11

Invited Paper – Computational models for open distributed systems

Elie Najm[a], *Jean-Bernard Stefani*[b]

(a) ENST - École Nationale Supérieure des Télécommunications.
46, Rue Barrault - 75014 Paris, France,
E-mail: najm@res.enst.fr

(b) France Telecom - CNET
38-40, rue du Général Leclerc - 92131 Issy-Les-Moulineaux, France
E-mail: stefani@issy.cnet.fr

Abstract

The notion of computational model is central to the Reference Model for Open Distributed Processing. It defines an abstract model for distributed computations and characterizes at the same time the functionality of a supporting distributed virtual machine. Most current distributed systems, including e.g. the CORBA platforms, have converged on a subset of this computational model. In this paper, we review the model, providing a formal characterization for it, and we discuss various directions for extensions, considering in particular different trends in distributed systems construction.

Keywords

ODP, objects, computational model, formal techniques, quality of service, reflection

1 INTRODUCTION

The notion of computational model* is central to the Reference Model for Open Distributed Processing (RM-ODP) [20]. It provides a model of distributed computations in an ODP environment and it describes, in a programming language-independent fashion, the interface of the distributed virtual machine whose structure is described by the engineering model of the RM-ODP.

The computational model made explicit in the RM-ODP underlies most of the distributed computational models which have been proposed, explicitly or implicitly, in recent distributed language or distributed system developments. For instance, the computational model adopted in the OMG CORBA specifications [29] corresponds* to the so-called operational subset of the RM-ODP computational model. The same is true for e.g. the Network Object system [7], the Java RMI system [33] which supports distributed Java programs.

Despite the differences between these systems, the convergence of all these proposals on the operational subset of the RM-ODP computational model as the basic model for distributed computations is remarkable. It suggests that some "universal" for open distributed system programming has been identified. In this paper, we begin by reviewing and formally characterizing the operational subset of the RM-ODP computational model. That it can be showed at least as expressive as the (asynchronous) π-calculus brings further evidence to its universality.

Apart from its well-understood operational subset, the ODP computational model contains additional features, such as binding objects and environment contracts which are probably less known but which provide important additions to the basic model. The introduction of binding objects extends the basic computational model with notions of explicit communication objects. The introduction of environment contracts allows the specification of quality of service constraints associated with objects. In this paper, we characterize formally the notion of binding object and we discuss the possibility of formally capturing the notion of environment contract using failure observers.

A computational model provides an abstraction of a distributed system programming interface. It is thus interesting to look at the current developments in distributed systems research to understand what a computational model for open distributed systems should contain, and whether extensions to the ODP computational model — taken as the reference computational model — are required. The current research on distributed systems covers a lot of ground. Among the issues covered, we can highlight issues of mobility, issues associated with real-time and fault-tolerant processing, support for different forms of communication and distributed control, support for different soft-

*The ODP Reference Model uses "language" instead of "model". We prefer to use the latter to emphasize the language- independent nature of the notion.

*As a first approximation, leaving typing issues — which we do not consider in this paper — aside.

ware architecture styles and programming with distributed components, etc. While some of these issues can be understood within the framework of the ODP computational model, the extent of these different developments calls for additional extensions to the ODP computational model, including, notably, notions of composition and reflection. We discuss in this paper these various aspects and comment on their possible formalization. In the process, we hope to bring some insights to further the quest for a flexible, universal basis for open distributed programming.

The paper is organized as follows:

- section 2 reviews the operational subset of the computational model described in the RM-ODP, and discusses some of its features;
- section 3 reviews binding object features of the RM-ODP computational model and characterizes them formally;
- section 4 reviews environment contracts and discusses a possible approach to their formalization;
- section 5 hints at various extensions of the model and at a reflective computational model that could provide the basis for a new computational model for adaptable open distributed systems.
- section 6 concludes the paper.

2 THE BASIC COMPUTATIONAL MODEL

Following [26], the operational subset of the ODP computational model can be described formally using rewriting logic [24]. More precisely, the model can be formalized by a rewrite rule schema, a rewriting theory and a predicate that characterize valid computations. We review them below, leaving aside typing aspects. Those are covered in the original paper [26]. We call $\mathrm{CM}_{\mathrm{basic}}$ the basic ODP computational model.

Since the ODP computational model abstracts away from the construction of individual objects, we consider objects as a given sort in the rewriting theory, characterizing them merely through their "visible" properties. We adopt an infix notation for operators and we use the same name for an operator and its extension to subsets of its domain. We use $\wp_f(S)$ to denote the set of finite subsets of a set S. We use S^* to denote the set of finite sequences of elements of a set S.

In what follows, we use the following sorts (to simplify notations we identify sorts and their carriers):

- Id denotes the set of interface identifiers; we use u, v, u_i, v_i to denote interface identifiers.
- Object denotes the set of objects; we use ω, ω', ω_i, ϖ, ϖ_i to denote objects.

- Signal denotes the set of signals; we use s, s_i to denote signals. Signals are units of interaction between objects and their environment.
- LocConf denotes the set of local configurations; we use c, c_i to denote local configurations. Local configurations are used to describe the behavior of objects.
- DisConf denotes the set of so-called "distributed configurations"; we use d, d_i to denote distributed configurations. Distributed configurations represent parallel compositions of objects that conform to the ODP computational model. The set of distributed configurations has two special elements, noted $\emptyset$ and $\perp$, that represent, respectively, an empty configuration and an invalid configuration.
- Message denotes the set of messages; we use m, m_i to denote messages. Messages represent asynchronous messages that are used to convey operation invocations between objects in the ODP computational model.
- Name denotes the set of signal names; we use n, n_i to denote signal names.

We use the following operators:

- $\|$: DisConf $\times$ DisConf $\rightarrow$ DisConf. $\|$ is the asynchronous parallel operator implied by the ODP computational model. A distributed configuration in the basic computational model consists in a set of objects and messages in parallel.
- Lhs : Object $\times$ $\wp_f$(Signal) $\rightarrow$ LocConf. Lhs is used to specify the behavior of objects.
- Rhs : Object $\times$ $\wp_f$(Signal) $\times$ $\wp_f$(Object) $\rightarrow$ LocConf. Rhs is used to specify the behavior of objects.
- L : Object $\rightarrow$ $\wp_f$(Id). $\omega.L$ yields the set of interface identifiers of object ω.
- K : Object $\rightarrow$ $\wp_f$(Id). $\omega.K$ yields the set of interface identifiers known by object ω.
- tgt : Signal $\rightarrow$ Id. $s.tgt$ is the identifier of the interface to which s is sent.
- arg : Signal $\rightarrow$ Id*. $s.arg$ is the sequence of interface identifiers that appear as arguments of s.
- nm : Signal $\rightarrow$ Name. $s.nm$ is the name of s.

