A Refinement Theory for Concurrent Object Oriented Languages*

Paulo Borba†
Departamento de Informática
Universidade Federal de Pernambuco

Joseph A. Goguen‡
Department of Computer Science and Engineering
University of California at San Diego

Abstract

A notion of refinement for concurrent object-oriented programs was originally presented in [3]. In this article we prove that the refinement relation originated from this notion is a congruence with respect to various standard programming language constructors, including parallel and sequential composition, conditional, and nondeterministic *internal* choice. We also establish a weaker compositionality result for the atomic evaluation constructor, and illustrate how novel compositionality properties can be derived from the basic congruence property.

1 Introduction

The importance of formal methods for software development is significantly recognized nowadays. This is mainly justified by the high level of reliability achieved by complex systems developed using languages having a clear mathematical semantics [14, 13, 7] and allowing formal proofs that design steps satisfy (refine) specifications [12].

However, among other factors, the industrial uptake of formal methods depends crucially on adequate refinement theories and associated proof techniques to support the use of those methods in practice [4]. In particular, formal (or even rigorous) software development is not at all practical unless there are theories justifying the *compositional* and *stepwise* refinement of specifications and implementations.

As originally presented in [3], based on the operational semantics [13] of an arbitrary object-oriented language, it is possible to directly define a notion of refinement for concurrent object-oriented programs written in that language. Moreover, this notion (relation) has some basic properties, such as reflexivity and transitivity, and comes up together with an effective proof technique for proving refinement. This refinement notion has been explored and proved to be a suitable basis for formal *stepwise* development of concurrent object-oriented programs, provided that the associated operational semantics satisfies some mild and natural conditions.

In this article we formalize those conditions and, provided that they are satisfied by the operational semantics, we prove that the refinement relation mentioned above is a congruence with respect to various standard programming language constructors, including parallel and sequential composition,

*This article was largely written while the authors were with the Programming Research Group of the Oxford University Computing Laboratory.

†Supported in part by CAPES, Grant 2184-91-8, and CNPq, Grant 301021/95-3. Electronic mail: `phmb@di.ufpe.br`.

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

conditional, and nondeterministic *internal* choice. We also establish a weaker compositionality result for the atomic evaluation constructor, which does not preserve refinement, and illustrate how novel compositionality properties can be derived from the basic congruence property. In this way we justify *compositional* development of concurrent object-oriented programs and set the basis for a refinement calculus in the style of [12].

We formalize only the essential concepts necessary for deriving the compositionality results. Most proofs are omitted for legibility and space reasons. Those proofs are presented in detail in [1], where it is clearly shown that they are not complicated and are based on very simple proof techniques such as case analysis and transition induction [11]; however, some of them are quite long and tedious. Indeed, the emphasis in this article is not on presenting the proofs, but on presenting and analysing the results and their intuitions, in addition to describing the mathematical machinery and approach used to derive the results in a simple way. By following this approach, anyone acquainted to that machinery can easily reproduce the results or adapt them for other contexts.

This article is structured in the following way. First we discuss basic aspects of the operational semantics of concurrent object oriented languages. Following that we overview the refinement notion presented in [3]. We then formalize the conditions that must be satisfied by a particular operational semantics and establish the congruence results for various constructors. Lastly, we illustrate how to derive compositionality results from the basic congruence property.

2 Operational Semantics

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

The operational semantics of an object-oriented language formalizes the notions of state and state transition due to the evaluation of expressions [13, 2]. For an object-oriented program P, let $\mathcal{T}_{\mathsf{P}}$ be the set of terms (expressions) constructed with the operations of P, arbitrary object identifiers, and programming language constructors such as sequential and parallel composition. Let $Db(\mathsf{P})$ be the set of database states associated to P. In particular, let the initial database state associated to a specification P be represented by $\emptyset_{\mathsf{P}}$, which contains only objects introduced by P and information about their respective attributes.

The set of states (or configurations) associated to a program P, denoted $Conf(\mathsf{P})$, is the set of pairs $\mathcal{T}_{\mathsf{P}} \times Db(\mathsf{P})$. States will be 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 will be called "states" as well.

For a program P, let state transitions be specified by a transition relation denoted by

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

This relation actually defines the operational semantics of 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 $\rightarrow$.

For a program P, the transition relation is inductively defined over the syntax of expressions by a set R_{P} of inference rules in the following form:

$$\frac{C_1 \Longrightarrow C_1' \quad C_2 \Longrightarrow C_2' \quad \ldots \quad C_n \Longrightarrow C_n'}{C \rightarrow C'} \qquad [cond]$$

where $\Longrightarrow$ stands for either $\rightarrow$ or $\rightarrow^*$, and C, C', C_i, C_i', for $i = 1 \,..\, n$, are elements of $Conf(\mathsf{P})$. Those rules 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 and the side-condition *cond* is valid. When n is 0 and *cond* is valid, we can directly infer that $C \rightarrow C'$.

