Formal Coordination of Communicating Entities described with Behavioural Interfaces

Pascal Poizat [1], Gwen Salaün [2]

[1] LaMI
Tour Évry 2
523 place des terrasses de l'Agora
91000 Évry Cedex, France
Email: poizat@lami.univ-evry.fr

[2] INRIA Rhône-Alpes, VASY Project
655 avenue de l'Europe
38330 Montbonnot Saint-Martin, France
Email: Gwen.Salaun@inrialpes.fr

— SPECIF —

Pascal Poizat, Gwen Salaün

Formal Coordination of Communicating Entities described with Behavioural Interfaces
22 p.

Formal Coordination of Communicating Entities described with Behavioural Interfaces

Pascal Poizat, Gwen Salaün

Abstract

The *interaction* paradigm is the foundation of many practical and theoretical computational issues. Roughly speaking, interaction involves communicating entities which evolve in parallel and may synchronize together. Communicating entities are usually viewed through public interfaces due to their black-box foundation. It is now widely accepted that such interfaces have to take into account behavioural information. In this report, we survey existing formal and abstract means to describe coordination of entities with behavioural interfaces, and we illustrate their use on simple e-business examples. We end with a comparison of such coordination means.

Contents

1 Introduction

Interaction is a central issue in computer science and particularly in distributed or concurrent programming. This topic has been widely studied and many works have been achieved in different areas such as coordination / composition of components, Architecture Description Languages (ADLs), middleware technologies, and more recently web services or even within the context of genetic regulatory networks study. Among many issues related to the interaction paradigm, one of the most outstanding ones is the way one can coordinate communicating entities (programs, components, services, processes, agents, etc), *i.e.*, describe, impose or restrict the way they interact together.

Let us introduce some related works mentioned before. *Coordination* aims at *"finding solutions to the problem of managing the interaction among concurrent programs"* [4]. A coordination model is constituted of three parts: entities, a media to connect the components, and the semantic framework of the model. Linda [16] and Manifold [14] are well-known examples of respectively data-driven and control-driven coordination languages. *Software architectures* [24] are organizations of systems as a collection of interacting components. ADLs [35] are modelling notations which focus on the high-level structure of the overall software application. They can be abstract like Wright [2] which is based on the process algebra CSP, or can be more related to the implementation level like Rapide [32]. *Web services* [28] emerged recently and are a promising way to develop applications through the web, their main specificity compared to more traditional software components. Web services are distributed, independent pieces of code which communicate with each other through the exchange of messages. One central issue is to make them working together to perform a given task. The two notions capturing the idea of composition in web services are *choreography* (ensuring correct interactions among services) and *orchestration* (developing new WSs managing other existing ones). Executable applications based on web services can be described using any programming language wrapped into a SOAP envelope, and composed using for example the XML-based language BPEL [3]. An emerging application domain of behavioural specification languages is the study of *genetic regulatory networks* which express interactions between genes and proteins. Here systems behaviours are studied (*e.g.*, model checking biological properties expressed in temporal logic) to propose experiments to biologists or to (in)validate biological theories [11]. Due to the combinatorial complexity of such systems (number of entities involved in, levels of expression for genes and proteins), an interesting direction is to express only the interactions and not the whole behaviours. Formal coordination is only beginning to be studied in this context [18]. To sum up, all these research areas aim at tackling interaction among entities considering a specific context, and taking into account all the constraints involved in it.

In this report, our purpose is to survey and compare efficient formal material to make concurrent entities interact properly. With regards to related work, our goal is to present coordination means in a general way, and therefore not related to a precise context. We introduce different solutions to this problem at an abstract level instead of the programming one (even though connections with the Cexecutable layer may be defined, see [33, 5] for instance). These abstract means are all formally-grounded, then enabling one to use existing verification tools. In such a way, properties of interest can be ensured such as right interaction rounds or correct composition of entities to solve a given task.

It is usually assumed that communicating entities, and in particular the interfaces they are accessed through, share an underlying LTS-based semantics (LTS stands for Labelled Transition System). We emphasize that entities and coordination means are completely independent from one another, but for the fact that the coordination of entities builds on these entities dynamic signatures (LTS labels). We privilege interaction models based on message exchange, compared to the shared memory ones, because in our opinion the former are more adequate to a distributed context, and do not oblige to assume imple-

mentation hypotheses concerning the underlying interaction models. Moreover, we focus on interaction approaches related to operational semantics (instead of coalgebraic [6] or institution-based ones [43] for instance) since we favour user-friendliness, expressiveness and executability criteria.

This survey is the result of experience acquired working on formal approaches to compose and integrate software entities, in particular software components, specification modules and web services [17, 39, 9, 41, 1, 40].

The outline of this report is as follows. In Section 2, we present a formal model of entities described as transition systems. Section 3 introduces and formalises, with respect to this formal model, means to express coordination of entities. In Section 4, we compare the coordination means presented beforehand. We end with concluding remarks in Section 5.

2 A Formal Model for Communicating Entities

In this section, a simple yet general formal model of communicating entities is defined to afterwards introduce and formalise different ways to describe their coordination.

It is now becoming accepted that entities and in particular their *public interfaces*, most of the time the only observable part of a component due to its black-box feature, have to be represented using dynamic behaviours [44, 22, 5, 36, 13]. Entities are described using a simple nondeterministic[1] LTS $< L, S, I, F, T >$ where L is the set of labels, S is the set of states, $I \in S$ is the initial state, $F \subseteq S$ is the set of final states, and T is the transition function of type $S \times L \to S$. Labels may be emissions or receptions denoted in the following using the CSP notation: l! for an emission and l? for a reception. It is sometimes useful to take `tau` actions into consideration to denote internal evolutions of the component. LTSs are introduced here as a semantic model, but may also be viewed as a notation.

