q}) \text{ and } I(\mathsf{P}) \subseteq I(\mathsf{Q}).$$

As $\mathcal{E}(\mathsf{P}) \subseteq \mathcal{E}(\mathsf{O})$, terms used to form experiments of P are also available to form experiments of O; hence it easily follows that

$$PS(\mathsf{P}) \subseteq PS(\mathsf{O}).$$

From this and $\mathsf{O} \leq_{PS(\mathsf{O})} \mathsf{Q}$, it is easy to check that

$$\mathsf{O} \leq_{PS(\mathsf{P})} \mathsf{Q}.$$

Finally, by the transitivity of $\leq_{PS(\mathsf{P})}$, we have $\mathsf{P} \leq_{PS(\mathsf{P})} \mathsf{Q}$. This concludes the proof that there are (P, Q)-simulations. We can now proceed to verify that $\mathcal{S}\ ;\ \mathcal{S}'$ is such a simulation.

Let us assume that

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

Thus, for some O,

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

Let $Q \ \rightarrow\ Q'$. As $(O,\ Q) \in \mathcal{S}'$, for some O',

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

As $O \rightarrow^* O'$ and $(P,\ O) \in \mathcal{S}$, it is straightforward to verify that

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

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

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

$$\begin{array}{rr} O\ \rightarrow^*\ O' \not\rightarrow, & \mathcal{FE}_{\mathsf{O}}(o') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(q), \\ \mathcal{FE}_{\mathsf{O}}(o') \Rightarrow o' =_{(\mathsf{O},\mathsf{Q})} q, & \text{and } (O',\ Q) \in \mathcal{S}'. \end{array}$$

As $O \rightarrow^* O'$ and $(P,\ O) \in \mathcal{S}$, it is easy to check that

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

for some P'. As $(P',\ O') \in \mathcal{S}$, for some P'', we have

$$\begin{array}{rr} P'\ \rightarrow^*\ P'' \not\rightarrow, & \mathcal{FE}_{\mathsf{P}}(p'') \Leftrightarrow \mathcal{FE}_{\mathsf{O}}(o'), \\ \mathcal{FE}_{\mathsf{P}}(p'') \Rightarrow p'' =_{(\mathsf{P},\mathsf{O})} o', & \text{and } (P'',\ O') \in \mathcal{S}. \end{array}$$

Therefore

$$\begin{array}{rr} P\ \rightarrow^*\ P'' \not\rightarrow, & \mathcal{FE}_{\mathsf{P}}(p'') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(q), \\ \mathcal{FE}_{\mathsf{P}}(p'') \Rightarrow p'' =_{(\mathsf{P},\mathsf{Q})} q, & \text{and } (P'',\ Q) \in \mathcal{S}\ ;\ \mathcal{S}', \end{array}$$

since $\rightarrow^*$ is transitive, and it is easy to verify that $p =_{(\mathsf{P},\mathsf{O})} o$ and $o =_{(\mathsf{O},\mathsf{Q})} q$ imply $p =_{(\mathsf{P},\mathsf{Q})} q$.

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

$$\begin{gathered} \langle exp,\ \mathcal{O}\rangle\ \rightarrow^*\ O' \not\rightarrow, \\ (O',\ Q') \in \mathcal{S}', \text{ and} \\ (\langle o,\ \mathcal{O}'\rangle,\ \langle q,\ \mathcal{Q}'\rangle) \in \mathcal{S}'. \end{gathered}$$

As $(P,\ O) \in \mathcal{S}$, for some P', we have

$$\begin{gathered} \langle exp,\ \mathcal{P}\rangle\ \rightarrow^*\ P' \not\rightarrow, \\ (P',\ O') \in \mathcal{S}, \text{ and} \\ (\langle p,\ \mathcal{P}'\rangle,\ \langle o,\ \mathcal{O}'\rangle) \in \mathcal{S}. \end{gathered}$$

Hence $(P',\ Q') \in \mathcal{S}\ ;\ \mathcal{S}'$ and $(\langle p,\ \mathcal{P}'\rangle,\ \langle q,\ \mathcal{Q}'\rangle) \in \mathcal{S}\ ;\ \mathcal{S}'$. $\square$

5.1.2 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 5.1.1. 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 5.1.2 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. $\Box$

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.

Properties

In this section we establish 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 Section 5.1.2.

Theorem 5.1.4 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 5.1.2, $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ is a $(\mathsf{P},\ \mathsf{Q})$-simulation and includes any other $(\mathsf{P},\ \mathsf{Q})$-simulation. $\Box$

Now we prove that the refinement relation on states has some reflexive and transitive properties.

Theorem 5.1.5 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 5.1.1, $\mathcal{ID}_{\mathsf{P}}$ 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 5.1.3, $\mathcal{S}\ ;\ \mathcal{S}'$ is a $(\mathsf{P},\ \mathsf{Q})$-simulation. $\Box$

In fact, the relation of refinement involving two specifications is only necessarily reflexive if the specifications are the same.

As the refinement relation on states $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ is a (P, Q)-simulation (see Theorem 5.1.4), it directly follows that any transition from a state Q can be simulated by a sequence of transitions from any state P such that $P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q$. The following result is a small extension of that; it indicates that any sequence of transitions from Q can be simulated by a sequence of transitions from P.

Lemma 5.1.6 Let P and Q be specifications. Then $P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q$ imply

1. Whenever $Q \rightarrow^* Q'$ then, for some P', $P \rightarrow^* P'$ and $P' \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q'$.
2. If $Q \not\rightarrow$ then, for some P',

$$\begin{array}{rr} P \rightarrow^* P' \not\rightarrow, & \mathcal{FE}_{\mathsf{P}}(p') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(q), \\ \mathcal{FE}_{\mathsf{P}}(p') \Rightarrow p' =_{(\mathsf{P},\mathsf{Q})} q, & \text{and } P' \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q. \end{array}$$

3. For any experiment $exp \in \mathcal{E}(\mathsf{P})$, whenever $\langle exp, \mathcal{Q}\rangle \rightarrow^* Q' \not\rightarrow$ then, for some P',

$$\begin{array}{c} \langle exp, \mathcal{P}\rangle \rightarrow^* P' \not\rightarrow, \\ P' \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q', \text{ and} \\ \langle p, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}'\rangle. \end{array}$$

Proof: Let $P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q$. So there is a (P, Q)-simulation $\mathcal{S}$ such that

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

Hence we have that, for any experiment $exp \in \mathcal{E}(\mathsf{P})$, whenever $\langle exp, \mathcal{Q}\rangle \rightarrow^* Q' \not\rightarrow$ then, for some P',

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

As $\mathcal{S}$ is a (P, Q)-simulation, the last part of the result easily follows since then $\mathcal{S} \subseteq \sqsubseteq_{(\mathsf{P},\mathsf{Q})}$.

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

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

Then the second part of the result easily follows since $\mathcal{S} \subseteq \sqsubseteq_{(\mathsf{P},\mathsf{Q})}$.

Finally let us consider the first part of the result. Let $Q \rightarrow^* Q'$. So either

$$Q \equiv Q' \text{ or } Q \rightarrow Q_1 \rightarrow Q_2 \rightarrow \cdots \rightarrow Q'.$$

We split the proof in two cases. First let $Q \equiv Q'$. Then the first part of the result follows trivially by taking P' to be P. Second let $Q \to Q_1 \to Q_2 \to \cdots \to Q'$. As $(P, Q) \in \mathcal{S}$, we can justify the following diagram:

$$\begin{tikzcd}
Q \arrow[r] \arrow[d, leftrightarrow, "\mathcal{S}"] & Q_1 \arrow[r] \arrow[d, leftrightarrow, "\mathcal{S}"] & Q_2 \arrow[r] \arrow[d, leftrightarrow, "\mathcal{S}"] & \cdots \arrow[r] & Q' \arrow[d, leftrightarrow, "\mathcal{S}"] \\
P \arrow[r, "*"] & P_1 \arrow[r, "*"] & P_2 \arrow[r, "*"] & \cdots \arrow[r, "*"] & P'
\end{tikzcd}$$

Therefore, as $\mathcal{S} \subseteq \sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ and $\to^*$ is transitive, we have $P \to^* P'$ and $P' \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q'$. $\Box$

5.1.3 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 (see Section 4.2 and note that in some pathological cases P might have no associated state; those cases are not considered here, since the refinement relations would be trivially empty). 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 5.1.3 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. $\Box$

Reflecting the notion of refinement of states, the notion of refinement of expressions requires the results yielded by the evaluation of related expressions to be the same.

Properties

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 5.1.7 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 5.1.5, $\langle p, \emptyset_{\mathsf{P}}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{P})} \langle p, \emptyset_{\mathsf{P}}\rangle$. This proves that 1 holds. Now we prove 2. Assume that

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

Hence

$$\langle p, \emptyset_{\mathsf{P}}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{O})} \langle o, \emptyset_{\mathsf{O}}\rangle \text{ and } \langle o, \emptyset_{\mathsf{O}}\rangle \sqsubseteq_{(\mathsf{O},\mathsf{Q})} \langle q, \emptyset_{\mathsf{Q}}\rangle.$$

Therefore, by Theorem 5.1.5, we have $\langle p, \emptyset_{\mathsf{P}}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \emptyset_{\mathsf{Q}}\rangle$. □

5.1.4 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 5.1.4 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 $\emptyset_{\mathsf{P}} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \emptyset_{\mathsf{Q}}$. This is justified by Corollary 6.1.7 (see Chapter 6).

Properties

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

Theorem 5.1.8 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 $\mathsf{P} \sqsubseteq \mathsf{Q}$.

Proof: Let $exp \in \mathcal{E}(\mathsf{P})$. By Theorem 5.1.7, $exp \sqsubseteq_{(\mathsf{P},\mathsf{P})} exp$. This proves that 1 holds. Now we prove 2. Assume that

$$\mathsf{P} \sqsubseteq \mathsf{O} \text{ and } \mathsf{O} \sqsubseteq \mathsf{Q}.$$

Hence, for any $expp \in \mathcal{E}(\mathsf{P})$ and $expo \in \mathcal{E}(\mathsf{O})$,

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

Now let $exp \in \mathcal{E}(\mathsf{P})$. Then we have

$$exp \sqsubseteq_{(\mathsf{P},\mathsf{O})} exp.$$

So we have $\mathcal{E}(\mathsf{P}) \subseteq \mathcal{E}(\mathsf{O})$ and

$$exp \in \mathcal{E}(\mathsf{O}).$$

Thus we have

$$exp \sqsubseteq_{(\mathsf{O},\mathsf{Q})} exp.$$

Therefore, by Theorem 5.1.7, we have $exp \sqsubseteq_{(\mathsf{P},\mathsf{Q})} exp$. □

5.2 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.

5.2.1 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 us examine the relationship between the expressions

$$\texttt{x := 0 ; x := x + 1} \tag{5.1}$$

and

$$\texttt{x := 1}, \tag{5.2}$$

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 is 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 (5.1) and (5.2) are not equivalent in a concurrent environment: just consider that the evaluation of (5.1) might be interfered—after the execution of `x := 0` and before `x := x + 1`—whereas (5.2) is atomic. For example, if the expressions (5.1) and (5.2) 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

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

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

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

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 (5.1) is not a refinement of (5.2), 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 (5.2) is not a refinement of (5.1). Let us investigate that here, by checking whether

$$\langle \texttt{x := 0 ; x := x + 1}, \mathcal{P} \rangle \tag{5.3}$$

is refined by

$$\langle \texttt{x := 1}, \mathcal{P} \rangle, \tag{5.4}$$

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

As discussed in Section 5.1.2, this can be verified by finding a (`CELL`, `CELL`)-simulation relating the configurations (states) described above. As proved by Theorem 5.1.1, such a simulation may exist because the restrictions on related specifications are trivially valid, since the specifications are the same.

A Simulation

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

$$(\langle \texttt{x := 0 ; x := x + 1}, \mathcal{P} \rangle \quad , \quad \langle \texttt{x := 1}, \mathcal{P} \rangle) \tag{5.5}$$

$$(\langle e, \mathcal{P} \rangle \quad , \quad \langle e, \mathcal{P} \rangle), \tag{5.6}$$

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 (5.3) is refined by (5.4), 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 $\rightarrow$. See Chapter 4 for the details.

By Definition 5.1.1, $\mathcal{S}$ 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 5.4 is

$$\langle \texttt{x := 1}, \mathcal{P} \rangle \rightarrow \langle \texttt{x}, \mathcal{P} \oplus \texttt{val x = 1} \rangle, \tag{5.7}$$

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

From configuration 5.3, 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 specified by `val x = 1` (indeed, this can be deduced from the formal definition of $\oplus$ given in Chapter 4). From this result and the sequence of transitions introduced above, we can say that

$$\langle \texttt{x := 0 ; x := x + 1}, \mathcal{P}\rangle \rightarrow^{*} \langle \texttt{x}, \mathcal{P} \oplus \texttt{val x = 1}\rangle. \tag{5.8}$$

After analysing the possible transitions from states, it is straightforward to verify that condition 1 (of Definition 5.1.1) is valid: just consider the unique transitions (5.7) and (5.8), 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 (5.7).

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 is certainly the case of $\mathcal{S}$. In fact, we do not even have to consider what are the experiments associated to `CELL`: let us 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 is 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 us assume that `x` is not in $\mathcal{P}$. Condition 1 becomes vacuously true since

$$\langle \texttt{x := 1}, \mathcal{P}\rangle \not\rightarrow \tag{5.9}$$

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,

$$\langle \texttt{x := 0 ; x := x + 1}, \mathcal{P}\rangle \not\rightarrow . \tag{5.10}$$

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

Based on (5.9) and (5.10), 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 (5.9) and (5.10) 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 5.1.1 is trivial; in fact, this is a particular case of what was done to prove Theorem 5.1.1.

Finally, we can concluded that $\mathcal{S}$ is a simulation. From this and Definition 5.1.2, 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

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

is refined by

```
x := 1.
```

Failing to be a Simulation

In the beginning of this section we claimed that

```
x := 1,
```

is not refined by

```
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 us check if

$$\langle \texttt{x := 1}, \mathcal{P} \rangle \tag{5.11}$$

is refined by

$$\langle \texttt{x := 0 ; x := x + 1}, \mathcal{P} \rangle, \tag{5.12}$$

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

$$\langle \texttt{x ; x := x + 1}, \mathcal{P} \oplus \texttt{val x = 0} \rangle \tag{5.13}$$

to some $\rightarrow^*$-derivative of

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

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

$$\langle \texttt{x}, \mathcal{P} \oplus \texttt{val x = 1} \rangle \tag{5.14}$$

(see transition (5.7)).

The first $\rightarrow^*$-derivative cannot be related to state (5.13) by any simulation: just consider performing the experiment `val(x)` with both states. The experiment performed with (5.13) yields the value `0`, whereas the experiment performed with $\langle \texttt{x := 1}, \mathcal{P} \rangle$ 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 (5.14) and (5.13) 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 (5.13) reached from (5.12), since it is not a simulation of any state reached from (5.11); this is why the behaviour of (5.12) is not a particular case of the behaviour of (5.11). As there is no simulation relating the pairs of states analysed here, we confirm the claim made at the beginning of the section.

Further Examples of Simulation

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

$$\texttt{x := x + 1 ; x := x - 1} \tag{5.15}$$

is refined by

$$\texttt{skip(x)}, \tag{5.16}$$

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 5.2.1, we can also verify that (5.16) is not refined by (5.15).

Perhaps more interesting is verifying that

$$\texttt{skip(x) ; x := 0}$$

is refined by

$$\texttt{x := 0},$$

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

$$(\langle \texttt{skip(x) ; x} := 0, \mathcal{P}\rangle \quad , \quad \langle \texttt{x := 0}, \mathcal{P}\rangle) \tag{5.17}$$

$$(\langle e, \mathcal{P}\rangle \quad , \quad \langle e, \mathcal{P}\rangle), \tag{5.18}$$

and

$$(\langle \texttt{x := 0}, \mathcal{P}\rangle \quad , \quad \langle \texttt{skip(x) ; x := 0}, \mathcal{P}\rangle) \tag{5.19}$$

$$(\langle \texttt{x := 0}, \mathcal{P}\rangle \quad , \quad \langle \texttt{x ; x := 0}, \mathcal{P}\rangle) \tag{5.20}$$

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

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

$$(\langle \texttt{x := 0 [] x := 1}, \mathcal{P}\rangle \quad , \quad \langle \texttt{x := 1}, \mathcal{P}\rangle) \tag{5.21}$$

$$(\langle e, \mathcal{P}\rangle \quad , \quad \langle e, \mathcal{P}\rangle), \tag{5.22}$$

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.

5.2.2 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 3.2, 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` (see Section 3.2), 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`:

```
var E : Elt .
var B : Buffer .
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.

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 provides 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 5.1.1), 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 is 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 5.1.1, $\mathcal{E}(\texttt{BUFFER})$ should be contained in $\mathcal{E}(\texttt{BUFFERA})$, and

$$\texttt{BUFFER} \leq_{PS(\texttt{BUFFER})} \texttt{BUFFERA} \tag{5.23}$$

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 (5.23) 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 (5.23) 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.

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 us 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 5.1.2, we can check if our intuitions are correct by trying to prove that a relation $\mathcal{S} \subseteq \mathit{Conf}(\texttt{BUFFER}) \times \mathit{Conf}(\texttt{BUFFERA})$ containing states in the following form is a simulation:

$$(\langle e, \mathcal{P}\rangle \quad , \quad \langle e, \mathcal{Q}\rangle), \tag{5.24}$$

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 5.1.1. 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

$$\langle \texttt{reset(b´)}, \mathcal{Q} \rangle \tag{5.25}$$

leads to

$$\langle \texttt{b´}, \mathcal{Q} \oplus \texttt{fst b´ = 0} \oplus \texttt{next b´ = 0} \oplus \texttt{array b´ = null} \rangle, \tag{5.26}$$

whereas the unique transition from

$$\langle \texttt{reset(b´)}, \mathcal{P} \rangle \tag{5.27}$$

leads to

$$\langle \texttt{b´}, \mathcal{P} \oplus \texttt{elems b´ = nil} \rangle. \tag{5.28}$$

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 (5.28) and (5.26) 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 (5.28) and (5.26). 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 (5.25).

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

$$\langle \texttt{empty?(b)},\ \mathcal{Q}\rangle \tag{5.29}$$

leads to

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

where `BA` is an abbreviation for FE(`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 (5.29). Similarly, no transition is possible from

$$\langle \texttt{empty?(b)},\ \mathcal{P}\rangle, \tag{5.30}$$

if `elems b` is not defined in $\mathcal{P}$. However, if this is not the case, there is a unique transition from (5.30); it leads to

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

where `B` is an abbreviation for FE(`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 $[\![\ e\ ]\!]_{\texttt{B}}$ is the same as $[\![\ e\ ]\!]_{\texttt{EB}}$ for any e of `BUFFER`. Therefore we have that $[\![\ \texttt{l == nil}\ ]\!]_{\texttt{B}}$ is the same as $[\![\ \texttt{i == j}\ ]\!]_{\texttt{B}}$. 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 [21] 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 us 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

$$\langle \texttt{put(b,e)},\ \mathcal{Q}\rangle \rightarrow \langle \texttt{b},\ \mathcal{Q} \oplus \texttt{next b = j + 1} \oplus \texttt{array b = a[j] := e}\rangle, \tag{5.31}$$

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

$$\langle \texttt{put(b,e)}, \mathcal{P}\rangle \to \langle \texttt{b}, \mathcal{P} \oplus \texttt{elems b = l . e}\rangle. \tag{5.32}$$

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

`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:

$$\langle \texttt{get(b)}, \mathcal{Q}\rangle \to \langle [\![\texttt{a[i]}]\!]_{\texttt{BA}}, \mathcal{Q} \oplus \texttt{fst b = i + 1}\rangle, \tag{5.33}$$

and

$$\langle \texttt{get(b)}, \mathcal{P}\rangle \to \langle [\![\texttt{head(l)}]\!]_{\texttt{B}}, \mathcal{P} \oplus \texttt{elems b = tail(l)}\rangle. \tag{5.34}$$

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 is 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´`.

5.2.3 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

$$p \ ; \ (q \ ; \ o) \sqsubseteq_{(\mathsf{P},\mathsf{P})} (p \ ; \ q) \ ; \ o \tag{5.35}$$

and

$$(p \ ; \ q) \ ; \ o \sqsubseteq_{(\mathsf{P},\mathsf{P})} p \ ; \ (q \ ; \ o), \tag{5.36}$$

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

In order to verify 5.35, 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

$$(\langle p \ ; \ (q \ ; \ o), \mathcal{D}\rangle \quad , \quad \langle (p \ ; \ q) \ ; \ o, \mathcal{D}\rangle), \tag{5.37}$$

for any $\mathcal{D} \in Db(\mathsf{P})$, p, q, and o, is a (P, P)-simulation. This is straightforward, by using Theorem 5.1.1 and the rules of the operational semantics for sequential composition (see Section 4.4.1). Basically, we just have to observe the following facts. First let p be a fully evaluated expression. By Rule 4.4.2, we have

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

Therefore, by Rule 4.4.1, we have

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

But, by Rule 4.4.2, we also have

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

Second let us 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 4.4.1, we have

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

Hence, by Rule 4.4.1, 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 5.36 and many other properties of method combiners can be proved in a similar way.

5.2.4 Parallel Composition Operators

Using the definition of refinement of states, we can finally formally prove Theorem 4.4.1 (stated in Section 4.4.2), which establishes the equivalence between two parallel composition operators for FOOPS. Here we state the theorem again:

$$\langle p \ \texttt{||} \ q, \mathcal{D}\rangle \approx \langle p \ \texttt{|||} \ q, \mathcal{D}\rangle,$$

for an arbitrary database state $\mathcal{D}$ and expressions p and q.

First we have to define our notion of equivalence of states. For a specification P, we say that a state P is **equivalent** to a state Q, denoted by $P \approx Q$, if $P \sqsubseteq_{(\mathsf{P},\mathsf{P})} Q$ and $Q \sqsubseteq_{(\mathsf{P},\mathsf{P})} P$. So in order to prove the theorem stated above we have only to find a (P, P)-simulation relating $\langle p \ \texttt{||} \ q, \mathcal{D}\rangle$ to $\langle p \ \texttt{|||} \ q, \mathcal{D}\rangle$, and vice versa, for any $\mathcal{D} \in Db(\mathsf{P})$. Thus let us consider the relation $\mathcal{S}$ formed by $\sqsubseteq_{(\mathsf{P},\mathsf{P})}$ and all pairs in the following forms:

$$\begin{array}{lcl} (\langle p \ \texttt{||} \ q, \mathcal{D}\rangle & , & \langle p \ \texttt{|||} \ q, \mathcal{D}\rangle), \text{ and} \\ (\langle p \ \texttt{|||} \ q, \mathcal{D}\rangle & , & \langle p \ \texttt{||} \ q, \mathcal{D}\rangle), \end{array}$$

for any $\mathcal{D} \in Db(\mathsf{P})$.

As $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ is included in $\mathcal{S}$, using Lemma 5.1.6 we can easily conclude that pairs of $\mathcal{S}$ related by $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ satisfy the simulation conditions (see Definition 5.1.1). We only concentrate on pairs in the form

$$(\langle p \;||\; q,\, \mathcal{D}\rangle,\, \langle p \;|||\; q,\, \mathcal{D}\rangle),$$

since the proof follows in a very similar way to pairs in the other form. First let $\langle p \;|||\; q,\, \mathcal{D}\rangle \rightarrow_{\mathsf{P}} \langle e,\, \mathcal{D}'\rangle$. Thus we consider the different cases justifying this transition:

Case 1 By the equivalent of Rule 4.4.3 for $|||$, where $e \equiv p' \;|||\; q$ and $\langle p,\, \mathcal{D}\rangle \rightarrow_{\mathsf{P}} \langle p',\, \mathcal{D}'\rangle$. Hence, by Rule 4.4.3, we have

$$\langle p \;||\; q,\, \mathcal{D}\rangle \rightarrow_{\mathsf{P}} \langle p' \;||\; q,\, \mathcal{D}'\rangle.$$

Therefore the simulation condition is valid in this case, since $Q \rightarrow Q'$ implies $Q \rightarrow^* Q'$.

Case 2 When the transition is justified by the equivalent of Rule 4.4.4 for $|||$, the proof follows as in case 1.

Case 3 By the equivalent of Rule 4.4.5 for $|||$, where $e \equiv q$, $\mathcal{D}' \equiv \mathcal{D}$, and $\mathcal{FE}_{\mathsf{P}}(p)$. Hence, by Rule 4.4.5, we have

$$\langle p \;||\; q,\, \mathcal{D}\rangle \rightarrow_{\mathsf{P}} \langle q,\, \mathcal{D}\rangle.$$

Therefore the simulation condition is valid in this case, since $Q \rightarrow Q'$ implies $Q \rightarrow^* Q'$, $\mathcal{S}$ includes $\sqsubseteq_{(\mathsf{P},\mathsf{P})}$, and $\langle q,\, \mathcal{D}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{P})} \langle q,\, \mathcal{D}\rangle$ (see Theorem 5.1.5).

Case 4 When the transition is justified by the equivalent of Rule 4.4.6 for $|||$, the proof follows as in case 3.

Case 5 By Rule 4.4.7, where $e \equiv p' \;|||\; q'$, $\langle p,\, \mathcal{D}\rangle \rightarrow_{\mathsf{P}} \langle p',\, \mathcal{D}_1\rangle$, and $\langle q,\, \mathcal{D}_1\rangle \rightarrow_{\mathsf{P}} \langle q',\, \mathcal{D}'\rangle$. Hence, by Rules 4.4.3 and 4.4.4, we have

$$\langle p \;||\; q,\, \mathcal{D}\rangle \rightarrow_{\mathsf{P}} \langle p' \;||\; q,\, \mathcal{D}_1\rangle \rightarrow_{\mathsf{P}} \langle p' \;||\; q',\, \mathcal{D}'\rangle,$$

Therefore $\langle p \;||\; q,\, \mathcal{D}\rangle \rightarrow^*_{\mathsf{P}} \langle p' \;||\; q',\, \mathcal{D}'\rangle$. Then it is easy to see that the simulation condition holds in this case.