For example, the semantics of the sequential composition constructor can be defined by the following two rules:

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

where v is a "fully evaluated expression"—an object identifier or an *evaluated* functional expression, such as an integer, a character, etc.—and it is specified that the argument on the left of the sequential composition constructor has to be fully evaluated before the evaluation of the other argument starts.

Similarly, the semantics of parallel composition can be defined by rules such as

$$\frac{\langle e,\, \mathcal{D}\rangle \to \langle e',\, \mathcal{D}'\rangle}{\langle e \mid\mid f,\, \mathcal{D}\rangle \to \langle e' \mid\mid f,\, \mathcal{D}'\rangle} \qquad \frac{}{\langle v \mid\mid e,\, \mathcal{D}\rangle \to \langle e,\, \mathcal{D}\rangle}$$

and the similar ones for evaluation of the argument on the right of the composition. This indicates that the evaluation of the expressions can be freely interleaved.

A nondeterministic *external* choice constructor can be specified by the rules

$$\frac{\langle e,\, \mathcal{D}\rangle \to \langle e',\, \mathcal{D}'\rangle}{\langle e\ [\,]\ f,\, \mathcal{D}\rangle \to \langle e',\, \mathcal{D}'\rangle} \qquad \frac{}{\langle v\ [\,]\ e,\, \mathcal{D}\rangle \to \langle v,\, \mathcal{D}\rangle}$$

and similar ones for evaluation of the expression on the right of the choice. Those rules indicate that transitions from the choice between two expressions correspond to transitions from one of the expressions. Note that a fully evaluated argument may be chosen by the choice without changing the database.

We can also define the semantics of an atomic evaluation constructor using rules such as the following:

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

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

The semantics of other constructors such as `if_then_else_fi` (conditional) and `result_;_` (evaluates its first argument and then evaluates the second, yielding the value resulting from the evaluation of the first) can be formally defined following the lines above. Contrasting, the semantics of method and attribute evaluation, and also object creation and deletion, will not even be discussed in this article since it may vary a lot depending on the language, and it is not directly relevant to the derivation of the compositionality results that we will present. The same holds for the representations of terms and database states. The semantics of those aspects are formally specified for a specific language in [2], for example.

3 Refinement

Based on the operational semantics of an arbitrary object-oriented language, the refinement notion originally presented in [3] considers refinement of states, expressions, and programs written in that language. In this article we discuss only refinement of states and expressions. Basically, a state P is simulated by a state Q if whatever can be observed by performing experiments with Q can also be observed by performing the same experiments with P. In this case, if experiments are done with Q, we cannot detect whether Q or P is being used, even though the experiments done with P might let us make more observations than if Q were used.

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

1. the values of the *visible* attributes of the objects in P;

2. the results yielded by the expressions being evaluated in P; and

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

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

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

3.1 Simulations

Now we formalize the finer notion of simulation originally presented in [3]. First let $\mathcal{E}(\mathsf{P})$ denote the set of **experiments** associated to a program P. Also, let $\mathcal{FE}_{\mathsf{P}}(e)$ be a predicate that indicates whether the expression $e \in \mathcal{T}_{\mathsf{P}}$ is fully evaluated; when P can be understood from the context we simply write $\mathcal{FE}(e)$. Furthermore, for fully evaluated terms p and q respectively related to specifications P and Q, let equality be denoted by $p =_{(\mathsf{P},\,\mathsf{Q})} q$; it holds when p and q are the same object identifiers or when they are the same element of an ADT that is defined by both P and Q.

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

1. Whenever $Q \rightarrow Q'$ 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 $\mathit{exp} \in \mathcal{E}(\mathsf{P})$, whenever $\langle \mathit{exp},\ \mathcal{Q}\rangle \rightarrow^* Q' \not\rightarrow$ then, for some P',

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

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

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

where P and Q denote configurations, $\mathcal{P}$ and $\mathcal{Q}$ denote their respective database states, and p and q denote their respective expressions. Relations having those special properties are called $(\mathsf{P},\ \mathsf{Q})$-**simulations**.

Condition 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, Condition 3 says that performing the same experiment with Q and P leads to states related by $\mathcal{S}$; by Condition 2, this implies that the experiments yield the same results when performed in both states. Furthermore, if an experiment cannot be performed with Q (its corresponding operation is not enabled because it is not allowed to be executed in a particular state), then performing the same experiment with P results in a state related to Q by $\mathcal{S}$; this usually means that the experiment cannot be performed with P as well.