Data information (data definitions, operations, parameters of messages) may be used. As far as the message parameters are concerned, their matching consists of making the number and type of parameter correspond [41]. Data description and manipulation is usually not useful to coordinate entities because from an external point of view we just have to deal with the exchange of messages (made explicit through behavioural interfaces). Furthermore, quality of service (*e.g.*, maximum response delay or quality of the result) [12] would be of interest to have at one's disposal an expressive description of interfaces. In our presentation, we focus on purely behavioural aspects as our issue is to describe entities interaction.

The *communication model* is an issue related to the means we use to compose the different entities, therefore this discussion is postponed to the next section. Message receivers are left implicit in the model so as to preserve a higher level of abstraction.

Let us finish with an example of two possible interfaces that will be used in the sequel as a running example (Fig. 1). The left-hand side behaviour describes a store which tries to buy a product, may receive refusals and terminates when it finds a product available. This behaviour takes into account a termination transition labelled with the `tau` action. It means that the store entity gives up purchasing the sought product. On the right-hand side, a supplier receives requests and replies with either a refusal or an acceptance. In the remainder of the article, we will survey several coordination formalisms to describe interactions between instances of such entities.

---

[1]Concrete applications are deterministic. However, nondeterminism may appear, especially when dealing with abstract descriptions of programs.
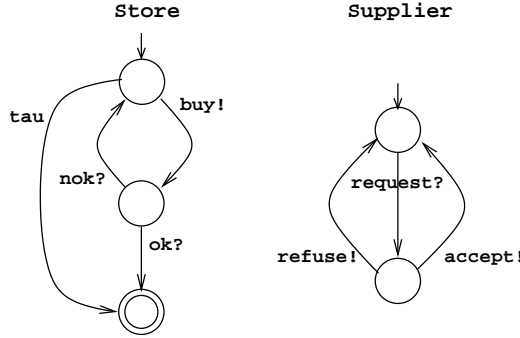
Figure 1: The `Store` and `Supplier` behavioural interfaces

3 Means of Coordination

Coordination may be *implicit*, when only semantic rules are used to coordinate the system entities. Coordination is said to be *explicit* when the coordination protocol is given separately from the entity interfaces. In this section, we introduce successively process algebra, synchronized products, interaction diagrams and temporal logic as explicit coordination means, and semantic glue as an implicit means to coordinate entities. It is worth noting that we will slightly emphasize the presentation of the subsection dedicated to the temporal logic because it is not usual to use it as a formal coordination means.

3.1 Process Algebra

Syntax and semantics. A process algebra [10], *e.g.*, CCS, (T)CSP, LOTOS, π-calculus, Promela, is an abstract language dedicated to the specification of concurrent processes. Process algebras are based on the notion of *action* corresponding to a local evolution of a process. Actions (emissions and receptions) are above all the way to express interactions between two (or more) processes. The basic operators to describe dynamic behaviours are sequence of actions, nondeterministic choice and parallel composition (denoted respectively `.`, `+` and `|` in the following). In addition, process algebras may introduce other constructs and may use slightly different communication models (*e.g.*, binary *vs* multi-way communication, synchronous *vs* asynchronous communication). Most of the time, these calculi are formalised operationally using a LTS-based semantics, as illustrated below with the basic CCS constructs. In the PAR2 rule, a and $'a$ being complementary actions can synchronize together.

$$\frac{}{\alpha.F \xrightarrow{\alpha} F} \quad \text{SEQ} \qquad \frac{F \xrightarrow{\alpha} F'}{F+G \xrightarrow{\alpha} F'} \quad \text{CH}$$

$$\frac{F \xrightarrow{\alpha} F'}{F|G \xrightarrow{\alpha} F'|G} \quad \text{PAR1} \qquad \frac{F \xrightarrow{a} F' \quad G \xrightarrow{'a} G'}{F|G \xrightarrow{\text{tau}} F'|G'} \quad \text{PAR2}$$

Such formalisms are tool-equipped (*e.g.*, CADP [23], CWB-NC [20], LTSA [33], SPIN [27]), making it possible to simulate the possible evolutions of processes, to check properties (*e.g.*, safety properties to ensure that a bad situation never happens) and to verify equivalences or behavioural subtyping [38].

Using process algebra for coordination. Interaction may sometimes be given implicitly using directly the process algebra parallel composition operators as a way to match inputs and outputs of entities. Depending on the operator used and its underlying semantics, many communication models can be

expressed. However, in many cases, a more expressive and explicit notation is required to take more complex coordination protocols into account. In such a case, process algebra may be used as an explicit coordination language. Coordination is then expressed using one (or several) first-class coordinator object(s), here a process algebraic term, possibly using all the available process algebra operators. The whole system is then given as the parallel composition of the entities and the coordinator(s).