We also have "injection" operators that turn a signal into a message, and that turn an object or a message into a distributed configuration. To simplify the notations, we do not make these obvious operators explicit and we just use objects and signals in these different contexts.

The following laws are assumed to hold. In the rest of this section, terms appearing in rewrite rules must be understood modulo the equations (E) below, i.e. as denoting their E-equivalence class.

- Abelian monoid laws for $\|$, with $\emptyset$ as a neutral element (i.e. $d \| \emptyset = d$) and $\perp$ as an absorbing element (i.e. $d \| \perp = \perp$).

- Reduction to $\perp$ for distributed configurations: $\omega_1.L \cap \omega_2.L \neq \emptyset \Rightarrow \omega_1 \parallel \omega_2 = \perp$.

The last law above specifies the conditions leading to an invalid distributed configuration. In other terms, no two objects in a valid distributed configuration may have an interface with the same identifier.

2.1 Characterizing objects

An object in $\mathrm{CM}_{\mathrm{basic}}$ (as well as in other computational models discussed later) is characterized by a set of rewrite rules that obey the rewrite rule schema ($\Diamond$), with Ω a (possibly empty) finite set of objects, and $\aleph$, $\aleph'$ (possibly empty) finite sets of signals:

$$(\Diamond) \quad Lhs(\omega, \aleph) \rightarrow Rhs(\omega', \aleph', \Omega)$$

where:

In the rest of the paper we use the notation $\Diamond(\omega, \aleph, \omega', \aleph', \Omega)$ to stand for $Lhs(\omega, \aleph) \rightarrow Rhs(\omega', \aleph', \Omega)$. In addition, the following conditions must hold, i.e. $\Diamond(\omega, \aleph, \omega', \aleph', \Omega) \Rightarrow C_0$, where C_0 is the conjunction of the conditions below:

- $\aleph.tgt \subseteq \omega.L$
  Signals on the left-hand side of the rule schema are targeted at interfaces of object ω.
- $\omega.L \subseteq \omega'.L$
  The set of interfaces of an object may increase over time. Notice that an object may have several interfaces (or access point). This condition indicates that ω *evolves* into ω' during the rewrite step, since the identity of an object is determined by its interfaces.
- $\neg(\omega' \parallel \Omega \rightarrow \perp)$
  The second side of the rule schema should not degenerate into an invalid distributed configuration. This is a local condition on the generation of new interface identifiers. It is complemented by the condition (†) introduced below, which prevents the evolution of distributed configurations into invalid ones.
- Let $A = \omega.K \cup \aleph.arg \cup \omega'.L \cup \Omega.L$. Then:

  $$\omega'.K \subseteq A \ \wedge \ \aleph'.tgt \subseteq A \ \wedge \ \aleph'.arg \subseteq A \ \wedge \ \Omega.K \subseteq A$$

  This constraint characterizes encapsulation, expressing that an object knowledge of its environment can grow only through interactions.
- Let ρ be a bijection on Id. We note $\omega.\rho$ the object resulting from substituting $\omega.L.\rho$ and $\omega.K.\rho$ to $\omega.L$ and $\omega.K$, respectively. In the same way, $s.\rho$ denotes the signal obtained from signal s by replacing $s.tgt$ by $s.tgt.\rho$ and $s.arg$ by

$s.arg.\rho$. Then:

$$\Diamond(\omega, \aleph, \omega', \aleph', \Omega) \Leftrightarrow \Diamond(\omega.\rho, \aleph.\rho, \omega'.\rho, \aleph'.\rho, \Omega.\rho)$$

This condition expresses the independence of object behavior and object definition from the actual choice of interface identifiers.

Intuitively, each rule conforming to ($\Diamond$) describes a possible state transition of an object ω. An equivalent reading is that an object is characterized by a transition system whose states are local configurations.

2.2 Characterizing distributed computations

Distributed configurations conforming to CM_{basic} can be characterized by a rewrite rule (♣) and an axiom (†). Rule (♣) is the conditional rewrite rule given below:

$$(\clubsuit) \qquad \frac{\Diamond(\omega, \aleph, \omega', \aleph', \Omega)}{\omega \parallel \aleph \rightarrow \omega' \parallel \aleph' \parallel \Omega}$$

Rule (♣) defines the allowed transitions between distributed configurations. It needs to be complemented with another constraint to ensure the uniqueness of interface identifiers in distributed configurations. This is the role of predicate WF, given by (†), that defines valid distributed configurations:

$$(\dagger) \quad \mathsf{WF}(d) \equiv \neg(d \rightarrow \bot)$$

A distributed configuration d that conforms to CM_{basic} is thus a valid distributed configuration (i.e. a distributed configuration d such that $\mathsf{WF}(d)$) that is obtained through the application of rule (♣).

2.3 Discussion

CM_{basic} described above corresponds to the so-called operational subset of the ODP computational model. Actually, the ODP computational model comprises two kinds of operation invocations: one way asynchronous invocations and two-way asynchronous invocations. Only the first have been presented in the formal model above but the second can easily be recovered as a specialization of the above rewrite rule schema*.

CM_{basic} exhibits "mobility" in much the same way as in the π-calculus. This statement can be made formal by comparing the expressive power of the (asynchronous) π-calculus (as defined e.g. in [3]) and of CM_{basic}. Let us say, as in [17], that a computational model M_2 equipped with equivalence $\approx_2$ is

*See [26] for details.

more expressive than a computational model M_1 equipped with equivalence $\approx_1$ when there is a fully abstract encoding $\mathcal{T}$ from M_1 to M_2, i.e. when for all P and Q in M_1 we have $P \approx_1 Q \Leftrightarrow \mathcal{T}(P) \approx_2 \mathcal{T}(Q)$

Theorem 1 *CM_{basic} is more expressive than the asynchronous π-calculus up to their weak barbed congruences.*

Weak barbed congruence is defined for the asynchronous π-calculus in [3]. It can be readily defined for CM_{basic} as follows. Observability in distributed configurations is first defined through predicates $\downarrow_s$ on distributed configurations, where $s \in \mathsf{Signal}$: $s \downarrow_s$, and $d_1 \parallel d_2 \downarrow_s$ if $d_1 \downarrow_s$ or $d_2 \downarrow_s$.