Those conditions reflect the ideas that we have introduced about simulation of states. However, they are based on a central assumption: operations used as experiments are atomic and terminating. That is why the formal definition of simulation does not consider the intermediate states reached during the evaluation of experiments (see Condition 3). However, note that the evaluation of experiments might take more than one transition, as long as the extra transitions do not change the database state nor depend on it. Also the assumption mentioned above does not imply that all methods have to be atomic and terminating; indeed, nonatomic and nonterminating methods can be defined as hidden operations. In [3], this assumption is analyzed and justified.

3.2 Refinement of States

Note that if the pair $(P,\ Q)$ is in a $(\mathsf{P},\ \mathsf{Q})$-**simulation** then whatever can be observed by performing experiments with Q can also be observed by performing experiments with P; this guarantees that one state simulates another. In fact, we say that a state $Q \in \mathit{Conf}(\mathsf{Q})$ **simulates** (or **refines**) a state $P \in \mathit{Conf}(\mathsf{P})$, denoted $P \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} Q$, if there is a $(\mathsf{P},\ \mathsf{Q})$-simulation containing the pair $(P,\ Q)$. Formally,

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

A direct result from this definition is the following:

Theorem 3.1 Assuming that the observations and experiments introduced by a specification P are also introduced by a specification Q, the refinement relation on states $\sqsubseteq_{(\mathsf{P},\,\mathsf{Q})}$ is the largest $(\mathsf{P},\ \mathsf{Q})$-simulation. □

This was proved in [3]. It is also easy to check that the identity relation on configurations of the same program is a simulation, and that $\sqsubseteq_{(\mathsf{P},\,\mathsf{P})}$ is reflexive and transitive; actually, we have that $P \sqsubseteq_{(\mathsf{P},\,\mathsf{O})} O$ and $O \sqsubseteq_{(\mathsf{O},\,\mathsf{Q})} Q$ imply $P \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} Q$.

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

Hereafter 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.

3.3 Refinement of Expressions

Based on the notion of refinement of states, we can define refinement of expressions. First recall that the initial database state associated to a program P is represented by $\emptyset_{\mathsf{P}}$. Also, assume that 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 $\mathcal{P}$[1]. 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

[1]By the definition of simulation, this already considers that the evaluation of the expressions might be interfered by experiments.

Q simulates the evaluation of p in the initial state of P. So, for specifications P and Q, 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

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

The reflexive and transitive properties of refinement of states are also valid for this definition of refinement of expressions.

3.4 Checking Refinement

In order to illustrate the notion of refinement presented in previous sections, let us consider an object-oriented program defining a class `Cell` of memory cells for storing natural numbers. We will be rather generic, instead of using a particular language to describe the example. Just assume that objects of `Cell` have the attribute `val` (for *val*ue), which indicates the natural number stored in a cell, and two atomic methods for assigning values to a cell: `assign(n)`, which assigns a given natural number to a cell, and `add(n)`, which assigns to a cell the value stored in it plus a given natural number.

For example, based on this program we can examine the relationship between the commands (expressions)

$$\texttt{x.assign(0) ; x.add(1)} \tag{1}$$

and

$$\texttt{x.assign(1)}, \tag{2}$$

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

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

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

$$\texttt{x.assign(1) || x.assign(1)}$$

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

$$\texttt{x.assign(1) || x.assign(0) ; x.add(1)}$$

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

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

$$\langle \texttt{x.assign(0) ; x.add(1)}, \mathcal{P} \rangle$$

is refined by

$$\langle \texttt{x.assign(1)}, \mathcal{P} \rangle,$$

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

As discussed in Section 3.2, this can be verified by finding a simulation relating the configurations described above. But it is easy to check that the relation $\mathcal{S}$, consisting of pairs in the forms

$$\begin{array}{rcl} (\langle \texttt{x.assign(0) ; x.add(1)}, \mathcal{P}\rangle & , & \langle \texttt{x.assign(1)}, \mathcal{P}\rangle) \\ (\langle e, \mathcal{P}\rangle & , & \langle e, \mathcal{P}\rangle), \end{array}$$

for any reachable database state $\mathcal{P}$ and any expression e, is indeed a simulation.

On the other hand, confirming the claim made at the beginning of the section, the relation containing pairs in the form

$$\begin{array}{rcl} (\langle \texttt{x.assign(1)}, \mathcal{P}\rangle & , & \langle \texttt{x.assign(0) ; x.add(1)}, \mathcal{P}\rangle) \\ (\langle e, \mathcal{P}\rangle & , & \langle e, \mathcal{P}\rangle), \end{array}$$

for any reachable database state $\mathcal{P}$ and any expression e, is not a simulation. In fact, the first transition from

$$\langle \texttt{x.assign(0) ; x.add(1)}, \mathcal{P}\rangle$$

cannot be matched by any transition from $\langle \texttt{x.assign(1)}, \mathcal{P}\rangle$; just observe that the first transition from the former configuration leads to an intermediate state where `x.val` is `0`, but no such state can be reached from the latter configuration.