Application. For illustration purposes, let us imagine a store which may request to a supplier a quantity of a product. Several suppliers (two below) are ready to receive requests and to reply depending on their local stock. A coordinator Coord is defined whose goal is to iterate the request on both suppliers. It terminates if a positive answer is received or both suppliers reply negatively. In Figure 2, we give an overview of the corresponding coordination schema.
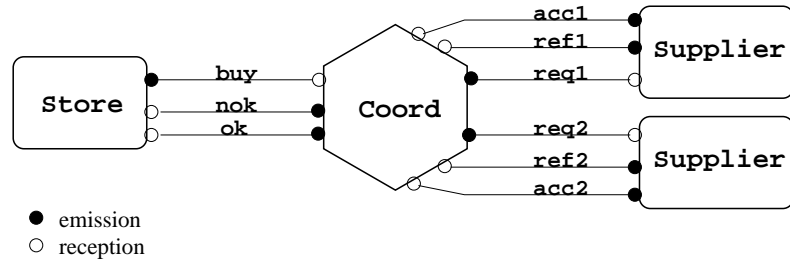

Figure 2: Coordination with process algebra

Now, we introduce the CCS code specifying the behaviour of the different involved entities, and in particular of the coordinator. Processes Supplier and Store correspond to a process algebraic encoding obtained from their automaton description (Fig. 1). In this CCS specification, nil stands for termination and t for the tau action. The [/] notation is used to rename some actions since synchronization occurs only for complementary actions in process algebra (rule PAR2, above). $S \backslash E$ enforces communication in S on actions appearing in the set E. The CWB-NC tool has been used to validate this coordinator specification (animation, deadlock-freeness, and temporal properties model-checking).

```
Store = 'buy.(ok.nil + nok.Store) + t.nil
Supplier = request.('refuse.Supplier + 'accept.Supplier)
Coord = buy.Coord1
Coord1 = 'req1.(acc1.'ok.nil + ref1.Coord2)
Coord2 = 'req2.(acc2.'ok.nil + ref2.'nok.nil)
S = Supplier[req1/request, ref1/refuse, acc1/accept]
  | Supplier[req2/request, ref2/refuse, acc2/accept]
  | Store
  | Coord
System == S \ {buy, ok, nok, req1, req2, acc1, acc2, ref1, ref2}
```

3.2 Synchronized Products

Synchronization vectors are simple and readable textual means to write interactions between entities. They have originally been introduced by Arnold and Nivat [7] and are used in the AltaRica formalism [8]. MEC 5 [25] is a model-checker which makes it possible to handle AltaRica models.

Syntax. Here, we present *ad-hoc* synchronization vectors [41], possibly managing data, which generalize [7, 8] ones and allow good levels of readability and expressiveness. We distinguish four different notations involving different communication models:

- synchronous, one with many: $<a!,\varepsilon,b?,\varepsilon,c?>$

- synchronous, *agreement*, many with many: $<a!,\varepsilon,b!,\varepsilon,c!>$

- synchronous, *negotiation*, many with many: $<a?,\varepsilon,b?,\varepsilon,c?>$

- asynchronous, one to many: $[a!,\varepsilon,b?,\varepsilon,c?]$

Vectors denoted with $< \ldots >$ deal with synchronous communication, whereas $[\ldots]$ ones deal with asynchronous communication. The order of events in vectors permits to identify involved entities. ε is used within synchronization vectors to express that the corresponding entity does nothing while others synchronize. The agreement and negotiation constructs are meaningful when managing data. Given two entities, E_1 and E_2, they respectively mean that: (i) if E_1 can do $p_1!$ and E_2 can do $p_2!$ ($<p_1!,p_2!>$), they have to send exactly the same value to synchronize (*e.g.*, some communicating devices agreeing on some frequency in order to communicate); (ii) if E_1 and E_2 can do respectively $p_1?$ and $p_2?$ ($<p_1?,p_2?>$), then, when they synchronize, some value is taken from the correct domain and passed to E_1 and E_2.

Semantics. $S[E/F]$ denotes the substitution of an entity F by an entity E in a system S made up of several entities (this is similar to the renaming construct for process algebra mentioned before). Given a set V of vectors, the rule for synchronous communication (it works for the three cases introduced above) is defined as:

$$\frac{\begin{array}{c} S = \{E_1, ..., E_n\} \\ V = \{V_1, ..., V_m\} \\ j \in 1..m \ . \ V_j \in V \ . \ V_j =<l_1, ..., l_n> \\ \forall i \in \{1..n\} \ . \ ((l_i \neq \varepsilon \wedge E_i \xrightarrow{l_i} E_i') \vee l_i = \varepsilon) \end{array}}{S \xRightarrow{V_j} S[E_i'/E_i]} \ \mathsf{SP-SYNC}$$

A vector $< l_1, ..., l_n >$ can be fired if for each action l_i different of ε pertaining to the vector, the corresponding entity can evolve along this action ($E_i \xrightarrow{l_i} E_i'$). The global result (conclusion of the rule) is an evolution of all the entities involved in the vector (denoted using $\xRightarrow{V_j}$).

The asynchronous communication rule is formalised adding queues to each entity and firing a transition only if an event with the same label belongs to the corresponding queue. Less strict versions of synchronization may be expressed with less strict versions of the vectors [41], such as multicast communication (*i.e.*, only some of the required receivers may really synchronize, as opposed to broadcast communication where all registered receivers have to synchronize).

Application. As an illustration, let us consider a slight modification of the observable interfaces presented previously. Figure 3 extends the store behaviour to enable it to receive several negative replies before posting another request.

