Characterizing Coordination Architectures According to Their Non-Functional Execution Properties

Valérie Issarny, Christophe Bidan, Titos Saridakis
IRISA / INRIA
Campus de Beaulieu, 35042 Rennes Cédex, France
{issarny, bidan, saridaki}@irisa.fr

Abstract

*A number of existing Distributed Processing Environments (*DPE*s) are eligible to serve as a coordination architecture. In order to ease the construction of distributed applications while exploiting existing* DPE*s, the developer should be provided with notations that allow him to characterize the coordination architecture that is the best suited to his application. Existing* DPE*s can be distinguished according to at least two criteria:* `i)` *the coordination protocols (e.g.* RPC*, pipe-filter, tuple space) they support, and* `ii)` *the non-functional execution properties (e.g. availability, security, responsiveness) they provide. This leads us to propose a twofold formal description of coordination architectures that characterizes the coordination protocols and non-functional execution properties that are expected from the underlying* DPE*, hence allowing to reason about the adequacy of a* DPE *for supporting a given coordination architecture.*

Keywords: *coordination protocols, coordination architecture, non-functional properties, software architecture, specification matching.*

1. Introduction

The coordination approach to the design of distributed applications provides a convenient framework to develop applications made of interacting activities. An application is described in terms of activities (e.g. processes, objects) that coordinate through the use of a *coordination architecture* that defines the communication patterns among the activities. The most common coordination architecture is the *tuple space* [6]: the tuple space is a shared data space, which exports a set of operations for the creation, synchronization, and communication of computational activities. However, any *Distributed Processing Environment* (DPE) providing support for the creation of activities, and their synchronization and communication is eligible as a coordination architecture. In that framework, there is a wide diversity of existing DPEs (e.g. Linda-like environments, RPC services, message passing environments) that can be used for developing coordination-based applications.

In order to ease the deployment of coordination-based applications over existing DPEs, notations should be provided for formal characterization of the coordination architecture that is the best suited to an application and dually that of the coordination architecture that is supported by a DPE. Such a characterization allows to reason about the adequacy of a DPE for supporting the execution of a given application, which promotes safe reuse of both applications and DPEs. In addition, the formal specification of DPEs may be used to reason about interoperability issues among heterogeneous DPEs, and ultimately to automatically customize existing DPEs according to the application needs. Formal description of DPEs together with its usage for easing the design, construction and deployment of distributed applications while exploiting existing DPEs and associated system services, constitute the core research of the Aster project[1]. In this paper we focus on the approach undertaken in Aster to the formal description of coordination architectures, hence providing the base ground for dealing with reuse, interconnection and customization of DPEs.

Existing DPEs can be distinguished according to at least the coordination protocols (e.g. RPC, pipe-filter, tuple space) they support. In addition, DPEs providing similar coordination protocols may differ from the standpoint of the non-functional execution properties (e.g. availability, security, responsiveness) they provide. For instance, the PLinda environment [10] extends a base Linda environment by providing availability guarantees through the implementation of transactional and checkpointing mechanisms. This leads us to propose a description of DPEs that subdivides into two complementary, yet independent formal specifica-

[1] `http://www.irisa.fr/solidor/work/aster`

tions that respectively relate to the coordination protocols and non-functional properties provided by the architecture. Practically, the twofold description of coordination architectures allows to characterize the architecture that best fits the application needs and dually to abstract the features of a given DPE from the standpoint of coordination management. As a result, this provides a sound basis to reason about the adequacy of a DPE for supporting the execution of a given application. Section 2 gives an overview of the proposed twofold description of coordination architectures. Sections 3 and 4 then detail the formal description of a coordination architecture from the standpoint of the provided coordination protocols and non-functional properties, respectively. The specification of coordination protocols is based on the work of [2]; it is however detailed for the sake of clarity. Finally, Section 5 gives conclusion, summarizing our contribution and addressing our current and future work.

2. Twofold Description of Coordination Architectures

Our approach to the description of coordination architectures builds on results produced in the field of software engineering that aims at providing a sound basis for the specification of various styles of software architectures (e.g. [1, 18, 21]).

2.1. Relating coordination architectures with software architectures

An architectural style identifies the set of patterns that should be followed by the system organization, that is, the kinds of components to be used and the way they interact. Although the software architecture field is continuously evolving, it is now accepted that the description of a system architecture decomposes into at least three abstractions (e.g. [21]):

(*i*) *Components* that abstractly define computational units written in any programming language.

(*ii*) *Connectors* that abstractly define types of communications (e.g. pipe, RPC) between components.