We can also verify that

$$\texttt{x.add(1) ; x.add(-1)} \tag{3}$$

is refined by

$$\texttt{x.skip()}, \tag{4}$$

where `skip()` is a method that does nothing, simply returns. In fact, a simulation relation very similar to $\mathcal{S}$ can be chosen as the basis for this proof. Also, it easily follows that (4) is not refined by (3).

Perhaps more interesting is verifying that

$$\texttt{x.skip() ; x.assign(0)}$$

is refined by

$$\texttt{x.assign(0)},$$

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

$$\begin{array}{rcl} (\langle \texttt{x.skip() ; x.assign(0)}, \mathcal{P}\rangle & , & \langle \texttt{x.assign(0)}, \mathcal{P}\rangle) \\ (\langle e, \mathcal{P}\rangle & , & \langle e, \mathcal{P}\rangle), \end{array}$$

and

$$\begin{array}{rcl} (\langle \texttt{x.assign(0)}, \mathcal{P}\rangle & , & \langle \texttt{x.skip() ; x.assign(0)}, \mathcal{P}\rangle) \\ (\langle \texttt{x.assign(0)}, \mathcal{P}\rangle & , & \langle \texttt{x ; x.assign(0)}, \mathcal{P}\rangle) \\ (\langle e, \mathcal{P}\rangle & , & \langle e, \mathcal{P}\rangle), \end{array}$$

for any reachable database state $\mathcal{P}$ and any expression e, are simulations.

Lastly, we can prove that the nondeterministic choice

$$\texttt{x.assign(0) [] x.assign(1)}$$

is refined by the assignment

$$\texttt{x.assign(1)},$$

by proving that the relation formed by pairs in the form

$$\begin{array}{rcl} (\langle \texttt{x.assign(0) [] x.assign(1)}, \mathcal{P}\rangle & , & \langle \texttt{x.assign(1)}, \mathcal{P}\rangle) \\ (\langle e, \mathcal{P}\rangle & , & \langle e, \mathcal{P}\rangle), \end{array}$$

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.

From those examples and the definition of refinement, we can observe several aspects relevant for refinement of concurrent object oriented programs: data refinement (including dynamic data structures), nondeterminism, concurrency, interference, and refinement of atomic operations. We can also observe that the definition of refinement is divergence insensitive; that is, it may relate an expression not having a divergent behavior to an expression having a divergent behavior. Just notice that, by the definition of simulation, a $\twoheadrightarrow$-transition from a state can be matched by no transitions from a related state.

4 Refinement Theory

Now we develop a theory in order to show that the relation of refinement of expressions introduced in the previous section is a congruence with respect to several programming language constructors. Precisely, taking parallel composition as example, we are interested in proving that

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

for any expression o formed by experiments of P and most of the constructors introduced in Section 2. Of course, we assume that the meaning of $\_ \mid\mid \_$ is the same for P and Q; that is, both R_{P} and R_{Q} have the same rules for defining the semantics of $\_ \mid\mid \_$.

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 behavior of states with respect to a restricted group of *visible* operations. This is definitely the case of the notion of refinement introduced in the previous section (see Condition 3c of the definition of simulation). 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* gives support for the compositional verification and derivation of implementations; that is essential for formal development of nontrivial systems. But note that this only supports compositionality *in the small* (i.e., regarding expressions and their composition [11]). Compositionality *in the large* (i.e., regarding modules and their composition [5]) should be supported by a congruence property of refinement of *programs* (*modules*) with respect to module interconnection operations (such as module importation and sum); however, this is out of the scope of this article which does not consider the semantics of module systems.

In order to derive the compositionality results we proceed as follows. First we formalize the constraints on experiments informally described at the end of Section 3.1. Those constraints allow us to prove that $\sqsubseteq_{(\mathsf{P},\,\mathsf{Q})}$ includes the identity on experiments whenever there is a $(\mathsf{P},\,\mathsf{Q})$-simulation relating initial states; that is, for any experiment *exp*, $exp \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} exp$ whenever there is a $(\mathsf{P},\,\mathsf{Q})$-simulation relating initial states. We also extend this result for composition of experiments using some programming language constructors. It then becomes easy to derive the desired congruence result; we consider each constructor separately.