We introduce another possible schema of coordination considering respectively a store and two suppliers. A request is emitted synchronously by the store to both suppliers. Afterwards, each supplier can send a refusal or acceptance to the store. The vectors corresponding to these communications are:
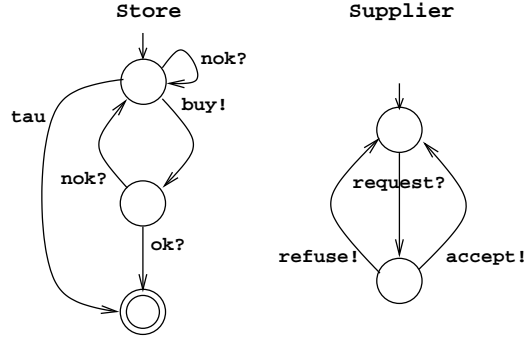
Figure 3: The `Store` and `Supplier` behavioural interfaces (extended)

$$< \texttt{buy!}, \texttt{request?}, \texttt{request?} >$$
$$< \texttt{nok?}, \varepsilon, \texttt{refuse!} >$$
$$< \texttt{nok?}, \texttt{refuse!}, \varepsilon >$$
$$< \texttt{ok?}, \varepsilon, \texttt{accept!} >$$
$$< \texttt{ok?}, \texttt{accept!}, \varepsilon >$$

As regards these observable interfaces and vectors, nothing prevents to send once more a request to both suppliers. This might be avoided if the store entity counts the number of `nok` messages received, and when completed, gives up firing the `tau` transition. Furthermore, all the messages are not captured with reference to the previous description, *e.g.*, when the first supplier to answer sends a positive reply, the message of the second supplier is not caught. This may be fixed extending the store entity to receive and manage messages `ok` and `nok` in its final state. Such a piece of work corresponds to adaptation [44] or compatibility issues [15], and is out of the scope of this report. To sum up, note that these vectors do not imply any order in the interaction firing, only legal synchronizations. This might be source of errors because the global evolution is therefore not made explicit.

3.3 Interaction Diagrams

Coordination may be described using interaction diagrams, this term taking into account dialects such as MSC or (UML) sequence diagrams (our focus in the sequel), and collaboration diagrams. MSC and sequence diagrams give a temporal view of interactions whereas collaboration diagrams deal with a spatial view.

Syntax and semantics. Many formalisations of interaction diagrams have been proposed so far, and may be used, for example [29, 34, 26, 31]. One MSC is made up of axes, each axis denoting the lifeline of one entity. Arrows express communications: one arrow links several entities, a sender and one or several receivers, and holds the name of the message. Arrows appearing in one MSC (from top to bottom) describe one scenario of possible communication sequences which may occur among entities. Proposals aimed at formalising MSCs in different way, *e.g.*, algebraically [29], operationally [34] or in a denotational way [31]. It would take too much space to introduce even partially semantic rules here, accordingly the reader may refer to the quoted papers.

Application. We show in Figure 4 the possible interactions among a store and two suppliers, using the broadcast MSC notation given in [31]. In Figure 4, `PAR` stands for a parallel composition (interleaving) and `ALT` for an alternative operator, *i.e.*, a choice. As MSCs do not enable components to synchronize on

events with different names, transitions of the store (`buy`, `nok` and `ok`) should be respectively renamed `request`, `refuse` and `accept`.

In this coordination example, we express the same interaction schema than in section 3.2 enhanced with the temporal application order of synchronizations which was not expressible using synchronization vectors. It may happen that one MSC is not sufficient to express all the interaction scenarios. In such a case one would have to define as many MSCs as possible interaction scenarios.
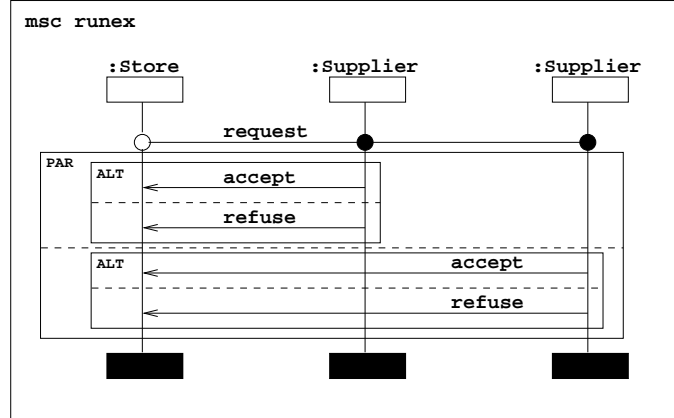

Figure 4: Coordination using MSC

Interaction diagrams provide a nice graphical representation of coordination. Moreover, they enable one to express a temporal ordering between basic communications. However, a shortcoming of interaction diagrams it that they do not enable name matching, and thus they harm the reusability level of components.

3.4   Temporal Logic

Modal logics are a very expressive means to express properties of systems. These logics deal with evolving truth values (using temporal modal operators) for properties on states of the dynamic models. Among them, different kinds of temporal logics exist. There is a first separation between linear (LTL) and branching time (CTL, CTL*) logics. Temporal logics have been extended with actions to take the labels of transitions into account (ACTL, TLA, HML). Here we aim at demonstrating that such logics are expressive means to coordinate entities. The idea is to be able first to denote sets of objects that are to be glued (in our model these objects are states and transitions). This denotation is achieved using logic formulas, with a set-theoretic semantics: given a set S and a formula f, the semantics of f over S is the subset of S elements which satisfy f. Then, the logic must also take into account coordination (using logical conjunction) and be able to lift the properties of the subcomponents of a composition up to the composition. This is achieved using indexed formulas.

