Coordination in Distributed Systems

Juan M. Cordero Mariano Gonzalez
Jesus Torres

Dep. de Lenguajes y Sistemas Informaticos
University of Seville

Keywords: Agent, Component, Coordination, Event, Java, Aspects Programming, Petri net, Dynamic System, Distributed System.

Abstract

There is a growing demand for programs which can inter-operate in distributed environments exchanging information and services. Most of the difficulty in developing these programs is found in specifying the coordination of those processes involved in reaching a common compute aim (to carry out a task). The coordination in environments where the processes have the same running context has already been studied. There are different architectural models and specification techniques to describe the concurrent access to shared resources and to synchronize processes. These techniques have been greatly used in the domain of operating systems. In the case of distributed systems there are certain proposals based on a client/server architecture. However, the use of dynamic distributed systems which can join, eliminate or substitute processes, force us to define new models. Moreover, these systems are usually characterised by the lack of a common space for names which leads to a model whose entities are greatly uncoupled. It is also necessary to find a solution based on components. In this paper we will propose a model and a language to specify coordination in distributed systems; as well as making the system design process as compatible as possible to the actual engineering methods of the software.

1 Introduction

A distributed system is a collection of processes running in parallel on different processors. These processors do not share common memory spaces, and they are only connected by a communication net.

An important aspect when specifying distributed systems is *coordinating* [25] the actions carried out by those entities participating in the system to reach a common compute aim. There are three reasons for which the actions must be coordinated [26]:

- When there is inter-dependence between actions carried out by multiple entities.
- When there are global restrictions: conditions imposed on the entities concerning the way they find the solution.
- The entities do not have enough information about the problem by themselves.

Malone and Crowston [27] identified the following types of inter-dependence:

- Shared resources: the same resource is necessary to carry out different activities.
- Prerequisite: an entity's activity must be carried out before another's.
- Transfer: an activity produced something, which is necessary for another.
- Usability: what is produced by an activity must be usable by another.
- Simultaneity: some activities must or must not happen, at the same time.
- Sub-tasks: a group of activities called sub-tasks do a part of a task.
- Group decisions: all decisions are taken collectively.

It is common to try to resolve only problems of synchronization restrictions, however we must not forget other problems related to prerequisite and simultaneity, which are connected to dependencies between activities.

Coordination gets complicated if the system is dynamic: we do not know the number of entities participating, or where they are (reference in the system).

The client/server model is the most used model in the distributed systems architecture. CORBA and JRMI [11], the most popular architecture used today to develop distributed systems, are based on this model. This model presumes that all the activities involved in a compute must recognize the other participants, or at least, those with which it has a client/server relationship.

We must take the following requirements into account in the system model:

- *The lack of global time.* Every processor has its own local clock; the time on the different processors in the system may vary.
- *Delays in messages between processes.* The messages are sent to the net and there are delays depending on how busy the net is.
- *Lack of global state.* The system global state is the union of process states, which participate in the system.
- *Autonomy.* A client/server relationship between those entities forming up the system does not necessarily have to exist.
- *Evolution.* Entities can join up, leave the system or be replaced dynamically.

- *Concurrency.* In theory, different entities participating in a compute in a distributed system carries out parallel activities, therefore they can be considered as concurrent activities. However, we have to take into account concurrency between those entities in the same processor, as well as concurrency between entities distributed by the system.

Taking these considerations into account we have developed a model and a language to specify coordination in dynamic distributed systems.

When we have finished specifying the system, and using additional information about those entities participating, it is possible to obtain a prototype.

Due to the fact that we can describe coordination characteristics independently from computed characteristics, coordination can be regarded as an orthogonal concept [23].

Thus, we can separate a concrete aspect from the code design: the coordination aspect. This idea of treating a concrete aspect of the problem when creating code can be found in [31].

We use two different languages to describe the separation between compute and coordination.

Coordination can be considered as having two parts: *configurations* and *interactions*. The former is structural relationships between entities. The latter are links between entities to coordinate the system.

The document is organized as follows. We describe the model used to design the specification language in section 2. In section 3 we define *Interaction Nets* as a way to define interactions, these will be used in section 4 to build *Agent Nets* in order to define configurations. The results obtained in prototyping the specifications can be seen in section 5. We present a brief summary of related works in section 6. Finally we show our conclusions and some advances for future work.