4.1 Experiments

As discussed in Section 3.1, 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 $\mathcal{D} \in Db(\mathsf{P})$, where a configuration is **terminating** if there is no infinite sequence of $\twoheadrightarrow_{\mathsf{P}}$-transitions from it. We drop the subscript from $\downarrow_{\mathsf{P}} e$ when it is not confusing.

Atomicity is not only achieved by ensuring that an expression is fully evaluated in one $\twoheadrightarrow$-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 $\twoheadrightarrow$-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 $\twoheadrightarrow$-transition without accessing the database state and without making choices. Formally,

$$p \rightrightarrows_{\mathsf{P}} q$$

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.

4.1.1 Transition Induction

The relation $\rightrightarrows^*$ can be formally defined by the following inference rules:

$$\frac{}{p \rightrightarrows^* p} \qquad \frac{p \rightrightarrows^* o \qquad o \rightrightarrows q}{p \rightrightarrows^* q}$$

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 [11], 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 that will be presented 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 4.1 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: The proof easily follows by transition induction on the inference of $p \rightrightarrows^* q$. $\square$

4.1.2 Properties of Experiments

Using Lemma 4.1, 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 4.2 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

$$P \not\rightarrow_{\mathsf{P}} \text{ and } \mathcal{FE}_{\mathsf{P}}(P).$$

□

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 $(\mathsf{P},\,\mathsf{Q})$-simulation relating the initial states of P and Q. In fact, as no meaningful 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 $(\mathsf{P},\,\mathsf{Q})$-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 the definition of simulation on Section 3.1).

Lemma 4.3 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 $\langle expp', \mathcal{P}'\rangle$,

$$\langle expp, \mathcal{P}\rangle \rightarrow^* \langle expp', \mathcal{P}'\rangle,$$
$$\langle p, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \langle q, \mathcal{Q}'\rangle,$$
$$expp' =_{(\mathsf{P},\,\mathsf{Q})} expq' \quad \text{or} \quad exp \rightrightarrows^*_{\mathsf{P}} expp' \text{ and } exp \rightrightarrows^*_{\mathsf{Q}} expq'.$$

□

Now we introduce the second lemma.

Lemma 4.4 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 $\langle expp', \mathcal{P}'\rangle$,

$$\langle expp, \mathcal{P}\rangle \rightarrow^* \langle expp', \mathcal{P}'\rangle \not\rightarrow,$$
$$\langle p, \mathcal{P}'\rangle \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \langle q, \mathcal{Q}\rangle,$$
$$\mathcal{FE}_{\mathsf{P}}(expp') \Leftrightarrow \mathcal{FE}_{\mathsf{Q}}(expq),$$
$$\mathcal{FE}_{\mathsf{P}}(expp') \text{ implies } expp' =_{(\mathsf{P},\,\mathsf{Q})} expq, \text{ and}$$
$$\neg\mathcal{FE}_{\mathsf{P}}(expp') \text{ implies } exp \rightrightarrows^*_{\mathsf{P}} expp'.$$

□

The following lemma introduces the simulation relating $\rightrightarrows$-derivatives of an experiment exp.

Lemma 4.5 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 $(\mathsf{P},\,\mathsf{Q})$-simulation.

Proof: It follows directly from Theorem 3.1, Lemma 4.3, Lemma 4.4, 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 $(\mathsf{P},\,\mathsf{Q})$-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 4.6 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 4.5, 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: the relation $\sqsubseteq_{(\mathsf{P},\mathsf{Q})}$ includes the identity on experiments whenever there is a (P, Q)-simulation relating initial states.

Corollary 4.7 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 4.6, since $\rightrightarrows^*_{\mathsf{P}}$ and $\rightrightarrows^*_{\mathsf{Q}}$ are reflexive. □

4.2 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. By composition of experiments we mean a term formed by composing experiments and fully evaluated expressions with the constructors described in Section 2, except the atomic evaluation and (external) nondeterministic choice constructors which only preserve refinement when their arguments satisfy some specific conditions, as we will discuss later. We use $\mathcal{CE}(\mathsf{P})$ to denote the set of **composition of experiments** of a specification P.