This concludes the case analysis. Now let $\langle p \;|||\; q,\, \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}}$. Hence, by the $|||$ version of Rules 4.4.3, 4.4.4, 4.4.5, and 4.4.6, we have

$$\begin{array}{cc} \langle p,\, \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}}, & \langle q,\, \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}}, \\ \neg\mathcal{FE}_{\mathsf{P}}(p), & \text{and } \neg\mathcal{FE}_{\mathsf{P}}(q). \end{array}$$

Hence, by Rules 4.4.3, 4.4.4, 4.4.5, and 4.4.6, we have

$$\langle p \;||\; q,\, \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}}\,.$$

Therefore the simulation condition holds in this case since $\rightarrow^*$ is reflexive and $p \;||\; q$ and $p \;|||\; q$ are clearly not fully evaluated.

Finally let $\langle exp,\, \mathcal{D}\rangle \rightarrow^*_{\mathsf{P}} \langle e,\, \mathcal{D}'\rangle \not\rightarrow_{\mathsf{P}}$, for any $exp \in \mathcal{E}(\mathsf{P})$. It is then easy to check that the simulation condition is valid in this case since, by Theorem 5.1.5, we have $\langle e,\, \mathcal{D}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{P})} \langle e,\, \mathcal{D}'\rangle$. This concludes the proof that $\mathcal{S}$ is a simulation and, therefore, the proof of Theorem 4.4.1.

5.3 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, besides the fact that whatever can be observed by performing experiments with Q can also be observed by 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 us assume that $P \sqsubseteq Q$.

It is well known in other frameworks that refinement can be achieved by reducing nondeterminism [69, 51, 71]. 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 5.2.1, where we check that a nondeterministic choice is refined by an assignment.

Also, data refinement [49, 54, 36] is supported 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 5.2.2, 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 5.2.1, the expression

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

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

In fact, refinement can 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 5.2.1, 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 $\twoheadrightarrow$-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 (in the sense of Section 5.2.4) 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.

Chapter 6

A Theory of Refinement for FOOPS

Now we develop a theory in order to show that the relation of refinement of expressions introduced in the last chapter is a congruence with respect to most FOOPS predefined method combiners; that is, choosing the parallel composition combiner as example, we are interested in proving that

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

implies

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

for any expression o formed by experiments of P and most predefined method combiners.

In fact, the congruence property only holds for contexts formed by *visible* operations (experiments). This is what should be expected for any notion of refinement based on the observational behaviour of states with respect to a restricted group of *visible* operations. This is definitely the case of the notion of refinement introduced in the last chapter (see condition 3c of Definition 5.1.1). So there is no guarantee that the evaluation of a *hidden* operation in two states related by $\sqsubseteq$ will not have completely different observable effects.

The congruence property of refinement of expressions is what supports the compositional verification and derivation of implementations; that is essential for formal development of large systems. But note that this only supports compositionality *in the small* (i.e., regarding expressions and their composition [69, 51]). Compositionality *in the large* (i.e., regarding modules and their composition [23]) should be supported by a congruence property of refinement of specifications with respect to FOOPS module interconnection operations (such as module importation); however, this is out of the scope of this work which does not consider the semantics of FOOPS module system.

We organize this chapter in the following way. First we formalize the constraints on experiments informally described in Section 5.1.1. This allows us to prove that $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ includes the identity on experiments whenever there is a (P, Q)-simulation relating initial states; that is, for any experiment *exp*, $exp \sqsubseteq_{(\mathsf{P},\mathsf{Q})} exp$ whenever there is a (P, Q)-simulation relating initial states. We also extend this result for composition of experiments using certain FOOPS predefined combiners. It then becomes easy to derive the desired congruence result; we consider each combiner separately. Lastly, we show other compositionality results.

6.1 Experiments

As discussed in Section 5.1.1, experiments should be terminating and atomic. An expression $e \in \mathcal{T}_{\mathsf{P}}$ is **terminating**, denoted $\downarrow_{\mathsf{P}} e$, if $\langle e, \mathcal{D}\rangle$ is terminating for any database state $\mathcal{D} \in Db(\mathsf{P})$ (see Section 4.4.6 for the definition of a terminating configuration). We drop the subscript from $\downarrow$ when it is not confusing.

Atomicity is not only achieved by ensuring that an expression is fully evaluated in one $\rightarrow$-transition. In fact, we consider that the atomic evaluation of an expression can take many transitions, as long as the only (if any) transition that depends on the state or corresponds to a choice is the last one; also, such a transition must yield a fully evaluated expression. Note that this still guarantees that any access to the database and any choices are made in only one $\rightarrow$-transition; no information is lost by the other transitions. In order to formalize those ideas, we introduce the relation $\rightrightarrows$, which indicates that an expression evaluates to another in one $\rightarrow$-transition without accessing the database state nor making choices. Here is the formal definition:

$$p \rightrightarrows_{\mathsf{P}} q$$

if, for any database state $\mathcal{D} \in Db(\mathsf{P})$,

$$\langle p, \mathcal{D}\rangle \rightarrow_{\mathsf{P}} \langle q, \mathcal{D}\rangle \text{ and } \langle p, \mathcal{D}\rangle \rightarrow_{\mathsf{P}} O \Rightarrow O \equiv \langle q, \mathcal{D}\rangle,$$

where $\equiv$ is the identity on configurations. An obvious extension of $\rightrightarrows$ for states is defined as follows:

$$P \rightrightarrows_{\mathsf{P}} Q$$

if

$$p \rightrightarrows_{\mathsf{P}} q \text{ and } P \rightarrow_{\mathsf{P}} Q,$$

assuming that p and q are respectively the expressions in P and Q. (Note that $P \rightrightarrows_{\mathsf{P}} Q$ imply $\mathcal{P} \equiv \mathcal{Q}$, where $\mathcal{P}$ and $\mathcal{Q}$ are respectively the database states in P and Q, and $\equiv$ denotes the identity on database states.) We drop the subscripts from those relations when not confusing. Also, when $P \rightrightarrows Q$ does not hold, we write $P \not\rightrightarrows Q$. A similar notation is used for the associated relation on expressions. Lastly, the reflexive and transitive closure of $\rightrightarrows$ is represented by $\rightrightarrows^*$.

We can then formalize the constraint on experiments, where $\mathcal{FE}_{\mathsf{P}}(P)$ is an abbreviation for $\mathcal{FE}_{\mathsf{P}}(p)$:

$$exp \in \mathcal{E}(\mathsf{P})$$

implies

$$\downarrow_{\mathsf{P}} exp$$

and, for any $\mathcal{D} \in Db(\mathsf{P})$, whenever $\langle exp, \mathcal{D}\rangle \rightarrow^*_{\mathsf{P}} O$ then either

$$\langle exp, \mathcal{D}\rangle \rightrightarrows^*_{\mathsf{P}} O$$

or for some Q,

$$\langle exp, \mathcal{D}\rangle \rightrightarrows^*_{\mathsf{P}} Q \rightarrow_{\mathsf{P}} O \text{ and } \mathcal{FE}_{\mathsf{P}}(O).$$

When reasoning about refinement involving two specifications, this property of experiments should be valid for experiments of both specifications.

6.1.1 Transition Induction

The relation $\rightrightarrows^*$ can be formally defined by the following inference rules:

Rule 6.1.1

$$\frac{}{p \rightrightarrows^* p}$$

□

and

Rule 6.1.2

$$\frac{p \rightrightarrows^* o \qquad o \rightrightarrows q}{p \rightrightarrows^* q}$$

□

Hence any transition $p \rightrightarrows^* q$ can be justified by a proof tree constructed using those rules. This allows us to prove properties of a $\rightrightarrows^*$-transition by induction on the depth of proof trees. This is a well known proof technique extensively used in [69], where it is called *proof by transition induction*; it will be quite useful here too. The same technique can also be used to prove properties of $\rightarrow$-transitions.

Using this proof technique, we now derive a property of $\rightrightarrows^*$ that will be useful for proving some results later. Informally, if $p \rightrightarrows^* q$ then, for any database state $\mathcal{D}$, there is a unique sequence of transitions from $\langle p, \mathcal{D}\rangle$ to $\langle q, \mathcal{D}\rangle$, since during the evaluation of p to q the database state cannot be modified and no choices can be made. So, whenever $\langle p, \mathcal{D}\rangle \rightarrow^* O$ then either O is one of the states in the sequence of transitions from $\langle p, \mathcal{D}\rangle$ to $\langle q, \mathcal{D}\rangle$, or O is a $\rightarrow^*$-derivative of $\langle q, \mathcal{D}\rangle$. This is formalized by the following lemma.

Lemma 6.1.1 If $p \rightrightarrows^*_{\mathsf{P}} q$ then, for any $\mathcal{D} \in Db(\mathsf{P})$, whenever

$$\langle p, \mathcal{D}\rangle \rightarrow^*_{\mathsf{P}} O$$

then

$$\langle q, \mathcal{D}\rangle \rightarrow^*_{\mathsf{P}} O \text{ or } O \rightrightarrows^*_{\mathsf{P}} \langle q, \mathcal{D}\rangle.$$

Proof: We omit the subscripts from the relations, assuming that they are associated to an arbitrary specification P. The proof follows by transition induction on the inference of $p \rightrightarrows^* q$. Let us assume that

$$p \rightrightarrows^* q.$$

We then analyse the possible cases justifying the last step of this inference:

Case 1 By Rule 6.1.1, where $p \equiv q$ ($\equiv$ denotes syntactical equality of expressions). Then the result trivially follows since what remains to be proved is an instance of the tautology $\alpha \Rightarrow \alpha \vee \beta$.

Case 2 By Rule 6.1.2, where $p \rightrightarrows^* e$ (by a shorter inference) and $e \rightrightarrows q$, for some e. Then, by induction, we have

$$\text{whenever } \langle p,\, \mathcal{D}\rangle \rightarrow^* O \text{ then } \langle e,\, \mathcal{D}\rangle \rightarrow^* O \text{ or } O \rightrightarrows^* \langle e,\, \mathcal{D}\rangle.$$

Let $\langle p,\, \mathcal{D}\rangle \rightarrow^* O$. Hence

$$\langle e,\, \mathcal{D}\rangle \rightarrow^* O \text{ or } O \rightrightarrows^* \langle e,\, \mathcal{D}\rangle.$$

We split the proof in two cases. First let $O \rightrightarrows^* \langle e,\, \mathcal{D}\rangle$. As $e \rightrightarrows q$, we have

$$\langle e,\, \mathcal{D}\rangle \rightrightarrows \langle q,\, \mathcal{D}\rangle.$$

Therefore, by transitivity, $O \rightrightarrows^* \langle q,\, \mathcal{D}\rangle$.

Now let $\langle e,\, \mathcal{D}\rangle \rightarrow^* O$. So we have

$$\langle e,\, \mathcal{D}\rangle \equiv O \quad \text{or} \quad \langle e,\, \mathcal{D}\rangle \rightarrow O' \rightarrow^* O,$$

for some O'. Again, we split the proof in two cases. First let $\langle e,\, \mathcal{D}\rangle \equiv O$. Then trivially $O \rightrightarrows^* \langle e,\, \mathcal{D}\rangle$, so the proof follows as in the previous main case. Second let $\langle e,\, \mathcal{D}\rangle \rightarrow O'$ and $O' \rightarrow^* O$. Then, as $e \rightrightarrows q$, we have

$$O' \equiv \langle q,\, \mathcal{D}\rangle.$$

Hence $\langle q,\, \mathcal{D}\rangle \rightarrow^* O$.

□

6.1.2 Properties of Experiments

Using Lemma 6.1.1, we can derive a direct consequence of the constraint on experiments: during the evaluation of an experiment, any transition that makes a choice or access the database leads to a fully evaluated state. This is formalized by the following lemma.

Lemma 6.1.2 Let $exp \in \mathcal{E}(\mathsf{P})$. Whenever $\langle exp,\, \mathcal{D}\rangle \rightrightarrows^*_{\mathsf{P}} O \rightarrow_{\mathsf{P}} P$ and $O \not\rightrightarrows_{\mathsf{P}} P$ then

$$P \not\rightarrow_{\mathsf{P}} \text{ and } \mathcal{FE}_{\mathsf{P}}(P).$$

Proof: We omit the subscripts from the relations, assuming that they are associated to an arbitrary specification P. Let us assume that

$$\langle exp,\, \mathcal{D}\rangle \rightrightarrows^* O \rightarrow P \text{ and } O \not\rightrightarrows P.$$

Hence, by the definition of $\rightrightarrows^*$, we have

$$\langle exp,\, \mathcal{D}\rangle \rightarrow^* O.$$

Thus, by transitivity, we have

$$\langle exp,\, \mathcal{D}\rangle \rightarrow^* P.$$

So, as $exp \in \mathcal{E}(\mathsf{P})$, either

$$\langle exp,\, \mathcal{D}\rangle \rightrightarrows^* P \quad \text{or} \quad \langle exp,\, \mathcal{D}\rangle \rightrightarrows^* Q \rightarrow P \text{ and } \mathcal{FE}_{\mathsf{P}}(P),$$

for some Q. We then split the proof in two cases. First let $\langle exp, \mathcal{D}\rangle \rightrightarrows^* Q \rightarrow P$ and $\mathcal{FE}_{\mathsf{P}}(P)$. Then the result easily follows since there is no rule in the operational semantics for evaluating fully evaluated expressions.

Now let $\langle exp, \mathcal{D}\rangle \rightrightarrows^* P$. Hence it is easy to check that

$$exp \rightrightarrows^* p \text{ and } P \equiv \langle p, \mathcal{D}\rangle.$$

Then, by Lemma 6.1.1, either

$$\langle p, \mathcal{D}\rangle \rightarrow^* O \text{ or } O \rightrightarrows^* \langle p, \mathcal{D}\rangle.$$

Again we split the proof in two cases. First let $O \rightrightarrows^* \langle p, \mathcal{D}\rangle$. As $O \rightarrow P$ and $O \not\rightrightarrows P$, it is then easy to infer

$$O \not\rightrightarrows .$$

Hence

$$O \equiv \langle p, \mathcal{D}\rangle.$$

Therefore, by transitivity and symmetry, we have $O \equiv P$. But, as $O \rightarrow P$, this indicates that the evaluation of exp might not terminate, contradicting our initial assumption: $exp \in \mathcal{E}(\mathsf{P})$.

Finally let $\langle p, \mathcal{D}\rangle \rightarrow^* O$. Hence

$$P \rightarrow^* O.$$

But, as $O \rightarrow P$, this contradicts our initial assumption: $exp \in \mathcal{E}(\mathsf{P})$. □

Motivated by condition 3c of the definition of simulation and the atomic nature of experiments, we now proceed to prove that, for any experiment $exp \in \mathcal{E}(\mathsf{P})$, $exp \sqsubseteq_{(\mathsf{P},\mathsf{Q})} exp$ whenever there is a (P, Q)-simulation relating initial states. In fact, as essentially no information is lost by $\rightrightarrows_{_}$transitions, we derive the more general result that any $\rightrightarrows_{\mathsf{P}}$-derivative of exp is simulated by any $\rightrightarrows_{\mathsf{Q}}$-derivative of exp, where o is a $\rightrightarrows_{\mathsf{P}}$-**derivative** of exp if $exp \rightrightarrows^*_{\mathsf{P}} o$. In order to accomplish that, we give a (P, Q)-simulation relating $\rightrightarrows$-derivatives of exp, for any $exp \in \mathcal{E}(\mathsf{P})$. But first we derive two auxiliary Lemmas which will help us to prove the first two conditions of this simulation relation (see Definition 5.1.1).

Lemma 6.1.3 Let $exp \in \mathcal{E}(\mathsf{P})$ and $P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q$. For any $expp$ and $expq$ such that $exp \rightrightarrows^*_{\mathsf{P}} expp$ and $exp \rightrightarrows^*_{\mathsf{Q}} expq$, whenever

$$\langle expq, \mathcal{Q}\rangle \rightarrow \langle expq', \mathcal{Q}'\rangle$$

then, for some $\langle expp', \mathcal{P}'\rangle$,

$$\langle expp, \mathcal{P}\rangle \rightarrow^* \langle expp', \mathcal{P}'\rangle,$$

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

$$expp' =_{(\mathsf{P},\mathsf{Q})} expq' \quad \text{or} \quad exp \rightrightarrows^*_{\mathsf{P}} expp' \text{ and } exp \rightrightarrows^*_{\mathsf{Q}} expq'.$$

Proof: Let us assume that

$$\langle expq, \mathcal{Q}\rangle \rightarrow \langle expq', \mathcal{Q}'\rangle.$$

Trivially we have

$$\langle expq,\ \mathcal{Q}\rangle \rightrightarrows_{\mathsf{Q}} \langle expq',\ \mathcal{Q}'\rangle \text{ or } \langle expq,\ \mathcal{Q}\rangle \not\rightrightarrows_{\mathsf{Q}} \langle expq',\ \mathcal{Q}'\rangle.$$

We split the proof in two cases. First let $\langle expq,\ \mathcal{Q}\rangle \rightrightarrows_{\mathsf{Q}} \langle expq',\ \mathcal{Q}'\rangle$. Hence

$$expq \rightrightarrows_{\mathsf{Q}} expq' \text{ and } \mathcal{Q} \equiv \mathcal{Q}'.$$

Then, by transitivity, we have

$$exp \rightrightarrows_{\mathsf{Q}}^{*} expq'.$$

Therefore the result easily follows by choosing $\langle expp',\ \mathcal{P}'\rangle$ to be $\langle expp,\ \mathcal{P}\rangle$, since $\rightarrow^{*}$ is reflexive.

Now let $\langle expq,\ \mathcal{Q}\rangle \not\rightrightarrows_{\mathsf{Q}} \langle expq',\ \mathcal{Q}'\rangle$. As $exp \rightrightarrows_{\mathsf{Q}}^{*} expq$, it is easy to check

$$\langle exp,\ \mathcal{Q}\rangle \rightarrow^{*} \langle expq,\ \mathcal{Q}\rangle.$$

Then, by Lemma 6.1.2, we have

$$\langle expq',\ \mathcal{Q}'\rangle \not\rightarrow \text{ and } \mathcal{FE}_{\mathsf{Q}}(\langle expq',\ \mathcal{Q}'\rangle).$$

By transitivity, we have

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

Hence, as $P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q$, for some $\langle expp',\ \mathcal{P}'\rangle$, we have

$$\langle exp,\ \mathcal{P}\rangle \rightarrow^{*} \langle expp',\ \mathcal{P}'\rangle \not\rightarrow,$$
$$\langle expp',\ \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle expq',\ \mathcal{Q}'\rangle \text{ and}$$
$$\langle p,\ \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\ \mathcal{Q}'\rangle.$$

So, as $exp \rightrightarrows_{\mathsf{P}}^{*} expp$, by Lemma 6.1.1, we have

$$\langle expp,\ \mathcal{P}\rangle \rightarrow^{*} \langle expp',\ \mathcal{P}'\rangle \text{ or } \langle expp',\ \mathcal{P}'\rangle \rightrightarrows_{\mathsf{P}}^{*} \langle expp,\ \mathcal{P}\rangle.$$

Thus we split the proof in two cases again. First let $\langle expp,\ \mathcal{P}\rangle \rightarrow^{*} \langle expp',\ \mathcal{P}'\rangle$. Therefore, as $\langle expp',\ \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle expq',\ \mathcal{Q}'\rangle$, we have $expp' =_{(\mathsf{P},\mathsf{Q})} expq'$.

Finally let $\langle expp',\ \mathcal{P}'\rangle \rightrightarrows_{\mathsf{P}}^{*} \langle expp,\ \mathcal{P}\rangle$. Hence, as $\langle expp',\ \mathcal{P}'\rangle \not\rightarrow$, we have

$$\langle expp',\ \mathcal{P}'\rangle \equiv \langle expp,\ \mathcal{P}\rangle.$$

Therefore $\langle expp,\ \mathcal{P}\rangle \rightarrow^{*} \langle expp',\ \mathcal{P}'\rangle$ and the result follows as in the last case. □

Now we establish the second lemma.

Lemma 6.1.4 Let $exp \in \mathcal{E}(\mathsf{P})$ and $P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q$. For any $expp$ and $expq$ such that $exp \rightrightarrows_{\mathsf{P}}^{*} expp$ and $exp \rightrightarrows_{\mathsf{Q}}^{*} expq$, if

$$\langle expq,\ \mathcal{Q}\rangle \not\rightarrow$$

then, for some $\langle expp', \mathcal{P}'\rangle$,

$$\langle expp, \mathcal{P}\rangle \to^* \langle expp', \mathcal{P}'\rangle \not\to,$$
$$\langle p, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}\rangle,$$
$$\mathcal{FE}_{\mathsf{P}}(expp') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(expq),$$
$$\mathcal{FE}_{\mathsf{P}}(expp') \text{ implies } expp' =_{(\mathsf{P},\mathsf{Q})} expq, \text{ and}$$
$$\neg\mathcal{FE}_{\mathsf{P}}(expp') \text{ implies } exp \rightrightarrows^*_{\mathsf{P}} expp'.$$

Proof: Let us assume that

$$\langle expq, \mathcal{Q}\rangle \not\to .$$

As $exp \rightrightarrows^*_{\mathsf{Q}} expq$, it is easy to check

$$\langle exp, \mathcal{Q}\rangle \to^* \langle expq, \mathcal{Q}\rangle \not\to.$$

Hence, as $P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q$, for some $\langle expp', \mathcal{P}'\rangle$, we have

$$\langle exp, \mathcal{P}\rangle \to^* \langle expp', \mathcal{P}'\rangle \not\to,$$
$$\langle expp', \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle expq, \mathcal{Q}\rangle \text{ and}$$
$$\langle p, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}\rangle.$$

So, as $\langle expp', \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle expq, \mathcal{Q}\rangle$, we have

$$\mathcal{FE}_{\mathsf{P}}(expp') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(expq), \text{ and}$$
$$\mathcal{FE}_{\mathsf{P}}(expp') \text{ implies } expp' =_{(\mathsf{P},\mathsf{Q})} expq.$$

As $\langle exp, \mathcal{P}\rangle \to^* \langle expp', \mathcal{P}'\rangle \not\to$ and $exp \rightrightarrows^*_{\mathsf{P}} expp$, by Lemma 6.1.1, we have

$$\langle expp, \mathcal{P}\rangle \to^* \langle expp', \mathcal{P}'\rangle \text{ or } \langle expp', \mathcal{P}'\rangle \rightrightarrows^*_{\mathsf{P}} \langle expp, \mathcal{P}\rangle.$$

Thus we split the proof in two cases. First let $\langle expp, \mathcal{P}\rangle \to^* \langle expp', \mathcal{P}'\rangle$. Then let $\neg\mathcal{FE}_{\mathsf{P}}(expp')$. As $exp \rightrightarrows^*_{\mathsf{P}} expp$, it is easy to check

$$\langle exp, \mathcal{P}\rangle \rightrightarrows^*_{\mathsf{P}} \langle expp, \mathcal{P}\rangle.$$

Hence, using Lemma 6.1.1 it becomes easy to see that

$$\langle expp, \mathcal{P}\rangle \rightrightarrows^*_{\mathsf{P}} \langle expp', \mathcal{P}'\rangle,$$

otherwise we would have that $\mathcal{FE}_{\mathsf{P}}(expp')$, contradicting our assumption. So we easily can infer

$$expp \rightrightarrows^*_{\mathsf{P}} expp'.$$

Therefore, by transitivity, $exp \rightrightarrows^*_{\mathsf{P}} expp'$.

Finally let $\langle expp', \mathcal{P}'\rangle \rightrightarrows^*_{\mathsf{P}} \langle expp, \mathcal{P}\rangle$. Hence, as $\langle expp', \mathcal{P}'\rangle \not\to$, we have

$$\langle expp', \mathcal{P}'\rangle \equiv \langle expp, \mathcal{P}\rangle.$$

Therefore the result follows as in the other case. □

The following lemma introduces the simulation relating $\rightrightarrows$-derivatives of an experiment *exp*, where we let $\mathcal{P} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q}$ hold whenever $\langle p, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}\rangle$ holds for some p and q.

Lemma 6.1.5 Let $\mathcal{S}$ be the relation consisting of $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ and all pairs in the form

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

where $\mathcal{P} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q}$ and there is an $exp \in \mathcal{E}(\mathsf{P})$ such that $exp \rightrightarrows^*_{\mathsf{P}} expp$ and $exp \rightrightarrows^*_{\mathsf{Q}} expq$. Then $\mathcal{S}$ is a (P, Q)-simulation.

Proof: It follows directly from Theorem 5.1.4, Lemma 6.1.3, Lemma 6.1.4, and by checking that whenever $p =_{(\mathsf{P},\mathsf{Q})} q$ then $\langle p, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}\rangle$, for any $\mathcal{P}$ and $\mathcal{Q}$ such that $\mathcal{P} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q}$. (This additional result can be trivially derived by showing that the relation formed by $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ and the set of all pairs in the form $(\langle p, \mathcal{P}\rangle, \langle q, \mathcal{Q}\rangle)$, where $p =_{(\mathsf{P},\mathsf{Q})} q$ and $\mathcal{P} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q}$, is a (P, Q)-simulation. Note that $p =_{(\mathsf{P},\mathsf{Q})} q$ implies that p and q are fully evaluated.) □

Now we can prove that any $\rightrightarrows_{\mathsf{P}}$-derivative of an experiment *exp* is simulated by any $\rightrightarrows_{\mathsf{Q}}$-derivative of *exp* whenever there is a (P, Q)-simulation relating initial states.

Theorem 6.1.6 Let $exp \in \mathcal{E}(\mathsf{P})$, $exp \rightrightarrows^*_{\mathsf{P}} expp$, and $exp \rightrightarrows^*_{\mathsf{Q}} expq$. Thus, if $\emptyset_{\mathsf{P}} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \emptyset_{\mathsf{Q}}$ then $expp \sqsubseteq_{(\mathsf{P},\mathsf{Q})} expq$.