2 Proposed Model

We propose an architecture of distributed systems which has no dependence, no previous knowledge, of those entities involved in the coordinated compute. This allows us to design dynamic systems where entities can join or leave the system without stopping or reconfiguring it.

We base the co-operation between entities on the interchange of information between processes by means of passing messages, in a synchronized or an asynchronized way. These messages represent the states the entities co-operating in the system are in and establish a protocol of information interchange. They also represent service requests between entities and the answers of the same. These messages are called *events* if we suppose that there is a compute model based on actions.

The model is based on *causal* relationships expressed in order of precedence between events. A system specification is therefore an *event algebra* [19].

Let us group the events together into high level abstractions, hiding their internal causal structure. In this way the designer can have simple and abstract

views of the system's behaviour, or of part of it, concentrating on the information which is considered relevant. The abstraction process may be repeated until the desired level of abstraction is reached.

In order to generate code after specifying the system using a programming based on aspects; we must take into account three parts when describing a system [31]):

- *Coordination.* The system is made up of objects, which must be coordinated to reach the compute aim. This part is resolved using the concept of *Agent Net* (see section 4).
- *Communication.* Objects communicate with each other through events described using an *Interaction Net* (see section 3).
- *Compute.* This part is express according to the technology of objects used. We use CORBA-IDL [11] in the examples in this paper.

It is necessary to use a formal technique to reason about the system.

Some of the most commonly used formalisms in distributed systems, due to their characteristics are: *Process Algebra* [12], *Modal Logic* [13] and *Petri net* [14].

We are interested in *causality* relationships between events therefore we will rule out the use of Modal Logic. We are also interested in expressing *true concurrency*, instead of *interleaving concurrency.* The interleaving concurrency models suppose that all events have a total order based on real time. However, this order is not very useful for analyzing causality, therefore only using Process Algebra is insufficient. Using Petri nets presents two drawbacks. The first being the size of the nets generated to model a system even when high level nets or hierarchical nets are used [8]. The second drawback is its static nature, which does not allow a dynamic system to be described.

This has led to a language, which allows us to have a specification based on algebraic terms of events [19], which describes implicitly a Petri net of the system model at the same time. In this way we can, on the one hand, check the specification using term rewriting techniques and equational reasoning by means of algebraic expressions, and on the other hand, validate and simulate the system through the Petri net model using tools such as *Design/CPN* [15].

We can thus define a partial labeled order, according to the underlying algebraic structure in the Winskel model [2], $(E, A, \leq, f)$, where E is the set of events partially ordered by the $\leq$ *order relation* which models the causal precedence described by Lamport [1], and f is the label function which relates the events with the actions of the A alphabet.

Finally we will have a language which allows us to build modular Petri nets as a collection of basic nets. The semantics associated with the net composition mechanism allows us to modify the system model, as well as to introduce new events, without changing their global behaviour.

3 Interaction Net

3.1 Interaction Pattern

An *Interaction Pattern* [30] can be defined as a set of events and causal relationships amongst said events. An instance of an interaction pattern in a distributed compute is a set of events whose labeled and causal relationships perfectly match the pattern. The same pattern can occur several times in the same compute. Interaction patterns can be used to build a hierarchy of interaction abstractions. In this process the interaction patterns can be interpreted as high level events.

Due to the fact that the precedence relationships in a distributed compute is a partial order, it would seem reasonable to demand that the set of events in an interaction pattern should have a partial order. That is to say, an interaction pattern is a partially ordered set of events.

3.2 Definition

An *Interaction Net* (INet) is a Petri net which is extended by introducing *incoming events* into the transitions. We will call these events *INet events*. INet events are used as a composition mechanism amongst INets. Composition between two INets is carried out by giving one of them an INet event as its incoming event, and the other generates that event. An INet event modeled by an INet is an interaction pattern, a compound or abstract event.

When an INet reaches its *fire condition*, that is, when an instance of a certain interaction pattern appears, a new INet event is produced. The new INet event causes a change in state in other INets, as well as allowing possible changes in state in certain objects which react to the stimulus the event causes.

Although the internal structure of an INet is a Petri net, it cannot be described explicitly by defining those elements which form a Petri net: places, transitions and arcs [14]. That is, we try to make the description less operational, allowing us to concentrate on the design in the causality relationships among INet events and in those actions associated with transitions.