Precisely, 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:

$$\begin{array}{l}
cexp \rightrightarrows^*_{\mathsf{P}} \;=\; \{e \mid cexp \rightrightarrows^*_{\mathsf{P}} e\}, \text{ if } cexp \in \mathcal{E}(\mathsf{P}). \\
cexp \rightrightarrows^*_{\mathsf{P}} \;=\; \{cexp\}, \text{ if } \mathcal{FE}_{\mathsf{P}}(cexp). \\
(cexp_1 \;\texttt{||}\; cexp_2) \rightrightarrows^*_{\mathsf{P}} \;=\; \{e_1 \;\texttt{||}\; e_2 \mid e_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}} \text{ and } e_2 \in cexp_2 \rightrightarrows^*_{\mathsf{P}}\} \\
(cexp_1 \;\texttt{;}\; cexp_2) \rightrightarrows^*_{\mathsf{P}} \;=\; \{e_1 \;\texttt{;}\; cexp_2 \mid e_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}}\} \\
(\texttt{result}\; cexp_1 \;\texttt{;}\; cexp_2) \rightrightarrows^*_{\mathsf{P}} \;= \\
\qquad \{\texttt{result}\; e_1 \;\texttt{;}\; cexp_2 \mid e_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}} \text{ and } \neg\mathcal{FE}_{\mathsf{P}}(cexp_1)\} \cup \\
\qquad \{\texttt{result}\; cexp_1 \;\texttt{;}\; e_2 \mid e_2 \in cexp_2 \rightrightarrows^*_{\mathsf{P}} \text{ and } \mathcal{FE}_{\mathsf{P}}(cexp_1)\} \\
(\texttt{if}\; cexp_1 \;\texttt{then}\; cexp_2 \;\texttt{else}\; cexp_3 \;\texttt{fi}) \rightrightarrows^*_{\mathsf{P}} \;= \\
\qquad \{\texttt{if}\; e_1 \;\texttt{then}\; cexp_2 \;\texttt{else}\; cexp_3 \;\texttt{fi} \mid e_1 \in cexp_1 \rightrightarrows^*_{\mathsf{P}}\}
\end{array}$$

for any $cexp_1, cexp_2, cexp_3 \in \mathcal{CE}(\mathsf{P})$. Note that $cexp \in cexp \rightrightarrows^*_{\mathsf{P}}$.

4.2.1 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. Those lemmas can be proved using the results introduced in Section 4.1.2.

Lemma 4.8 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 $cexp' \in \mathcal{CE}(\mathsf{P})$. □

Now we present the second lemma.

Lemma 4.9 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 $\mathcal{D}$. □

We can then introduce the simulation relating $\rightrightarrows$-derivatives of a composition of experiments $cexp$.

Lemma 4.10 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. □

This lemma can be proved similarly to Lemma 4.5. 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 4.11 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 $cexpp \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} cexpq$. □

This theorem can be proved similarly to Theorem 4.6. Then we can easily derive another 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 4.12 Let $cexp \in \mathcal{CE}(\mathsf{P})$. Thus, if $\emptyset_{\mathsf{P}} \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \emptyset_{\mathsf{Q}}$ then $cexp \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} cexp$. □

4.3 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 constructors introduced in Section 2. This is realized by presenting specific simulation relations.

4.3.1 Parallel Composition

First let us consider parallel composition. The following lemma is the essence of the proof that refinement is a congruence with respect to parallel composition.

Lemma 4.13 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. □

Using this lemma we can easily verify that refinement is preserved by the parallel composition of derivatives of the same composition of experiments. This is a general compositionality result.

Theorem 4.14 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

$$p \;||\; cexpp \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q \;||\; cexpq.$$

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 4.13, there is a $(\mathsf{P},\,\mathsf{Q})$-simulation $\mathcal{S}$ such that

$$(\langle p \;||\; cexpp,\, \emptyset_{\mathsf{P}}\rangle,\, \langle q \;||\; cexpq,\, \emptyset_{\mathsf{Q}}\rangle) \in \mathcal{S}.$$

So we have

$$\langle p \;||\; cexpp,\, \emptyset_{\mathsf{P}}\rangle \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \langle q \;||\; cexpq,\, \emptyset_{\mathsf{Q}}\rangle.$$

Therefore $p \;||\; cexpp \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q \;||\; cexpq$. □

Finally, by specializing the result above, we obtain the promised compositionality result.

Corollary 4.15 Let $cexp \in \mathcal{CE}(\mathsf{P})$. If

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

then

$$p \;||\; cexp \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q \;||\; cexp.$$

Proof: Directly by Theorem 4.14, since $cexp \in cexp \rightrightarrows^*_{\mathsf{P}}$ and $cexp \in cexp \rightrightarrows^*_{\mathsf{Q}}$. □

There is also another result for parallel composition:

Theorem 4.16 Let $cexp \in \mathcal{CE}(\mathsf{P})$. If

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

then

$$cexp \;||\; p \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} cexp \;||\; q.$$

□

It can be derived following exactly the same line sketched above.

4.3.2 Sequential Composition

Congruence results can be derived for the sequential composition constructor, using essentially the same approach used for parallel composition. So first we introduce a special simulation.

Lemma 4.17 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. □

Then we derive a lemma showing that refinement is preserved by sequential composition using derivatives of the same composition of experiments.