We then define weak barbed bisimulation for distributed configurations as follows:

Definition 1 *A symmetric relation R_ρ on distributed configurations, indexed by a bijection ρ on* Id*, is a weak barbed bisimulation if, for all $(d_1, d_2) \in R_\rho$, we have:*

1. *if $d_1 \rightarrow d_1'$, then $d_2 \rightarrow d_2'$ such that $(d_1', d_2') \in R_\rho$;*
2. *if $d_1 \downarrow_s$ then $d_2 \downarrow_{s.\rho}$.*

Two distributed configurations d_1 and d_2 are weak barbed bisimilar if there is a weak barbed bisimulation R_ρ such that $(d_1, d_2) \in R_\rho$.

Finally we define weak barbed congruence for distributed configurations:

Definition 2 *Two distributed configurations d_1 and d_2 are weak barbed congruent if, for all distributed configurations d such that $d.K \subseteq d.L \cup d_1.L$ and $d \parallel d_1 \neq \perp$, there is a weak barbed bisimulation R_ρ such that $(d \parallel d_1, d.\rho \parallel d_2) \in R_\rho$.*

Loosely speaking, we can paraphrase the above in saying that two distributed configurations are weak barbed congruent if they are bisimilar under all closed*, valid contexts.

The proof of Theorem1 is too long to be reproduced here in detail but it is relatively standard. It involves the encoding of each π-calculus process as an object, the interpretation of a term $P_1 \mid P_2$ of the π-calculus as the creation of a an object associated with process P_2, and the modeling of each π-calculus

*Intuitively, a closed context for a distributed configuration d_1 is a distributed configuration d that does not have any references to other objects than those appearing in d itself or in d_1. Notice that it is always possible to close a distributed configuration with appropriate dummy objects (i.e. objects that cannot change state) bearing the required interface identifiers. The constraint that contexts be closed is required to avoid the illicit "guess", by the context, of future interface identifiers generated by transitions from d_1.

channel by an object that provides a non-deterministic dispatching of signals towards registered objects. The latter is required to capture the ability of a π-calculus channel to be used for reception by several distinct processes. The appendix to this paper provides more details on this encoding.

Mobility in CM_{basic} is achieved through the communication of interface identifiers, but it may be implemented in different ways. For instance, the model may, classically, be implemented using some form of RPC protocol. But it may also be implemented by moving (client) objects to the site where their correspondents (servers) reside (e.g. as an instance of the so-called client-agent-server architecture formalized in [18]). As such, CM_{basic} does not provide explicit control over the physical locations or resources required to support objects and distributed configurations. Instead, the model can be understood as taking a view of "maximum distribution" of objects: an object performs local computations (as represented by rules conforming to the schema $(\Diamond)$) and communicate asynchronously by exchanging signals with a priori distant other objects.

The formal model presented in the previous sections has a lot in common with the abstract actor model described in [34]. In fact, except for the specification of the fairness conditions for actor computations, the basic computational model provides a direct modeling of actors as objects with a single interface.

3 BINDING OBJECTS

CM_{basic} is limited in its ability to directly capture features of distributed platforms and in its modeling capabilities. In particular, even though it makes an assumption of maximum distribution, it remains limited in its capacity to describe the different forms of communications and of failures that may occur in an actual distributed setting. Failures that can be taken into account are essentially local failures (through the definition of appropriate faulty object behavior). The model does not account for the diversity of network situations that may arise in a real environment, with different assumptions and guarantees being provided by underlying networks. The basic form of communication supported by CM_{basic} does not account for other forms of communications required and supported in actual distributed environments, that exhibit a large variety of communication semantics and topologies (e.g. point-to-point or multipoint continuous media communication, blackboard communication schemes, group communication schemes, etc.). Even within the basic RPC-like model of asynchronous operation invocation, a large variety of semantics can be provided and made available to applications (see e.g. systems such as Spring [19], SOR [30], or Spin [6]. A distributed system that has adopted the notion of binding as a means to provide a highly flexible distributed platform is described in [11].

For all these reasons, the ODP computational model has been defined to

comprise an explicit notion of communication object, called a binding object. A binding object mediates communication between other objects. The ODP computational model does not prescribe a particular communication semantics for binding objects. Instead, a binding object, just as any other object, may have an arbitrary behavior. If necessary, a binding object can encapsulate behavior corresponding to faulty network behavior. A formal presentation of the full ODP computational model with binding objects can be given using the elements introduced in the previous section. The resulting model is called $\mathrm{CM}_{\mathrm{bindings}}$.

Distributed configurations are now restricted to be finite sets of objects in parallel, i.e. terms of the form $\omega_1 \parallel \ldots \parallel \omega_n$. Distributed configurations d conforming to $\mathrm{CM}_{\mathrm{bindings}}$ are characterized by the conditional rewrite rule (♠) together with the constraint that they be valid (i.e. that $\mathsf{WF}(d) \equiv \mathsf{true}$, where WF is given by (†)):

$$(\spadesuit) \qquad \frac{\Diamond(\omega, \aleph, \omega', \aleph', \Omega) \ \wedge \ \forall j \in J \ \Diamond(\omega_j, \aleph_j, \omega'_j, \aleph'_j, \Omega_j) \ \wedge \ C_1}{\omega \ \|_{j\in J} \ \omega_j \rightarrow \omega' \parallel \Omega \ \|_{j\in J} \ (\omega'_j \parallel \Omega_j)}$$

where J is a finite index set and C_1 is the condition defined as follows:

$$C_1 \ \equiv \ (\aleph = \bigcup_{j\in J} \aleph'_j) \ \wedge \ (\aleph' = \bigcup_{j\in J} \aleph_j)$$

The resulting model is thus a synchronous model, where, intuitively, synchronous interaction are local interactions, and where communications between distant objects are systematically mediated by bindings. Notice that the above formalization does not make any difference between objects and binding objects. In the general case, it is merely the intention associated with the conveyance of information from one object to another that characterizes a binding object.