In order to achieve this goal the specification language translates the INet description into a high-level Petri net, connecting an input place to the INet events, an output place to the fire condition of the modeled INet event, and a place for every transition. Place tokens are value structures. These values can be references to objects or primitive data of the object language used. Net marking and its dynamics depend upon how the transitions receive the INet events.

We will see the notation used to specify an INet. To be as brief as possible we have chosen a very simple example, to show our ideas:

event LeftButtonClick {
  var time : *Clock*;
  trans Pressed-Left (c : *short*)

```
      when [ PressedLeftButton ]
      being [ c := time.getTime(); ];
    trans Released-Left ( c1 : short )
      when [ Pressed-Left ( c2 ), ReleasedLeftButton ]
      being [ c1 := time.getTime() - c2; ];
    fire when [ Released-Left( c ) ]
      if [ c <= 500 ];
}
```

For this example, the Java code generated by means of the specification can be found in [16].

4 Agent Net

4.1 Agent

An agent is an abstraction of an autonomous and co-operative entity. To develop a system using agents is to model it as a set of active autonomous objects. These objects are pro-active, that is, they can decide which actions they must carry out whilst acting independently and concurrently, co-operating through messages.

Agent based software engineering is mentioned in [24] as a facility to create software which can inter-operate in heterogeneous environments (with different operative systems, with programs written in different programming languages and using different machines). In this approach to software development, applications are written in as software agents, that is, software components which communicate with other components by interchanging messages.

One of the main differences between objects and agents is that, although both offer an interface based on messages which are independent to their internal data structures, in objects their meaning of the messages can vary from one object to another. On the other hand agents use a common language with semantics which are independent from the agent.

Another difference between objects and agents is that agents carry out objectives autonomously and independently whilst objects are naturally passive (they follow a client/server model).

This concept of an agent is close to the concept of an active object, but it is different.

4.2 Definition of Agent Net

An Agent Net is a group of objects and INets. Its state is determined by the states of the objects which it is composed of and by the states of the INets. The internal structure of an Agent Net is a Petri net, as happens with an INet. This Petri net is composed of INets forming the Agent Net.

There are two composition mechanisms between INets which create an Agent Net:

- *Aggregation.* Let us use two INets, neither of them has an input INet event which generates the other. We will get two Petri nets working parallely. This composition mode does not mean that the groups of input INet events are disjoint.

- *Interaction.* Let us take two INets, at least one of them has the INet event generated by the other as the input INet event. We will get a single Petri net. The interaction is modeled in said Petri net as a common place for the two sub-nets corresponding to each INet.

An Agent Net defines an output interface of INet events which are visible from the outside, and the input interface where the types of INet events which can be heard are indicated. It is possible to rename the input INet events expected by those INets forming the Agent Net.

All the INet events generated inside an Agent Net are visible within the Agent Net.

There are three types of objects forming an Agent Net:

- *Encapsulated objects.* These are handled and controlled by the Agent Net. Actions carried out on these objects depend on the state of the INets which the Agent Net is made up of.

- *Shared objects.* These are found in certain places in the system and controlled by a group of Agent Nets. These Agent Nets use a concurrent access protocol determined by the model of Petri net they configure. The access is controlled by giving a token as a parameter through the INet events. The algorithm determining which Agent Net has access to a shared object is based on [17], an algorithm which applies criteria of fair selection.

- *Control objects.* These arrive to the Agent Net through the INet event parameters, and they do not go through a fair selection algorithm as a copy of them is made for all INets involved.

4.3 Notation

Let us see an outline of the language used to specify Agent Nets.

agent agent name {
  require non-integrated agents;
  compose integrated agents;
  var encapsulated objects;
  input INet events;
  output INet events;
  init { agent initialization }
  event INet event name { ... }
  method name of method (parameters) { ... }
}