Theorem 4.18 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

$$p\ ;\ cexpp \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q\ ;\ cexpq.$$

□

Anf finally we derive the expected compositionality result.

Corollary 4.19 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

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

□

Similarly, we can prove that refinement of expressions is a congruence with respect to the constructors `if_then_else_fi` and `result_;_`; we omit the details in this article.

4.3.3 Atomic Evaluation

Contrasting with the other constructors, atomic evaluation is unary. So the congruence property is expressed in the following way:

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

implies

$$\mathtt{[}\,p\,\mathtt{]} \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \mathtt{[}\,q\,\mathtt{]}.$$

Indeed, this congruence property does not hold because the behavior of the atomic evaluation constructor depends on whether its argument always terminates and whether it can be fully evaluated, whereas those aspects are abstracted by the definition of refinement considered in this article. In fact, the evaluation of an expression $\mathtt{[}\,q\,\mathtt{]}$ in a state $\mathcal{D}$ only blocks (i.e., it does not lead to any state) 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, $\langle \mathtt{[}\,q\,\mathtt{]}, \mathcal{D}\rangle$ is not a refinement of $\langle \mathtt{[}\,p\,\mathtt{]}, \mathcal{D}\rangle$, since the first configuration blocks whereas the second does not.

However, using the definition of refinement considered in this article we can establish a weaker compositionality result. In fact, refinement is only preserved by the atomic evaluation constructor if the argument of this constructor satisfies 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 4.20 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. □

Now we can directly establish the weaker compositionality result for the atomic evaluation constructor.

Theorem 4.21 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 4.20, 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 $\texttt{[}\,p\,\texttt{]} \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \texttt{[}\,q\,\texttt{]}$. □

Naturally, a notion of refinement considering the conditions expressed above would be a congruence with respect to the atomic evaluation constructor. In fact, such a more concrete notion of refinement can be obtained by extending, in a very simple and natural way, the definition of refinement considered in this article.

4.3.4 Nondeterministic Choice

The notion of refinement discussed in Section 3 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 only leads to P, since by the definition of simulation $\rightarrow$-transitions from Q may be matched by zero $\rightarrow$-transitions from P.

This suggests that the notion of refinement is not preserved by external choice. In fact, it is not. For instance, 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 is not matched by p `[]` $cexp$, we must conclude that the former expression is not a refinement of the latter.

A notion of refinement that does not abstract from the power of expressions of making external choices can be easily derived from the notion of refinement presented in this article, by essentially requiring the initial transitions from a state to be matched by at least one transition (instead of zero or many) from a related state. This derived notion would be preserved by the *external* choice method combiner.

Here we shall be content to show that the notion of refinement we are using in this article is preserved by the *internal* choice method combiner, which can be defined in the following way:

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

where `P` and `Q` are meta-variables ranging over expressions of the programming language, and **`skip`** is any operation that does nothing and always terminates. This constructor preserves the notion of refinement because its behavior does not depend on the behavior of its arguments; the choice is made by **`skip`**, not by `P` or `Q`. This is a quite satisfactory result since the internal choice constructor seems more useful than the external choice constructor in the context of object-oriented languages [2]. The following simulation is the first step towards deriving this result.

Lemma 4.22 Let $\mathcal{S}$ be the relation consisting of $\sqsubseteq_{(\mathsf{P},\,\mathsf{Q})}$ and all pairs in the forms

$$\begin{array}{rcl} (\langle p \;\texttt{Or}\; cexpp,\, \mathcal{P}\rangle & , & \langle q \;\texttt{Or}\; cexpq,\, \mathcal{Q}\rangle), \\ (\langle \texttt{skip ; } p \texttt{ [] skip ; } cexpp,\, \mathcal{P}\rangle & , & \langle \texttt{skip ; } q \texttt{ [] skip ; } cexpq,\, \mathcal{Q}\rangle), \\ (\langle cexpp \;\texttt{Or}\; p,\, \mathcal{P}\rangle & , & \langle cexpq \;\texttt{Or}\; q,\, \mathcal{Q}\rangle), \text{ and} \\ (\langle \texttt{skip ; } cexpp \texttt{ [] skip ; } p,\, \mathcal{P}\rangle & , & \langle \texttt{skip ; } cexpq \texttt{ [] skip ; } q,\, \mathcal{Q}\rangle), \end{array}$$

where $\langle p,\, \mathcal{P}\rangle \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} \langle q,\, \mathcal{Q}\rangle$ and there is a $cexp \in \mathcal{CE}(\mathsf{P})$ 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. □

We can then easily verify that the internal choice method combiner preserves refinement, following an approach similar to the one followed for parallel composition. First we derive a general compositionality result.