Syntax. We first define means to express properties of transitions. This is achieved using *transition formulas (Λ)* which are defined as:

$$\lambda ::= \textbf{true} \mid \text{label} \mid \neg\lambda \mid \lambda_1 \wedge \lambda_2$$

where a label is an event pattern, *i.e.*, an event name and a direction (*e.g.*, e? or e!). Such patterns are used

to match a subset of the transitions. Here with our simple model, patterns do syntactically correspond[2] to transition labels (*e.g.*, pattern e? matches all the e? transitions).

Properties of states are expressed using *state formulas* (Ψ):

$$\psi ::= P \mid \mathbf{true} \mid [\lambda]\psi \mid \neg\psi \mid \psi_1 \wedge \psi_2$$

where P is a state property (any first order formula[2]) and λ a transition formula. Temporal modalities are inspired from HML. $[\lambda]\psi$ means that any outsourcing transition that satisfies λ will lead to a state that satisfies ψ. $\langle\lambda\rangle\psi$, which may be defined as $\neg[\lambda]\neg\psi$, means that there is one outsourcing transition that satisfies λ which leads to a state that satisfies ψ.

Transition and state formulas express properties of basic entities. They are then lifted to compositions (and hence glue) using *composition-oriented transition formulas ($\overline{\Lambda}$)* and *composition-oriented state formulas ($\overline{\Psi}$)*:

$$\overline{\lambda} ::= \mathbf{true} \mid x.\lambda \mid \neg\overline{\lambda} \mid \overline{\lambda}_1 \wedge \overline{\lambda}_2$$
$$\overline{\psi} ::= \mathbf{true} \mid x.\psi \mid \neg\overline{\psi} \mid \overline{\psi}_1 \wedge \overline{\psi}_2$$

with x being the identifier of one of the composition components (taken in a set Id). In any type of formula, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ can be defined as usual. Some shortcuts can also be defined: for $\phi \in \Lambda \cup \Psi$ and $C \subseteq Id$ (a subset of the composition identifiers), $\mathbf{ALL}(C).\phi \overset{def}{=} \bigwedge_{x \in C} x.\phi$, and $\mathbf{ONE}(C).\phi \overset{def}{=} \bigvee_{x \in C}(x.\phi \bigwedge_{y \in C, x \neq y} \neg y.\phi)$.

Semantics. The semantics of transition and state formulas are defined given a model $M =< L, S, I, F, T >$ and denote respectively sets of transitions and sets of states. The semantics of transition formulas is defined as:

$$\|\mathbf{true}\|_M = T$$
$$\|\mathrm{label}\|_M = \{t \in T \mid t = (s, \mathrm{label}, s')\}$$
$$\|\neg\lambda\|_M = \|\mathbf{true}\|_M \backslash \|\lambda\|_M$$
$$\|\lambda_1 \wedge \lambda_2\|_M = \|\lambda_1\|_M \cap \|\lambda_2\|_M$$

true denotes all transitions, whereas a *label* denotes only the transitions that have this label. Then, other formulas denotations are obtained from these two base definitions using a set-theoretical approach (difference for $\neg$ and intersection for $\wedge$). As usual, $M \models_t \lambda$ iff $t \in \|\lambda\|_M$ and $M \models \lambda$ iff $M \models_t \lambda$ for every t in T.

The semantics of state formulas is defined in the same way as:

$$\|P\|_M = \{s \in S \mid P(s)\}$$
$$\|\mathbf{true}\|_M = S$$
$$\|[\lambda]\psi\|_M = \{s \in S \mid \forall t = (s, l, s') \in \|\lambda\|_M . s' \in \|\psi\|_M\}$$
$$\|\neg\psi\|_M = \|\mathbf{true}\|_M \backslash \|\psi\|_M$$
$$\|\psi_1 \wedge \psi_2\|_M = \|\psi_1\|_M \cap \|\psi_2\|_M$$

A state property denotes all states for which it is true. A formula $[\lambda]\psi$ denotes (source) states such that all (target) states related by transitions of the λ denotation are in the ψ denotation. Other denotations are defined in the same way than for transition formulas. As usual, $M \models_s \psi$ iff $s \in \|\psi\|_M$ and $M \models \psi$ iff $M \models_s \lambda$ for every s in $S(M)$.

---

[2]These are important extension points of our logics. Here data may also be taken into account, together with the usual $\exists x$ and $\forall x$ quantifiers on data.

Coordination using temporal logic is given using a $\overline{\lambda}$ composition-oriented transition formula (possibly **true**) and three $\overline{\psi}$, $\overline{\psi_{\mathrm{I}}}$, and $\overline{\psi_{\mathrm{F}}}$ composition-oriented state formula (all possibly **true**). Given n identified (over a set Id) component models $M_{j \in \mathrm{Id}} = < L_j, S_j, I_j, F_j, T_j >$, a *global model* $\Pi_{j \in \mathrm{Id}}(\overline{\lambda}, \overline{\psi}, \overline{\psi_{\mathrm{I}}}, \overline{\psi_{\mathrm{F}}}) M_j$ is a transition system

$$\overline{M} = < \overline{L}, \overline{S}, \overline{I}, \overline{F}, \overline{T} >$$

such that:

$$\overline{S} \subseteq \{\overline{s} \in \Pi_{j \in \mathrm{Id}} \; j.S_j \mid M_j \models_{\overline{s}} \overline{\psi}\}$$
$$\overline{I} \subseteq \{\overline{i} \in \Pi_{j \in \mathrm{Id}} \; j.I_j \mid M_j \models_{\overline{i}} \overline{\psi_{\mathrm{I}}}\}$$
$$\overline{F} \subseteq \{\overline{f} \in \Pi_{j \in \mathrm{Id}} \; j.F_j \mid M_j \models_{\overline{f}} \overline{\psi_{\mathrm{F}}}\}$$
$$\overline{T} \subseteq \{\overline{t} \in \Pi_{j \in \mathrm{Id}} \; j.T_j \mid M_j \models_{\overline{t}} \overline{\lambda}\}$$
$\overline{L}$ corresponds to the labels from $\overline{T}$.