A more detailed specification for a distributed system of lift control [18], can be found in [16]. In this document we will only show a part to be brief:

```
typedef enum Displacement { upwards, downwards }; typedef enum
Rotation { lev-rotation, dex-rotation };
```

agent Floor-Detector {
  require floor : *short*;
  require inferior : Sensor **rename** [Excited **as** Excited-Inferior];
  require superior : Sensor **rename** [Excited **as** Excited-Superior];
  output At-Floor (*short*);
  event At-Floor (level) {
    fire when [Excited-Inferior, Excited-Superior] **being** [level := floor;];
  }
}

agent Cabin-Buttons {
  compose level-1 : Lift-Button **rename** [Pressed **as** Pressed-1];
  compose level-2 : Lift-Button **rename** [Pressed **as** Pressed-2];
  compose level-3 : Lift-Button **rename** [Pressed **as** Pressed-3];
  compose door : Lift-Button;
  var active-button : *boolean*[4];
  input Attended-Floor (*short*, *Displacement*);
  output Attend-Cabin (*short*);
  output Open-Door;
  init { disable-buttons(); };
  event Not-Light **toward** level-1 {
    fire when [Attended-Floor (level, direction)]
      if [level == 1] **do** [active-button[level] := *false*;];
  }
  event Light **toward** level-1 { **fire when** [Pressed-1]; }
  event Attend-Car (level) {
    fire when [Pressed-1]
      if [**not** active-button[1]] **do** [active-button[1] := true;] **being** [level := 1;];
  }
  event Open-Door { **fire when** [door::Pressed]; }
  method disable-buttons () {
    for(i=1 ; i¡=4 ; i++) active-button[i] := *false*;
  }
}

4.4 Approach based on Components

Using Agent Nets to describe a system allows us to see said system based on reactive components [5], given that:

- They are continuously operating, continuously interacting with the environment, without stopping as happens with other concepts of components

which are activated when they receive data and they finish when a result is produced.

- In response to a trigger, and depending on their present state, they modify their state.

4.5 Communication

Let us suppose that there is a communication type broadcast among the Agent Nets of the system. Differently to a more common mechanism of communication such as passing on a message or rendezvous, a broadcast communication allows us the following advantages:

- The same information is sent to several Agent Nets in only one operation.
- It allows a dynamic approach as new Agent Nets can be added dynamically without changing sources.

Although it makes us consider the following:

- An INet event can be present, absent or undefined (it is not persistent data).
- An INet event cannot be both present and absent in the same instant (coherency property).

Therefore broadcast communication will be considered in our paradigm as instant: an INet event is received by all receptors in the same instant it is generated. Moreover if we consider that an event can be generated and received in the same instant we will have a perfectly synchronous hypothesis.

5 Formal Definition

5.1 Basic Preliminaries

- An elementary *colour set* is a finite set of elements called *colors*, as for instance

  $$integer == \{..., -10, ..., 0, 1, ...\}$$
  $$\mathbb{N} == \{0, 1, 2, 3, ...\}$$

- A *color domain* can be an elementary color set or a cartesian product of countably many such elementary color sets, as for instance

  $$integer$$
  $$\mathbb{N} \times integer$$

- Γ is a set of elementary color sets, as for instance

$$\Gamma == \{integer,\ string,\ boolean\}$$

- $\Gamma^n == \underbrace{\Gamma \times ... \times \Gamma}_{n}$
- $\Gamma^* == \bigcup_{n \in \mathbb{N}} \Gamma^n$
- $\pi_i : \gamma_1 \times ... \times \gamma_n \rightarrow \gamma_i$, where $\gamma_1 \times ... \times \gamma_n \in \Gamma^n$ and π_i denotes the projection on the i^{th} dimension of $\gamma_1 \times ... \times \gamma_n$ color domain
- C_γ is the set of all the constants of the elementary color set γ
- V_γ is the set of variables over the elementary color set γ.
- $Symb_\gamma == C_\gamma \cup V_\gamma$
- $C_{(\gamma_1 \times ... \times \gamma_n)} == C_{\gamma_1} \times ... \times C_{\gamma_n}$ for a color domain $\gamma_1 \times ... \times \gamma_n \in \Gamma^n$
- $V_{(\gamma_1 \times ... \times \gamma_n)} == V_{\gamma_1} \times ... \times V_{\gamma_n}$ for a color domain $\gamma_1 \times ... \times \gamma_n \in \Gamma^n$
- $Symb_{(\gamma_1 \times ... \times \gamma_n)} == Symb_{\gamma_1} \times ... \times Symb_{\gamma_n}$ for a color domain $\gamma_1 \times ... \times \gamma_n \in \Gamma^n$
- The *type of a variable* v is denoted by $Type(v)$
- The *type of an expression* $expr$ is denoted by $Type(expr)$
- The *set of variables* in an expression $expr$ is denoted by $Var(expr)$. $Var(expr)$ however only includes free variables, i.e. those which are not bound.
- On elements of Γ we assume to be defined the **subtyping relations** $\preceq$ and $\prec$.