Proof: Let us assume that

$$\emptyset_{\mathsf{P}} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \emptyset_{\mathsf{Q}}.$$

Then, by Lemma 6.1.5, there is a (P, Q)-simulation $\mathcal{S}$ such that

$$(\langle expp, \emptyset_{\mathsf{P}}\rangle, \langle expq, \emptyset_{\mathsf{Q}}\rangle) \in \mathcal{S}.$$

Hence

$$\langle expp, \emptyset_{\mathsf{P}}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle expq, \emptyset_{\mathsf{Q}}\rangle.$$

Therefore $expp \sqsubseteq_{(\mathsf{P},\mathsf{Q})} expq$. □

Then we can easily derive the expected result: $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ includes the identity on experiments whenever there is a (P, Q)-simulation relating initial states.

Corollary 6.1.7 Let $exp \in \mathcal{E}(\mathsf{P})$. Thus, if $\emptyset_{\mathsf{P}} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \emptyset_{\mathsf{Q}}$ then $exp \sqsubseteq_{(\mathsf{P},\mathsf{Q})} exp$.

Proof: Directly by Theorem 6.1.6, since $\rightrightarrows^*_{\mathsf{P}}$ and $\rightrightarrows^*_{\mathsf{Q}}$ are reflexive. □

6.2 Composition of Experiments

Following the lines of the presentation that $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ includes the identity on *experiments* whenever there is a (P, Q)-simulation relating initial states, now we prove the more general result that $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ includes the identity on *composition of experiments* whenever there is a (P, Q)-simulation relating initial states.

A composition of experiments is a term formed by composing experiments and fully evaluated expressions of primary sorts with some of FOOPS predefined method combiners. The atomic evaluation and (external) nondeterministic choice method combiners are not considered for composing experiments because they only preserve refinement when their arguments satisfy some specific conditions, as we will discuss later. Then, the set of **composition of experiments** of a specification P, denoted by $\mathcal{CE}(\mathsf{P})$, is the least set satisfying the following conditions:

- if $e \in I(\mathsf{P}) \cup \mathcal{E}(\mathsf{P})$ then $e \in \mathcal{CE}(\mathsf{P})$;
- if $e \in \mathcal{T}_{F(\mathsf{P})}$, $LS(e) \in PS(\mathsf{P})$, and $e \equiv [\![e]\!]_{FE(\mathsf{P})}$ then $e \in \mathcal{CE}(\mathsf{P})$; and
- $e_1 \ \texttt{||}\ e_2$, $e_1 \ \texttt{;}\ e_2$, $\texttt{result}\ e_1 \ \texttt{;}\ e_2$, $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi} \in \mathcal{CE}(\mathsf{P})$,

for any $e_1, e_2, e_3 \in \mathcal{CE}(\mathsf{P})$. Note that $\mathcal{E}(\mathsf{P}) \subseteq \mathcal{E}(\mathsf{Q})$, $I(\mathsf{P}) \subseteq I(\mathsf{Q})$, and $\mathsf{P} \leq_{PS(\mathsf{P})} \mathsf{Q}$ imply $\mathcal{CE}(\mathsf{P}) \subseteq \mathcal{CE}(\mathsf{Q})$.

As made explicit at the beginning of this section, we are interested in proving that $cexp \sqsubseteq_{(\mathsf{P},\mathsf{Q})} cexp$, for any $cexp \in \mathcal{CE}(\mathsf{P})$, whenever there is a (P, Q)-simulation relating initial states. But, in fact, we prove the more general result that any $\rightrightarrows_{\mathsf{P}}$-derivative of $cexp$ is simulated by any $\rightrightarrows_{\mathsf{Q}}$-derivative of $cexp$, where the set of $\rightrightarrows_{\mathsf{P}}$-**derivatives** of a composition of experiments $cexp$, denoted $cexp \rightrightarrows^*_{\mathsf{P}}$, is defined as follows:

$$cexp \rightrightarrows^*_{\mathsf{P}} \;=\; \{e \mid cexp \rightrightarrows^*_{\mathsf{P}} e\}, \text{ if } cexp \in \mathcal{E}(\mathsf{P}).$$

$$cexp \rightrightarrows^*_{\mathsf{P}} \;=\; \{cexp\}, \text{ if } \mathcal{FE}_{\mathsf{P}}(cexp).$$

$$(cexp_1 \ \texttt{||}\ cexp_2) \rightrightarrows^*_{\mathsf{P}} \;=\; \{e_1 \ \texttt{||}\ e_2 \mid e_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}} \text{ and } e_2 \in cexp_2 \rightrightarrows^*_{\mathsf{P}}\}$$

$$(cexp_1 \ \texttt{;}\ cexp_2) \rightrightarrows^*_{\mathsf{P}} \;=\; \{e_1 \ \texttt{;}\ cexp_2 \mid e_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}}\}$$

$$(\texttt{result}\ cexp_1 \ \texttt{;}\ cexp_2) \rightrightarrows^*_{\mathsf{P}} \;=\;$$

$$\{\texttt{result}\ e_1 \ \texttt{;}\ cexp_2 \mid e_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}} \text{ and } \neg\mathcal{FE}_{\mathsf{P}}(cexp_1)\} \cup$$

$$\{\texttt{result}\ cexp_1 \ \texttt{;}\ e_2 \mid e_2 \in cexp_2 \rightrightarrows^*_{\mathsf{P}} \text{ and } \mathcal{FE}_{\mathsf{P}}(cexp_1)\}$$

$$(\texttt{if}\ cexp_1\ \texttt{then}\ cexp_2\ \texttt{else}\ cexp_3\ \texttt{fi}) \rightrightarrows^*_{\mathsf{P}} \;=\;$$

$$\{\texttt{if}\ e_1\ \texttt{then}\ cexp_2\ \texttt{else}\ cexp_3\ \texttt{fi} \mid e_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}}\}$$

for any $cexp_1, cexp_2, cexp_3 \in \mathcal{CE}(\mathsf{P})$. Note that $cexp \in cexp \rightrightarrows^*_{\mathsf{P}}$.

6.2.1 Properties of Composition of Experiments

In order to prove the more general result, we give a (P, Q)-simulation relating $\rightrightarrows$-derivatives of $cexp$ for any $cexp \in \mathcal{CE}(\mathsf{P})$. But before we derive two auxiliary Lemmas which will help us to prove the first two conditions of this simulation relation.

Lemma 6.2.1 Whenever

$$\langle cexpq,\ \mathcal{Q}\rangle \rightarrow_{\mathsf{Q}} \langle cexpq',\ \mathcal{Q}'\rangle$$

then, if there is some $cexp \in \mathcal{CE}(\mathsf{P})$ such that $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$, for any $P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q$ and $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$ we have

$$\langle cexpp,\ \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp',\ \mathcal{P}'\rangle,$$

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

$$cexpp' \in cexp' \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpq' \in cexp' \rightrightarrows^*_{\mathsf{Q}},$$

for some $\langle cexpp',\ \mathcal{P}'\rangle$ and $cexp' \in \mathcal{CE}(\mathsf{P})$.

Proof: The proof follows by transition induction on the inference of $\langle cexpq,\ \mathcal{Q}\rangle \rightarrow_{\mathsf{Q}} \langle cexpq',\ \mathcal{Q}'\rangle$. Let

$$\langle cexpq,\ \mathcal{Q}\rangle \rightarrow_{\mathsf{Q}} \langle cexpq',\ \mathcal{Q}'\rangle, \qquad cexp \in \mathcal{CE}(\mathsf{P}),$$
$$cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}, \qquad P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q, \text{ and}$$
$$cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}\ .$$

Trivially we have

$$cexp \in \mathcal{E}(\mathsf{P}) \text{ or } cexp \notin \mathcal{E}(\mathsf{P}).$$

We split the proof in two cases. First let $cexp \in \mathcal{E}(\mathsf{P})$. So, as $P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q$, we have $\mathcal{E}(\mathsf{P}) \subseteq \mathcal{E}(\mathsf{Q})$ and then

$$cexp \in \mathcal{E}(\mathsf{Q}).$$

Thus, as $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$ and $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$, we have

$$cexp \rightrightarrows^*_{\mathsf{Q}} cexpq \text{ and } cexp \rightrightarrows^*_{\mathsf{P}} cexpp.$$

So, by Lemma 6.1.3, for some $\langle cexpp',\ \mathcal{P}'\rangle$, we have

$$\langle cexpp,\ \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp',\ \mathcal{P}'\rangle,$$
$$\langle p,\ \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\ \mathcal{Q}'\rangle,$$
$$cexpp' =_{(\mathsf{P},\mathsf{Q})} cexpq' \quad \text{or} \quad cexp \rightrightarrows^*_{\mathsf{P}} cexpp' \text{ and } cexp \rightrightarrows^*_{\mathsf{Q}} cexpq'.$$

Thus we split the proof again in two cases. First let $cexp \rightrightarrows^*_{\mathsf{P}} cexpp'$ and $cexp \rightrightarrows^*_{\mathsf{Q}} cexpq'$. Therefore the result easily follows by taking $cexp'$ to be $cexp$. Second let $cexpp' =_{(\mathsf{P},\mathsf{Q})} cexpq'$. As $cexp \rightrightarrows^*_{\mathsf{P}} cexpp$, it is easy to check

$$\langle cexp,\ \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp,\ \mathcal{P}\rangle.$$

So, by transitivity, we have

$$\langle cexp,\ \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp',\ \mathcal{P}'\rangle.$$

Hence $cexpp'$ and $cexpq'$ are either the same object identifier or the same fully evaluated functional expression of a primary sort of P. Therefore the result easily follows by taking $cexp'$ to be $cexpp'$.

Now let $cexp \notin \mathcal{E}(\mathsf{P})$. So, as $cexp \in \mathcal{CE}(\mathsf{P})$, it is easy to check

$$cexp \notin \mathcal{E}(\mathsf{Q}).$$

Then we analyse the possible cases justifying the last step of the inference of $\langle cexpq,\ \mathcal{Q}\rangle \rightarrow_{\mathsf{Q}} \langle cexpq',\ \mathcal{Q}'\rangle$:

Case 1 By Rule 4.3.1, where $cexpq \in \mathcal{T}_{F(\mathsf{Q})}$ and $\neg(cexpq \equiv [\![cexpq]\!]_{FE(\mathsf{Q})})$. Hence, as $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$, we have that either $cexp \in \mathcal{E}(\mathsf{Q})$ or $\mathcal{FE}_{\mathsf{Q}}(cexpq)$. But this contradicts our assumptions. Therefore Rule 4.3.1 cannot be used to justify the transition.

Case 2 By one of the following rules: 4.3.2, 4.3.3, 4.3.4, 4.3.5, 4.5.1, 4.5.2, 4.5.3, or 4.5.4, where $cexpq \equiv op(\tilde{e})$, for a method, attribute, or object creation or deletion operation op. Hence, as $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$, we have $cexp \in \mathcal{E}(\mathsf{Q})$. Therefore the rules above cannot be used to justify the transition.

Case 3 By Rule 4.4.3, where $cexpq \equiv cexpq_1 \ || \ cexpq_2$, $cexpq' \equiv cexpq'_1 \ || \ cexpq_2$, and, by a shorter transition, $\langle cexpq_1,\ \mathcal{Q}\rangle \rightarrow_{\mathsf{Q}} \langle cexpq'_1,\ \mathcal{Q}'\rangle$. Hence, as $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$ and $cexp \notin \mathcal{E}(\mathsf{Q})$, we have

$$cexp \ \equiv \ cexp_1 \ || \ cexp_2,$$

for some $cexp_1, cexp_2 \in \mathcal{CE}(\mathsf{Q})$ such that

$$cexpq_1 \in cexp_1 \rightrightarrows^*_{\mathsf{Q}} \text{ and } cexpq_2 \in cexp_2 \rightrightarrows^*_{\mathsf{Q}} .$$

Then, as $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$, we have

$$cexpp \ \equiv \ cexpp_1 \ || \ cexpp_2,$$

for some $cexpp_1$ and $cexpp_2$ such that

$$cexpp_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpp_2 \in cexp_2 \rightrightarrows^*_{\mathsf{P}} .$$

Indeed, as $cexp \in \mathcal{CE}(\mathsf{P})$, we have

$$cexp_1, cexp_2 \in \mathcal{CE}(\mathsf{P}).$$

So, by induction, for some $\langle cexpp'_1,\ \mathcal{P}'\rangle$ and $cexp'_1 \in \mathcal{CE}(\mathsf{P})$, we have

$$\langle cexpp_1,\ \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp'_1,\ \mathcal{P}'\rangle,$$
$$\langle p,\ \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\ \mathcal{Q}'\rangle,$$
$$cexpp'_1 \in cexp'_1 \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpq'_1 \in cexp'_1 \rightrightarrows^*_{\mathsf{Q}} .$$

Then, by Rule 4.4.3, it is easy to check

$$\langle cexpp_1 \ || \ cexpp_2,\ \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp'_1 \ || \ cexpp_2,\ \mathcal{P}'\rangle.$$

Therefore the result easily follows by choosing $cexpp'$ to be $cexpp'_1 \ || \ cexpp_2$ and $cexp'$ to be $cexp'_1 \ || \ cexp_2$.

The proof considering Rule 4.4.4 follows in a similar way.

Case 4 By Rule 4.4.5, assuming that $cexpq \equiv cexpq_1 \ || \ cexpq_2$, $cexpq' \equiv cexpq_2$, $\mathcal{Q} \equiv \mathcal{Q}'$, and $\mathcal{FE}_{\mathsf{Q}}(cexpq_1)$. Exactly as in case 3, we can infer

$$cexp \ \equiv \ cexp_1 \ || \ cexp_2,$$
$$cexpq_1 \in cexp_1 \rightrightarrows^*_{\mathsf{Q}} \text{ and } cexpq_2 \in cexp_2 \rightrightarrows^*_{\mathsf{Q}},$$
$$cexpp \ \equiv \ cexpp_1 \ || \ cexpp_2,$$
$$cexpp_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpp_2 \in cexp_2 \rightrightarrows^*_{\mathsf{P}},$$

for $cexpp_1$, $cexpp_2$, and $cexp_1, cexp_2 \in \mathcal{CE}(\mathsf{P})$. Then, as $cexpq_1 \in cexp_1 \rightrightarrows^*_{\mathsf{Q}}$ and $\mathcal{FE}_{\mathsf{Q}}(cexpq_1)$, we have

$$cexp_1 \ \equiv \ cexpq_1 \text{ or } cexp_1 \in \mathcal{E}(\mathsf{Q}).$$

So we split the proof in two cases. First let $cexp_1 \equiv cexpq_1$. Then, as $cexp_1 \in \mathcal{CE}(\mathsf{P})$ and $\mathcal{FE}_{\mathsf{Q}}(cexp_1)$, it is easy to check

$$\mathcal{FE}_{\mathsf{P}}(cexp_1).$$

Hence, as $cexpp_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}}$, we have

$$cexpp_1 \equiv cexp_1.$$

Thus, by Rule 4.4.5, we have

$$\langle cexpp_1 \mid\mid cexpp_2, \mathcal{P}\rangle \rightarrow_{\mathsf{P}} \langle cexpp_2, \mathcal{P}\rangle.$$

Therefore the result easily follows by choosing $\langle cexpp', \mathcal{P}'\rangle$ to be $\langle cexpp_2, \mathcal{P}\rangle$ and $cexp'$ to be $cexp_2$.

Now let $cexp_1 \in \mathcal{E}(\mathsf{Q})$. Then, as $cexp_1 \in \mathcal{CE}(\mathsf{P})$, it is easy to check

$$cexp_1 \in \mathcal{E}(\mathsf{P}).$$

So, as $cexpp_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}}$ and $cexpq_1 \in cexp_1 \rightrightarrows^*_{\mathsf{Q}}$, we have

$$cexp_1 \rightrightarrows^*_{\mathsf{P}} cexpp_1 \text{ and } cexp_1 \rightrightarrows^*_{\mathsf{Q}} cexpq_1.$$

Also, as $\mathcal{FE}_{\mathsf{Q}}(cexpq_1)$, it is easy to check

$$\langle cexpq_1, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}} .$$

Hence, by Lemma 6.1.4, for some $\langle cexpp'_1, \mathcal{P}'\rangle$, we have

$$\begin{gathered}\langle cexpp_1, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp'_1, \mathcal{P}'\rangle \not\rightarrow,\\ \langle p, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}\rangle,\\ \mathcal{FE}_{\mathsf{P}}(cexpp'_1) \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(cexpq_1).\end{gathered}$$

So we have

$$\mathcal{FE}_{\mathsf{P}}(cexpp'_1).$$

Then, by Rules 4.4.3 and 4.4.5, we have

$$\langle cexpp_1 \mid\mid cexpp_2, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp'_1 \mid\mid cexpp_2, \mathcal{P}'\rangle \rightarrow_{\mathsf{P}} \langle cexpp_2, \mathcal{P}'\rangle.$$

Therefore the result easily follows by choosing $cexpp'$ to be $cexpp_2$ and $cexp'$ to be $cexp_2$.

The proof considering Rule 4.4.6 follows in a similar way.

Case 5 The proof follows in a similar way to case 3 when Rules 4.4.1, 4.4.15, or 4.4.12 are used for justification, whereas the proof follows as in case 4 when Rules 4.4.2, 4.4.16, or 4.4.17 are used. We omit the details here.

Case 6 By Rule 4.4.13, where $cexpq \equiv \texttt{result}\ cexpq_1\ ;\ cexpq_2$, $cexpq' \equiv \texttt{result}\ cexpq_1\ ;\ cexpq'_2$, $\mathcal{FE}_{\mathsf{Q}}(cexpq_1)$, and, by a shorter inference, $\langle cexpq_2,\, \mathcal{Q}\rangle \to_{\mathsf{Q}} \langle cexpq'_2,\, \mathcal{Q}'\rangle$. Hence, as $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$ and $cexp \notin \mathcal{E}(\mathsf{Q})$, we have

$$cexp \equiv \texttt{result}\ cexp_1\ ;\ cexp_2,$$

for some $cexp_1, cexp_2 \in \mathcal{CE}(\mathsf{Q})$ such that

$$cexpq_1 \in cexp_1 \rightrightarrows^*_{\mathsf{Q}} \text{ and } cexpq_2 \in cexp_2 \rightrightarrows^*_{\mathsf{Q}}.$$

Then, as $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$, we have

$$cexpp \equiv \texttt{result}\ cexpp_1\ ;\ cexpp_2,$$

for some $cexpp_1$ and $cexpp_2$ such that

$$cexpp_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpp_2 \in cexp_2 \rightrightarrows^*_{\mathsf{P}}.$$

Indeed, as $cexp \in \mathcal{CE}(\mathsf{P})$, we have

$$cexp_1, cexp_2 \in \mathcal{CE}(\mathsf{P}).$$

Also, as $cexpq_1 \in cexp_1 \rightrightarrows^*_{\mathsf{Q}}$ and $\mathcal{FE}_{\mathsf{Q}}(cexpq_1)$, we have

$$cexp_1 \equiv cexpq_1 \text{ or } cexp_1 \in \mathcal{E}(\mathsf{Q}).$$

So we split the proof in two cases. First let $cexp_1 \equiv cexpq_1$. Then $\mathcal{FE}_{\mathsf{Q}}(cexp_1)$ and, as $cexp_1 \in \mathcal{CE}(\mathsf{P})$, it is easy to check

$$\mathcal{FE}_{\mathsf{P}}(cexp_1).$$

Hence, as $cexpp_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}}$, we have

$$cexpp_1 \equiv cexp_1 \text{ and then } \mathcal{FE}_{\mathsf{P}}(cexpp_1).$$

Also, by induction, for some $\langle cexpp'_2,\, \mathcal{P}'\rangle$ and $cexp'_2 \in \mathcal{CE}(\mathsf{P})$, we have

$$\langle cexpp_2,\, \mathcal{P}\rangle \to^*_{\mathsf{P}} \langle cexpp'_2,\, \mathcal{P}'\rangle,$$
$$\langle p,\, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}'\rangle,$$
$$cexpp'_2 \in cexp'_2 \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpq'_2 \in cexp'_2 \rightrightarrows^*_{\mathsf{Q}}.$$

Then, as $\mathcal{FE}_{\mathsf{P}}(cexpp_1)$, by Rule 4.4.13, it is easy to check

$$\langle \texttt{result}\ cexpp_1\ ;\ cexpp_2,\, \mathcal{P}\rangle \to^*_{\mathsf{P}} \langle \texttt{result}\ cexpp_1\ ;\ cexpp'_2,\, \mathcal{P}'\rangle.$$

Therefore the result easily follows by choosing $cexpp'$ to be $\texttt{result}\ cexpp_1\ ;\ cexpp'_2$ and $cexp'$ to be $\texttt{result}\ cexp_1\ ;\ cexp'_2$.

Now let $cexp_1 \in \mathcal{E}(\mathsf{Q})$. Then, as $cexp_1 \in \mathcal{CE}(\mathsf{P})$, it is easy to check

$$cexp_1 \in \mathcal{E}(\mathsf{P}).$$

So, as $cexpp_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}}$ and $cexpq_1 \in cexp_1 \rightrightarrows^*_{\mathsf{Q}}$, we have

$$cexp_1 \rightrightarrows^*_{\mathsf{P}} cexpp_1 \text{ and } cexp_1 \rightrightarrows^*_{\mathsf{Q}} cexpq_1.$$

Also, as $\mathcal{FE}_{\mathsf{Q}}(cexpq_1)$, it is easy to check

$$\langle cexpq_1,\, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}} .$$

Hence, by Lemma 6.1.4, for some $\langle cexpp'_1,\, \mathcal{P}'\rangle$, we have

$$\langle cexpp_1,\, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp'_1,\, \mathcal{P}'\rangle \not\rightarrow,$$
$$\langle p,\, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle,$$
$$\mathcal{FE}_{\mathsf{P}}(cexpp'_1) \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(cexpq_1)$$
$$\mathcal{FE}_{\mathsf{P}}(cexpp'_1) \text{ implies } cexpp'_1 =_{(\mathsf{P},\mathsf{Q})} cexpq_1.$$

So we have

$$\mathcal{FE}_{\mathsf{P}}(cexpp'_1) \text{ and } cexpp'_1 =_{(\mathsf{P},\mathsf{Q})} cexpq_1.$$

By induction, for some $\langle cexpp'_2,\, \mathcal{P}''\rangle$ and $cexp'_2 \in \mathcal{CE}(\mathsf{P})$, we have

$$\langle cexpp_2,\, \mathcal{P}'\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp'_2,\, \mathcal{P}''\rangle,$$
$$\langle p,\, \mathcal{P}''\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}'\rangle,$$
$$cexpp'_2 \in cexp'_2 \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpq'_2 \in cexp'_2 \rightrightarrows^*_{\mathsf{Q}} .$$

Then, as $\mathcal{FE}_{\mathsf{P}}(cexpp'_1)$, by Rules 4.4.12 and 4.4.13, it is easy to check

$$\begin{aligned}&\langle \texttt{result}\ cexpp_1\ ;\ cexpp_2,\, \mathcal{P}\rangle\\ &\qquad \rightarrow^*_{\mathsf{P}} \langle \texttt{result}\ cexpp'_1\ ;\ cexpp_2,\, \mathcal{P}'\rangle\\ &\qquad \rightarrow^*_{\mathsf{P}} \langle \texttt{result}\ cexpp'_1\ ;\ cexpp'_2,\, \mathcal{P}''\rangle.\end{aligned}$$

Therefore the result easily follows by choosing $\langle cexpp',\, \mathcal{P}'\rangle$ to be $\langle \texttt{result}\ cexpp'_1\ ;\ cexpp'_2,\, \mathcal{P}''\rangle$ and $cexp'$ to be $\texttt{result}\ cexpp'_1\ ;\ cexp'_2$, since, as $cexp_1 \rightrightarrows^*_{\mathsf{P}} cexpp_1$, it is easy to check

$$\langle cexp_1,\, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp_1,\, \mathcal{P}\rangle,$$

and, by transitivity,

$$\langle cexp_1,\, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp'_1,\, \mathcal{P}'\rangle.$$

Hence $cexpp'_1$ and $cexpq_1$ are either the same object identifier or the same fully evaluated functional expression of a primary sort of P.

Case 7 If we consider Rule 4.4.14 for justification, the proof follows using the same kind of reasoning used in case 6.

Case 8 The justification using Rules 4.4.8, 4.4.9, 4.4.10, 4.4.11, 4.4.18, 4.4.19, and 4.4.20 is not appropriate here. In fact, as $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$, it is easy to see that using any of the rules above we can conclude that $cexp \in \mathcal{E}(\mathsf{Q})$, contradicting our initial assumption.

$\square$

Now we give the second lemma. Its proof follows essentially the same line of the previous proof; it can be found in Appendix A.

Lemma 6.2.2 Let $cexp \in \mathcal{CE}(\mathsf{P})$ and $P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q$. For any $cexpp$ and $cexpq$ such that $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$ and $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$, if

$$\langle cexpq,\ \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}}$$

then, for some $\langle cexpp',\ \mathcal{P}'\rangle$ and $cexp' \in \mathcal{CE}(\mathsf{P})$,

$$\langle cexpp,\ \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp',\ \mathcal{P}'\rangle \not\rightarrow_{\mathsf{P}},$$

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

$$\mathcal{FE}_{\mathsf{P}}(cexpp') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(cexpq),$$

$$\mathcal{FE}_{\mathsf{P}}(cexpp') \text{ implies } cexpp' =_{(\mathsf{P},\mathsf{Q})} cexpq,$$

$$cexpp' \in cexp' \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpq \in cexp' \rightrightarrows^*_{\mathsf{Q}}, \text{ and}$$

$$\mathcal{D} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q} \text{ implies } \langle cexpp',\ \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}},$$

for any $\mathcal{D}$. $\square$

Now we can finally introduce the simulation relating $\rightrightarrows$-derivatives of a composition of experiments $cexp$.

Lemma 6.2.3 Let $\mathcal{S}$ be the relation consisting of $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ and all pairs in the form

$$(\langle cexpp,\ \mathcal{P}\rangle,\ \langle cexpq,\ \mathcal{Q}\rangle),$$

where $\mathcal{P} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q}$ and there is a $cexp \in \mathcal{CE}(\mathsf{P})$ such that $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$ and $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$. Then $\mathcal{S}$ is a $(\mathsf{P},\ \mathsf{Q})$-simulation.