A global composition model is made up of indexed state and transition product built from the composition subcomponent states and transitions. Indexing is used to be able to denote, at the composition level, a property of a given (named) subcomponent. An interesting property of global composition models is that they have the same transition system structure than component models, hence composites can be used as components in larger compositions.

The semantics of composition-oriented transition formulas is defined on global models in the following way:

$$\overline{M} \models_{\overline{t}} \mathbf{true} \text{ for all } \overline{t} \in \overline{T}$$
$$\overline{M} \models_{\overline{t}} x.\lambda \text{ iff } x \in Id \text{ and } M_x \models_{t_x} \lambda$$
$$\overline{M} \models_{\overline{t}} \neg\overline{\lambda} \text{ iff } \overline{M} \not\models_{\overline{t}} \overline{\lambda}$$
$$\overline{M} \models_{\overline{t}} \overline{\lambda}_1 \wedge \overline{\lambda}_2 \text{ iff } \overline{M} \models_{\overline{t}} \overline{\lambda}_1 \text{ and } \overline{M} \models_{\overline{t}} \overline{\lambda}_2$$

The interesting part here are the indexed formulas $(x.\lambda)$ which are true for a given t transition of the global model only if there is a subcomponent identified by x in the global model and if λ is true for this subcomponent part of the t transition. As usual, $\overline{M} \models \overline{\lambda}$ iff $\overline{M} \models_{\overline{t}} \overline{\lambda}$ for every $\overline{t}$ in $\overline{T}$. Composition-oriented state formulas are handled in the same way.

Application. In our example, to express the same communications than in subsection 3.2 but generalized to N `Supplier` components, the glue would be defined using the following composition-oriented transition property:

$$\mathtt{Store.buy!} \Leftrightarrow \mathbf{ALL}(\{i : [1..N]\mathtt{Supplier}_i\}).\mathtt{request?}$$
$$\vee \;\; \mathtt{Store.ok?} \Leftrightarrow \mathbf{ONE}(\{i : [1..N]\mathtt{Supplier}_i\}).\mathtt{accept!}$$
$$\vee \;\; \mathtt{Store.nok?} \Leftrightarrow \mathbf{ONE}(\{i : [1..N]\mathtt{Supplier}_i\}).\mathtt{refuse!}$$

Temporal logic is an expressive way to specify interaction: matching actions with different names and expressing communication patterns on N entities, such as broadcasting the requests to suppliers in our example, is possible. Other complex coordination patterns (value-passing, exclusive 1-to-N communication, exclusive states, dynamic coordination reconfiguration) can be defined using this kind of logic [1]. It also proves to be more synthetic and abstract than synchronized vectors for example (no need to make explicit the fact that some entities do nothing while other synchronize, N entities coordinated easily).

3.5 Semantic Glue

The idea is to *implicitly* describe the way entities interact together through semantic rules. This is an helpful approach when interactions are not given explicitly using a specific language but though have

to be formalised [9]. We show here how such inference rules may be written to express a two-way asynchronous communication between entities.

S corresponds to a system made up of couples (E_i, Q_i) where E_i are entities and Q_i are associated queues. Queues are sequences of events. QQ' denotes the overloading of the queue Q by Q', whereas $Q - Q'$ denotes the difference between Q and Q'.

The SEND rule describes how queues are modified when an emission takes place. In the conclusion of the rule, the sent event, $l!$, is appended to the queue of the E_j receiver, written $Q_j\{l!\}$ (explicit receivers are being used in this example), and E_i' substitutes E_i. Note that a simple arrow ($\xrightarrow{E_j.l!}$) stands for an entity evolution while a double one ($\xRightarrow{l}$) stands for a global evolution of all entities (as for synchronized vectors).

$$\frac{\begin{array}{c} S = \{(E_1, Q_1), ..., (E_n, Q_n)\} \\ i, j \in \{1, .., n\} \\ E_i \xrightarrow{E_j.l!} E_i' \end{array}}{S \xRightarrow{l} S[(E_i', Q_i)/(E_i, Q_i), (E_j, Q_j\{l!\})/(E_j, Q_j)]} \quad \text{SEND}$$

The second rule, CONSUME, formalises how one entity consumes an event belonging to its input queue. Thus, if the event $l!$ belongs to the Q_i queue and the entity E_i may evolve in E_i' receiving a value along the label l, then the interaction occurs, the entity E_i evolves and the event is removed from the Q_i queue.

$$\frac{\begin{array}{c} S = \{(E_1, Q_1), ..., (E_n, Q_n)\} \\ i \in \{1, .., n\} \\ l! \in Q_i \\ E_i \xrightarrow{l?} E_i' \end{array}}{S \xRightarrow{l} S[(E_i', Q_i - \{l!\})/(E_i, Q_i)]} \quad \text{CONSUME}$$

Such semantic rules are sufficient to make one store and several suppliers cooperate together: no additional pieces of notation are necessary. The only notation we introduced in the rules above is explicit receivers, even though this is not mandatory at all (but it made the formalisation easier).