$$T_1 \preceq T_2 \Leftrightarrow T_1 = T_2 \vee T_1 \prec T_2$$
$$T_1 \prec T_2 \Leftrightarrow \textit{ every element of } T_1 \textit{ is a valid element of } T_2$$

- A *binding of a set of variables* $\mathcal{V} == \{v_1, ..., v_n\}$ is denoted by $b == \{v_1 \mapsto c_1, ..., v_n \mapsto c_n\}$, where $v_i \in V_{\gamma_i}$ and $c_i \in C_{\gamma_i}$. If we express the set $\mathcal{V}$ as a multi-demensional variable as $\mathcal{V} == (v_1, ..., v_n)$, a valid binding b for that variable is a n-tuple of constants $\mathcal{C} == (c_1, ..., c_n)$ such that $\pi_i(\mathcal{V}) \in C_{\gamma_i}$ then $\pi_i(\mathcal{V}) = \pi_i(\mathcal{C})$, that is, $\forall\, v : \mathcal{V} \bullet b(v) \in Type(v)$. It is demanded that $Type(c_i) \preceq Type(v_i)$ for each v in $\mathcal{V}$.
- The *value obtained by evaluating an expression* $expr$ *in a binding* b is denoted by $expr < b >$. It is demanded that $Var(expr)$ is a subset of the variables of b, and the evaluation is performed by substituting each variable $v_i \in Var(expr)$ with the value $c_i \in Type(v_i)$ determined by the binding b.

5.2 Coloured Petri Net

- As usually, ${}^\bullet x$ and $x^\bullet$ are the pre and post sets of a place or a transition in a Petri net. If S is a set, ${}^\bullet S$ and $S^\bullet$ are the union of pre and post sets of elements of S
- A **token distribution** is a function M defined on P. We define the relations $\neq$ and $\leq$ as:

  $$M_1 \neq M_2 \Leftrightarrow \exists\, p : P \bullet M_1(p) \neq M_2(p)$$
  $$M_1 \leq M_2 \Leftrightarrow \forall\, p : P \bullet M_1(p) \leq M_2(p)$$

  The relations $<$, $>$, $\geq$ and $=$ are defined analogously to $\leq$.
- A **binding distribution** is a function Y defined on T such that $\forall\, t : T \bullet Y(t) \in b(t)$
- A **marking** of a CPN is a token distribution.
- The **initial marking** is the marking obtained by evaluating the initialization expressions.
- A **step** is a non-empty binding distribution.
- A step Y is enabled in a marking M iff:

  $$\forall\, p : P \bullet \textstyle\sum_{(t,b)\in Y} E(p,t) < b > \leq M(p)$$

  When a step is enabled, it may occur. When a step occurs, tokens are removed from the input places and added to the output places of the occurring transitions, based on the transition expression evaluated for the occurrence bindings.
- A transition t of a CPN is *enabled* at marking M iff ${}^\bullet t \subseteq M$. The set of all enabled transitions at marking M is denoted by $Enabled(M)$. In CPNs, a marking is not sufficient information to describe a complete state of the system. The state must also include timing information. This is given as a clock function that, for each enabled transition, gives the amount of time that has passed since it has become enabled.
- A *state* of an CPN is a pair $S = (M, I)$, where M is a marking, and $I : Enabled(M) \rightarrow \mathcal{T}$ is called the *clock function*. The *initial state* of the CPN is $S_0 = (M_0, I_0)$, where $I_0(t) = \varnothing$ for all $t \in Enabled(M_0)$.