Proof: It follows directly from Theorem 5.1.4, Lemma 6.2.1, Lemma 6.2.2, and by checking that whenever $p =_{(\mathsf{P},\mathsf{Q})} q$ then $\langle p,\ \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\ \mathcal{Q}\rangle$, for any $\mathcal{P}$ and $\mathcal{Q}$ such that $\mathcal{P} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q}$ (see the proof of Lemma 6.1.5). $\square$

This allows us to prove that any $\rightrightarrows_{\mathsf{P}}$-derivative of a composition of experiments $cexp$ is simulated by any $\rightrightarrows_{\mathsf{Q}}$-derivative of $cexp$ whenever there is a $(\mathsf{P},\ \mathsf{Q})$-simulation relating initial states.

Theorem 6.2.4 Let $cexp \in \mathcal{CE}(\mathsf{P})$, $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$, and $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$. Thus, if $\emptyset_{\mathsf{P}} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \emptyset_{\mathsf{Q}}$ then $cexpp \sqsubseteq_{(\mathsf{P},\mathsf{Q})} cexpq$.

Proof: Let us assume that

$$\emptyset_{\mathsf{P}} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \emptyset_{\mathsf{Q}}.$$

Then, by Lemma 6.2.3, there is a $(\mathsf{P},\ \mathsf{Q})$-simulation $\mathcal{S}$ such that

$$(\langle cexpp,\ \emptyset_{\mathsf{P}}\rangle,\ \langle cexpq,\ \emptyset_{\mathsf{Q}}\rangle) \in \mathcal{S}.$$

Hence

$$\langle cexpp,\ \emptyset_{\mathsf{P}}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle cexpq,\ \emptyset_{\mathsf{Q}}\rangle.$$

Therefore $cexpp \sqsubseteq_{(\mathsf{P},\mathsf{Q})} cexpq$. $\square$

Then we can easily derive another promised result: $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ includes the identity on composition of experiments whenever there is a $(\mathsf{P},\, \mathsf{Q})$-simulation relating initial states.

Corollary 6.2.5 Let $cexp \in \mathcal{CE}(\mathsf{P})$. Thus, if $\emptyset_\mathsf{P} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \emptyset_\mathsf{Q}$ then $cexp \sqsubseteq_{(\mathsf{P},\mathsf{Q})} cexp$.

Proof: Directly from Theorem 6.2.4, since it is easy to check that $cexp \in cexp \rightrightarrows^*_\mathsf{P}$ and $cexp \in cexp \rightrightarrows^*_\mathsf{Q}$. $\square$

6.3 Congruence Property of Refinement

Using the results introduced so far, it becomes easy to derive the expected congruence result. Essentially, we verify that refinement is preserved by several predefined method combiners. This is realized by presenting specific simulation relations.

6.3.1 Parallel Composition

First let us consider parallel composition.

Lemma 6.3.1 Let $\mathcal{S}$ be the relation consisting of $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ and all pairs in the form

$$(\langle p \;||\; cexpp,\, \mathcal{P}\rangle,\, \langle q \;||\; cexpq,\, \mathcal{Q}\rangle),$$

where $\langle p,\, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle$ and there is a $cexp \in \mathcal{CE}(\mathsf{P})$ such that $cexpp \in cexp \rightrightarrows^*_\mathsf{P}$ and $cexpq \in cexp \rightrightarrows^*_\mathsf{Q}$. Then $\mathcal{S}$ is a $(\mathsf{P},\, \mathsf{Q})$-simulation.

Proof: As $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ is included in $\mathcal{S}$, using Lemma 5.1.6 we can easily conclude that pairs related by $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ satisfy the simulation conditions (see Definition 5.1.1). Let us then concentrate on pairs in the form

$$(\langle p \;||\; cexpp,\, \mathcal{P}\rangle,\, \langle q \;||\; cexpq,\, \mathcal{Q}\rangle).$$

First let $\langle q \;||\; cexpq,\, \mathcal{Q}\rangle \rightarrow_\mathsf{Q} \langle e,\, \mathcal{Q}'\rangle$. Thus we consider the different cases justifying this transition.

Case 1 By Rule 4.4.4, where $e \equiv q \;||\; cexpq'$ and $\langle cexpq,\, \mathcal{Q}\rangle \rightarrow_\mathsf{Q} \langle cexpq',\, \mathcal{Q}'\rangle$. Then, for some $\langle cexpp',\, \mathcal{P}'\rangle$ and $cexp' \in \mathcal{CE}(\mathsf{P})$, by Lemma 6.2.1 we have

$$\langle cexpp,\, \mathcal{P}\rangle \rightarrow^*_\mathsf{P} \langle cexpp',\, \mathcal{P}'\rangle,$$

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

$$cexpp' \in cexp' \rightrightarrows^*_\mathsf{P} \text{ and } cexpq' \in cexp' \rightrightarrows^*_\mathsf{Q}\ .$$

Hence, by Rule 4.4.4, it is easy to check

$$\langle p \;||\; cexpp,\, \mathcal{P}\rangle \rightarrow^*_\mathsf{P} \langle p \;||\; cexpp',\, \mathcal{P}'\rangle.$$

Therefore the result follows since $\langle p \;||\; cexpp',\, \mathcal{P}'\rangle$ is clearly related by $\mathcal{S}$ to $\langle q \;||\; cexpq',\, \mathcal{Q}'\rangle$.

Case 2 By Rule 4.4.3, where $e \equiv q' \;||\; cexpq$ and $\langle q, \mathcal{Q}\rangle \rightarrow_{\mathsf{Q}} \langle q', \mathcal{Q}'\rangle$. Then, by Lemma 5.1.6, for some $\langle p', \mathcal{P}'\rangle$, we have

$$\langle p, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle p', \mathcal{P}'\rangle \text{ and } \langle p', \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q', \mathcal{Q}'\rangle.$$

Hence, by Rule 4.4.3, it is easy to check

$$\langle p \;||\; cexpp, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle p' \;||\; cexpp, \mathcal{P}'\rangle.$$

Therefore the result follows since $\langle p' \;||\; cexpp, \mathcal{P}'\rangle$ is clearly related by $\mathcal{S}$ to $\langle q' \;||\; cexpq, \mathcal{Q}'\rangle$.

Case 3 By Rule 4.4.6, where $e \equiv q$, $\mathcal{Q}' \equiv \mathcal{Q}$, and $\mathcal{FE}_{\mathsf{Q}}(cexpq)$. Hence

$$\langle cexpq, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}} .$$

Then, by Lemma 6.2.2, for some $\langle cexpp', \mathcal{P}'\rangle$,

$$\langle cexpp, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp', \mathcal{P}'\rangle \not\rightarrow_{\mathsf{P}},$$
$$\langle p, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}\rangle, \text{ and}$$
$$\mathcal{FE}_{\mathsf{P}}(cexpp') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(cexpq).$$

So we have

$$\mathcal{FE}_{\mathsf{P}}(cexpp').$$

Then, by Rules 4.4.4 and 4.4.6, we have

$$\langle p \;||\; cexpp, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle p \;||\; cexpp', \mathcal{P}'\rangle \rightarrow_{\mathsf{P}} \langle p, \mathcal{P}'\rangle.$$

Therefore the result easily follows since $\langle p, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}\rangle$ and $\mathcal{S}$ includes $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$.

Case 4 By Rule 4.4.5, where $e \equiv cexpq$, $\mathcal{Q}' \equiv \mathcal{Q}$, and $\mathcal{FE}_{\mathsf{Q}}(q)$. Hence

$$\langle q, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}} .$$

Then, by Lemma 5.1.6, for some $\langle p', \mathcal{P}'\rangle$,

$$\langle p, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle p', \mathcal{P}'\rangle \not\rightarrow_{\mathsf{P}},$$
$$\langle p', \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}\rangle, \text{ and}$$
$$\mathcal{FE}_{\mathsf{P}}(p') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(q).$$

So we have

$$\mathcal{FE}_{\mathsf{P}}(p').$$

Then, by Rules 4.4.3 and 4.4.5, we have

$$\langle p \;||\; cexpp, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle p' \;||\; cexpp, \mathcal{P}'\rangle \rightarrow_{\mathsf{P}} \langle cexpp, \mathcal{P}'\rangle.$$

Therefore the result easily follows since $\mathcal{S}$ includes $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ and, by Lemma 6.2.3, it is easy to check that $\langle cexpp, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle cexpq, \mathcal{Q}\rangle$.

This concludes the case analysis. Now let $\langle q \;||\; cexpq,\, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}}$. Hence, by Rules 4.4.3, 4.4.4, 4.4.5, and 4.4.6, we have

$$\langle q,\, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}}, \qquad \langle cexpq,\, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}},$$
$$\neg\mathcal{FE}_{\mathsf{Q}}(q), \qquad \text{and } \neg\mathcal{FE}_{\mathsf{Q}}(cexpq).$$

Then, by Lemma 6.2.2, for some $\langle cexpp',\, \mathcal{P}'\rangle$ and $cexp' \in \mathcal{CE}(\mathsf{P})$, we have

$$\langle cexpp,\, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp',\, \mathcal{P}'\rangle \not\rightarrow_{\mathsf{P}},$$
$$\langle p,\, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle,$$
$$\mathcal{FE}_{\mathsf{P}}(cexpp') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(cexpq),$$
$$cexpp' \in cexp' \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpq \in cexp' \rightrightarrows^*_{\mathsf{Q}}, \text{ and}$$
$$\mathcal{D} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q} \text{ implies } \langle cexpp',\, \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}},$$

for any $\mathcal{D}$. So we have

$$\neg\mathcal{FE}_{\mathsf{P}}(cexpp').$$

Also, as $\langle p,\, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle$, by Lemma 5.1.6, for some $\langle p',\, \mathcal{P}''\rangle$, we have

$$\langle p,\, \mathcal{P}'\rangle \rightarrow^*_{\mathsf{P}} \langle p',\, \mathcal{P}''\rangle \not\rightarrow_{\mathsf{P}},$$
$$\langle p',\, \mathcal{P}''\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle, \text{ and}$$
$$\mathcal{FE}_{\mathsf{P}}(p') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(q).$$

So we have

$$\neg\mathcal{FE}_{\mathsf{P}}(p').$$

Also, as $\langle p',\, \mathcal{P}''\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle$, we have

$$\langle cexpp',\, \mathcal{P}''\rangle \not\rightarrow_{\mathsf{P}}.$$

Then, by Rules 4.4.3, 4.4.4, 4.4.5, and 4.4.6, we have

$$\langle p \;||\; cexpp,\, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle p \;||\; cexpp',\, \mathcal{P}'\rangle \rightarrow^*_{\mathsf{P}} \langle p' \;||\; cexpp',\, \mathcal{P}''\rangle \not\rightarrow_{\mathsf{P}}.$$

Therefore the result easily follows since $p' \;||\; cexpp'$ and $q \;||\; cexpq$ are clearly not fully evaluated, and $\langle p' \;||\; cexpp',\, \mathcal{P}''\rangle$ is related by $\mathcal{S}$ to $\langle q \;||\; cexpq,\, \mathcal{Q}\rangle$.

Finally let $\langle exp,\, \mathcal{Q}\rangle \rightarrow^*_{\mathsf{Q}} Q' \not\rightarrow_{\mathsf{Q}}$, for any $exp \in \mathcal{E}(\mathsf{P})$. Hence, as $\langle p,\, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle$, by Lemma 5.1.6, for some P', we have

$$\langle exp,\, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} P' \not\rightarrow_{\mathsf{P}},$$
$$P' \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q', \text{ and}$$
$$\langle p,\, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}'\rangle.$$

Therefore the result easily follows since $\mathcal{S}$ includes $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$, and it is easy to check that

$$\langle p \;||\; cexpp,\, \mathcal{P}'\rangle$$

is related by $\mathcal{S}$ to $\langle q \;||\; cexpq,\, \mathcal{Q}'\rangle$. $\square$

We can then verify that the parallel composition method combiner preserves refinement.

Theorem 6.3.2 Let $cexp \in \mathcal{CE}(\mathsf{P})$, $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$, and $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$. If

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

then

$$p \mid\mid cexpp \sqsubseteq_{(\mathsf{P},\mathsf{Q})} q \mid\mid cexpq.$$

Proof: Let $p \sqsubseteq_{(\mathsf{P},\mathsf{Q})} q$. Hence

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

Then, by Lemma 6.3.1, there is a (P, Q)-simulation $\mathcal{S}$ such that

$$(\langle p \mid\mid cexpp, \emptyset_{\mathsf{P}}\rangle, \langle q \mid\mid cexpq, \emptyset_{\mathsf{Q}}\rangle) \in \mathcal{S}.$$

So we have

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

Therefore $p \mid\mid cexpp \sqsubseteq_{(\mathsf{P},\mathsf{Q})} q \mid\mid cexpq$. □

Finally we obtain the promised result.

Corollary 6.3.3 Let $cexp \in \mathcal{CE}(\mathsf{P})$. If

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

then

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

Proof: Directly by Theorem 6.3.2, since it is easy to check that $cexp \in cexp \rightrightarrows^*_{\mathsf{P}}$ and $cexp \in cexp \rightrightarrows^*_{\mathsf{Q}}$. □

The following results for parallel composition can also be derived in a very similar way to Lemma 6.3.1, Theorem 6.3.2, and Corollary 6.3.3. We omit the proofs here.

Lemma 6.3.4 Let $\mathcal{S}$ be the relation consisting of $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ and all pairs in the form

$$(\langle cexpp \mid\mid p, \mathcal{P}\rangle, \langle cexpq \mid\mid q, \mathcal{Q}\rangle),$$

where $\langle p, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}\rangle$ and there is a $cexp \in \mathcal{CE}(\mathsf{P})$ such that $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$ and $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$. Then $\mathcal{S}$ is a (P, Q)-simulation. □

Theorem 6.3.5 Let $cexp \in \mathcal{CE}(\mathsf{P})$, $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$, and $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$. If

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

then

$$cexpp \mid\mid p \sqsubseteq_{(\mathsf{P},\mathsf{Q})} cexpq \mid\mid q.$$

□

Corollary 6.3.6 Let $cexp \in \mathcal{CE}(\mathsf{P})$. If

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

then

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

□

6.3.2 Sequential Composition

Compositional results can also be easily derived for the sequential composition operator. We omit the proofs here since they are very similar to the ones for parallel composition.

Lemma 6.3.7 Let $\mathcal{S}$ be the relation consisting of $\sqsubseteq_{(\mathsf{P},\,\mathsf{Q})}$ and all pairs in the forms

$$\begin{array}{lcl} (\langle cexpp\ ;\ p,\, \mathcal{P}\rangle & , & \langle cexpq\ ;\ q,\, \mathcal{Q}\rangle), \text{ and} \\ (\langle p\ ;\ cexpp,\, \mathcal{P}\rangle & , & \langle q\ ;\ cexpq,\, \mathcal{Q}\rangle), \end{array}$$

where $\langle p,\, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle$ and there is a $cexp \in \mathcal{CE}(\mathsf{P})$ such that $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$ and $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$. Then $\mathcal{S}$ is a $(\mathsf{P},\,\mathsf{Q})$-simulation. □

Theorem 6.3.8 Let $cexp \in \mathcal{CE}(\mathsf{P})$, $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$, and $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$. If

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

then

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

and

$$p\ ;\ cexpp \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q\ ;\ cexpq.$$

□

Corollary 6.3.9 Let $cexp \in \mathcal{CE}(\mathsf{P})$. If

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

then

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

and

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

□

6.3.3 Conditional

Deriving compositionality results for the conditional method combiner is as easy as deriving compositionality results for parallel composition. In fact, the proofs are quite similar; we omit them here.

Lemma 6.3.10 Let $\mathcal{S}$ be the relation consisting of $\sqsubseteq_{(\mathsf{P},\,\mathsf{Q})}$ and all pairs in the forms

$$\begin{array}{lcl} (\langle \mathtt{if}\ p\ \mathtt{then}\ cexpp_1\ \mathtt{else}\ cexpp_2\ \mathtt{fi},\, \mathcal{P}\rangle & , & \langle \mathtt{if}\ q\ \mathtt{then}\ cexpq_1\ \mathtt{else}\ cexpq_2\ \mathtt{fi},\, \mathcal{Q}\rangle), \\ (\langle \mathtt{if}\ cexpp_1\ \mathtt{then}\ p\ \mathtt{else}\ cexpp_2\ \mathtt{fi},\, \mathcal{P}\rangle & , & \langle \mathtt{if}\ cexpq_1\ \mathtt{then}\ q\ \mathtt{else}\ cexpq_2\ \mathtt{fi},\, \mathcal{Q}\rangle), \text{ and} \\ (\langle \mathtt{if}\ cexpp_1\ \mathtt{then}\ cexpp_2\ \mathtt{else}\ p\ \mathtt{fi},\, \mathcal{P}\rangle & , & \langle \mathtt{if}\ cexpq_1\ \mathtt{then}\ cexpq_2\ \mathtt{else}\ q\ \mathtt{fi},\, \mathcal{Q}\rangle), \end{array}$$

where $\langle p,\, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle$ and there are $cexp_1, cexp_2 \in \mathcal{CE}(\mathsf{P})$ such that $cexpp_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}}$, $cexpq_1 \in cexp_1 \rightrightarrows^*_{\mathsf{Q}}$, $cexpp_2 \in cexp_2 \rightrightarrows^*_{\mathsf{P}}$, and $cexpq_2 \in cexp_2 \rightrightarrows^*_{\mathsf{Q}}$. Then $\mathcal{S}$ is a $(\mathsf{P},\,\mathsf{Q})$-simulation. □

Theorem 6.3.11 Let $cexp_1, cexp_2 \in \mathcal{CE}(\mathsf{P})$, $cexpp_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}}$, $cexpq_1 \in cexp_1 \rightrightarrows^*_{\mathsf{Q}}$, $cexpp_2 \in cexp_2 \rightrightarrows^*_{\mathsf{P}}$, and $cexpq_2 \in cexp_2 \rightrightarrows^*_{\mathsf{Q}}$. If

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

then

$$\texttt{if } p \texttt{ then } cexpp_1 \texttt{ else } cexpp_2 \texttt{ fi} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \texttt{if } q \texttt{ then } cexpq_1 \texttt{ else } cexpq_2 \texttt{ fi},$$

$$\texttt{if } cexpp_1 \texttt{ then } p \texttt{ else } cexpp_2 \texttt{ fi} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \texttt{if } cexpq_1 \texttt{ then } q \texttt{ else } cexpq_2 \texttt{ fi},$$

and

$$\texttt{if } cexpp_1 \texttt{ then } cexpp_2 \texttt{ else } p \texttt{ fi} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \texttt{if } cexpq_1 \texttt{ then } cexpq_2 \texttt{ else } q \texttt{ fi}.$$

□

Corollary 6.3.12 Let $cexp_1, cexp_2 \in \mathcal{CE}(\mathsf{P})$. If

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

then

$$\texttt{if } p \texttt{ then } cexp_1 \texttt{ else } cexp_2 \texttt{ fi} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \texttt{if } q \texttt{ then } cexp_1 \texttt{ else } cexp_2 \texttt{ fi},$$

$$\texttt{if } cexp_1 \texttt{ then } p \texttt{ else } cexp_2 \texttt{ fi} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \texttt{if } cexp_1 \texttt{ then } q \texttt{ else } cexp_2 \texttt{ fi},$$

and

$$\texttt{if } cexp_1 \texttt{ then } cexp_2 \texttt{ else } p \texttt{ fi} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \texttt{if } cexp_1 \texttt{ then } cexp_2 \texttt{ else } q \texttt{ fi}.$$

□

6.3.4 Result

Now we prove that the result method combiner preserves refinement. The proofs follow using the same kind of reasoning used to verify that the parallel composition combiner preserves refinement. So first we introduce a simulation.

Lemma 6.3.13 Let $\mathcal{S}$ be the relation consisting of $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ and all pairs in the forms

$$(\langle \texttt{result } cexpp \texttt{ ; } p, \mathcal{P}\rangle \quad , \quad \langle \texttt{result } cexpq \texttt{ ; } q, \mathcal{Q}\rangle), \text{ and}$$
$$(\langle \texttt{result } p \texttt{ ; } cexpp, \mathcal{P}\rangle \quad , \quad \langle \texttt{result } q \texttt{ ; } cexpq, \mathcal{Q}\rangle),$$

where $\langle p, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}\rangle$ and there is a $cexp \in \mathcal{CE}(\mathsf{P})$ such that $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$ and $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$. Then $\mathcal{S}$ is a (P, Q)-simulation.

Proof: The proof follows essentially the same lines of the proof of Lemma 6.3.1; it is deferred to Appendix A. □

Then it becomes easy to derive a general compositionality result.

Theorem 6.3.14 Let $cexp \in \mathcal{CE}(\mathsf{P})$, $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$, and $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$. If

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

then

$$\texttt{result } cexpp \texttt{ ; } p \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \texttt{result } cexpq \texttt{ ; } q$$

and

$$\texttt{result}\ p\ \texttt{;}\ cexpp \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \texttt{result}\ q\ \texttt{;}\ cexpq.$$

Proof: It easily follows from Lemma 6.3.13 (see the proof of Theorem 6.3.2). □

Finally we establish the expected compositionality result.

Corollary 6.3.15 Let $cexp \in \mathcal{CE}(\mathsf{P})$. If

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

then

$$\texttt{result}\ cexp\ \texttt{;}\ p \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \texttt{result}\ cexp\ \texttt{;}\ q$$

and

$$\texttt{result}\ p\ \texttt{;}\ cexp \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \texttt{result}\ q\ \texttt{;}\ cexp.$$

Proof: Directly by Theorem 6.3.14, since it is easy to check that $cexp \in cexp \rightrightarrows^*_{\mathsf{P}}$ and $cexp \in cexp \rightrightarrows^*_{\mathsf{Q}}$. □

6.3.5 Atomic Evaluation

Contrasting with the other combiners, the atomic evaluation operator is unary. So the congruence property is expressed in the following way:

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

implies

$$\texttt{[}\,p\,\texttt{]} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \texttt{[}\,q\,\texttt{]}.$$

Note that, by Corollary 6.2.5, this property immediately implies that

$$\texttt{[}\,cexp\,\texttt{]} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \texttt{[}\,cexp\,\texttt{]},$$

for any $cexp \in \mathcal{CE}(\mathsf{P})$, whenever $\emptyset_{\mathsf{P}} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \emptyset_{\mathsf{Q}}$.

Indeed, the congruence property stated above does not hold because the behaviour of the atomic evaluation operator depends on whether its argument always terminates and whether it can be fully evaluated, whereas those aspects are abstracted by our definition of refinement. In fact, an expression $\texttt{[}\,q\,\texttt{]}$ only blocks (i.e., its evaluation does not lead to any state and the expression is not fully evaluated) in a state $\mathcal{D}$ if the evaluation of q in this state always terminates and never yields a fully evaluated expression. On the other hand, it is easy to check that a blocking state $\langle q, \mathcal{D}\rangle$ may be a refinement of a state $\langle p, \mathcal{D}\rangle$ that might either lead to $\langle q, \mathcal{D}\rangle$, yield a result, or diverge. In this case, $\langle \texttt{[}\,q\,\texttt{]}, \mathcal{D}\rangle$ is not a refinement of $\langle \texttt{[}\,p\,\texttt{]}, \mathcal{D}\rangle$, since the first configuration blocks whereas the second does not.

Naturally, a notion of refinement considering the aspects discussed above would be a congruence with respect to the atomic evaluation operator. In fact, such a more concrete notion of refinement can be obtained by extending our definition of refinement in a very simple and natural way. Using our definition of refinement, we can only establish a weaker compositionality result. Indeed, refinement is only preserved by the atomic evaluation operator if the arguments of this operator satisfy some conditions, corresponding to the aspects discussed above. This is formalized by the following simulation, where we write $\mathcal{NFE}_{\mathsf{P}}(P)$ to indicate that P never fully evaluates; formally, $\mathcal{NFE}_{\mathsf{P}}(P)$ if

$$\text{whenever } P \rightarrow^*_{\mathsf{P}} P' \not\rightarrow_{\mathsf{P}} \text{ then } \neg\mathcal{FE}_{\mathsf{P}}(P').$$

Also, we write $\downarrow_{\mathsf{P}} P$ to indicate that P is terminating.

Lemma 6.3.16 Let $\mathcal{S}$ be the relation consisting of $\sqsubseteq_{(\mathsf{P},\,\mathsf{Q})}$ and all pairs in the form

$$(\langle \texttt{[} \, p \, \texttt{]},\, \mathcal{P}\rangle,\, \langle \texttt{[} \, q \, \texttt{]},\, \mathcal{Q}\rangle),$$

where $\langle p,\, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle$ and, for any $\mathcal{D}$ and $\mathcal{D}'$ such that $\mathcal{D} \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \mathcal{D}'$,

- $\downarrow_{\mathsf{Q}} \langle q,\, \mathcal{D}'\rangle$ implies $\downarrow_{\mathsf{P}} \langle p,\, \mathcal{D}\rangle$, and
- $\mathcal{NFE}_{\mathsf{Q}}(\langle q,\, \mathcal{D}'\rangle)$ implies $\mathcal{NFE}_{\mathsf{P}}(\langle p,\, \mathcal{D}\rangle)$.

Then $\mathcal{S}$ is a $(\mathsf{P},\, \mathsf{Q})$-simulation.

Proof: As $\sqsubseteq_{(\mathsf{P},\,\mathsf{Q})}$ is included in $\mathcal{S}$, using Lemma 5.1.6 we can easily conclude that pairs related by $\sqsubseteq_{(\mathsf{P},\,\mathsf{Q})}$ satisfy the simulation conditions. Let us then concentrate on pairs in the form

$$(\langle \texttt{[} \, p \, \texttt{]},\, \mathcal{P}\rangle,\, \langle \texttt{[} \, q \, \texttt{]},\, \mathcal{Q}\rangle).$$

First let $\langle \texttt{[} \, q \, \texttt{]},\, \mathcal{Q}\rangle \rightarrow_{\mathsf{Q}} \langle q',\, \mathcal{Q}'\rangle$. Thus we consider the different cases justifying this transition.