Theorem 4.23 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

$$p \;\texttt{Or}\; cexpp \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q \;\texttt{Or}\; cexpq$$

and

$$cexpp \;\texttt{Or}\; p \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} cexpq \;\texttt{Or}\; q.$$

□

And finally, by a trivial specialization, the desired compositionality result.

Corollary 4.24 Let $cexp \in \mathcal{CE}(\mathsf{P})$. If

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

then

$$p \;\texttt{Or}\; cexp \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q \;\texttt{Or}\; cexp$$

and

$$cexp \;\texttt{Or}\; p \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} cexp \;\texttt{Or}\; q.$$

□

5 Other Compositionality Properties

The basic compositionality properties of standard programming language constructors were established in previous sections, by showing that refinement is preserved by them. In this section we show how other compositionality properties can be derived from the basic properties. We use the sequential composition constructor in the examples. Similar results can be derived for other constructors.

First let us state the basic compositionality properties of sequential composition given in Section 4.3.2: if

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

then, for any $cexp \in \mathcal{CE}(\mathsf{P})$,

$$p \ ; \ cexp \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q \ ; \ cexp \quad \text{and} \quad cexp \ ; \ p \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} cexp \ ; \ q.$$

As discussed before, this result depends crucially on the condition $cexp \in \mathcal{CE}(\mathsf{P})$, since the notion of refinement presented in this article is based on the observational behavior of states with respect to a restricted group of operations.

Now consider the following compositionality property:

$$p \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q \quad \text{and} \quad p' \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q'$$

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 $q \ ; \ q'$. In fact, the congruence result for sequential composition 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 4.10), if reachable or not.

Although the compositionality property stated above is not valid, the following weaker version of it is indeed valid: for $q, p' \in \mathcal{CE}(\mathsf{P})$,

$$p \sqsubseteq_{(\mathsf{P},\,\mathsf{P})} q \quad \text{and} \quad p' \sqsubseteq_{(\mathsf{P},\,\mathsf{Q})} q'$$

implies

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

It follows from the general transitivity property of $\sqsubseteq_{(\mathsf{P},\,\mathsf{Q})}$ (discussed on Sections 3.2 and 3.3) 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 object-oriented software. In this article we limit ourselves to give the basis for deriving those properties.

6 Conclusions

We established several properties of the notion of refinement originally introduced in [3]. In particular, we proved that the relation of refinement of expressions is a congruence with respect to several standard programming language constructors. We also explained how to derive novel compositionality results from the basic congruence result. The congruence property justifies the compositional verification and derivation of implementations; this is essential for formal development of nontrivial systems in practice. However, we only considered compositionality *in the small*; properties of compositionality *in the large* are related to the semantics of module systems, which was not discussed in this article.

The proof of the congruence result justified and provided a deep insight into some of the technical decisions taken in [3]. 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.

By observing the proofs it is also clear 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 behavior of states with respect to a restricted group of *visible* operations.

We explained why refinement is not a congruence with respect to the atomic evaluation and non-deterministic *external* choice constructors. Basically, the behavior of those constructors depends on some aspects that are abstracted by the notion of refinement explored in this article. However, we verified that nondeterministic *internal* choice preserves refinement, and we proved a weaker compositionality result for the atomic evaluation constructor. Indeed, this weaker result indicates how the atomic evaluation constructor should be used in practice.

We can also conclude that refinement is not in general 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; in fact, this depends on the type system of a particular language. So, for a visible operation op, $op(p)$ might be a well formed term whereas $op(q)$ might not.

The results presented in this article are not proposed as the only tools that should be available for formal software development; instead, they are just a basis for the definition of refinement calculi [12] and methodologies for formal development of concurrent object-oriented programs.

In fact, just a few notions of refinement of concurrent object-oriented programs have been proposed so far. We will comment on two of those notions, concluding that none of them provide a general definition of refinement with compositionality results such as the ones presented here. One of them [8] considers only refinement of object *based* concurrent programs, presenting many examples of formal development of programs. Most examples are simple and elegant, and use a few refinement preserving transformation rules. However, no general definition of refinement is proposed; although the semantics of the language presented in [8] is given in terms of a process algebra, the notions of equivalence and refinement of the algebra cannot be directly used to define the notion of refinement of the language [9].

Another approach for refinement of object oriented concurrent programs is described in [10], which gives an overview of a general notion of refinement for a concurrent object oriented specification language. This notion of refinement is compositional *in the large*. However, compositionality *in the small*, as considered here, and proof techniques are not discussed.

References

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

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

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

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

[5] Joseph Goguen. Parameterized programming. *Transactions on Software Engineering,* SE–10(5):528–543, September 1984.

[6] 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.

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

[8] 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.

[9] 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.

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

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

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

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

Contents