5.3 Interaction-Net

An Interaction-Net is 7-tuple $INet = (Net, P_{acc}, T_{res}, T_{acc}, Class, Time, Code)$ where:

- Net is a Coloured Petri Net, $Net = (P, T, Dom, Pre, Post, Guard, M_0)$ with :

- P is the finite set of places (ordinary places)
- T is the set of transitions
- $P \cap T = \varnothing$
- $Dom_P : P \cup T \rightarrow \Gamma^*$ defines the colour domains for places
- $Pre : P \times T \rightarrow \text{bag}\, Symb_{Dom_P}$ defines the backward incidence color function
- $Post : P \times T \rightarrow \text{bag}\, Symb_{Dom_P}$ defines the forward incidence color function
- *Guard* defines the guards on transitions

  $$Guard : T \rightarrow (\text{bag}\, Symb_{Dom_t} \rightarrow \mathbb{B})$$

  That is, the **guard** function *Guard* maps each transition t into a predicate, i.e. an expression yielding a boolean.
- M_0 is a marking for *Net* and $\forall\, p : P \bullet M_0(p) \,\text{in bag}\, C_{Dom_p}$ where $\forall\, p : P \bullet Var(M_0(p)) = \varnothing$
  That is, the **initialization** function M_0 maps each place p into an expression which is a bag of tokens. The expression is not allowed to contain any variables.

- P_{acc} is the set of *accept places* or *interface places* holding the tokens modeling requests accepted from the environment, where $P_{acc} \subset P$ is a set of places such that

  $$\forall\, p_{acc} : P_{acc} \bullet^{\bullet} p_{acc} = \varnothing \wedge M_0(p_{acc}) = [\![\,]\!]$$

- T_{res} is the set of *result transitions* emitting the tokens modeling results issued for requests accepted from the environment, where $T_{res} \subset T$ is a set of transitions such that $\forall\, t_{res} : T_{res} \bullet t_{res}^{\bullet} = \varnothing$

- $T_{acc} \subset T$ is a set of transitions such that
  - $\forall\, p_{acc} : P_{acc} \bullet \exists\, t_{acc} : T_{acc} \bullet t_{acc} \in p_{acc}^{\bullet}$
  - $\forall\, t_{acc} : T_{acc} \bullet \exists\, p_{acc} : P_{acc} \bullet p_{acc} \in^{\bullet} t_{acc}$
  - $\Upsilon_{acc-res} : P_{acc} \rightarrow T_{res}$ is a bijection such that $\forall\, p_{acc} : P_{acc} \bullet \exists\, t_1 ... t_n : T, 1 \leq i \leq n-1 \bullet t_n \in T_{res} \wedge t_i^{\bullet} \cap^{\bullet} t_{i+1} \neq \varnothing$
    The bijection $\Upsilon_{acc-res}$ ensures the correspondence between incoming requests and outgoing results, and give an optional semantics to *INet*. It ensures that every transition producing an outgoing result belongs to a potential sequence containing a transition that consumes an incoming request.

- *Class* is a set of object types

- *Time* is a **timeout** function

  $$Time : T \rightarrow \mathbb{N}$$

  That is, the timeout function maps each transition into a natural number, denoting the timeout value for timeout exception.

- *Code* is a **code** function,

  $$Code : T \rightarrow E$$

  That is, the code function associates an expression with each transition. The expression is a function representing the piece of code that is executed on transition occurrence.

5.4 Agent-Net

- **Composition of Agent-Nets**. Two Agent-Nets A_1 and A_2 can be combined if there is a mapping

  $$\varphi : T_{res}(A_1) \rightarrow P_{acc}(A_2)$$

  such that

  $$\forall\, t : T_{res}(A_1) \bullet Dom(t) = Dom(\varphi(t))$$

  Such constructions allow to build ad-hoc composite components and subsystems.

- We call $Clients(A_1) == \{A_i \mid \exists\, \varphi_{A_1 \rightarrow A_i}\}$, that is the set of components that can act as clients of A_1.

- We call $Servers(A_2) == \{A_i \mid \exists\, \varphi_{A_i \rightarrow A_2}\}$, that is the set of components that can act as servers of A_2.

6 Results: generation of prototypes

Nowadays the generation of prototypes is not an automatic process. By means of a specification and using translation schemes and design patterns [33], we can generate code Java to get a system prototype.

Initially we used JavaSpace [32] to coordinate the system, but it was substituted by the class packages Jada and SugarCubes, which are based on generative communication [22]. This is a closer concept to the model described here.

Mainly due to its capacity to pass objects according to their value we use the client/server model of JRMI to communicate.

We have chosen the JDK1.2 name service [32] for present implementations although a version using CORBA is being prepared.