Case 1 By Rule 4.4.18, where $\langle q,\, \mathcal{Q}\rangle \rightarrow^*_{\mathsf{Q}} \langle q',\, \mathcal{Q}'\rangle$ and $\mathcal{FE}_{\mathsf{Q}}(q')$. Hence, by Lemma 5.1.6, for some $\langle p',\, \mathcal{P}'\rangle$, we have

$$\langle p,\, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle p',\, \mathcal{P}'\rangle \text{ and } \langle p',\, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \langle q',\, \mathcal{Q}'\rangle.$$

As $\mathcal{FE}_{\mathsf{Q}}(q')$, we have $\langle q',\, \mathcal{Q}'\rangle \not\rightarrow_{\mathsf{Q}}$. Then, by Lemma 5.1.6 again, for some $\langle p'',\, \mathcal{P}''\rangle$, we have

$$\langle p',\, \mathcal{P}'\rangle \rightarrow^*_{\mathsf{P}} \langle p'',\, \mathcal{P}''\rangle \not\rightarrow_{\mathsf{P}},$$
$$\langle p'',\, \mathcal{P}''\rangle \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \langle q',\, \mathcal{Q}'\rangle, \text{ and}$$
$$\mathcal{FE}_{\mathsf{P}}(p'') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(q').$$

So we have

$$\mathcal{FE}_{\mathsf{P}}(p'').$$

Also, by transitivity, we have

$$\langle p,\, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle p'',\, \mathcal{P}''\rangle.$$

Thus, by Rule 4.4.18, we have

$$\langle \texttt{[} \, p \, \texttt{]},\, \mathcal{P}\rangle \rightarrow_{\mathsf{P}} \langle p'',\, \mathcal{P}''\rangle.$$

Therefore the result easily follows since $\mathcal{S}$ includes $\sqsubseteq_{(\mathsf{P},\,\mathsf{Q})}$.

Case 2 By Rule 4.4.19, where $q' \equiv \texttt{[} \, q \, \texttt{]}$ and $\mathcal{Q} \equiv \mathcal{Q}'$. Hence, the result trivially follows since $\langle \texttt{[} \, p \, \texttt{]},\, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle \texttt{[} \, p \, \texttt{]},\, \mathcal{P}\rangle$.

This concludes the case analysis. Now let $\langle \texttt{[}\, q\, \texttt{]},\, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}}$. Hence by Rules 4.4.18 and 4.4.19, we have

$$\mathcal{NFE}_{\mathsf{Q}}(\langle q,\, \mathcal{Q}\rangle) \text{ and } \downarrow_{\mathsf{Q}} \langle q,\, \mathcal{Q}\rangle.$$

So, as $\langle p,\, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle$, we have

$$\mathcal{NFE}_{\mathsf{P}}(\langle p,\, \mathcal{P}\rangle) \text{ and } \downarrow_{\mathsf{P}} \langle p,\, \mathcal{P}\rangle.$$

Then, by Rules 4.4.18 and 4.4.19, we have

$$\langle \texttt{[}\, p\, \texttt{]},\, \mathcal{P}\rangle \not\rightarrow_{\mathsf{P}} .$$

Therefore the result easily follows since $\rightarrow_{\mathsf{P}}^{*}$ is reflexive and $\texttt{[}\, p\, \texttt{]}$ and $\texttt{[}\, q\, \texttt{]}$ are clearly not fully evaluated.

Finally let $\langle exp,\, \mathcal{Q}\rangle \rightarrow_{\mathsf{Q}}^{*} Q' \not\rightarrow_{\mathsf{Q}}$, for any $exp \in \mathcal{E}(\mathsf{P})$. Hence, as $\langle p,\, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle$, by Lemma 5.1.6, for some P', we have

$$\langle exp,\, \mathcal{P}\rangle \rightarrow_{\mathsf{P}}^{*} P' \not\rightarrow_{\mathsf{P}},$$

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

$$\langle p,\, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}'\rangle.$$

Therefore the result easily follows since $\mathcal{S}$ includes $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$, and it is easy to check that $\langle \texttt{[}\, p\, \texttt{]},\, \mathcal{P}'\rangle$ is related by $\mathcal{S}$ to $\langle \texttt{[}\, q\, \texttt{]},\, \mathcal{Q}'\rangle$. $\square$

Now we can easily establish the weaker compositionality result for the atomic evaluation operator.

Theorem 6.3.17 For any $\mathcal{D}$ and $\mathcal{D}'$ such that $\mathcal{D} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{D}'$, if

$$\mathcal{NFE}_{\mathsf{Q}}(\langle q,\, \mathcal{D}'\rangle) \text{ implies } \mathcal{NFE}_{\mathsf{P}}(\langle p,\, \mathcal{D}\rangle),$$

and $\downarrow_{\mathsf{Q}} \langle q,\, \mathcal{D}'\rangle$ implies $\downarrow_{\mathsf{P}} \langle p,\, \mathcal{D}\rangle$ then

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

implies

$$\texttt{[}\, p\, \texttt{]} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \texttt{[}\, q\, \texttt{]}.$$

Proof: Let us assume that

$$\mathcal{NFE}_{\mathsf{Q}}(\langle q,\, \mathcal{D}'\rangle) \text{ implies } \mathcal{NFE}_{\mathsf{P}}(\langle p,\, \mathcal{D}\rangle), \text{ and}$$

$$\downarrow_{\mathsf{Q}} \langle q,\, \mathcal{D}'\rangle \text{ implies } \downarrow_{\mathsf{P}} \langle p,\, \mathcal{D}\rangle,$$

for any $\mathcal{D}$ and $\mathcal{D}'$ such that $\mathcal{D} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{D}'$. Now let $p \sqsubseteq_{(\mathsf{P},\mathsf{Q})} q$. Hence

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

Then, by Lemma 6.3.16, there is a $(\mathsf{P},\, \mathsf{Q})$-simulation $\mathcal{S}$ such that

$$(\langle \texttt{[}\, p\, \texttt{]},\, \emptyset_{\mathsf{P}}\rangle,\, \langle \texttt{[}\, q\, \texttt{]},\, \emptyset_{\mathsf{Q}}\rangle) \in \mathcal{S}.$$

So we have

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

Therefore $\texttt{[}\, p\, \texttt{]} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \texttt{[}\, q\, \texttt{]}$. $\square$

6.3.6 Nondeterministic Choice

The notion of refinement discussed here abstracts from the power of expressions of making external choices; that is, an expression that cannot make an external choice (it always blocks) may be refined by an expression that can always make choices (it never blocks). Indeed, a blocked state P may be refined by a state Q that eventually reaches P, since by the definition of simulation $\rightarrow$-transitions from Q may be matched by zero $\rightarrow$-transitions from P.

This suggests that our notion of refinement is not preserved by external choice. In fact, it is not. For example, let p and q be expressions such that $p \sqsubseteq q$ and q can make choices whereas p cannot. Also, assume that *cexp* is a non fully evaluated composition of experiments. So p `[]` *cexp* can only lead to an expression *cexp*$'$ derived from *cexp*, whereas q `[]` *cexp* can either lead to *cexp*$'$ or to an expression q' derived from q. Therefore, as there is a transition from q `[]` *cexp* that cannot be matched by p `[]` *cexp*, we must conclude that the first expression is not a refinement of the second.

A notion of refinement that does not abstract from the power of expressions of making external choices can be easily derived from our notion of refinement, by essentially requiring initial transitions from an expression to be matched by at least one transition (instead of zero or many) from a related expression. Here we shall be content to show that our notion of refinement is preserved by the *internal* choice method combiner, which is defined by the axiom

```
ax P Or Q = (skip ; P) [] (skip ; Q) .
```

where `P` and `Q` are variables and `skip` is any functional constant. This operator preserves our notion of refinement because its behaviour does not depend on the behaviour of its arguments. This is a quite satisfactory result since internal choice seems to be more useful than external choice in the context of FOOPS (see Section 4.4.3). The following simulation is the first step towards deriving this result. Hereafter, we assume that any specification has the axiom above (and no other) defining the operator `_Or_`.

Lemma 6.3.18 Let $\mathcal{S}$ be the relation consisting of $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ and all pairs in the forms

$$\begin{array}{rcl} (\langle p \texttt{ Or } \mathit{cexpp},\, \mathcal{P}\rangle & , & \langle q \texttt{ Or } \mathit{cexpq},\, \mathcal{Q}\rangle), \\ (\langle \texttt{skip ; } p \texttt{ [] skip ; } \mathit{cexpp},\, \mathcal{P}\rangle & , & \langle \texttt{skip ; } q \texttt{ [] skip ; } \mathit{cexpq},\, \mathcal{Q}\rangle), \\ (\langle \mathit{cexpp} \texttt{ Or } p,\, \mathcal{P}\rangle & , & \langle \mathit{cexpq} \texttt{ Or } q,\, \mathcal{Q}\rangle), \text{ and} \\ (\langle \texttt{skip ; } \mathit{cexpp} \texttt{ [] skip ; } p,\, \mathcal{P}\rangle & , & \langle \texttt{skip ; } \mathit{cexpq} \texttt{ [] skip ; } q,\, \mathcal{Q}\rangle), \end{array}$$

where $\langle p, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}\rangle$ and there is a $\mathit{cexp} \in \mathcal{CE}(\mathsf{P})$ such that $\mathit{cexpp} \in \mathit{cexp} \rightrightarrows^*_{\mathsf{P}}$ and $\mathit{cexpq} \in \mathit{cexp} \rightrightarrows^*_{\mathsf{Q}}$. Then $\mathcal{S}$ is a (P, Q)-simulation.

Proof: As $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ is included in $\mathcal{S}$, using Lemma 5.1.6 we can easily conclude that pairs related by $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ satisfy the simulation conditions. Here we only analyse pairs in the first and second forms. The proof considering pairs in the third and fourth forms follow by a very similar argument. Let us then analyse pairs in the form

$$(\langle p \texttt{ Or } \mathit{cexpp},\, \mathcal{P}\rangle, \langle q \texttt{ Or } \mathit{cexpq},\, \mathcal{Q}\rangle).$$

The unique transitions from the configurations above are

$$\langle q \texttt{ Or } \mathit{cexpq},\, \mathcal{Q}\rangle \rightarrow_{\mathsf{Q}} \langle \texttt{skip ; } q \texttt{ [] skip ; } \mathit{cexpq},\, \mathcal{Q}\rangle$$

and

$$\langle p \ \texttt{Or} \ cexpp, \mathcal{P}\rangle \rightarrow_{\mathsf{P}} \langle \texttt{skip ;} \ p \ \texttt{[] skip ;} \ cexpp, \mathcal{P}\rangle.$$

(They are justified by Rule 4.4.20). Therefore the result trivially holds. Also, as shown above, it is not the case that $\langle q \ \texttt{Or} \ cexpq, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}}$. So it remains only to consider that $\langle exp, \mathcal{Q}\rangle \rightarrow^{*}_{\mathsf{Q}} Q' \not\rightarrow_{\mathsf{Q}}$, for any $exp \in \mathcal{E}(\mathsf{P})$. Hence, as $\langle p, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}\rangle$, by Lemma 5.1.6, for some P', we have

$$\langle exp, \mathcal{P}\rangle \rightarrow^{*}_{\mathsf{P}} P' \not\rightarrow_{\mathsf{P}},$$
$$P' \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q', \text{ and}$$
$$\langle p, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}'\rangle.$$

Therefore the result easily follows since $\mathcal{S}$ includes $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$, and it is easy to check that

$$\langle p \ \texttt{Or} \ cexpp, \mathcal{P}'\rangle$$

is related by $\mathcal{S}$ to $\langle q \ \texttt{Or} \ cexpq, \mathcal{Q}'\rangle$.

Now let us consider pairs in the form

$$(\langle \texttt{skip ;} \ p \ \texttt{[] skip ;} \ cexpp, \mathcal{P}\rangle, \langle \texttt{skip ;} \ q \ \texttt{[] skip ;} \ cexpq, \mathcal{Q}\rangle).$$

By Rules 4.4.2, 4.4.8, 4.4.9, 4.4.10, and 4.4.11, there are only two possible transitions from each of the configurations above:

$$\begin{tikzcd} & \langle \texttt{skip ;} \ q \ \texttt{[] skip ;} \ cexpq, \mathcal{Q}\rangle \arrow[dl, "\mathsf{Q}"'] \arrow[dr, "\mathsf{Q}"] & \\ \langle q, \mathcal{Q}\rangle & & \langle cexpq, \mathcal{Q}\rangle \end{tikzcd}$$

and

$$\begin{tikzcd} & \langle \texttt{skip ;} \ p \ \texttt{[] skip ;} \ cexpp, \mathcal{P}\rangle \arrow[dl, "\mathsf{P}"'] \arrow[dr, "\mathsf{P}"] & \\ \langle p, \mathcal{P}\rangle & & \langle cexpp, \mathcal{P}\rangle \end{tikzcd}$$

Therefore the result easily follows since $\mathcal{S}$ includes $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ and, by Lemma 6.2.3, it is easy to check that $\langle cexpp, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle cexpq, \mathcal{Q}\rangle$. Also, as shown above, it is not the case that

$$\langle \texttt{skip ;} \ q \ \texttt{[] skip ;} \ cexpq, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}} .$$

Lastly, we have to consider that $\langle exp, \mathcal{Q}\rangle \rightarrow^{*}_{\mathsf{Q}} Q' \not\rightarrow_{\mathsf{Q}}$, for any $exp \in \mathcal{E}(\mathsf{P})$. Hence, as

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

by Lemma 5.1.6, for some P', we have

$$\langle exp, \mathcal{P}\rangle \rightarrow^{*}_{\mathsf{P}} P' \not\rightarrow_{\mathsf{P}},$$
$$P' \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q', \text{ and}$$
$$\langle p, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}'\rangle.$$

Therefore the result easily follows since $\mathcal{S}$ includes $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$, and it is easy to check that

$$\langle \texttt{skip ; } p \texttt{ [] skip ; } \mathit{cexpp},\ \mathcal{P}'\rangle$$

is related by $\mathcal{S}$ to $\langle \texttt{skip ; } q \texttt{ [] skip ; } \mathit{cexpq},\ \mathcal{Q}'\rangle$. □

We can then verify that the internal choice method combiner preserves refinement.

Theorem 6.3.19 Let $\mathit{cexp} \in \mathcal{CE}(\mathsf{P})$, $\mathit{cexpp} \in \mathit{cexp} \rightrightarrows^*_{\mathsf{P}}$, and $\mathit{cexpq} \in \mathit{cexp} \rightrightarrows^*_{\mathsf{Q}}$. If

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

then

$$p \texttt{ Or } \mathit{cexpp} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} q \texttt{ Or } \mathit{cexpq}$$

and

$$\mathit{cexpp} \texttt{ Or } p \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathit{cexpq} \texttt{ Or } q.$$

Proof: Using Lemma 6.3.18, it follows as easily as the proof of Theorem 6.3.2. □

Corollary 6.3.20 Let $\mathit{cexp} \in \mathcal{CE}(\mathsf{P})$. If

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

then

$$p \texttt{ Or } \mathit{cexp} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} q \texttt{ Or } \mathit{cexp}$$

and

$$\mathit{cexp} \texttt{ Or } p \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathit{cexp} \texttt{ Or } q.$$

Proof: Directly by Theorem 6.3.19, since it is easy to check that $\mathit{cexp} \in \mathit{cexp} \rightrightarrows^*_{\mathsf{P}}$ and $\mathit{cexp} \in \mathit{cexp} \rightrightarrows^*_{\mathsf{Q}}$. □

6.4 Other Compositionality Properties

The basic compositionality properties of FOOPS operators were established in previous sections, by showing that refinement is preserved by those operators. Here we show how other compositionality properties can be derived from the basic properties. We use the sequential composition method combiner in the examples. Similar results can be derived for other combiners.

First let us recall the basic compositionality properties of sequential composition given in Section 6.3.2: if

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

then, for any $\mathit{cexp} \in \mathcal{CE}(\mathsf{P})$,

$$p \texttt{ ; } \mathit{cexp} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} q \texttt{ ; } \mathit{cexp} \quad \text{and} \quad \mathit{cexp} \texttt{ ; } p \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathit{cexp} \texttt{ ; } q.$$

As briefly discussed before, this result depends crucially on the condition $\mathit{cexp} \in \mathcal{CE}(\mathsf{P})$, since our notion of refinement is based on the observational behaviour of states with respect to a restricted group of experiments.

Now consider the following compositionality property:

$$p \sqsubseteq_{(\mathsf{P},\mathsf{Q})} q \;\text{ and }\; p' \sqsubseteq_{(\mathsf{P},\mathsf{Q})} q'$$

implies

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

In general, this property is not valid because $p' \sqsubseteq_{(\mathsf{P},\mathsf{Q})} q'$ essentially indicates that p' is simulated by q' when they are evaluated in *reachable* database states related by $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$, whereas $p \sqsubseteq_{(\mathsf{P},\mathsf{Q})} q$ just indicates that the evaluation of p and q in reachable states related by $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ leads to states related by $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$; but note that those resulting states are not necessarily reachable, since p and q might not be formed exclusively by experiments. So we cannot always expect $p \;;\; p'$ to be simulated by $q \;;\; q'$. On the other hand, the congruence result is valid because $cexp \in \mathcal{CE}(\mathsf{P})$ is simulated by $cexp$ when they are evaluated in any states related by $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ (see Lemma 6.2.3), if reachable or not.

Although the compositionality property stated above is not valid, the following weaker version of it is indeed valid: for $q, p' \in \mathcal{CE}(\mathsf{P})$,

$$p \sqsubseteq_{(\mathsf{P},\mathsf{P})} q \;\text{ and }\; p' \sqsubseteq_{(\mathsf{P},\mathsf{Q})} q'$$

implies

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

It follows from Theorem 5.1.7 since, under the conditions stated above, by the basic compositionality properties of sequential composition, we have

$$p \;;\; p' \sqsubseteq_{(\mathsf{P},\mathsf{P})} q \;;\; p' \text{ and } q \;;\; p' \sqsubseteq_{(\mathsf{P},\mathsf{Q})} q \;;\; q'.$$

A similar property can be derived in the same way by assuming that $q', p \in \mathcal{CE}(\mathsf{P})$ instead of $q, p' \in \mathcal{CE}(\mathsf{P})$. In fact, those properties would even be valid for $\rightrightarrows$-derivatives of the composition of experiments.

Many other compositionality properties could be easily derived as well. It is important to note that a particular group of those properties can be used to suggest a specific methodology for formal development of software in FOOPS. Here we limit ourselves to give the basis for deriving those properties.

Chapter 7

Mechanization of Refinement Proofs

In this chapter we describe tools for mechanizing refinement proofs such as the ones presented in Chapter 5. First we describe a symbolic simulator, which does most of the routine work of checking the transitions from a given state, according to the FOOPS operational semantics given in Chapter 4. After that we describe a proof assistant, which supports mechanical formal reasoning based on the definitions and propositions introduced in previous chapters (e.g., definition and properties of refinement of states). We also show how the proof assistant interacts with the simulator in order to automatically prove some theorems.

The presentation concentrates on the use of the tools, rather than on their implementation. In fact, both tools are just prototypes developed using the OBJ language [45] and the 2OBJ meta-logical framework [42, 83]. Besides that, the tools are quite useful for partially validating our approach for refinement and motivating its use in practice.

7.1 A Symbolic Simulator for FOOPS

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

The simulator was derived from the inference rules defining FOOPS operational semantics (i.e., the transition relation $\rightarrow_{\mathtt{SP}}$ for a specification SP; see Chapter 4) in such a way that any transition indicated by the simulator can be justified by these rules. The converse cannot be valid in general since the simulator cannot decide upon termination of programs and this would be necessary in order to automatically justify some of the transitions of the operational semantics (see Rule 4.4.19). Nevertheless, under certain circumstances, the simulator can be used without loss of generality.

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

However, transitions depending on arbitrary properties of symbolic values (e.g., transitions corresponding to the evaluation of conditional axioms) cannot be automatically justified; in general, this would require automatic theorem proving involving non trivial theories. Hence the simulator has to indicate transitions together with the conditions that have to be valid in order to justify those transitions. In fact, the prototype of the simulator is implemented by a function `simulate`, where `simulate(SP,P,H)`—for a group of sentences `H`, a specification `SP`, and an associated state `P`—denotes a set of pairs in the form `[ H´ , P´ ]` such that `P` $\rightarrow_{\texttt{SP}}$ `P´` if the sentences in `H´` are valid[1]. The sentences in `H` establish properties of the symbolic values in `P`, whereas `H´` contains `H` and additional conditions.

Having described the main functionality of the simulator, we now proceed to show how the specifications that we are interested in simulating can be encoded into the simulator. After that, we show how to represent expressions, states, and sentences. Then we illustrate the use of the simulator. Lastly, we indicate in which circumstances the prototype of the simulator conforms to the operational semantics.

7.1.1 Specifications

The simulator accepts specifications in a small extension of FOOPS syntax, which was informally described in Chapter 3. The extra information available in those specifications could actually be automatically derived from FOOPS specifications, but in order to simplify the prototype implementation of the simulator we assume this information is explicitly provided.

In order to illustrate the encoding of specifications into the simulator, we progressively introduce an encoding of the specification of buffers given in Section 3.2. First we consider auxiliary theories. Let us start with the module `NAT-SIM`, defining natural numbers:

```
fmod NAT-SIM is
  ex BASIC .
```

Any specification encoded into the simulator should import the module `BASIC`, which introduces the universal sort `` `Universal``, defines an ADT of booleans (with the usual operations: `` `true``, `` `false``, `` `and``, `` `not``, etc.), and provides the equality and inequality operations (`` `==`` and `` `=/=``). Those definitions are available in any FOOPS specification, except that the operation and sort symbols are not prefixed by '`'[2]. However, they have to be explicitly imported when using the simulator.

Sorts, operations, and variables are declared in the usual way, but all sorts in the specification have to be declared subsorts (subtypes) of `` `Universal``:

```
sorts `Nat `NzNat `Zero .
subsorts `Zero `NzNat < `Nat < `Universal .
fn `0 : -> `Zero .
fn s_ : `Nat -> `NzNat .
```

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

[2]Different names have to be used because the simulator is an extension of a FOOPS interpreter [82, 79] that introduces predefined modules; so we have to avoid naming conflicts with the operations defined in those modules. Also, we use the name `NAT-SIM` because `NAT` is a predefined module.

```
    fn p_ : `NzNat -> `Nat .
    fn _`+_ : `Nat `Nat -> `Nat [assoc comm] .
    ⋮
    vars N M : `Nat .
    vars N´ M´ : `NzNat .
```

Functional level axioms must be preceeded by the declaration `[axiom,fun]`, as shown below:

```
    [axiom,fun] ax p s N = N .
    [axiom,fun] ax N `+ `0 = N .
    [axiom,fun] ax (s N) `+ (s M) = s s(N `+ M) .
    ⋮
  endfmod
```

The extra declaration indicates that the equation is a functional level axiom and, as such, can only be used for equational reasoning or term rewriting when the associated variables are instantiated with fully evaluated terms (see Section 4.3.1).

Similarly, we can write the encoding of the module `LIST` specified in Section 3.1:

```
  fmod LIST-SIM[E :: TRIV] is
    pr NAT-SIM .
    sorts List NeList .
    subsorts Elt < NeList < List < `Universal .
    fn `nil : -> List .
    fn _._ : List List -> List [assoc id: `nil] .
    ⋮
    var E : Elt .
    vars L L´ : List .
    ⋮
    [axiom,fun] ax # `nil = `0 .
    [axiom,fun] ax # (E . L) = # L `+ (s `0) .
  endfmod
```

The functional theory `MAX` can also be encoded in the same way; we omit the details here.

Now we consider the object level parameterized specification `BUFFER` (see Section 3.2). Instead of importing the module `BASIC`, in the encoding of object level specifications we import the module `MC`, which includes `BASIC` and declares FOOPS method combiners (i.e., _;_, _||_, etc.). We have also to declare that any class is a subtype of `` `Universal``:

```
  omod BUFFER-SIM[E :: TRIV, M :: MAX] is
    ex MC .
    pr LIST-SIM[E] * (op (_._) to (_:_)) .
    class Buffer .
    subsort Buffer < `Universal .
```

As indicated by FOOPS module renaming operator *, here we use a different notation for list concatenation, comparing with the one introduced in `LIST-SIM`.

Operations and variables are declared in the usual way, except that the declaration `[private]` should be used in place of `[hidden]`:

```
  at elems_ : Buffer -> List [private] .
  at empty?_ : Buffer -> `Bool .
  me get : Buffer -> Elt .
  me na-get : Buffer -> Elt .
  me put : Buffer Elt -> Buffer .
  ⋮
  var B : Buffer .
  var E : Elt .
```

We also have to add the declaration `[axiom,attr]` to axioms defining attributes, whereas IMAs and DMAs must be preceeded by `[axiom,me]`, as shown below:

```
  [axiom,attr] ax empty? B = (elems B) `== `nil .
  [axiom,me] cx elems put(B,E) = (elems B) : E if `not(full? B) .
  [axiom,me] cx na-get(B) = [ result head(elems B) ; del(B) ]
             if `not(empty? B) .
  [axiom,me] ax get(B) = [ na-get(B) ] .
  ⋮
endomod
```

Observe that we must explicitly add the declarations corresponding to the abbreviation introduced by axioms having the keyword `acx` (see Section 3.2.1).

Now we can instantiate `BUFFER-SIM` in order to obtain a specification that can be simulated. For instance, we consider a specification of buffers that are able to store six natural numbers:

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

This should then be loaded into the simulator. The underlying system supporting the simulator is an extension of the FOOPS interpreter described in [82, 79] to accept the syntax of specifications discussed in this section. Moreover, the core of the simulator is actually implemented by a FOOPS functional level theory: `SIMULATOR`. So `SP` should be loaded into the extension of the interpreter and then integrated into the core of the simulator in the following way:

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

As we will show later, this module can then be used to simulate the specification `SP`, by using the functions provided by `SIMULATOR`, and the extended FOOPS interpreter for carrying on computations.