4 Comparison

Table 5 summarizes facts about the explicit coordination formalisms we have presented in this report. Indeed, they are, in our opinion, the main candidates for a use in the Component Based Software Engineering community, the building of semantic glues being more restricted to the Formal Methods community. Comparison is done using several criteria. The first ones address expressiveness: kind of communication (binary, n-ary all entities having to synchronize, n-ary with possible synchronization of a subset of the entities), possibility to make interface labels with different names match, possibility to deal with data in the coordination (value passing, agreement and negotiation), and possibility to express an order between synchronizations in the coordination. Then criteria related to automated techniques and tools, and user-friendliness are presented: reasoning techniques supported by tools for the coordinator specifications, executability of the coordinator specifications (is the simulation or code generation of the coordinators

		Process Algebras	Vectors	Interaction Diagrams	Logics
Communication Expressiveness	1 to 1	yes	yes	yes	yes
	1 to N	yes	yes	yes	yes
	1 to M in N	extension (E–LOTOS)	yes	yes	yes
	Name matching	no	yes	no	yes
	Data	yes	extension	no	yes
	Order	yes	no	yes	no
User Friendliness	Tools	animation equivalence checking model–checking	animation equivalence checking model–checking	animation model–checking	embeddings
	Executability	yes	no	yes	no
	Graphical notations	extension (GCCS)	no	yes	no

Figure 5: Comparison of the proposals

possible?), existence of user-friendly graphical notations. *Extension* denotes that only specific semantics (or dialects) of a coordination means support the criteria.

Some comments can be made on this table. Process algebra provides a good trade-off between expressiveness and user-friendliness. In addition, existing tools such as CADP [23] or SPIN [27] allow the use of efficient formal verification means (from interactive animation to advanced model-checking techniques). Graphical notations [33, 19] and development processes [33, 30, 40] have been proposed for process algebra. Vectors provide a simple and yet expressive language for simple coordination protocols. However, they do not take more advanced coordination patterns (choice, order) into account. Toolboxes such as MEC 5 [25] enable one to specify processes interactions using synchronized products, and then to reason on such descriptions. Interaction diagrams are quite expressive (no name matching however) and propose a rich graphical notation. The LTSA-MSC tool [42] makes it possible to synthetize automata from MSC. It does not enable one to reason directly on the MSC specifications. The real interaction diagrams flaw is the lack of a widely admitted formal semantics which would open the way to the development of formal and powerful reasoning tools. Logics are pretty expressive and allow the writing of any communication models and interaction means. On the other hand, they are not easily readable and not tool equipped (even if embeddings into higher-order logic tools such as PVS [21] or Isabelle [37] may be undertaken).

Evaluation has been done for each coordination technique taking only its basic constructs into account. As far as process algebras are concerned, this yields a first class coordinator: the process describing the interaction can be itself considered as an entity (taking the operational semantics of the process into account to obtain a LTS). First class coordinators provide the specifier with more expressiveness as they enable, for example, the definition of a specific order between interactions. As far as the other coordination means (vectors, interaction diagrams, temporal logic) are concerned, the definition of first class coordinators would also be possible. However, this would require, separately from the interaction definition, the definition of at least one new entity. For example, dealing with temporal logic, one could first define a new entity corresponding to the process algebraic coordinator and then define the temporal

glue over the supplier, the two stores and this new coordinator entity.

5   Concluding Remarks

In this report, our goal was to overview abstract and formal means that can be used to describe coordination of communicating entities. All the coordination means introduced here may be used and applied to many contexts. For instance, we defined in previous proposals expressive mechanisms [17, 9, 41] to compose heterogeneous entities mixing behaviours and data descriptions. Thereby, such a survey is useful as a starting point to assess possible material to be used afterwards in a particular application domain.

These formal means have different advantages, and judicious choices have to be done (when using them) depending on the context, application domain, or goals of the approach at hand. Therefore, even if we cannot claim that one approach is the best in any case, we can emphasize the convenient trade-off that process algebra proposes as a coordination means (expressiveness, user-friendliness, tool support). Transversely, we stress that temporal logic is an adequate formalism to describe expressive interaction models, even if some work has still to be done to make them more usable and equipped with reasoning tools.

References

[1] M. Aiguier, F. Barbier, and P. Poizat. A Logic with Temporal Glue for Mixed Specifications. In *Proc. of FOCLASA'2003*, volume 97 of *ENTCS*, pages 155–174, 2004.

[2] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Specification: Business Process Execution Language for Web Services Version 1.1. 2003.

[4] F. Arbab. What Do You Mean, Coordination? In the *March '98 Issue of the Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, 1998.

[5] F. Arbab, F. S. de Boer, M. M. Bonsangue, and J. V. Guillen Scholten. A Channel-based Coordination Model for Components. In *Proc. of FOCLASA'02*, volume 68(3) of *ENTCS*, 2002.

[6] F. Arbab and J. J. M. M. Rutten. A Coinductive Calculus of Component Connectors. In *Proc. of WADT'02, Revised Selected Papers*, volume 2755 of *LNCS*, pages 34–55. Springer-Verlag, 2003.

[7] A. Arnold. *Finite Transition Systems*. Prentice-Hall, 1994.

[8] A. Arnold, G. Point, A. Griffault, and A. Rauzy. The AltaRica Formalism for Describing Concurrent Systems. *Fundamenta Informaticae*, 40:109–124, 2000.