The compute part is directly obtained from the specification. An Agent Net is translated as a Thread of Java. We thus improve the system's workload in comparison with the former translation scheme which created a Thread for each INet.

Each INet changes into an event monitor class. The INet events generated are translated into Java events. We use JavaBeans [32] type construction for this, which allows for introspection of those events required and generated by an Agent Net. Each INet stores a list of those INets interested in its INet event. Also a CORBA event service is being prepared.

In order to improve the systems's workload, object communication between Agent Net groups is done by means of a JavaBus [32].

7 Related works

To be brief we will only show some of the works proposing different coordination models. Some can be grouped together in the category of mathematical models (they describe coordination by means of abstract mathematical terms), others as operational models (they concentrate on the necessary operations for coordination) and the rest as structural models (they concentrate on relationships among entities taking part in coordination).

We can also differentiate between proposals based on centralized coordination and those based on distributed coordination.

- *Actors* [7] are autonomous parallel agents distributed in space with their own execution flow, they communicate using asynchronous messages. The message sending primitive is similar to an unlock call to a procedure.

- Bates [3] described an automata model called *shuffle automata* to recognize abstract events. This model cannot be used to recognize true concurrency, but only interleaving concurrency. This model imposes an event time order instead of a causal order.

- Hseush and Kaise [4] introduced another type of automata called *predecessor automata*. They use *Data-Path Expressions* which are basically regular concurrent expressions.

- *CO-OPN/2* [8] generalizes mechanisms of transition fusion and of hierarchical nets using the description of a modular approach for Petri nets, where high-level Petri nets are inside the objects, and using synchronisation mechanism to control co-operation among objects.

- *Cooperative Objects* [9] define control structures through high-level Petri nets in which the tokens are value tuples, and where objects interact, cooperate with each other through a client/server relationship.

- *GEM* [20] is an interpreted declarative language used to monitor events. It uses primitive events to form compound events.

- *Objective Linda* [21] was designed to find the requirements of a coordination model for open systems. It is based on the existence of *object spaces*, these are shared data structures to which you can have access with a minimum set of primitives. Communication is carried out inserting, reading or extracting elements from a shared data structure.

- *Darwin* [28] is a coordination language which proposes the construction of distributed programs out of hierarchical structured specifications of components. The components interact having access to the services offered by other components.

- The language *IOA* [29] is for distributed programming based on the I/O automata model. Together with the *IOA toolset* it supplies a variety of validation methods: theorem demonstrator, model checkers and simulators. These can be used to make sure that the generated programs are correct.

8 Conclusions

The main advantages of the proposed model is a clear separation from the coordination aspect of the system. The system is seen as a group of anonymous components which cooperate with each other to reach a common compute goal.

The components observe events produced in the system and they react, according to interaction patterns, producing changes in the coordinated objects.

The model allows us to establish contexts in the operation of the components, therefore one component can have different behaviours depending on the contexts it is in, but it always acts according to defined interaction patterns.

The prototype generation, by means of the specification of a system, makes this work a realistic approach to the development of software based on engineering techniques.

9 Future work

Some of the topics, we are working on are:

- Refinement of the model and extension of the language.

- By means of a Category Theory the definition of semantics of the proposed Petri net model, as well as defining implied inheritance relationship between INets compositions.

- Automation of the prototype generation process.

- Definition of a methodology to cope with the specification design.

- Checking and Simulation, generating an input for any tool of Petri net analysis.

References

[1] L. Lamport, *Time, Clocks and the Ordering of Events in a Distributed System*, Communications of the ACM, 21(7):558-565, July 1978.

[2] G. Winskel, *An Introduction to Event Structures*, Vol. 354 of Lecture Notes in Computer Science, pages 364-397, Springer Verlag, 1989.

[3] P.C. Bates, *Shuffle Automata: A Formal model for Behaviour Recognition in Distributed Systems*, Technical Report 87-27, University of Massachusetts, Computer and Information Science Department, Amherst, Massachusetts, USA, 1987.

[4] W. Hseush y G.E. Kaiser, *Modeling Concurrency in Parallel Debugging*, ACM SIGPLAN Notices, 25(3):11-20, March 1990.

[5] D. Harel y A. Pnueli, *On the Development of Reactive Systems*, NATO ASI Series F, Vol. 13, Springer Verlag, 1985.