7.1.2 Expressions

Essentially, any expression of type `Universal can be simulated. Considering the module SIM-TH (introduced in the last section), any term formed with the operations of SP can be simulated, since any type of SP is a subtype of `Universal. But in order to simulate more useful expressions, we have to indicate the object identifiers associated to the class Buffer. This can be easily done by extending SIM-TH[3] with the following declaration:

```
mes b b´ b1 b2 : -> Buffer .
```

This indicates that b, b´, b1, and b2 are object identifiers associated to Buffer. We can then simulate expressions such as put(b,`0).

7.1.3 States

Besides defining the simulation function simulate, the simulator provides an ADT of database states (for short, just states). Here we shall be interested in three operations from this ADT: an operation for creating the initial state associated to a specification; an operation for adding an object to a state; and an operation for updating the value of an attribute. Those operations are respectively declared as follows:

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

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

In order to use the function initial-state to create the initial state associated to SP (defined in Section 7.1.1), we have to obtain a representation of the module SP as an element of the sort Specification. This can be done by extending SIM-TH in the following way:

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

The function module-to-specification is provided by the simulator; it yields the element of Specification denoted by the FOOPS module whose name is given as argument[4]. Note that we are using the symbol SP both for the FOOPS module and its representation as an element of Specification. The declaration [memo] indicates that the value associated to SP is computed once and then memorized; this is used here for efficiency reasons.

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

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

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

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

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

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

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

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

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

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

7.1.4 Sentences

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

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

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

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

are typical equations associated to the specification `SP` (defined in Section 7.1.1).

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

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

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

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

is formed by the sentences ‘`` `not(`0 `== `6) = `true ``’ and ‘`` `not(s(`0) `== `6) = `true ``’.

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

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

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

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

where `conf` is the configuration defined in the last section.

7.1.5 Simulations

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

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

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

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

For a specification SP, a group of sentences H, and a configuration P, if [H´ , P´] is in final-states(SP,[H , P]) then P $\rightarrow^{*}_{\mathtt{SP}}$ P´ $\not\rightarrow_{\mathtt{SP}}$ unless one of the sentences in H´ is not valid.

Now we illustrate the application of those functions[5] using the module SIM-TH (introduced at the end of Section 7.1.1) and the declarations introduced in previous sections. We give examples using some of the transitions discussed in Section 5.2.2 (from page 85 to page 88). Let us then consider a database state stP having only an unitialized object b´:

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

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

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

This yields the following:

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

where 'b´ elems_: `nil' represents a database state containing the object b´ with its attribute elems set to `nil, {} denotes the empty set of pairs of sentences and configurations, and e + s denotes the set containing e and all elements in the set s. So there is only one transition from the simulated state; the execution of reset(b´) is deterministic. Also, note that the unique transition depends only on the condition expressed by the tautology `true = `true[6]; hence it is always possible.

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

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

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

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

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

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

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

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

indicating that there is only one transition from the simulated state, and this transition does not depend on any condition. In the simulator, the expression $[\![$`l == nil`$]\!]_{SP}$ is represented as `[| l `== `nil |]-{SP}`.

The execution of `put(b,e)` in `stP2` is illustrated by evaluating

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

which results in

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

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

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

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

yields

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

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

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

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

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

evaluates to

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

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

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

which results in the following:

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

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

7.1.6 Correctness

Ideally, the transitions indicated by the simulator should exactly be the transitions defined by FOOPS operational semantics; that is, the simulator should be correct with respect to the semantics, it should conform to the semantics. This is not possible here because the definition of the operational semantics depends on issues such as termination of programs (see Rule 4.4.19). In fact, any transition indicated by the simulator is a transition specified by the operational semantics, but not vice versa. However, the reverse also holds under certain circumstances, which we will describe now.

Obviously, the simulator cannot decide upon termination of programs. So it cannot automatically justify transitions specified by Rule 4.4.19; in fact, those transitions are not indicated by the simulator. Therefore the simulator can only conform to the semantics when expressions to be atomically evaluated terminate in the contexts where they are used, since only in these cases the transitions specified by Rule 4.4.19 cannot happen.

Actually, the simulator could indicate transitions specified by Rule 4.4.19 provided that these transitions were conditional on termination assumptions. But, as shown in Section 7.1.4, assumptions indicated by the simulator should be equations relating expressions of the specification being simulated. Moreover, extending the language of assumptions to include termination predicates would not solve other problems related to termination issues, as we will describe next.

In order to indicate transitions specified by Rule 4.4.18, the simulator essentially has to compute the transitive closure of $\rightarrow_{\mathtt{SP}}$ for a given state. But this computation only terminates if there is no infinite sequence of transitions from that state. In fact, the evaluation of `simulate` might not terminate. Again, we can conclude that the simulator can only conform to the semantics when expressions to be atomically evaluated terminate in the contexts where they are used, since only in this circumstance the evaluation of `simulate` terminates.

Unbounded nondeterminism also precludes the simulator from conforming to the semantics. Indeed, by Rule 4.5.1, the execution of `new.C()`, for a class `C`, leads to infinitely many states whenever the set of object identifiers associated to `C` is infinite. This possibly infinite set of transitions is not indicated by the prototype of the simulator. In fact, only one of those transitions is nondeterministically chosen and indicated by the simulator. Although this is not the behaviour specified by the operational semantics, it gives a similar effect which can be used in some applications without loss of generality.

Lastly, we have to note that the simulator does not conform to the semantics when considering specifications with contradictory DMAs (see Section 4.3.3). This is not actually a problem since those specifications should not be of interest for simulations.

7.2 A Proof Assistant for FOOPS

As shown in Chapter 5, in order to prove refinement we have to reason about many different concepts (e.g., simulation relation, state transitions, and fully evaluated terms). Hereafter we use the name "refinement theory" for the collection of those concepts' properties presented in previous chapters. The FOOPS proof assistant (for short, proof assistant) supports mechanical formal reasoning based on most aspects of the refinement theory.

The proof assistant is based on the 2OBJ system, which is a theorem prover that can be tailored to prove theorems in any logical system. By loading the encoding of a logic into 2OBJ, we automatically obtain a theorem prover for that logic. Once 2OBJ is tailored for a particular logic, we can define theories based on this logic and build proofs over these theories. Usually, proofs are interactively built through 2OBJ's graphical interface. Here we simply indicate the proof steps necessary for completing a proof, not how those can be given as commands to 2OBJ.

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

Here we present aspects of the encodings of the refinement theory and first order predicate logic. We discuss only the aspects that we have to know in order to build refinement proofs. First we consider the encoding of the logic, as originally described in [83]; this should give a general overview of the basic machinery available for constructing refinement proofs. Then we consider the aspects of the encoding of the refinement theory that are necessary for proving condition 1 of the definition of simulation (see Definition 5.1.1 on page 66); the other conditions can be proved by using essentially the same techniques, so we omit the details here. Lastly, we illustrate how the proof assistant can be used for proving refinement.

7.2.1 First Order Predicate Logic

In 2OBJ, the encoding of a logic is given by ADTs specified in OBJ (the functional level of FOOPS), where sentences and proofs are represented as terms of this ADT, and operations on sentences and proofs are defined as operations of the ADT. In particular, the first order predicate logic with equality is encoded by the OBJ module `FOPC`. Here we introduce some of the definitions in this module. A more detailed description can be found in [83]. First we explain how sentences and sequents are encoded. Then we describe the inference rules provided by `FOPC`. Lastly, we show how to define tactics in 2OBJ.

Sentences

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

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

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

Sequents

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

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

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

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

Inference Rules

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

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

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

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

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

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

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

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

Here is the list of inference rules from `FOPC` that are useful for proving refinement; the rules are written in the usual notation of first order predicate logic:

Conjunction:

$$\frac{H \vdash A \quad H \vdash B}{H \vdash A \wedge B}\texttt{andi} \qquad \frac{H_1; \ldots; H_n; A; B \vdash C}{H_1; \ldots; H_n \vdash C}\texttt{ande(}i\texttt{)} \quad (H_i = A \wedge B)$$

Disjunction:

$$\frac{H \vdash A}{H \vdash A \vee B}\texttt{oril} \qquad \frac{H \vdash B}{H \vdash A \vee B}\texttt{orir}$$

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

Implication:

$$\frac{H; A \vdash B}{H \vdash A \rightarrow B}\texttt{impi} \qquad \frac{H_1; \ldots; H_n \vdash A \quad H_1; \ldots; H_n; B \vdash C}{H_1; \ldots; H_n \vdash C}\texttt{impe(}i\texttt{)} \quad (H_i = A \rightarrow B)$$

Negation:

$$\frac{H; A \vdash False}{H \vdash \neg A}\texttt{noti} \qquad \frac{H_1; \ldots; H_n \vdash A}{H_1; \ldots; H_n \vdash C}\texttt{note(}i\texttt{)} \quad (H_i = \neg A)$$

$$\frac{H_1; \ldots; H_n; A \vdash C}{H_1; \ldots; H_n \vdash C}\texttt{notcc(}i\texttt{)} \quad (H_i = \neg\neg A)$$

Trivial truth and absurdity:

$$\frac{}{H \vdash True}\texttt{Truei} \qquad \frac{}{H_1; \ldots; H_n \vdash C}\texttt{Falsee(}i\texttt{)} \quad (H_i = False)$$

Universal Quantification:

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

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

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

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

Existential Quantification:

$$\frac{H \vdash A[t/V]}{H \vdash \exists V.A}\texttt{Ei(}t\texttt{)}$$

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

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

Miscellaneous:

$$\frac{}{H_1; \ldots; H_n \vdash C}\texttt{hyp(}i\texttt{)} \quad (H_i = C) \qquad \frac{}{H \vdash A \vee \neg A}\texttt{exmid}$$

$$\frac{H \vdash X \qquad H; X \vdash C}{H \vdash C}\texttt{cut(}X\texttt{)} \qquad \frac{\vdash C}{H \vdash C}\texttt{thin}$$

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

Tactics

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

- The application of the tactic `t THEN t´` to a goal corresponds to the application of the tactic `t` to the same goal and, when `t` succeeds, `t´` is applied to the subgoals resulting from the application of `t`. If `t` or `t´` fails then `t THEN t´` fails as well.
- The application of the tactic `t THENL (`t_1`,...,`t_n`)` to a goal corresponds to the application of the `t` to the same goal and, when `t` succeeds, the tactics t_1,..., and t_n are applied to the corresponding subgoals created by the application of `t`. Note that the number of subgoals must be `n`; otherwise, `t THENL (`t_1`,...,`t_n`)` fails. It also fails if `t` or any of the tactics t_1, ..., or t_n fails.
- The application of the tactic `t ELSE t´` to a goal corresponds to the application of `t` to the same goal and, if `t` fails, `t´` is applied to that goal. If `t´` fails then `t ELSE t´` fails as well.
- The application of the tactic `idtac` to a goal yields the same goal as a subgoal. This tactic is very useful in tactic programming; it always succeeds.

A tactic is defined by an equation specifying the effect of applying the tactic to a specific goal. For example, a tactic for introducing case analysis in a proof can be easily defined by using the inference rules of `FOPC` and the built-in tacticals:

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

where `SQ` is a variable of sort `Sequent`. By applying the tactic `CASE(sent)`, we split the proof in two parts: in one of them, it is assumed that `sent` is valid; in the other, it is assumed that `~sent` is valid.

More elaborated examples of how new tactics can be programmed are given in [83]. Here we just mention some tactics that can be defined:

- The application of the tactic `TRY(t)` to a goal corresponds to the application of `t` to the same goal. If `t` fails, `TRY(t)` behaves as `idtac`.
- The application of the tactic `EXHAUST(t)` to a goal corresponds to the exhaustive application of `t` to that goal; that is, `t` is applied to the goal and then `EXHAUST(t)` is recursively applied to the generated subgoals, until `t` cannot be applied anymore.

7.2.2 Refinement Theory

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

Observe that most objects and functions of the refinement theory are defined by the symbolic simulator described in Section 7.1. In particular, the simulator provides OBJ implementations of ADTs of terms, specifications, database states, and configurations; those have an essential role in the refinement theory. So the encoding of those objects and functions is essentially given by importing the module `SIMULATOR`, which defines the simulator, into the module `REF-TH`. Besides that, extra elements should be added to sorts such as `State`, `Specification`, etc; those elements correspond to first order predicate logic variables (elements of sort `Var`, not OBJ variables) ranging over the elements of the corresponding ADTs of states, specifications, etc.

Database States

In the encoding of the refinement theory, we refer to the operations on states introduced in Section 7.1.3 and the following ones:

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

where the sort `` `UniversalSet `` corresponds to an ADT of sets of elements of `` `Universal ``. The first function gives the set of identifiers of the objects in a database state. The second function yields the value of an attribute expression in a particular state. The last function indicates whether an element of `State` is a "syntactic state" (i.e, a variable) or a "semantic state" (i.e., an element of the ADT of states defined in `SIMULATOR`).

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

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

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

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

Configuration

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

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

where p, pp, and dp respectively are elements of sorts Configuration-Var, `Universal-Var, and State-Var; that is, they are first order predicate logic variables respectively ranging over configurations, expressions, and database states. As described in Section 7.1.3, the operation < _ , _ > creates configurations.

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

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

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

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

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

Relation on Configurations

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

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

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

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

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

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

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

where syntactical equality is represented by the operation

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

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

Equational Reasoning

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and it is similar to the second rule described in this section, except that the selected element of the hypothesis list should be a term in the form `t =~= t´`, rather than an equation. Lastly, the rule `synt-eq` can be used to derive a sentence in the form `t =~= t´`, provided that the terms `t` and `t´` are syntactically equal.

State Transition

In the encoding of the refinement theory, sentences indicating state transitions can be created using the following operations:

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

where `p -->{sp} q` is the encoding of $\mathtt{p} \rightarrow_{\mathtt{sp}} \mathtt{q}$, and `p -->*{sp} q` is the encoding of $\mathtt{p} \rightarrow^{*}_{\mathtt{sp}} \mathtt{q}$. The axioms and inference rules expressing properties of sentences formed with the operations above are not of interest for us here.

Simulation Relation

We can indicate that a relation on configurations is a simulation by constructing sentences with the following operation:

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

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

As here we consider only the details necessary for proving condition 1 of the definition of simulation (see Definition 5.1.1 on page 66), for simplicity we assume that the following inference rule can be used in order to prove that a relation is a simulation:

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

where p, q, p´, and q´ are first order predicate logic variables ranging over configurations. This rule essentially corresponds to a simplified definition of simulation, considering only condition 1, and omitting the constraints on specifications.

Now observe that a predicate in the form

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

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

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

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

Sometimes we actually have to check state transitions in order to prove that a relation is a simulation. As discussed in Section 7.1, under certain circumstances, we can use the FOOPS symbolic simulator in order to automatically check state transitions. In fact, we can invoke the simulator from the proof assistant since the module `SIMULATOR` is imported into the module `REF-TH`, which should then have an inference rule for automatically justifying state transitions based on the simulator's output.

For using the simulator to check a transition in the form `p -->{spp} p´`, the following condition has to be observed: the expressions to be atomically evaluated in `p` should terminate. In the encoding, this is formalized by the following sentence

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

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

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

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

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

The function `object-ids` yields the set of object identifiers in a given expression. As specified by the axiom `is-related-to` (omitted here), a sentence in the form

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

is an abbreviation for

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

This is what really provides access to the simulator, by indicating that all transitions from `Q`—as given by the simulator—can be derived from `P`, according to a sentence in the form

$$\texttt{are (SC}_1\texttt{ + } \cdots \texttt{ + SC}_n\texttt{ + \{\}) derived-from P by(S,spP,dP,dQ)}$$

which, as specified by the axioms `are-derived-from-e` and `are-derived-from` (omitted here), stands for

$$\begin{array}{l} \quad \texttt{is SC}_1\texttt{ derived-from [* , P] by(S,spP,dP,dQ)} \\ \hat{}\ \vdots \\ \hat{}\ \texttt{is SC}_n\texttt{ derived-from [* , P] by(S,spP,dP,dQ)} \\ \hat{}\ \texttt{True} \end{array}$$

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

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

A sentence in the form

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

abbreviates the sentence

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

whereas a sentence in the form `is [ SSL , Q ] together-with [ SSL´ , P ] in(S,dP,dQ)` is defined as follows:

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

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

7.2.3 Proving Refinement

In order to illustrate how the proof assistant can be used for proving refinement, we consider examples involving different specifications of bounded buffers, as described in Section 5.2.2. First we explain how the FOOPS specifications that we want to reason about are encoded and integrated with the proof assistant. Then we show how to define relations on configurations associated to those specifications and then prove that the relations are simulations.

Specifications

In order to encode a specification into the proof assistant, we follow essentially the same rules used to encode a specification into the simulator, as described in Section 7.1.1. However, for each type (sort or class) in the FOOPS specification, we should also define a type in the encoding representing the set of first order predicate logic variables ranging over elements of the corresponding type in the specification. Moreover, all types denoting variables should be a subtype of `` `Universal-Var``, which is a subtype of `` `Universal``. For example, the encoding of a specification of natural numbers into the proof assistant is exactly as the encoding of this specification into the simulator (given by the module `NAT-SIM`, introduced in Section 7.1.1), except for the information shown below:

```
fmod NAT-ASS is
  ex BASIC-VAR .
  ⋮
  sorts `Zero-Var `NzNat-Var `Nat-Var .
  subsorts `Zero-Var `NzNat-Var < `Nat-Var < `Universal-Var .
```

```
    subsort `Zero-Var < `Zero .
    subsort `NzNat-Var < `NzNat .
    subsort `Nat-Var < `Nat .
    ⋮
  endfmod
```

Note that the module BASIC-VAR, instead of BASIC, is imported into NAT − ASS; it is an extension of BASIC with the definition of the sort `Universal-Var.

Based on the module LIST-SIM defined in Section 7.1.1, we can similarly define a module LIST-ASS suitable to be integrated with the proof assistant. However, note that LIST-ASS should be parameterized by the following extension of TRIV:

```
  fth TRIV-VAR is
    ex BASIC-VAR .
    sorts Elt-Var Elt .
    subsorts Elt-Var < Elt < `Universal .
    subsort Elt-Var < `Universal-Var .
  endfth
```

which indicates that the ADTs used to instantiate LIST-ASS should have an associated sort of first order predicate logic variables.

By using TRIV-VAR, we can also define a module BUFFERL-ASS, which is similar to BUFFER-SIM (see Section 7.1.1), except for the following declarations:

```
  omod BUFFERL-ASS[E :: TRIV-VAR, M :: MAX] is
    ex MC-VAR .
    pr LIST-ASS[E] * (fn (_._) to (_:_)) .

    classes BufferL-Var BufferL .
    subsort BufferL-Var < BufferL < `Universal .
    subsort BufferL-Var < `Universal-Var .
    ⋮
  endomod
```

Instead of MC, the module MC-VAR, which is an extension of MC with BASIC-VAR, is imported. We can then instantiate BUFFERL-ASS in the following way:

```
  omod SP-LIST is
    pr BUFFERL-ASS[view to NAT-ASS is sort Elt to `Nat .
                                      sort Elt-Var to `Nat-Var . endv,
                   view to NAT-ASS is fn max to `6 . endv] .
  endomod
```

Note that the view also indicates the interpretation of the sort Elt-Var.

In a similar way to how LIST-ASS was defined from LIST-SIM (and LIST, introduced in Section 3.2), we can define the module ARRAY-ASS encoding the specification of arrays ARRAY, which was described in Section 5.2.2. Also, the specification BUFFERA, described in Section 5.2.2, can be encoded just like BUFFERL-ASS was encoded from BUFFER-SIM (and BUFFER, defined in Section 3.2). Here we omit the details and just assume that the module BUFFERA-ASS introduces the class BufferA of buffers implemented by arrays and pointers, and that it can be instantiated in the following way:

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

similarly to how SP-LIST was defined.

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

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

This FOOPS object level module can then be loaded into a version of 2OBJ built on top of a FOOPS interpreter, rather than on top of an OBJ interpreter.

A Simulation

By extending the module ASS-TH introduced in the last section, we can define relations and check whether they are simulations. But first we have to represent the modules SP-LIST and SP-ARRAY as elements of the sort Specification:

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

The following declarations define the relation sim1, which is similar to the relation considered in Section 5.2.2. The definition of sim1 consists of three parts: constraints on expressions, general constraints on configurations, and constraints on database states:

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

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

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

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

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

The following sentence formalizes the constraints on states related by `sim1`; it is basically an encoding of the constraints on states related by the relation $\mathcal{S}$ defined in Section 5.2.2, on page 84:

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

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

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

are defined as follows:

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

Note that we check if an attribute `at(e)` is not defined in a state `db` by checking if the expression '`eval at(e) in db`' yields `at(e)`.

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

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

By applying the inference rule `simulation?` to the goal above, we obtain the following subgoal:

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

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

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

In order to unfold the definition of `sim1`, we use equational reasoning by applying the rule `app(´ASS-TH,´sim1,fwd,at,nosub,top,1)`. Thus we obtain the subgoal

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

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

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

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

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

yielding the subgoal

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

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

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

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

Another Simulation

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

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

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

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

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

Assuming that `b` and `e` are arbitrary, if we prove that `sim2` is a simulation then we also prove that the extension of `sim2` that relates expressions in the form `put(b,e)`, for any `b` and `e`, is also a simulation.