Note that the rule (♠) constitutes a particular modeling choice, since the RM-ODP only specifies, informally, that a "signal occurrence is an atomic action". We could have made other choices, compatible with the informal assertions of the RM-ODP. For instance, we could have an alternative rule where the interactions remain synchronous but exhibit a clear causal relation, with some object being the initiator of the transition.

Because of its synchronous nature, rule (♠) may be difficult or even impossible to implement in presence of arbitrary failures if ω, appearing in the rule, is to be implemented as a distributed object: in this case, the rule would require a form of distributed rendez-vous. The condition Dis defined below imposes an additional condition on distributed objects that solves the problem, confining synchronous interaction as a primitive for local communication only.

The rule (♠) does not capture a distinction between local and distant objects. In $\mathrm{CM}_{\mathrm{basic}}$, the distinction was made explicit through the use of asynchronous messages. We can capture the basic asynchrony present in $\mathrm{CM}_{\mathrm{basic}}$ through the predicate Dis, defined as follows. Let ω be an object and let

$J = \omega.L$. Let t_j $(j \in J)$ be triples of the form $(\omega_j, \aleph'_j, \Omega_j)$. Let $\aleph_0$, $\aleph_j$ $(j \in J)$ be finite sets of signals such that $\forall j \in J$, $\aleph_j.tgt = j$ and $\aleph_0 = \bigcup_{j \in J} \aleph_j$. Finally let $\alpha_1, \ldots, \alpha_n, \beta, \gamma_1, \ldots, \gamma_n$ be the following assertions:

- $\alpha_j \equiv Lhs(\omega, \aleph_j) \rightarrow Rhs(t_j)$
- $\beta \equiv Lhs(\omega, \aleph_0) \rightarrow Rhs(t_0)$
- $\gamma_j \equiv Lhs(\omega, \aleph_0 \setminus \aleph_j) \rightarrow Rhs(t_0)$

We can then define:

$$\mathsf{Dis}(\omega) \equiv \wedge \bigwedge_{j \in J} \alpha_j \Rightarrow \exists t_0, \beta \wedge (\bigwedge_{j \in J} \gamma_j)$$
$$\wedge \beta \Rightarrow \exists t_1, \ldots, t_n, (\bigwedge_{j \in J} \alpha_j) \wedge (\bigwedge_{j \in J} \gamma_j)$$

Predicate Dis corresponds to a kind of confluence property and constitutes a condition for objects to be implementable in a general distributed setting, in presence of arbitrary forms of failures. Interfaces of object ω in the above definition can be construed as distant interfaces, located on different sites*. Interactions on one interface do not depend on interactions on other distant interfaces. Such interactions can be implemented without the recourse to a distributed consensus protocol which would be expensive or even impossible [16, 21] to implement in the presence of arbitrary failures. In other terms, in presence of arbitrary failures, a distributed atomic interaction — as would be mandated by the general case of rule (♠) — cannot be taken as primitive and must be explicitly modeled as a set of more primitive actions. Predicate Dis, as a characterization of distributable objects, confines synchronous interactions to the local case, thus making rule (♠) a suitable, primitive model for distributed computations.

$\mathrm{CM}_{\mathrm{bindings}}$ constitutes a conservative extension of $\mathrm{CM}_{\mathrm{basic}}$ in that all features of $\mathrm{CM}_{\mathrm{basic}}$ can be fully recovered* in $\mathrm{CM}_{\mathrm{bindings}}$. This merely involves modeling the sending of an asynchronous message in $\mathrm{CM}_{\mathrm{basic}}$ as the creation of an elementary binding object whose sole responsibility is to invoke a signal on a given target interface.

4 QUALITY OF SERVICE

Apart from binding objects, the ODP computational model also identifies the notion of *environment contracts*. Intuitively, an environment contract specifies non-functional aspects of an object behavior, notably its quality of ser-

*ω is thus a distributed object, with each interface on a different site. This is again taking a view of maximum distribution.

*I.e. by a fully abstract encoding.

vice (QoS) constraints. The notion of quality of service should be understood here in a broad sense, covering both real-time constraints such as delay or throughput constraints, and dependability constraints such as availability or fault-tolerance constraints. The ODP computational model [20] thus indicates that a computational object specification may contain additional behavioral constraints in the form of an environment contract specification. The introduction of environment contracts is motivated by the desire to provide a complete characterization of a distributed system or application from a computational point of view, abstracting away from the detailed mechanisms used to meet quality of service constraints. Such a characterization is required for a wide variety of applications, from real-time process control, and mission-critical applications to interactive multimedia applications, whose correct behavior depend on the provision of a suitable quality of service. The introduction of environment contracts also follows an obvious trend in distributed systems construction to explicitly support quality of service constraints. A detailed presentation of system issues involved and some proposals can be found in the recent book [8].

Whereas the notion of binding is well specified in [20], the notion of environment contract associated with an object remains under-specified. We discuss in this section some possibilities for formalizing the notion.

The term *contract* emphasizes a deontic view of such constraints (see e.g. [32] for a discussion): an environment contract specifies at the same time expectations from an object on its environment and guarantees or obligations that the object will fulfill in return. Informally, the behavior of an object conforms to a given contract if it does not violate the obligations of the contracts at least as long as the contract expectations are fulfilled by the object environment.

Formalizing this notion of contract requires, as a minimum, the capture of appropriate notions of failures, and the introduction of an explicit notion of time in the computational model. For instance, capturing dependability constraints requires the definition of certain failure modes for objects and groups of objects. First attempts in this direction, that introduce simple primitives for failures* in the context of mobile process calculi include [2] and [18]. Another attempt, dealing especially with real-time constraints, is [12], which extends the ODP computational model with time and interaction failures, and formalizes environment contracts as specific (failure) observers.