[6] D. Gelernter, *Generative Communication in Linda*, ACM Transactions on Programming Languages and Systems, 7(1), 1985.

[7] S. Frolund, *Coordinating Distributed Objects*, The MIT Press, 1996.

[8] O. Biberstein y Didier Buchs, *CO-OPN/2: a specification language for distributed systems engineering*, University of Geneva, 1995.

[9] R. Bastide, *Cooperative Objects: A concurrent, Petri Net Based, Object-Oriented Language*, University of Toulousse, 1992.

[10] K. Jensen, *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical use*, Vol. 1, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1992.

[11] R. Orfali y D. Harkey, *Client/Server Programming with Java and CORBA*, Wiley, 1998.

[12] M. Hennessy, *Algebraic Theory of Processes*, MIT Press, 1988.

[13] S. Abramsky, D.M. Gabbay y T.S. Maibaum, *Handbook of Logic in Computer Science*, Oxford Science Publications, Volumen 2, 1992.

[14] M. Silva, *Las Redes de Petri: en la Automática y la Informática*, Edited by AC, 1985.

[15] K. Jensen, S. Christensen, P. Huber y M. Holla, *Design/CPN*, Meta Software Corporation, Cambridge, Massachusetts, USA, 1991.

[16] J.M. Cordero, *ftp://www.lsi.us.es/ cordero/RedI/ejemplos*

[17] R. Corchuelo, O. Martn, M. Toro, A. Ruiz y J.M. Prieto, *Weak Fairness in the Context of Constraint-Based Multiparty Interactions*, Proceedings of Spanish Symposium of Distributed Informatic, Editor: S. Barro, N.R. Brisaboa, J.M. Busta y F. F. Rivera, Santiago de Compostela, february 1999.

[18] Francez, *Interacting Processes*

[19] H. Alexander, *Formally-Based Tools and Techniques for Human-Computer Dialogues*, Ellis Horwood Limited, 1987.

[20] M. Mansouri-Samani y M. Sloman, *GEM. A Generalised Event Monitoring Language for Distributed Systems*, IEE/IOP/BCS Distributed Systems Engineering Journal, Volumen 4, Number 2, june 1997.

[21] T. Kielmann, *Designing a Coordination Model for Open Systems*, Lecture Notes in Computer Science, pages 267-284, Springer, 1996.

[22] C.F. Tschudin, *On the Structuring of Computer Communications*, PhD thesis, University of Geneva, 1993.

[23] P. Ciancarini, K.K. Jensen, y D. Yankelevich, *On the operational semantics of a coordination language*, LNCS 924, pages 77-106, Springer-Varlag, 1995.

[24] M.R. Genesereth y S.P. Ketchpel, *Software Agents*, Communications of the ACM 37 (7), pag. 48-53, 1994.

[25] T.W. Malone y K. Crowston, *The interdisciplinary study of coordination*, ACM Computing Surveys, 26(1):87-119, march 1994.

[26] N. R. Jennings, *Coordination Techniques for Distributed Artificial Intelligence*, Foundations of Distributed Artificial Intelligence, pages 187-210, John Willey & Sons, 1996.

[27] T. Malone y K. Crowston, *The Interdisciplinary Study of Coordination*, ACM Computing Surveys, vol. 26, num. 1, march 1994.

[28] J. Magee, J. Kramer y N. Dulay, *Darwin/mp: An environment for parallel and distributed programming*, Proceedings of 26th Annual Hawaii International Conference on System Sciences, volumen 2, IEEE Computer Society Press, 1993.

[29] S.J. Garland y N. A. Lynch, *The IOA Language and Toolset: Support for Designing, Analysing, and Building Distributed Systems*, MIT Laboratory for Computer Science, Cambridge, MA, 1998.

[30] G. Agha, *Abstracting interaction patterns: A programming paradigm for open distributed systems*, FMOODS'97 proceedings, 1997.

[31] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier y J. Irwin, *Aspect-oriented programming*, ECOOP'97 proceedings, LNCS 1241, pages 220-242, Springer-Verlag, june 1997.

[32] J. Jaworski, *Java 1.2 Al descubierto*, Prentice Hall, 1999.

[33] D. Lea, *Concurrent Programming in Java. Design Principles and Patterns*, Addison Wesley, 1997.