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

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

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

```
fn db-inst-ba : State BufferA Nat Nat Nat -> Rule .
cx db-inst-ba(DB,BA,M,N,I) (H |- G) =
   H ; DB = (DB ++ add-object BA to initial-state(SP-ARRAY)) |- G
```

```
if hyp(N,H) == (BA In? objects(DB)) and
   hyp(M,H) == (((((eval array(BA) in DB) `== array(BA)) = true)) ^
               (((eval fst(BA) in DB) `== fst(BA)) = true) ^
               (((eval next(BA) in DB) `== next(BA)) = true)) and
   matches((is < pP , DB > compatible-with SP-ARRAY ?) = true,hyp(I,H)) .
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 8

Conclusions

In this thesis we presented a framework for supporting formal development of software in the concurrent object oriented language FOOPS. In particular, we described the following aspects:

- An operational semantics.
- A notion of refinement for concurrent object oriented programs.
- A theory establishing congruence properties of refinement.
- Tools for mechanizing proofs of refinement.

By developing this framework, many questions and concepts were raised and clarified. This motivated modifications and extensions to FOOPS, suggested appropriate programming techniques, and justified many technical points of FOOPS design. Those aspects were also described here.

Now we summarize and discuss the main results described in this thesis. After that, we compare the work presented here with related approaches. Lastly, we indicate directions for further research.

8.1 Summary and Conclusions

FOOPS

In Chapter 3 we introduced a natural extension of FOOPS with features for specifying systems of concurrent, distributed, and autonomous objects. This extension is based on ideas from [73] and [29] for adding concurrency, nondeterminism, and auto-methods for FOOPS. Besides that, we extended the language with the following features: atomic execution of expressions, definition of method combiners, evaluation of expressions as background processes, and a mechanism for object protection. This extension offers a general computational model including most aspects of concurrent object oriented programming.

Operational Semantics

In Chapter 4 we described a structural operational semantics for FOOPS, considering features such as: classes of objects with associated methods and attributes, object identity, dynamic object creation and deletion, overloading, polymorphism, inheritance with overriding, concurrency, nondeterminism, atomic execution, definition of method combiners, evaluation of expressions as background processes, auto-methods, nonterminating methods, and a mechanism for object protection. This semantics is the core of a formal definition of FOOPS.

We concentrated on the object level of FOOPS. The semantics of other aspects, like the functional level and the module system, are discussed elsewhere [79, 60]. Here we only considered the semantics of "flat" specifications; that is, specifications without module importation or generic parameters.

The usual features of object oriented languages were explained in detail, in a simple and abstract way. This was made possible by adopting a special approach for modelling states of the operational semantics: we adapted some ideas from [39], where states are represented as order sorted theory presentations [40]. This approach uses the power of the theory of ATDs for defining operations on states and reasoning about them; in particular, the semantics of inheritance, and evaluation and dynamic binding of stored attributes is directly provided by OSA [40]. It then became simple to define the basic operations on states. In fact, lots of complications were avoided, a concise semantic definition could be obtained, and many concepts usually confusing in other frameworks were clarified. Indeed, this approach seems appropriate to define the operational semantics of other object oriented languages as well.

We also justified the semantics adopted for some constructs, comparing with alternatives. In special, we analysed two parallel composition operators: the interleaving operator and an operator which allows simultaneous evaluation of expressions. We argued that those two operators are equivalent in the context of FOOPS, given some mild assumptions on the notion of equivalence of programs adopted for the language. In Chapter 5, we actually proved that this holds considering a particular definition of equivalence. That should be expected since it is desirable to specify systems of distributed objects without worrying whether other computations are being carried out simultaneously. This result could probably be generalized for other concurrent object oriented languages.

Along with the semantic description, we clarified many concepts and phenomena related to object oriented languages and FOOPS. In particular, the definition of the operational semantics motivated small modifications to FOOPS design and informal semantics of FOOPS described in [39]:

- there must be syntactical constraints on signatures, specifications, axioms' conditions, and the RHS of DMAs;
- attributes do not necessarily must have an associated value in database states;
- the object creation operations should not have arguments corresponding to values for initializing attributes; and
- class invariants are just annotations.

We also briefly discussed how the semantics suggests an appropriate programming style for FOOPS, and how to avoid introducing inconsistencies in FOOPS specifications.

Definition of Refinement

In Chapter 5 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.

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 [21]. Although we focused on FOOPS, our definition of refinement only depends on minimal assumptions about FOOPS and its operational semantics; those assumptions were described in this thesis. 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.

Our notion of refinement is based on the operational semantics described in Chapter 4. This was made possible due to the simplicity of the semantic definition. Notions of refinement and equivalence based on operational semantics have been successively developed for other languages as well [69, 72, 48]. Our notion of refinement was developed by adapting ideas from [69], [68], and [77], where a general approach for defining equivalence of transition systems is developed and justified; this approach is quite simple and provides an effective proof technique for verifying refinement. We also considered related ideas from work on data refinement [49, 54, 36].

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), non-determinism, concurrency, interference, and refinement of atomic operations. We also concluded that the definition of refinement presented in this thesis 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.

Refinement Theory

In Chapter 6 we established several properties of FOOPS' notion of refinement. In particular, we proved that the relation of refinement of expressions is a congruence with respect to most FOOPS predefined method combiners. We also explained how to derive other compositionality results. The congruence property justifies the compositional verification and derivation of implementations; this is essential for formal development of large systems. However, we only considered compositionality *in the small*; properties of compositionality *in the large* are related to the semantics of FOOPS module system, which was not described here.

The proof of the congruence result justified and provided a better insight into some of the technical decisions adopted in this thesis. For instance, by analysing the proof we could conclude that the congruence result can only be obtained if experiments have an atomic and terminating nature. Moreover, we indicated that atomicity of experiments does not necessarily mean that experiments have to be evaluated in one state transition. Instead, the fully evaluation of an experiment may take many transitions, as long as the only (if any) transition that depends on the state or corresponds to a nondeterministic choice is the last one. That is the essence of atomicity.

We can also conclude that the congruence property is only valid for *contexts* formed by *visible* operations (experiments). In fact, this should be expected for any notion of refinement based on the observational behaviour of states with respect to a restricted group of *visible* operations. In particular, refinement is not a congruence with respect to contexts formed by user defined method combiners, since they might just be abbreviations for a combination of hidden operations.

We explained why refinement is not a congruence with respect to the atomic evaluation and nondeterministic *external* choice operators. Basically, the behaviour of those operators depends on some aspects which are abstracted by our notion of refinement. However, we verified that nondeterministic *internal* choice preserves refinement, and a weaker compositionality result is valid for the atomic evaluation operator. Indeed, this weaker result indicates how the atomic evaluation operator should be used in practice. Lastly, we concluded that refinement is not a congruence with respect to visible operations (methods and attributes), since $p \sqsubseteq q$ does not necessarily imply that p and q have the same or related types; so, for a visible operation op, $op(p)$ might be a well formed term whereas $op(q)$ is not.

Tools

In Chapter 7 we described a symbolic simulator and a proof assistant for FOOPS. We presented the main features of both tools and explained how they can be used for mechanizing refinement proofs. Those tools are just prototypes developed using the OBJ language [45] and the 2OBJ meta-logical framework [42, 83]. However, they turned out to be useful for partially validating our approach for refinement and motivating its use in practice.

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

The proof assistant supports mechanical formal reasoning based on an encoding of the definitions and propositions introduced throughout this thesis. This encoding is an extension of the encoding of first order predicate logic with equality presented in [83]. We described the encodings' details that are necessary to prove condition 1 of the definition of simulation relation. In particular, we showed how the proof assistant interacts with the simulator in order to automatically prove some theorems. The details of the encoding related to the remaining conditions of the definition of simulation were implemented in a similar way and can be proved by using essentially the same techniques.

We only indicated the steps necessary for interactively completing proofs. However, the proofs presented here could be mostly automated by defining suitable tactics in 2OBJ. In fact, the proofs described in Section 5.2.1 can be automatically done using the proof assistant, whereas the proofs described in Section 5.2.2 can be mostly automated, leaving to the user only functional level theorems which can be proved using standard equational reasoning and induction. In most cases, the assistant helps to reduce object level theorems to functional level theorems, which result from proofs that a configuration pair belongs to a given relation, and proofs that transitions from one state can happen whenever transitions from another state can happen. In general, those functional level theorems are related to non trivial functional theories, so that they cannot be automatically proved.

8.2 Related Work

FOOPS

The original design of FOOPS was introduced in [39]. A detailed description of a revised design of the language was given in [79] and [82], including some ideas about different approaches for its formal semantics: reflective semantics based on order sorted algebra [39, 40], hidden order sorted algebra [28, 36], and sheaf theory [90, 64, 32]. Some aspects of FOOPS design have been further explored in several places [46, 41, 73]. In [82], FOOPS is generally evaluated and compared with other languages. A language that evolved from FOOPS is described in [1].

Now we comment on some alternatives to the original approach for object protection that we proposed for FOOPS. First, we consider the introduction of *root objects*, as in C++ [84] and Eiffel [67]. Basically, only root objects may be interfered in an arbitrary way; interference in non root objects is derived from interference in root objects. In FOOPS without object protection, any object is a root object. Usually, there is more than one root object in a distributed system, but only one in a sequential system.

This mechanism only protects non root objects from arbitrary interference. In fact, we have to check if the interference propagated by root objects does not violate the protection rules that we want to enforce for non root objects. In general, this might generate complicated proof obligations. Moreover, this only assures protection for a system in isolation, it does not guarantee protection if this system is included as part of a larger system, where more root objects might be available and additional propagated interference might occur. Clearly, this is not appropriate for a compositional development method.

In order to achieve compositionality with this approach, it would be necessary to verify that the objects that should be protected cannot be accessed by new objects in a larger system. This can be enforced by making sure that the identifiers of the protected objects are not yielded by any operation that may be invoked by objects in the larger system. Again, this might lead to complicated proof obligations. In fact, it seems simpler to support a mechanism that directly enforces object protection, rather than writing some specific code for guaranteeing that, and discharging complicated proof obligations.

As briefly discussed in this thesis, the second alternative for object protection is the support for private references [58]. However, it is not appropriate in general. In fact, this is a particular case of our mechanism for object protection. Considering the examples given in Section 3.2.5, private references are appropriate for modelling linked lists (since an intermediate cell should only be accessed by its precedent in the sequence); however, private references are too restrictive for modelling the communication protocol (since the channel should be protected, but shared by both agents). Hence, it seems useful to have a more general mechanism for object protection.

Another general approach for object protection was independently introduced by Hogg [53]. In our approach, one explicitly restricts the group of objects that can directly invoke methods of a protected object. In Hogg's approach, one explicitly indicate a so called *bridge* object, defining an associated group of protected objects (*island*); then any direct access to a protected object (an object in an island) must be indirectly derived from an access to an associated bridge object. There are subtle differences between the two approaches. Whereas Hogg's approach seems more abstract, it seems that a finer level of protection can be specified by our approach. In fact, more experiments would be necessary to give us more confidence that Hogg's approach should not be supported by FOOPS as well.

Operational Semantics

Most of the FOOPS features considered by the operational semantics described in this thesis are not considered by other alternatives for the semantics of FOOPS, like the reflective semantics [39] and the sheaf semantics [90, 64, 12]. In fact, we are not aware of a formal semantics for a language integrating all those features.

As described in [39], the basic idea about FOOPS reflective semantics, is to represent FOOPS programs and database states as abstract data types (ADTs) in such a way that the queries and modifications to the database are encoded as functions of these data types. Essentially, this defines an operational semantics for FOOPS using the functional part of the language, which has a denotational semantics based on order sorted algebra [40] and an operational semantics given by order sorted rewriting [60]. In other words, this can be seen as providing a simulator for FOOPS written in OBJ. Using this approach, reasoning about FOOPS programs is essentially reduced to order sorted deduction [40], where part of the equational theory, given by the information stored in the database, changes with time. In this thesis we use a more abstract approach [78], which is also more appropriate for giving the semantics of concurrency, nondeterminism, dynamic object creation and deletion, dynamic binding, and other aspects of FOOPS not discussed in [39].

Currently, the most complete work on a sheaf theoretic semantics for FOOPS is [12], which also explores the relation between the sheaf theoretic semantics and the operational semantics described here and in [7]. In [12], only a subset of FOOPS is considered; in particular, inheritance and dynamic binding are not considered. Contrasting to the semantics presented here, it seems that the semantics of the atomicity operator cannot be defined in a simple and abstract way using sheaf theory. Also, the semantic description using sheaf theory is much longer than its equivalent operational semantics description. The advantage of using sheaf theory for defining the semantic of FOOPS seems to be the rich mathematical structure associated to sheafs. However, it is not clear yet how this can be used in a practical way; some research in this direction has been reported in [63].

A lot of effort has been done in order to give semantics for object oriented languages. Most of the work in this area defines semantics in terms of mathematical models based on set theory [18, 14], metric spaces [3], category theory [65, 19, 35], and hidden order sorted algebra [31, 28, 36, 35]. Exceptions are [15], which defines an assertional style proof system, and [88] and [59], which give the semantics in terms of a process algebra based on operational semantics [70]. By contrast, the semantics developed here is based on the simple frameworks of structural operational semantics [78] and OSA [40]. The work described in [20] is also based on operational semantics.

Because of the lack of a fully abstract mathematical model for interleaving, and the intrinsic details of the semantics of object identification and dynamic object creation and deletion, the framework of operational semantics seems to be adequate for specifying the semantics of a concurrent language like FOOPS. In fact, as demonstrated in this thesis, the semantics could be easily and concisely defined, being still possible to reason about it in a pragmatical way. Also, by using a process algebra to give the semantics of a language like FOOPS, one is not able to use the algebra, or its notions of equivalence and refinement, in order to reason about the semantics and programs; for that purpose, one actually would have to use the (operational) semantics associated to the process algebra—see [59], for example. So it seems more effective to define the semantics directly.

Refinement

Well established approaches for refinement and equivalence of sequential programs [71, 52, 4] 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 [78] and bisimulations [77]. Based on those frameworks, notions of equivalence have also been developed for process algebras [69, 74] and a general notation for giving the semantics of programming languages [72].

The notion of refinement introduced in this thesis is the first notion of refinement for FOOPS. In fact, just a few notions of refinement of concurrent object oriented programs have been proposed so far. Here we comment on two of those notions, concluding that none of them provide both a general definition of refinement and an effective proof technique such as the one presented here. Preliminary research on a third approach has been reported in [16].

An approach for refinement of object *based* concurrent programs is described in [58], 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 [55] are also used for proving refinement. However, no general definition of refinement is proposed; although the semantics of the language presented in [58] is given in terms of a process algebra [70], the notions of equivalence and refinement of the algebra cannot be directly used to define the notion of refinement of the language [59].

Another approach for refinement of object oriented concurrent programs is described in [61], 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.

Tools

The adequacy of OBJ for implementing prototypes of simulation tools has been discussed and confirmed in [85] and [75]. Here we follow a similar approach. In [12], the Eqlog language (an extension of OBJ with logic programming features) [37, 38] has been used in an interesting way to illustrate how programs written in a subset of FOOPS can be simulated, following the sheaf semantics described in [12].

OBJ is a quite effective tool for formal reasoning, as reported in several places [26, 30, 21]. Ideally, we should use OBJ for formally reasoning about refinement. However, it seems that some of the concepts of the refinement theory cannot be directly defined by equations as supported by OBJ. That is why we use the 2OBJ system for implementing the proof assistant.

8.3 Further Research

In order to further evaluate the work presented in this thesis, it would be necessary to formally develop large applications using FOOPS. This should give us more feedback on the efficiency of our approach and suggest modifications to the language, its semantics, and its notion of refinement.

FOOPS

After formally developing large applications using FOOPS, we would also be able to better compare this language with other object oriented specification languages [58, 11, 80, 65, 2, 61] and process algebras [51, 69, 5]. Moreover, we could then explore the suitability and expressiveness of the mechanism for object protection, and consider whether it should be extended. Here we just considered a few examples using this mechanism.

Operational Semantics

The operational semantics described in this thesis could be extended and revised as follows:

- In addition to DMAs and IMAs, it would be interesting to investigate the semantics of more general axioms; this might be useful for specifications in general.
- Perhaps it would be more appropriate to understand DMAs and IMAs' conditions as *pre-conditions*[1], rather than as *enabling conditions*[2]; this would allow a higher degree of under-specification in FOOPS specifications.
- It might be worth investigating the consequences of adopting a more restricted computational model for FOOPS; for example, this could be achieved by allowing only one method to be executed in an object at a given time, as adopted in [58] for instance.
- The semantics of dynamic binding of derived attributes could be specified by adding a rule for evaluation of derived attributes, similar to the rule for evaluation of methods specified by IMAs; however, note that attributes should be atomically evaluated.

[1] The meaning of an operation is undefined for a particular state if its related axioms have pre-conditions that are not valid in that state

[2] An operation blocks in a state where its related axioms have enabling conditions which are not valid in that state

- A complete semantics for updating of multi-argument stored attributes should be developed.

Refinement

Some aspects of our approach for refinement need to be further developed. First, we should use signature morphisms [21] for defining refinement, so that it would be possible to abstract from the names of operations that are used for making experiments, and the names used to identify objects.

As a natural extension of the framework presented in this thesis, we should provide techniques for reasoning about refinement without directly using the operational semantics and the definition of refinement. In fact, as suggested in several places [48, 51, 69, 58], it is desirable to have different proof techniques for refinement, even though the simulation technique is quite effective and often used to prove refinement of complex examples (see [69]). Indeed, algebraic reasoning is a very useful proof technique; so we should develop a set of algebraic laws (and refinement preserving transformation rules) for FOOPS operators, as illustrated in Section 5.2.3. Other possibilities are the following:

- Refinement rules in the style of [71].
- Derived compositionality properties, as illustrated in Section 6.4.
- Compositionality rules establishing how refinement of expressions can imply refinement of specifications.
- An assertional style proof system for FOOPS.

Those techniques could be developed and proved sound with respect to the definition of refinement introduced here.

We should also introduce rules indicating how the relation of refinement of functional level specifications interacts with the relation of refinement of object level specifications. This depends on the semantics of module composition, which was not considered here.

In this thesis we explored in detail a particular notion of refinement, so that it would give us confidence that other notions could be developed in a similar way. In fact, several notions of refinement should be available for formal development of programs in practice. In particular, we should provide notions of refinement that are easier to check and useful only for specific cases. For instance, this could be achieved by modifying our notion of simulation relation to consider only the input/output behaviour of programs, or to require one $\rightarrow$-transition from a state to be matched by one $\rightarrow$-transition from a related state; both options could be used for refinement of expressions to be atomically evaluated. Also, notions of refinement considering divergence [87] and the context where an expression is going to be used [62] could be defined by adapting the simulation technique described here. Other techniques for defining refinement, such as the one introduced in [48], could probably be applied to FOOPS as well. We chose the simulation approach mainly for its simplicity and effective proof technique.

We should also introduce notions of refinement considering aspects such as object protection and evaluation in the background. Those notions should be based on a representation of states that allows us to distinguish between expressions whose result can be observed from expressions whose result cannot be observed. This distinction could then be considered by an extended definition of simulation and refinement, where the results yielded by some expressions would not be compared. In particular, this would be useful for reasoning about hidden operations, when we are only interested in analysing the effects generated by expressions.

In general, the constraints on experiments enforced by our definition of refinement are too restrictive, since visible methods must be atomic. We should try to develop a notion of refinement that enforces weaker constraints on experiments (such as the ones adopted in [58]), but still offering simple proof techniques.

Tools

The tools described in this thesis are really prototypes. In order to be used in practice, those tools should be more efficient, provide more features, and offer a better user interface. The simulator should be extended to support a few FOOPS' features which have not been implemented, such as the mechanism for object protection. The proof assistant should be extended to provide an encoding of some aspects of the refinement theory, such as the refinement relation on functional specifications and the inference rules of the operational semantics. We could also easily extend the proof assistant to support formal reasoning with the laws of FOOPS combiners and compositionality properties. The proof assistant should also provide tactics for mostly automating refinement proofs.

Ideally, the correctness of the tools should be proved, so that they could be used for formal development of software in practice. Aspects of the correctness of the simulator with respect to the operational semantics were discussed in Section 7.1.6. Provided we assume the correctness of the LISP [89] code that implements some of the simulator's functionality, a proof of correctness of the simulator should not be difficult since its definition was directly derived from the inference rules of the operational semantics and OBJ, with its declarative style and simple semantics, is used for most of its implementation. The correctness of the proof assistant essentially depends on the correctness of the encoding of the refinement theory; we would then have to verify the validity of the inference rules described in this thesis.

Appendix A

Deferred Proofs

Here we give the proofs of some results stated in Chapter 6.

A.1 Composition of Experiments

Lemma 6.2.2 (restated) Let $cexp \in \mathcal{CE}(\mathsf{P})$ and $P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q$. For any $cexpp$ and $cexpq$ such that $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$ and $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$, if

$$\langle cexpq,\, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}}$$

then, for some $\langle cexpp',\, \mathcal{P}'\rangle$ and $cexp' \in \mathcal{CE}(\mathsf{P})$,

$$\langle cexpp,\, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp',\, \mathcal{P}'\rangle \not\rightarrow_{\mathsf{P}},$$

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

$$\mathcal{FE}_{\mathsf{P}}(cexpp') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(cexpq),$$

$$\mathcal{FE}_{\mathsf{P}}(cexpp') \text{ implies } cexpp' =_{(\mathsf{P},\mathsf{Q})} cexpq,$$

$$cexpp' \in cexp' \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpq \in cexp' \rightrightarrows^*_{\mathsf{Q}}, \text{ and}$$

$$\mathcal{D} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q} \text{ implies } \langle cexpp',\, \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}},$$

for any $\mathcal{D}$.

Proof: The proof follows by structural induction on $cexp$. Let us assume that

$$\langle cexpq,\, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}}\ .$$

We then analyse the possible forms of $cexp \in \mathcal{CE}(\mathsf{P})$:

Case 1 Let $cexp \in \mathcal{T}_{F(\mathsf{P})}$, where $LS(cexp) \in PS(\mathsf{P})$ and $cexp \equiv [\![\, cexp\,]\!]_{FE(\mathsf{P})}$. Hence

$$\mathcal{FE}_{\mathsf{P}}(cexp).$$

Then, as $P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q$, we have $\mathsf{P} \leq_{PS(\mathsf{P})} \mathsf{Q}$ and then

$$\mathcal{FE}_{\mathsf{Q}}(cexp).$$

So, as $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$ and $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$, we have

$$cexpp \equiv cexpq \equiv cexp.$$

Thus we have

$$\mathcal{FE}_{\mathsf{P}}(cexpp).$$

It is then easy to check that, for any $\mathcal{D}$,

$$\langle cexpp, \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}} .$$

Therefore the result follows by chosing $\langle cexpp', \mathcal{P}'\rangle$ to be $\langle cexpp, \mathcal{P}\rangle$ and $cexp'$ to be $cexp$, since $\rightarrow^*$ is reflexive, and $cexpp$ and $cexpq$ are the same fully evaluated expression of a primary sort of P.