Drawing on these ideas, it would be interesting to define an extension of $\mathrm{CM}_{\mathrm{bindings}}$ dealing explicitly with the possibility of failures. Let us call $\mathrm{CM}_{\mathrm{failures}}$ the envisaged model, and let us consider some possible features of the model. A first feature of $\mathrm{CM}_{\mathrm{failures}}$ would be the explicit presence of time in the model. This can be done using timed rewriting logic models discussed in [25] or directly on our object model. For instance, adopting the basic ideas behind timed automata (see e.g. [1]), we could consider pairs (ω, u) as the

*Essentially, primitives for fail-silent objects.

basic units in distributed computations, where ω is an object whose state depends on u, a clock variable that takes its values in a time domain Time. Object behaviors would now be described using two kinds of rewrite rules:

- a modified ($\Diamond$) rule schema of the form:

$$(\Diamond_t) \quad Lhs((\omega, u), \aleph) \rightarrow Rhs((\omega', u), \aleph', \Omega)$$

  where Ω is now a finite set of pairs of the form (ϖ, v).
- new labeled rules for allowing time to pass of the form:

$$t : (\omega, u) \rightarrow (\omega, u + t)$$

  where $t \in$ Time.

Notice that the explicit presence of rules with time labels is required in the definition of an object to allow time to pass. This allows an object to not let time pass, which in turn allows to express urgency for transitions. The model could be completed with time-related axioms such as *time determinism*, *time additivity*, *deadlock-freeness*, *action persistence*, etc. (see e.g. [1, 27] for a discussion of properties of timed models).

We would then have the following modified ($\spadesuit$) rule for distributed configurations:

$$(\spadesuit_t) \quad \frac{\Diamond_t(\omega, u, \aleph, \omega', \aleph', \Omega) \ \wedge \ \forall j \in J \ \Diamond_t(\omega_j, u_j, \aleph_j, \omega'_j, \aleph'_j, \Omega_j) \ \wedge \ C_1}{(\omega, u) \ \|_{j \in J} \ (\omega_j, u_j) \rightarrow (\omega', u) \ \| \ \Omega \ \|_{j \in J} \ ((\omega'_j, u_j) \ \| \ \Omega_j)}$$

And a new labeled rule for timed distributed configurations:

$$\frac{t : d_1 \rightarrow d'_1 \ \wedge \ t : d_2 \rightarrow d'_2}{t : d_1 \ \| \ d_2 \rightarrow d'_1 \ \| \ d'_2}$$

A second feature of $\mathrm{CM}_{\mathrm{failures}}$ would be to provide the ability to capture and detect various forms of failures. For instance, we could capture the notion of a fail-silent object through the following rules:

1. $\Downarrow (\omega) : (\omega, u) \rightarrow (\Downarrow \omega, u)$
2. the behavior of $\Downarrow \omega$ is given by the set of rules: for all $t \in$ Time, $t : (\Downarrow \omega, u) \rightarrow (\Downarrow \omega, u + t)$

Added to the behavior of an object ω, these rules allow ω to fail silently (i.e. to evolve into the failed object $\Downarrow \omega$). Note that the only behavior of a failed object is to let time pass. More complex behaviors would be possible to reflect different failure modes, or to model the possibility of recovery. The label $\Downarrow (\omega)$ can be interpreted as a predicate asserting that object ω has failed.

In the same vein, we could capture the notion of interaction failure, or missed rendez-vous, through the definition of a predicate $\Downarrow (s)$, where s is a signal, asserting that s could not be exchanged successfully between two

objects. Let $\mathsf{locked}_t(\omega, u)$ assert that (ω, u) can only progress by letting time pass:

$$\mathsf{locked}_t(\omega, u) \equiv \neg\, \exists \aleph, \omega', \aleph', \Omega,\ \Diamond_t(\omega, u, \aleph, \omega', \aleph', \Omega)$$

Let us define also a predicate urgent as:

$$\begin{aligned} \mathsf{urgent}(\omega, u, \omega', s, \Omega) \equiv\ & \wedge\, \Diamond_t(\omega, u, \emptyset, \omega', \{s\}, \Omega) && (1) \\ & \wedge\, \neg(\exists t,\ t :\ (\omega, u) \rightarrow (\omega, u + t)) && (2) \end{aligned}$$

We can now define the following rule:

$$\frac{\mathsf{urgent}(\omega_1, u_1, \omega_1', s, \Omega_1)\ \wedge\ s.tgt \in \omega_2.L\ \wedge\ \mathsf{locked}_t(\omega_2, u_2)}{\Downarrow(s) :\ \omega_1 \parallel \omega_2 \rightarrow \omega_1' \parallel \omega_2 \parallel \Omega_1}$$

The rule expresses the fact that an interaction failure occurs, with signal s failing to be transmitted from ω_1 to ω_2, when ω_1 must urgently transmit s while ω_2 cannot receive it.

The $\Downarrow$ predicates just introduced, can be used to define failure detectors, much in the same sense as failure detectors introduced in [9]. $\Downarrow(s)$, in particular, provides just the basic observation of interaction failure that is required by the calculus of object contracts introduced in [12]. A contract, as defined in this work, is merely a process which observes and arbitrates the collective behavior of configurations of objects. A contract observes objects and depending on the outcome of their interactions (success or failure) may identify and incriminate faulty objects. [12] also defines other contract-related notions such as assumption, obligation, realization an refinement, and it provides a composition theorem which gives sufficient conditions for collections of objects, each satisfying a given contract, to be composed. A similar, more general endeavor may be attempted for $\mathrm{CM}_{\mathsf{failures}}$, thus resulting in a formalization of environment contracts as specialized failure observers. In particular, one could envisage introducing observers corresponding to the different forms of failure detectors discussed in [9], providing a suitable basis for specifying fault-tolerant distributed systems.

5 TOWARDS A REFLECTIVE COMPUTATIONAL MODEL

We have reviewed in previous sections the present contents of the ODP computational model. With its combination of binding objects and environment contracts, the model obtained is already quite rich, but is it sufficient ? Considering the ODP computational model as the abstract programming interface of a (distributed) virtual machine incites us to review different issues currently under investigation in the distributed systems and distributed programming language communities.

5.1 Components and software architecture styles

The ODP computational model as defined by $CM_{bindings}$ remains fairly low-level. Several authors argue in favor of a higher-level, *component-based* approach to the construction of complex software systems, and distributed systems in particular (see e.g. [23, 28, 5, 31]). They also point out the necessity to support different architecture styles such as those discussed in [31], e.g. dataflow, object-based, event-based, data-centered, rule-based, to account for the diversity of design possibilities. Most of these proposals suggest the introduction of different forms of connectors to construct arbitrary configurations of components. Leaving aside language issues, which do not concern us here, we can remark that components in these proposals, whether primitives or composite, can invariably be modeled as computational objects, i.e. whose behavior can be defined as a set of transitions conforming to the ($\Diamond$) schema. A more important observation is that the notion of binding object, as defined in section3, subsumes that of connector, e.g. as envisaged in [23] or [31]. The ODP computational model does not prescribe the way objects are constructed, but nothing prevents the specification of an object as a parallel composition of a set of computational objects interconnected by binding objects. Since binding objects can embody any communication semantics, any form of connector can be construed as a binding object. For instance, binding objects can be defined that provide event multicasting facilities (e.g. similar to event channels defined in [29]), or a form of generative communication à la Linda [14]. The latter example could be realized, e.g. following the formal specification of generative communication provided in [10]. In that respect, $CM_{bindings}$ provides an excellent support for component-based approaches to distributed system programming.