[9] C. Attiogbé, P. Poizat, and G. Salaün. Integration of Formal Datatypes within State Diagrams. In *Proc. of FASE'03*, volume 2621 of *LNCS*, pages 341–355. Springer-Verlag, 2003.

[10] J. A. Bergstra, A. Ponse, and S. A. Smolka, eds.. *Handbook of Process Algebra*. Elsevier, 2001.

[11] G. Bernot, J.-P. Comet, A. Richard, and J. Guespin. Application of Formal Methods to Biological Regulatory Networks: Extending Thomas' Asynchronous Logical Approach with Temporal Logic. *Journal of Theoretical Biology*, 229(3):339–347, 2004.

[12] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, 1999.

[13] D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web Service Interfaces. In *Proc. of WWW'05*. ACM Press, 2005.

[14] M. M. Bonsangue, F. Arbab, J. W. de Bakker, J. J. M. M. Rutten, A. Secutella, and G. Zavattaro. A Transition System Semantics for the Control-driven Coordination Language MANIFOLD. *Theoretical Computer Science*, 240(1):3–47, 2000.

[15] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are Two Web Services Compatible? In *Proc. of TES'04*, volume 3324 of *LNCS*, pages 15–28. Springer-Verlag, 2004.

[16] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

[17] C. Choppy, P. Poizat, and J.-C. Royer. A Global Semantics for Views. In *Proc. of AMAST'00*, volume 1816 of *LNCS*, pages 165–180. Springer-Verlag, 2000.

[18] D. Clarke, D. Costa, and F. Arbab. Modelling Coordination in Biological Systems. In *Proc. of ISoLA'04*, 2004.

[19] R. Cleaveland, X. Du, and S. A. Smolka. GCCS: A Graphical Coordination Language for System Specification. In *Proc. of COORDINATION'00*, volume 1906 of *LNCS*, pages 284–298. Springer-Verlag, 2000.

[20] R. Cleaveland, T. Li, and S. Sims. *The Concurrency Workbench of the New Century (Version 1.2)*. Department of Computer Science, North Carolina State University, 2000.

[21] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. In *Proc. of WIFT'95*. Computer Science Laboratory, SRI International, 1995.

[22] L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proc. of ESEC/FSE'01*, pages 109–120. ACM Press, 2001.

[23] H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, 2001.

[24] D. Garlan and M. Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, eds., *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.

[25] A. Griffault and A. Vincent. The MEC 5 Model-Checker. In *Proc. of CAV'04*, volume 3114 of *LNCS*, pages 488–491. Springer-Verlag, 2004.

[26] R. Heckel and S. Sauer. Strengthening UML Collaboration Diagrams by State Transformations. In *Proc. of FASE'01*, volume 2029 of *LNCS*, pages 109–123. Springer-Verlag, 2001.

[27] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[28] M. N. Huhns and M. P. Singh. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[29] ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart 1996 (MSC'96) – Annex B: Algebraic Semantics of Message Sequence Charts. Technical report, ITU-TS, Geneva, 1996.

[30] J. Kramer, J. Magee, and S. Uchitel. Software Architecture Modeling and Analysis: A Rigorous Approach. In *Proc. of SFM'03*, volume 2804 of *LNCS*, pages 44–51. Springer-Verlag, 2003.

[31] I. Krüger, W. Prenninger, and R. Sandner. Broadcast MSCs. *Formal Aspects of Computing*, 16(3):194–209, 2004.

[32] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–54, 1995.

[33] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. Wiley, 1999.

[34] S. Mauw and M. A. Reniers. Operational Semantics for MSC'96. *Computer Networks and ISDN Systems*, 31(17):1785–1799, 1999.

[35] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[36] S. Moschoyiannis, M. W. Shields, and P. J. Krause. Modelling Component Behaviour with Concurrent Automata. In *Proc. of FESCA'05*, 2005. To appear in *ENTCS*.

[37] T. Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[38] P. Poizat, J.-C. Royer, and G. Salaün. Formal Methods for Component Description, Coordination and Adaptation. In *Proc. of WCAT'04 – ECOOP'04*, 2004.

[39] G. Salaün, M. Allemand, and C. Attiogbé. Foundations for a Combination of Heterogeneous Specification Components. In *Proc. of FMCI'02*, volume 66(4) of *ENTCS*, 2002.

[40] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *Proc. of ICWS'04*, pages 43–51. IEEE CSP, 2004. Extended version to appear in the *IJBPIM* journal.

[41] G. Salaün and P. Poizat. Interacting Extended State Diagrams. In *Proc. of SFEDL'04*, volume 115 of *ENTCS*, pages 49–57, 2004.

[42] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios. In *Proc. of TACAS'03*, volume 2619 of *LNCS*, pages 597–601. Springer-Verlag, 2003.

[43] W. W. Vasconcelos. Logic-Based Electronic Institutions. In *Proc. of DALT'03*, volume 2990 of *LNCS*, pages 221–242. Springer-Verlag, 2004.

[44] D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

Formal Coordination of Communicating Entities described with Behavioural Interfaces

Pascal Poizat, Gwen Salaün

Abstract

The *interaction* paradigm is the foundation of many practical and theoretical computational issues. Roughly speaking, interaction involves communicating entities which evolve in parallel and may synchronize together. Communicating entities are usually viewed through public interfaces due to their black-box foundation. It is now widely accepted that such interfaces have to take into account behavioural information. In this report, we survey existing formal and abstract means to describe coordination of entities with behavioural interfaces, and we illustrate their use on simple e-business examples. We end with a comparison of such coordination means.