Case 2 Let $cexp \in I(\mathsf{P})$. Hence $\mathcal{FE}_{\mathsf{P}}(cexp)$ and, as $P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q$, $I(\mathsf{P}) \subseteq I(\mathsf{Q})$ and $cexp \in I(\mathsf{Q})$. Therefore $\mathcal{FE}_{\mathsf{Q}}(cexp)$. The proof then follows as in case 1, by verifying that $cexpp$ and $cexpq$ are the same object identifier.

Case 3 Let $cexp \in \mathcal{E}(\mathsf{P})$. So, as $P \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q$, we have $\mathcal{E}(\mathsf{P}) \subseteq \mathcal{E}(\mathsf{Q})$ and

$$cexp \in \mathcal{E}(\mathsf{Q}).$$

Thus, as $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$ and $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$, we have

$$cexp \rightrightarrows^*_{\mathsf{Q}} cexpq \text{ and } cexp \rightrightarrows^*_{\mathsf{P}} cexpp.$$

So, by Lemma 6.1.4, for some $\langle cexpp', \mathcal{P}'\rangle$, we have

$$\langle cexpp, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp', \mathcal{P}'\rangle \not\rightarrow_{\mathsf{P}},$$

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

$$\mathcal{FE}_{\mathsf{P}}(cexpp') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(cexpq),$$

$$\mathcal{FE}_{\mathsf{P}}(cexpp') \text{ implies } cexpp' =_{(\mathsf{P},\mathsf{Q})} cexpq, \text{ and}$$

$$\neg\mathcal{FE}_{\mathsf{P}}(cexpp') \text{ implies } cexp \rightrightarrows^*_{\mathsf{P}} cexpp'.$$

We split the proof in two cases. First let $\mathcal{FE}_{\mathsf{P}}(cexpp')$. Hence

$$cexpp' =_{(\mathsf{P},\mathsf{Q})} cexpq.$$

Also, it is easy to check that, for any $\mathcal{D}$,

$$\langle cexpp', \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}} .$$

Therefore the result follows by chosing $cexp'$ to be $cexpp'$, since, as $cexp \rightrightarrows^*_{\mathsf{P}} cexpp$, it is easy to check

$$\langle cexp, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp, \mathcal{P}\rangle,$$

and, by transitivity,

$$\langle cexp, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp', \mathcal{P}'\rangle.$$

Hence $cexpp'$ and $cexpq$ are either the same object identifier or the same fully evaluated functional expression of a primary sort of P.

Second let us consider that $\neg\mathcal{FE}_{\mathsf{P}}(cexpp')$. Hence

$$cexp \rightrightarrows^*_{\mathsf{P}} cexpp' \text{ and } \neg\mathcal{FE}_{\mathsf{Q}}(cexpq).$$

Let $\mathcal{D} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q}$, for an arbitrary $\mathcal{D}$. Then it is easy to check

$$\langle cexp, \mathcal{D}\rangle \rightrightarrows^*_{\mathsf{P}} \langle cexpp', \mathcal{D}\rangle.$$

Now let us assume it is not the case that $\langle cexpp', \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}}$. So let

$$\langle cexp, \mathcal{D}\rangle \rightrightarrows^*_{\mathsf{P}} \langle cexpp', \mathcal{D}\rangle \rightarrow_{\mathsf{P}} O,$$

for an arbitrary O. As $\langle cexpp', \mathcal{P}'\rangle \not\rightarrow_{\mathsf{P}}$, it is easy to check

$$cexpp' \not\rightrightarrows_{\mathsf{P}} .$$

Hence the following easily follows:

$$\langle cexpp', \mathcal{D}\rangle \not\rightrightarrows_{\mathsf{P}} O.$$

So, by Lemma 6.1.2, we have

$$\langle cexp, \mathcal{D}\rangle \rightrightarrows^*_{\mathsf{P}} \langle cexpp', \mathcal{D}\rangle \rightarrow_{\mathsf{P}} O \not\rightarrow_{\mathsf{P}} \text{ and } \mathcal{FE}_{\mathsf{P}}(O).$$

But then, as $cexp \rightrightarrows^*_{\mathsf{Q}} cexpq$, we have

$$\langle cexp, \mathcal{Q}\rangle \rightarrow_{\mathsf{Q}} \langle cexpq, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}},$$

contradicting the fact that $\mathcal{D} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q}$, since $\langle cexpq, \mathcal{Q}\rangle$ cannot be a simulation of O because $\mathcal{FE}_{\mathsf{P}}(O)$ whereas $\neg\mathcal{FE}_{\mathsf{Q}}(cexpq)$. Therefore we must conclude that $\langle cexpp', \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}}$. Then the result follows by chosing $cexp'$ to be $cexp$.

Case 4 Let $cexp \equiv cexp_1 \mid\mid cexp_2$, where $cexp_1, cexp_2 \in \mathcal{CE}(\mathsf{P})$. Hence, as $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$, we have

$$cexpq \equiv cexpq_1 \mid\mid cexpq_2,$$

for some $cexpq_1$ and $cexpq_2$ such that

$$cexpq_1 \in cexp_1 \rightrightarrows^*_{\mathsf{Q}} \text{ and } cexpq_2 \in cexp_2 \rightrightarrows^*_{\mathsf{Q}} .$$

Similarly, as $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$, we have

$$cexpp \equiv cexpp_1 \mid\mid cexpp_2,$$

for some $cexpp_1$ and $cexpp_2$ such that

$$cexpp_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpp_2 \in cexp_2 \rightrightarrows^*_{\mathsf{P}} .$$

As $\langle cexpq_1 \;||\; cexpq_2,\, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}}$, by Rules 4.4.3, 4.4.4, 4.4.5, and 4.4.6 we have

$$\begin{array}{ll} \langle cexpq_1,\, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}}, & \langle cexpq_2,\, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}}, \\ \neg\mathcal{FE}_{\mathsf{Q}}(cexpq_1), & \text{and } \neg\mathcal{FE}_{\mathsf{Q}}(cexpq_2). \end{array}$$

So, by induction, for some $\langle cexpp'_1,\, \mathcal{P}'\rangle$ and $cexp'_1 \in \mathcal{CE}(\mathsf{P})$, we have

$$\langle cexpp_1,\, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp'_1,\, \mathcal{P}'\rangle \not\rightarrow_{\mathsf{P}},$$
$$\langle p,\, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle,$$
$$\mathcal{FE}_{\mathsf{P}}(cexpp'_1) \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(cexpq_1),$$
$$cexpp'_1 \in cexp'_1 \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpq_1 \in cexp'_1 \rightrightarrows^*_{\mathsf{Q}}, \text{ and}$$
$$\mathcal{D} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q} \text{ implies } \langle cexpp'_1,\, \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}},$$

for any $\mathcal{D}$. Hence

$$\neg\mathcal{FE}_{\mathsf{P}}(cexpp'_1).$$

By induction again, for some $\langle cexpp'_2,\, \mathcal{P}''\rangle$ and $cexp'_2 \in \mathcal{CE}(\mathsf{P})$, we have

$$\langle cexpp_2,\, \mathcal{P}'\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp'_2,\, \mathcal{P}''\rangle \not\rightarrow_{\mathsf{P}},$$
$$\langle p,\, \mathcal{P}''\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle,$$
$$\mathcal{FE}_{\mathsf{P}}(cexpp'_2) \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(cexpq_2),$$
$$cexpp'_2 \in cexp'_2 \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpq_2 \in cexp'_2 \rightrightarrows^*_{\mathsf{Q}}, \text{ and}$$
$$\mathcal{D} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q} \text{ implies } \langle cexpp'_2,\, \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}},$$

for any $\mathcal{D}$. Hence

$$\neg\mathcal{FE}_{\mathsf{P}}(cexpp'_2).$$

By Rules 4.4.3 and 4.4.4 it is easy to check

$$\langle cexpp_1 \;||\; cexpp_2,\, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp'_1 \;||\; cexpp_2,\, \mathcal{P}'\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp'_1 \;||\; cexpp'_2,\, \mathcal{P}''\rangle.$$

Lastly, let $\mathcal{D} \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q}$. So,

$$\langle cexpp'_1,\, \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}} \text{ and } \langle cexpp'_2,\, \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}}\ .$$

Hence, by Rules 4.4.3, 4.4.4, 4.4.5, and 4.4.6, we have

$$\langle cexpp'_1 \;||\; cexpp'_2,\, \mathcal{D}\rangle \not\rightarrow_{\mathsf{P}}\ .$$

Therefore the result follows by taking $\langle cexpp',\, \mathcal{P}'\rangle$ to be $\langle cexpp'_1 \;||\; cexpp'_2,\, \mathcal{P}''\rangle$ and $cexp'$ to be $cexp'_1 \;||\; cexp'_2$, since $\mathcal{P}'' \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \mathcal{Q}$ and $cexpp'_1 \;||\; cexpp'_2$ and $cexpq_1 \;||\; cexpq_2$ are clearly not fully evaluated.

Case 5 The proof follows in a similar way to case 4 when $cexp$ is in one of the following forms: $cexp_1$; $cexp_2$, `if` $cexp_1$ `then` $cexp_2$ `else` $cexp_3$ `fi`, or `result` $cexp_1$; $cexp_2$, for $cexp_1, cexp_2, cexp_3 \in \mathcal{CE}(\mathsf{P})$. Contrasting to case 4, in some of those cases, it will be enough to apply induction just once. We omit the details here.

□

A.2 The Result Method Combiner

Lemma 6.3.13 (restated) Let $\mathcal{S}$ be the relation consisting of $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ and all pairs in the forms

$$(\langle \texttt{result } cexpp \ ; \ p, \mathcal{P}\rangle \quad , \quad \langle \texttt{result } cexpq \ ; \ q, \mathcal{Q}\rangle), \text{ and}$$
$$(\langle \texttt{result } p \ ; \ cexpp, \mathcal{P}\rangle \quad , \quad \langle \texttt{result } q \ ; \ cexpq, \mathcal{Q}\rangle),$$

where $\langle p, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}\rangle$ and there is a $cexp \in \mathcal{CE}(\mathsf{P})$ such that $cexpp \in cexp \rightrightarrows^*_{\mathsf{P}}$ and $cexpq \in cexp \rightrightarrows^*_{\mathsf{Q}}$. Then $\mathcal{S}$ is a (P, Q)-simulation.

Proof: As $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ is included in $\mathcal{S}$, using Lemma 5.1.6 we can easily conclude that pairs related by $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ satify the simulation conditions. We concentrate on pairs in the following form:

$$(\langle \texttt{result } p \ ; \ cexpp, \mathcal{P}\rangle, \langle \texttt{result } q \ ; \ cexpq, \mathcal{Q}\rangle).$$

The proof follows in a similar way for pairs in the first form. First let $\langle \texttt{result } q \ ; \ cexpq, \mathcal{Q}\rangle \rightarrow_{\mathsf{Q}} \langle e, \mathcal{Q}'\rangle$. Thus we consider the different cases justifying this transition.

Case 1 By Rule 4.4.12, where $e \equiv \texttt{result } q' \ ; \ cexpq$ and $\langle q, \mathcal{Q}\rangle \rightarrow_{\mathsf{Q}} \langle q', \mathcal{Q}'\rangle$. Then, for some $\langle p', \mathcal{P}'\rangle$, by Lemma 5.1.6, we have

$$\langle p, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle p', \mathcal{P}'\rangle \text{ and } \langle p', \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q', \mathcal{Q}'\rangle.$$

Hence, by Rule 4.4.12, it is easy to check

$$\langle \texttt{result } p \ ; \ cexpp, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle \texttt{result } p' \ ; \ cexpp, \mathcal{P}'\rangle.$$

Therefore the result follows since

$$\langle \texttt{result } p' \ ; \ cexpp, \mathcal{P}'\rangle$$

is clearly related by $\mathcal{S}$ to $\langle \texttt{result } q' \ ; \ cexpq, \mathcal{Q}'\rangle$.

Case 2 By Rule 4.4.13, where $e \equiv \texttt{result } q \ ; \ cexpq'$, $\mathcal{FE}_{\mathsf{Q}}(q)$, and $\langle cexpq, \mathcal{Q}\rangle \rightarrow_{\mathsf{Q}} \langle cexpq', \mathcal{Q}'\rangle$. Hence it is easy to check

$$\langle q, \mathcal{Q}\rangle \not\rightarrow_{\mathsf{Q}} .$$

Then, by Lemma 5.1.6, for some $\langle p', \mathcal{P}'\rangle$,

$$\langle p, \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}} \langle p', \mathcal{P}'\rangle \not\rightarrow_{\mathsf{P}},$$
$$\langle p', \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}\rangle, \text{ and}$$
$$\mathcal{FE}_{\mathsf{P}}(p') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(q).$$

So we have

$$\mathcal{FE}_{\mathsf{P}}(p').$$

Also, by Lemma 6.2.1, for some $\langle cexpp', \mathcal{P}''\rangle$ and $cexp' \in \mathcal{CE}(\mathsf{P})$, we have

$$\langle cexpp, \mathcal{P}'\rangle \rightarrow^*_{\mathsf{P}} \langle cexpp', \mathcal{P}''\rangle,$$
$$\langle p', \mathcal{P}''\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q, \mathcal{Q}'\rangle,$$
$$cexpp' \in cexp' \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpq' \in cexp' \rightrightarrows^*_{\mathsf{Q}} .$$

Hence, by Rules 4.4.12 and 4.4.13, it is easy to check

$$\begin{aligned}&\langle \texttt{result}\ p\ ;\ cexpp,\ \mathcal{P}\rangle\\ &\qquad \to_{\mathsf{P}}^{*} \langle \texttt{result}\ p'\ ;\ cexpp,\ \mathcal{P}'\rangle\\ &\qquad \to_{\mathsf{P}}^{*} \langle \texttt{result}\ p'\ ;\ cexpp',\ \mathcal{P}''\rangle.\end{aligned}$$

Therefore the result follows since

$$\langle \texttt{result}\ p'\ ;\ cexpp',\ \mathcal{P}''\rangle$$

is clearly related by $\mathcal{S}$ to $\langle \texttt{result}\ q\ ;\ cexpq',\ \mathcal{Q}'\rangle$.

Case 3 By Rule 4.4.14, where $e \equiv q$, $\mathcal{Q} \equiv \mathcal{Q}'$, $\mathcal{FE}_{\mathsf{Q}}(q)$, and $\mathcal{FE}_{\mathsf{Q}}(cexpq)$. Hence it is easy to check

$$\langle q,\ \mathcal{Q}\rangle \not\to_{\mathsf{Q}} \text{ and } \langle cexpq,\ \mathcal{Q}\rangle \not\to_{\mathsf{Q}}\ .$$

Then, by Lemma 5.1.6, for some $\langle p',\ \mathcal{P}'\rangle$,

$$\begin{gathered}\langle p,\ \mathcal{P}\rangle \to_{\mathsf{P}}^{*} \langle p',\ \mathcal{P}'\rangle \not\to_{\mathsf{P}},\\ \langle p',\ \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\ \mathcal{Q}\rangle,\\ \mathcal{FE}_{\mathsf{P}}(p') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(q), \text{ and}\\ \mathcal{FE}_{\mathsf{P}}(p') \Rightarrow p' =_{(\mathsf{P},\mathsf{Q})} q.\end{gathered}$$

So we have

$$\mathcal{FE}_{\mathsf{P}}(p') \text{ and } p' =_{(\mathsf{P},\mathsf{Q})} q.$$

Also, by Lemma 6.2.2, for some $\langle cexpp',\ \mathcal{P}''\rangle$ and $cexp' \in \mathcal{CE}(\mathsf{P})$, we have

$$\begin{gathered}\langle cexpp,\ \mathcal{P}'\rangle \to_{\mathsf{P}}^{*} \langle cexpp',\ \mathcal{P}''\rangle \not\to_{\mathsf{P}},\\ \langle p',\ \mathcal{P}''\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\ \mathcal{Q}\rangle,\\ \mathcal{FE}_{\mathsf{P}}(cexpp') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(cexpq),\\ cexpp' \in cexp' \rightrightarrows_{\mathsf{P}}^{*} \text{ and } cexpq' \in cexp' \rightrightarrows_{\mathsf{Q}}^{*}\ .\end{gathered}$$

Thus we have

$$\mathcal{FE}_{\mathsf{P}}(cexpp').$$

Hence, by Rules 4.4.12, 4.4.13, and 4.4.14 it is easy to check

$$\begin{aligned}&\langle \texttt{result}\ p\ ;\ cexpp,\ \mathcal{P}\rangle\\ &\qquad \to_{\mathsf{P}}^{*} \langle \texttt{result}\ p'\ ;\ cexpp,\ \mathcal{P}'\rangle\\ &\qquad \to_{\mathsf{P}}^{*} \langle \texttt{result}\ p'\ ;\ cexpp',\ \mathcal{P}''\rangle\\ &\qquad \to_{\mathsf{P}} \langle p',\ \mathcal{P}''\rangle.\end{aligned}$$

Therefore the result follows since, as $\mathcal{S}$ includes $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$, $\langle p',\ \mathcal{P}''\rangle$ is related by $\mathcal{S}$ to $\langle q,\ \mathcal{Q}\rangle$ (see the proof of Lemma 6.1.5).

This concludes the case analysis. Now let $\langle \texttt{result}\ q\ ;\ cexpq,\ \mathcal{Q}\rangle \not\to_{\mathsf{Q}}$. Hence, by Rules 4.4.12, 4.4.13, and 4.4.14, we have

$$\neg\mathcal{FE}_{\mathsf{Q}}(q) \text{ and } \langle q,\ \mathcal{Q}\rangle \not\to_{\mathsf{Q}}$$

or

$$\mathcal{FE}_{\mathsf{Q}}(q), \neg\mathcal{FE}_{\mathsf{Q}}(cexpq), \text{ and } \langle cexpq,\ \mathcal{Q}\rangle \not\to_{\mathsf{Q}}\ .$$

We split the proof in two cases. First let $\neg\mathcal{FE}_{\mathsf{Q}}(q)$ and $\langle q,\ \mathcal{Q}\rangle \not\to_{\mathsf{Q}}$. Then, by Lemma 5.1.6, for some $\langle p',\ \mathcal{P}'\rangle$, we have

$$\langle p,\ \mathcal{P}'\rangle \to^*_{\mathsf{P}} \langle p',\ \mathcal{P}'\rangle \not\to_{\mathsf{P}},$$

$$\langle p',\ \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\ \mathcal{Q}\rangle, \text{ and}$$

$$\mathcal{FE}_{\mathsf{P}}(p') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(q).$$

So we have

$$\neg\mathcal{FE}_{\mathsf{P}}(p').$$

So, by Rules 4.4.12, 4.4.13, and 4.4.14, we have

$$\langle \texttt{result}\ p\ ;\ cexpp,\ \mathcal{P}\rangle \to^*_{\mathsf{P}} \langle \texttt{result}\ p'\ ;\ cexpp,\ \mathcal{P}'\rangle \not\to_{\mathsf{P}}.$$

Therefore the result easily follows since $\texttt{result}\ p'\ ;\ cexpp$ and $\texttt{result}\ q\ ;\ cexpq$ are clearly not fully evaluated, and $\langle \texttt{result}\ p'\ ;\ cexpp,\ \mathcal{P}'\rangle$ is related by $\mathcal{S}$ to $\langle \texttt{result}\ q\ ;\ cexpq,\ \mathcal{Q}\rangle$.

Second let $\mathcal{FE}_{\mathsf{Q}}(q)$, $\neg\mathcal{FE}_{\mathsf{Q}}(cexpq)$, and $\langle cexpq,\ \mathcal{Q}\rangle \not\to_{\mathsf{Q}}$. Hence it is easy to check

$$\langle q,\ \mathcal{Q}\rangle \not\to_{\mathsf{Q}}\ .$$

Then, by Lemma 5.1.6, for some $\langle p',\ \mathcal{P}'\rangle$,

$$\langle p,\ \mathcal{P}\rangle \to^*_{\mathsf{P}} \langle p',\ \mathcal{P}'\rangle \not\to_{\mathsf{P}},$$

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

$$\mathcal{FE}_{\mathsf{P}}(p') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(q), \text{ and}$$

$$\mathcal{FE}_{\mathsf{P}}(p') \Rightarrow p' =_{(\mathsf{P},\mathsf{Q})} q.$$

So we have

$$\mathcal{FE}_{\mathsf{P}}(p') \text{ and } p' =_{(\mathsf{P},\mathsf{Q})} q.$$

Also, by Lemma 6.2.2, for some $\langle cexpp',\ \mathcal{P}''\rangle$ and $cexp' \in \mathcal{CE}(\mathsf{P})$, we have

$$\langle cexpp,\ \mathcal{P}'\rangle \to^*_{\mathsf{P}} \langle cexpp',\ \mathcal{P}''\rangle \not\to_{\mathsf{P}},$$

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

$$\mathcal{FE}_{\mathsf{P}}(cexpp') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(cexpq),$$

$$cexpp' \in cexp' \rightrightarrows^*_{\mathsf{P}} \text{ and } cexpq \in cexp' \rightrightarrows^*_{\mathsf{Q}}\ .$$

Thus we have

$$\neg\mathcal{FE}_{\mathsf{P}}(cexpp').$$

Hence, by Rules 4.4.12, 4.4.13, and 4.4.14 it is easy to check

$$\begin{aligned}&\langle \texttt{result}\ p\ ;\ cexpp,\ \mathcal{P}\rangle\\ &\quad \rightarrow^*_{\mathsf{P}} \langle \texttt{result}\ p'\ ;\ cexpp,\ \mathcal{P}'\rangle\\ &\quad \rightarrow^*_{\mathsf{P}} \langle \texttt{result}\ p'\ ;\ cexpp',\ \mathcal{P}''\rangle \not\rightarrow_{\mathsf{P}}\ .\end{aligned}$$

Therefore the result easily follows since $\texttt{result}\ p'\ ;\ cexpp'$ and $\texttt{result}\ q\ ;\ cexpq$ are clearly not fully evaluated, and it is easy to check that

$$\langle \texttt{result}\ p'\ ;\ cexpp',\ \mathcal{P}''\rangle$$

is related by $\mathcal{S}$ to $\langle \texttt{result}\ q\ ;\ cexpq,\ \mathcal{Q}\rangle$.

Finally let $\langle exp,\ \mathcal{Q}\rangle \rightarrow^*_{\mathsf{Q}}\ Q' \not\rightarrow_{\mathsf{Q}}$, for any $exp \in \mathcal{E}(\mathsf{P})$. Hence, as $\langle p,\ \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\ \mathcal{Q}\rangle$, by Lemma 5.1.6, for some P', we have

$$\begin{aligned}&\langle exp,\ \mathcal{P}\rangle \rightarrow^*_{\mathsf{P}}\ P' \not\rightarrow_{\mathsf{P}},\\ &P' \sqsubseteq_{(\mathsf{P},\mathsf{Q})} Q',\ \text{and}\\ &\langle p,\ \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\mathsf{Q})} \langle q,\ \mathcal{Q}'\rangle.\end{aligned}$$

Therefore the result easily follows since $\mathcal{S}$ includes $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$, and it is easy to check that

$$\langle \texttt{result}\ p\ ;\ cexpp,\ \mathcal{P}'\rangle$$

is related by $\mathcal{S}$ to $\langle \texttt{result}\ q\ ;\ cexpq,\ \mathcal{Q}'\rangle$. $\Box$

Bibliography

[1] Antonio Alencar and Joseph Goguen. OOZE: An object-oriented Z environment. In Pierre America, editor, *European Conference on Object Oriented Programming*. Springer-Verlag, 1991. Lecture Notes in Computer Science, Volume 512.

[2] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1:366–411, 1989.

[3] Pierre America and J.J.M.M. Rutten. A parallel object-oriefnted language: Design and semantic foundations. Technical Report CS-R8953, Centre for Mathematics and Computer Science, 1989.

[4] 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.

[5] Jan Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60:190–137, 1984.

[6] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition, 1994.

[7] Paulo Borba and Joseph Goguen. An operational semantics for FOOPS. Technical Monograph PRG-115. Oxford University, Computing Laboratory, Programming Research Group. November, 1994. An earlier version appeared as PRG-TR-16-94. A shorter version appeared in Working papers of the International Workshop on Information Systems—Correctness and Reusability, IS-CORE'94, edited by Roel Wieringa and Remco Feenstra, Technical Report IR-357, Vrije Universiteit, Amsterdam, September, 1994.

[8] Paulo Borba and Joseph Goguen. Refinement of concurrent object oriented programs. Technical Report PRG-TR-18-95, Oxford University, Computing Laboratory, Programming Research Group, March 1995. A draft version appeared as "On Refinement and FOOPS", PRG-TR-17-94. To appear in the proceedings of the BCS/FACS workshop on formal aspects of object-oriented programming, London, December 1993.

[9] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.

[10] Rod Burstall and Joseph Goguen. An informal introduction to specifications using Clear. In Robert Boyer and J Moore, editors, *The Correctness Problem in Computer Science*, pages 185–213. Academic, 1981. Reprinted in *Software Specification Techniques*, Narain Gehani and Andrew McGettrick, editors, Addison-Wesley, 1985, pages 363–390.

[11] David Carrington et al. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Proceedings of FORTE'89*, pages 281–296. North-Holland, 1990.

[12] Corina Cîrstea. A distributed semantics for FOOPS. To appear, 1995. Oxford University, Computing Laboratory, Programming Research Group.

[13] Stephen Coffin. *UNIX System V, Release 4, the Complete Reference*. Osborne–McGraw-Hill, 1991.

[14] Stefan Conrad. Developing model-theory for compositional specification of object communities. Technische Universität Braunschweig, Informatik, Abt. Datenbanken, Germany. To appear.

[15] Frank de Boer. *Reasoning about dynamically evolving process structures —A proof theory for the parallel object-oriented language POOL*. PhD thesis, Vrije Universiteit, Amsterdam, 1991.

[16] Grit Denker and Hans-Dieter Ehrich. Action reification in object-oriented specifications. In Roel Wieringa and Remco Feenstra, editors, *Working papers of the International Workshop on Information Systems—Correctness and Reusability, IS-CORE'94*. Vrije Universiteit, Amsterdam, September 1994.

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

[18] David Duke and Roger Duke. Towards a semantics for Object-Z. In Dines Bjorner, C.A.R. Hoare, and Hans Langmaack, editors, *Proceedings, VDM'90: VDM and Z—Formal Methods in Software Development*, pages 242–262. Springer-Verlag, 1990. Lecture Notes in Computer Science, Volume 428.

[19] Hans-Dieter Ehrich, Joseph Goguen, and Amilcar Sernadas. A categorial theory of objects as observed processes. In J.W. de Bakker, Willem P. de Roever, and Gregorz Rozenberg, editors, *Foundations of Object Oriented Languages*, pages 203–228. Springer-Verlag, 1991. Lecture Notes in Computer Science, Volune 489; Proceedings, REX/FOOL Workshopfli, Noordwijkerhout, the Netherlands, May/June 1990.

[20] Martin Gogolla, Stefan Conrad, and Rudolf Herzig. Sketching concepts and computational model of TROLL *light*. Technische Universität Braunschweig, Informatik, Abt. Datenbanken, Germany. To appear.

[21] Joseph Goguen. *Theorem Proving and Algebra*. Draft version. Oxford University, Computing Laboratory, Programming Research Group.

[22] Joseph Goguen. Order sorted algebra. Technical Report 14, UCLA Computer Science Department, 1978. Semantics and Theory of Computation Series.

[23] Joseph Goguen. Parameterized programming. *Transactions on Software Engineering*, SE–10(5):528–543, September 1984.

[24] Joseph Goguen. Suggestions for using and organizing libraries in software development. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, First International Conference on Supercomputing Systems*, pages 349–360. IEEE Computer Society, 1985. Also in *Supercomputing Systems*, Steven and Svetlana Kartashev, Eds., Elsevier, 1986.

[25] Joseph Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, February 1986. Reprinted in *Tutorial: Software Reusability*, Peter Freeman, editor, IEEE Computer Society, 1987, pages 251–263, and in *Domain Analysis and Software Systems Modelling*, Ruben Prieto-Diaz and Guillermo Arango, editors, IEEE Computer Society, 1991, pages 125–137.

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

[27] 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.

[28] Joseph Goguen. An algebraic approach to refinement. In Dines Bjorner, C.A.R. Hoare, and Hans Langmaack, editors, *Proceedings, VDM'90: VDM and Z—Formal Methods in Software Development*, pages 12–28. Springer-Verlag, 1990. Lecture Notes in Computer Science, Volume 428.

[29] Joseph Goguen. Hyperprogramming: A formal approach to software environments. In *Proceedings, Symposium on Formal Approaches to Software Environment Technology.* Joint System Development Corporation, Tokyo, Japan, January 1990.

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

[31] Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991.

[32] Joseph Goguen. Sheaf semantics for concurrent interacting objects. *Mathematical Structures in Computer Science*, 11:159–191, 1992.

[33] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992. Draft appears as Report ECS-LFCS-90-106, Computer Science Department, University of Edinburgh, January 1990; an early ancestor is "Introducing Institutions," in *Proceedings, Logics of Programming Workshop*, Edward Clarke and Dexter Kozen, Eds., Springer Lecture Notes in Computer Science, Volume 164, pages 221–256, 1984.

[34] Joseph Goguen and Răzvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, to appear.

[35] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In *Proceedings, Tenth Workshop on Abstract Data Types*. Springer, to appear 1993.

[36] 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.

[37] Joseph Goguen and José Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179–210, September 1984.

[38] Joseph Goguen and José Meseguer. Models and equality for logical programming. In Hartmut Ehrig, Giorgio Levi, Robert Kowalski, and Ugo Montanari, editors, *Proceedings, 1987 TAPSOFT*, pages 1–22. Springer, 1987. Lecture Notes in Computer Science, Volume 250.

[39] 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.

[40] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 2(105), 1992.

[41] Joseph Goguen and Adolfo Socorro. Module composition and system design for the object paradigm. *Journal of Object-Oriented Programming*, 1994.

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

[43] Joseph Goguen and Joseph Tardo. An introduction to OBJ: A language for writing and testing software specifications. In Marvin Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189. IEEE, 1979. Reprinted in *Software Specification Techniques*, Nehan Gehani and Andrew McGettrick, editors, Addison Wesley, 1985, pages 391–420.

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

[45] 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.

[46] 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.

[47] Ian Hayes. *Specification Case Studies.* Prentice-Hall, 1987.

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

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

[50] C.A.R. Hoare. Programming: Sorcery or science? *IEEE Software*, April 1984.

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

[52] 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.

[53] John Hogg. Islands: Aliasing protection in object-oriented languages. In *[76]*, 1991.

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

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

[56] Cliff Jones. *Systematic Software Development using VDM.* Prentice-Hall, second edition, 1990.

[57] Cliff Jones. Does the O-O community need formal methods? In *TOOLS4: Technology of O-O Languages and Systems*, pages 15–22. Prentice Hall, 1991.

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

[59] 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.

[60] Claude Kirchner, Hélène Kirchner, and José Meseguer. Operational semantics of OBJ3. In T. Lepistö and Aarturo Salomaa, editors, *Proceedings, 15th International Colloquium on Automata, Languages and Programming, Tampere, Finland, July 11-15, 1988*, pages 287–301. Springer-Verlag, 1988. Lecture Notes in Computer Science, Volume 317.

[61] 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.

[62] Kim Larsen and Robin Milner. A compositional protocol verification using relativized bisimulation. *Information and Computation*, 99:80–108, 1992.

[63] Grant Malcolm. Interconnections of object specifications. To appear in the proceedings of the BCS/FACS workshop on formal aspects of object-oriented programming, London, December 1993. Oxford University, Computing Laboratory, Programming Research Group.

[64] Grant Malcolm. A sheaf semantics for FOOPS. To appear, 1995. Oxford University, Computing Laboratory, Programming Research Group.

[65] José Meseguer. A logical theory of concurrent objects. In *Proceedings of ECOOP-OOPSLA90 Conference on Object Oriented Programming*, pages 101–115. ACM, 1990.

[66] Bertrand Meyer. On formalism in specifications. *IEEE Software*, January 1985.

[67] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

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

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

[70] 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.

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

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

[73] Ellen Munthe-Kaas, Joseph Goguen, and José Meseguer. Method expressions and default values for object-valued attributes. SRI International, Computer Science Laboratory. Draft Version, 1989.

[74] Elie Najm and Jean-Bernard Stefani. Object-based concurrency: a process calculus analysis. In S. Abramsky and T.S.E. Maibaum, editors, *TAPSOFT'91, Theory and Practice of Software Development*, volume 494 (1) of *Lecture Notes in Computer Science*, pages 359–380. Springer-Verlag, 1991.

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

[76] Andreas Paepcke, editor. *OOPSLA'91.* ACM, ACM Press, November 1991.

[77] 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.

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

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

[80] Gunter Saake, Thorsten Hartman, Ralf Jungclaus, and Hans-Dieter Ehrich. Object-oriented design of information systems: TROLL language features. Technische Universität Braunschweig, Informatik, Abt. Datenbanken, Germany. To appear.

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

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

[83] Andrew Stevens, Keith Hobley, and Joseph Goguen. Mechanised theorem proving with 2OBJ: A tutorial introduction. Technical report, Oxford University, Computing Laboratory, Programming Research Group. To appear.

[84] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1991.

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

[86] Kim Waldén and Jean-Marc Nerson. *Seamless Object-Oriented Software Architectures—Analysis and Design of Reliable Systems.* Prentice Hall, 1995.

[87] David Walker. Bisimulation and divergence. In *Proceedings of Logic in Computer Science*, 1988.

[88] David Walker. π-Calculus semantics of object-oriented programming languages. In *TACS'91 – Proceedings of the international Conference on Theoretical Aspects of Computer Science*, volume 526 of *Lecture Notes in Computer Science*, pages 532–547. Springer-Verlag, 1991.

[89] Robert Wilensky. *Common LISPcraft.* W.W. Norton & Company, 1986.

[90] David Wolfram and Joseph Goguen. A sheaf semantics for FOOPS expressions. In Mario Tokoro, Oscar Nierstrasz, and Peter Wegner, editors, *Object-Based Concurrent Computation*, pages 81–98. Springer-Verlag, 1992. Proceedings, ECOOP'91 Workshop, Geneva, July 1991.