5.2 Object migration, resource control and reflection

The ODP computational model adopts a standpoint which we have qualified as a *maximum distribution* standpoint, where each object is potentially located on a different site. In fact, since each object may indeed be a distributed object, as we saw in section 3, each interface can be considered as potentially located on a different site. This view is appropriate if no finer control over object location is required or can be left for a later phase in system construction. An explicit modeling of locations is necessary, however, if such a control is required — as would be the case e.g. for purposes of explicit load-balancing. [2] and [18] provide two examples of how locations can be introduced in mobile process-calculi. In our context, the notion of location could be introduced in a similar way to that of the join-calculus [18]. To provide even more flexibility and control, we could go beyond the mere modeling of locations as names and

consider locations as objects in their own right, with a specific operator for associating (standard) computational objects and locations.

As a simple illustration, we consider the following extension to $\mathrm{CM}_{\mathrm{basic}}$. Distributed configurations are now parallel ($\|$) compositions of messages and of localized configurations of the form $l : [d]$, where l is a location and d is a standard $\mathrm{CM}_{\mathrm{basic}}$ distributed configuration*. We extend the notion of message to cover also messages of the form $l.n\langle a_1, \ldots, a_n\rangle$, where l is a location, $n \in \mathsf{Name}$ is an operation name, and each of the a_i is either an interface identifier or an object. Object transitions now obey the following rewrite rule schema:

$$(\Diamond_l) \quad Lhs(\omega, \aleph) \rightarrow Rhs(\omega', \aleph', \Omega)$$

where Ω is a set of localized configurations of the form $l : [\varpi_1 \parallel \ldots \parallel \varpi_n]$. Assuming that all locations respond at least to the message $\mathsf{go}\langle . \rangle$, our new model can be defined by the following rewrite rules:

- Let Ω be such that $\Omega = \Omega_l \cup \Omega_{\bar{l}}$ with $\Omega_l = l : [\omega_1 \parallel \ldots \parallel \omega_n]$ and $\Omega_{\bar{l}} = \{l_1 : [d_1], \ldots, l_p : [d_p]\}$ with $l_j \neq l$ for all $j \in \{1, \ldots, p\}$. Let $\aleph'$ be such that $\aleph' = \aleph'_l \cup \aleph'_{\bar{l}}$ with $\aleph'_l.tgt \subseteq (\omega' \parallel \Omega_l).L$, and $\aleph'_{\bar{l}}.tgt \cap (\omega' \parallel \Omega_l).L = \emptyset$. Then:

$$(\clubsuit_l) \quad \frac{\Diamond_l(\omega, \aleph, \omega', \aleph', \Omega)}{l : [\omega \parallel \aleph] \rightarrow l : [\omega \parallel \aleph'_l \parallel_{i=1\ldots n} \omega_i] \parallel \aleph'_{\bar{l}} \parallel \Omega_{\bar{l}}}$$

  Rule schema $(\Diamond_l)$ specifies the general form object transitions may take. Notice that new objects may be created in different locations (which must be known to the object ω that create them, i.e. one must have $\{l_1, \ldots, l_p\} \subseteq \omega.K$*). Object behaviors may additionally comprise transitions of the form:

$$\omega \rightarrow l.\mathsf{go}\langle\omega\rangle$$

  With the above rewriting rule an object may send itself to a different location.

- $$\frac{\aleph.tgt \subseteq d.L}{l : [d] \parallel \aleph \rightarrow l : [d \parallel \aleph]}$$

  This rule just specifies that messages flow asynchronously from locations to locations.

- $$l : [d] \parallel l.\mathsf{go}\langle\omega\rangle \rightarrow l : [d \parallel \omega]$$

  This rule specifies the semantics of the go message, which is to transmit an object to a target location.

---

*Notice that we could just as simply have allowed extended distributed configurations inside a $l : [.]$ context, thereby allowing trees of locations, much as in [18].

*To simplify notations, we identify here locations and identifiers of their pre-determined interface, i.e. that which supports the **go** operation.

Specifying locations on which objects may reside provides applications explicit control over the placement of objects, presumably for purpose of load-balancing, security, fault-tolerance or availability. The ability to migrate objects from location to location as illustrated above with the $\mathsf{go}\langle\rangle$ message, offers dynamic control over object placement, allowing communication trade-offs. Some comments are in order. Note first that the last two rules are in fact instances of ($\clubsuit_l$) if one note that $l : [d]$ corresponds to a composite object as per the above section, with d as an internal state, and which "exports" interfaces from d. Note then that the resulting model is a conservative extension of $\mathrm{CM}_{\mathrm{basic}}$: the latter can be recovered simply by having one object per location, formalizing the implicit "maximum distribution" principle discussed in section 2.

Controlling the placement of objects is not sufficient for applications that require quality of service guarantees, notably real-time guarantees. Such applications typically require the ability to access and control the characteristics of the resources they use, e.g. processors, memory, I/O. Even in non real-time contexts, applications may require e.g. fine-grained control over the degree of parallelism with which they run, thus requiring access to thread and process-like entities within a given node or location. More generally, providing applications with fine-grained control over their implementation can be useful to easily adapt applications to different environments, and to different non-functional requirements. This has lead researchers to consider using reflection in distributed systems, e.g. [4, 13, 15, 22, 35]. Having reflection as a feature of our reference computational model for open distributed systems is attractive for it promises to unify in one setting the different extensions we have discussed in this paper. Let us just mention what that would entail. First, introducing reflection in the model will require reifying the different mechanisms implicit in distributed computations, corresponding roughly to the seven *aspects* of [22]: sending, accepting, queuing, and receiving messages; describing object state and object transitions; executing object transitions. Second, we can note that reifying execution aspects, would mean reifying most of the ODP engineering model [20], with its notions of nodes, capsules, clusters, and threads. Locations, as described above, already provide a first approximation of RM-ODP nodes. Capsules, clusters and threads can be introduced similarly. In particular, note that all can be understood as specialized forms of locations, with exclusive access in the case of threads. Lastly, let us note that the above rules for location, already capture some features of message acceptance and queuing.

6 CONCLUSION

We have reviewed the ODP computational model and its formal semantics, developed using a conditional rewriting logic approach. This lead us to a series of computational models, from the most basic one, $\mathrm{CM}_{\mathrm{basic}}$, that cor-

responds to the operational subset of the ODP computational model, and $CM_{bindings}$ that captures the full model with binding objects. Capturing the ODP notion of environment contracts and their associated quality of service constraints has been discussed as part of $CM_{failures}$. Prominent features of $CM_{failures}$ were discussed, suggesting ways to model failures in the model and suggesting the use of specific failure observers to implement or characterize environment contracts. Finally, we have discussed several issues in open distributed systems construction which, taken together, call for a more radical approach to the flexibility and adaptability — in other terms, to the openness — of distributed systems. This leads us to consider elements of a reflective computational model, $CM_{reflect}$, which may provide an elegant way to integrate numerous features under one highly flexible distributed computational model. Going beyond "basic" and "bindings", towards "failures" and "reflect", looks like a promising research agenda.

REFERENCES

[1] R. Alur, D. Hill: "A theory of timed automata" – Theoretical Computer Science 126, pp 183-235, 1994.

[2] R. Amadio, S. Prasad: "Localities and Failures" – in Proceedings 14th Conference on Foundations of Software Technology and Theoretical Computer Science, LNCS 880, Springer Verlag, 1994.

[3] R. Amadio, I. Castellani, D. Sangiorgi: "On bisimulations for the asynchronous π-calculus" – in Proceedings CONCUR '96, LNCS 1119, Springer Verlag, 1996.

[4] Barga, R., C. Pu, "Reflection on a Legacy Transaction Processing Monitor" – Proceedings of Reflection 96, G. Kiczales (ed), pp 63-78, San Francisco, USA, April 1996.

[5] L. Bellissard, S. Ben Atallah, A. Kerbrat, M. Riveill: "Component-based programming and application management with Olan" – Object-based parallel and distributed computation, LNCS 1107, Springer Verlag, 1995.

[6] Bershad, B.N., S. Savage, P.Przemyslaw, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, S. Eggers, S.: "Extensibility, Safety and Performance in the SPIN Operating System" – in Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95), pp 267-284, Copper Mountain CO, U.S.A., December 1995.

[7] A. Birrell, Greg Nelson, Susan Owicki, Edward Wobber: "Network Objects", SRC Research Report 115, Digital Systems Research Center, December 1995.

[8] G. Blair, J.B. Stefani: "Open Distributed Processing and Multimedia" – Addison-Wesley 1997.

[9] T.D. Chandra, S. Toueg: "Unreliable failure detectors for reliable distributed systems" – Journal of the ACM, Vol. 43, No. 2, pp. 225-267,

March 1996.

[10] P. Ciancarini, R. Gorrieri, G. Zavattaro: "Towards a calculus for generative communication" – Proceedings 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS '96), E. Najm and J.B. Stefani eds, Chapman & Hall 1996.

[11] F. Dang Tran, V. Perebaskine, J.B. Stefani, B. Crawford, A. Kramer, D. Otway: "Binding and Streams: the ReTINA approach", in Proceedings TINA '96 International Conference, Heidelberg, Germany, September 1996.

[12] A. Fevrier, E. Najm, J.B. Stefani: "Contracts for ODP" – in Proceedings 4th AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software, Ciudad de Mallorca, Mallorca, Spain, May 1997.

[13] Forman, I. R., S. Danford, H. Madduri, "Composition of Before/ After Metaclasses in SOM" – Proceedings of OOPSLAÕ94, pp427-439, ACM, 1994.

[14] D. Gelernter: "Generative communication in Linda" – ACM Transactions on Programming Languages and Systems 7(1), pp. 80-112, 1985.

[15] Gowing, B., V. Cahill, "Meta-Object Protocols for C++: The Iguana Approach" – Proceedings of Reflection 96, G. Kiczales (ed), pp 137-152, San Francisco, USA, April 1996.

[16] M. Fischer, N. Lynch, M. Paterson: "Impossibility of distributed consensus with one faulty process" – Journal of the ACM 32(2), pp. 374-382, April 1985.

[17] C. Fournet, G. Gonthier: "The reflexive chemical abstract machine and he join-calculus" – 23rd ACM Symposium on Principles of Programming Languages (POPL), January 1996.

[18] C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, D. Remy: "A calculus of mobile agents" – in Proceedings CONCUR '96, LNCS 1119, Springer Verlag, 1996.

[19] G. Hamilton, M. Powell, J. Mitchell: "Subcontract: a flexible base for distributed programming", Proceedings of the 14th Symposium on Operating Systems Principles, Asheville NC, December 1993.

[20] ITU-T Recommendation X.903 | ISO/IEC International Standard 10746-3: "ODP Reference Model: Prescriptive Model" – 1995.

[21] N. Lynch: "Distributed Algorithms" – Morgan Kaufmann, 1996.

[22] McAffer, J., "Meta-Level Architecture Support for Distributed Objects" – Proceedings of Reflection 96, G. Kiczales (ed), pp 39-62, San Francisco, USA, April 1996.

[23] J. Magee, N. Dulay, J. Kramer: "Specifying distributed software architectures" – Proceedings European Software Engineering Conference, LNCS, Springer Verlag, 1995.

[24] J. Meseguer: "Conditional rewriting logic as a unified model of concurrency" – Theoretical Computer Science 96, pp. 73-155, 1992.

[25] J. Meseguer: "Rewriting logic as a semantic framework for concurrency: a

progress report" – in Proceedings CONCUR '96, LNCS 1119, Springer Verlag, 1996.

[26] E. Najm, J.B. Stefani: "A formal semantics for the ODP computational model" – Computer Networks and ISDN Systems 27, pp.1305-1329, 1995.

[27] X. Nicollin, J. Sifakis: "An overview and synthesis on timed process algebras" – Proceedings 3rd Workshop on Computer-Aided Verification, Alborg, Denmark, July 1991.

[28] O. Nierstrasz, J.G. Schneider, M. Lumpe: "Formalizing composable software systems - A research agenda" – Proceedings 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS '96), E. Najm and J.B. Stefani eds, Chapman & Hall 1996.

[29] Object Management Group: "The Common Object request Broker: Architecture and Specification", CORBA V2.0, July 1995.

[30] M. Shapiro: "A binding protocol for distributed shared objects", 14th International Conference on Distributed Computer Systems (ICDCS), Poznan, Poland, June 1994.

[31] M. Shaw, D. Garlan: "Software architecture: perspectives on an emerging discipline" – Prentice-Hall 1996.

[32] J.B. Stefani: "Computational Aspects of QoS in an object-based, distributed systems architecture" – Proceedings 3rd International Workshop on Responsive Computer Systems, Lincoln, NH, USA, September 1993.

[33] Sun Microsystems: "Java Remote Method Invocation Specification", Technical Report, Sun Microsystems, Moutain View CA, USA, May 1996.

[34] C. Talcott: "Interaction semantics for components of distributed systems" – Proceedings 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS '96), E. Najm and J.B. Stefani eds, Chapman & Hall 1996.

[35] Yokote, Y., "The Apertos Reflective Operating System: The Concept and Its Implementation" – Proceedings of OOPSLAÕ92, vol. 28 of ACM SIGPLAN Notices, pp 414-434, ACM Press, 1992.

APPENDIX 1 A FULLY ABSTRACT ENCODING OF THE π-CALCULUS IN CM_{BASIC}

In this section, we sketch an encoding of the asynchronous π-calculus in CM_{basic}. We consider the version of the calculus described in [3], with a guarded choice operator. The exact syntax used is given by the following grammar:

$$P ::= \bar{a}b \mid P \mid P \mid \nu aP \mid !G \mid G \tag{3}$$

$$G ::= 0 \mid a(b).P \mid \tau.P \mid G + G \tag{4}$$

The encoding, T, uses an auxiliary environment ρ, to record the association of π-calculus names with certain interface identifiers. An environment ρ thus takes the form: $\rho = \{a_1 \mapsto u_1, \ldots, a_n \mapsto u_n\}$, where $a_i \in \mathsf{Name}$ (Name the set of π-calculus names) and where $u_i \in \mathsf{Id}$. We note $\rho[a \mapsto u]$ the environment which is like ρ except on a: if $a \mapsto v \in \rho$, then $\rho[a \mapsto u] = \rho \backslash \{a \mapsto v\} \cup \{a \mapsto u\}$; if $a \notin \mathit{domain}(\rho)$, then $\rho[a \mapsto u] = \rho \cup \{a \mapsto u\}$. We note $\bar{\rho}$ an environment that has the same domain than ρ and a range that comprises entirely fresh identifiers.

In the definition of the encoding, $\mathcal{T}_\rho(P)$ denotes an object ω such that $\omega.L = \psi(\mathit{range}(\rho))$, and $\omega.K = \mathit{domain}(\rho)$. We assume the existence of two injections ϕ and ψ, from Name into Id, such that $\mathit{range}(\phi) \cap \mathit{range}(\psi) = \emptyset$. We also denote messages thus: $t.u$, where t is the target of the message and u is the (only) argument. Notice that we do not make use of signal names: we just assume a default signal name used for each message. Notice also that, as a convention, when we make use of identifier w in the encoding, it refers to a fresh identifier (i.e. one which is different from the interface identifiers of the current object, and whose uniqueness is ensured through (†)). Instead of using the cumbersome $\Diamond$ notation, we specify directly the behavior of auxiliary objects in the encoding using rewrite rules between distributed configurations, an abuse of notation that is fully justified by rule (♣). Finally, we assume that in a π-calculus process P, all variables have been given suitably different names to avoid name capture.

The encoding is defined inductively as follows ($I = \{1, \ldots, n\}$):

- $\mathcal{T}_\rho(\bar{a}b) = \psi(a).\psi(b)$ if $a, b \in \mathit{domain}(\rho)$.
- $\mathcal{T}_\rho(P) = D_a^\emptyset \parallel \mathcal{T}_{\rho[a \mapsto w]}(P)$ if $a \notin \mathit{domain}(\rho)$ and $a \in FV(P)$.
- $\mathcal{T}_\rho(P_1 \mid P_2) = \mathcal{T}_\rho(P_1) \parallel \mathcal{T}_{\bar{\rho}}(P_2)$ if $\forall a \in FV(P_1 \mid P_2)$, $a \in \mathit{domain}(\rho)$
- $\mathcal{T}_\rho(\nu a P) = \mathsf{nu}(a, P, \rho)$
- $\mathcal{T}_\rho(!G) = \mathcal{T}_\rho(!G) \parallel \mathcal{T}_{\bar{\rho}}(G)$
- $\mathcal{T}_\rho(\sum_{i \in I} \alpha_i.P_i) = \mathsf{choice}(\sum_{i \in I} \alpha_i.P_i, \rho) \parallel_{i \in I} m_i(\alpha)$ with $m(\tau) = \emptyset$ and $m(a(b)) = \phi(a).\rho(a)$

with auxiliary objects defined thus (to simplify notations, we identify objects which can no longer evolve with $\emptyset$):

- $\mathsf{nu}(a, P, \rho) \to D_a^\emptyset \parallel \mathcal{T}_{\rho[a \mapsto w]}(P)$
- $\mathsf{choice}(\sum_{i \in I} \alpha_i.P_i, \rho) \to \mathcal{T}_\rho(P_i)$ if $\alpha_i = \tau$
- $\mathsf{choice}(\sum_{i \in I} \alpha_i.P_i, \rho) \parallel \rho(a).\psi(c) \to \mathcal{T}_{\rho[c \mapsto w]}(P_i\{c/b\})$ if $\alpha_i = a(b)$
- $D_a^A \parallel \phi(a).u \to D_a^{A \cup \{u\}}$
- $D_a^A \parallel \psi(a).u \to D_a^A \parallel v.u$ with $v \in A$

Notice in the encoding above that there are no rules pertaining to $\tau.P$ and $a(b).P$ as these are covered by the rule for the choice operator $\sum_{i \in I} \alpha_i.P_i$.