(*iii*) *Configuration* that defines a system structure (i.e. a software architecture) in terms of the interconnections of components through connectors.

Development environments that are based on the software architecture paradigm integrate an Architectural Description Language (ADL) that allows system specification in terms of the three above abstractions together with runtime libraries that implement base system services (including primitive connectors).

The software architecture abstractions relate to coordination architectures as follows: components and connectors respectively correspond to activities and coordination architectures; and a configuration defines the organization of the application activities together with requirements upon the underlying coordination architecture. Furthermore, if we consider that existing DPEs can be characterized by the connectors they provide, the selection of a DPE for implementing the coordination architecture required by an application consists of retrieving the DPE connectors that *match* the connectors required by the application.

2.2. Connector matching

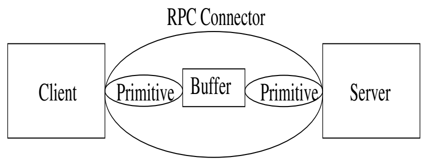
A straightforward way to reason about connector matching is to verify the compatibility of the connectors signatures (i.e. application programmer interfaces). However, such a matching relation is independent of connectors behaviors. A more powerful definition of connector matching consists in introducing a relation among specifications of connectors, hence taking into account the semantics of connectors and not just their signatures. The definition of specification matching relations has been extensively dealt with in the literature to reason about components compatibility (e.g. [11, 17, 22]). On the other hand, the definition of connector matching relations has received little attention.

A definition of connector matching has been introduced in [2], and is aimed at checking the correct use of connectors by components with respect to the protocols supported by the connectors. The component and connector behaviors are formally defined using a CSP-based algebra and the refinement concept is used to verify compatibility between components and connectors. As discussed in Section 3, this proposal forms the basis of connector specification matching from the standpoint of coordination protocols.

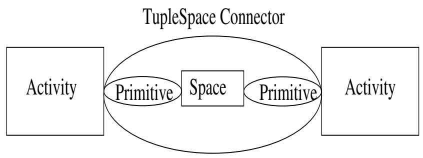
Matching of connector specifications with respect to their non-functional properties is addressed in Section 4. The undertaken approach is based on our proposal of [9] that introduces the formal specification of non-functional properties provided by a specific RPC system. The connector behaviors are formally defined in terms of first order logic formulas that characterize non-functional properties, and the specification matching relation is defined in terms of implications among formulas.

2.3. Description of coordination-based connectors

For illustration, let us consider the two software architectures depicted in Figure 1 that respectively use coordination architectures based on RPC and tuple space connectors. Both coordination architectures are characterized by connectors, qualified as *complex*, that are made of more primitive connectors (e.g. message passing, shared memory) and



a) A connector based on RPC



b) A connector based on tuple space

Figure 1. Coordination based connectors

components (e.g. buffer, memory space), which may themselves be complex.

As stated earlier, our approach to the specification of connectors consists of specifying separately their associated coordination protocols and non-functional properties. For instance, let us focus on the coordination architecture of Figure 1.*b*. Let us further assume two instances of this architecture that are Linda [6] and PLinda [10]. We get the following descriptions for the corresponding connectors:

Connector Linda = {
  Protocol TupleSpace;
  NonFunctional Null;}

Connector PLinda = {
  Protocol TupleSpace;
  NonFunctional FlatTransaction, Checkpointing;}

where the *Protocol* and *NonFunctional* clauses are used to declare the specifications of coordination protocols and non-functional properties respectively.

The advantage of a twofold description of connectors is that the specifications of coordination protocols and non-functional properties are addressed separately. As a result, the specification process becomes more powerful since both specifications can be reused and combined according to the connector semantics. In addition, let us notice that the connector matching process which verifies that a DPE connector matches an application connector can be automated. Model checking technology can be used to check matching between coordination protocol specifications [2] while a theorem prover can be used to verify matching between the specifications of non-functional properties [9].

The next two sections detail the specifications of connectors where we concentrate on the coordination among existing activities. Hence, we do not deal with the creation of activities, and we focus on the communication and synchronization aspects of coordination architectures.

3. Describing Coordination Protocols

For the specification of connectors from the standpoint of coordination protocols, we use the notations of the Wright ADL, introduced in [2]. Compared to other ADLs, Wright introduces a CSP-like notation [7] for the formal specification of the behaviors of components and connectors with respect to their communication patterns. Hence, this allows to verify the correctness of a configuration with respect to the communication protocols that are used.

3.1. Specifying coordination protocols

Using Wright, a component description embeds a set of *port* processes that are the component interaction points, and a *coordination* process defining coordination among ports. The behavior of a component is then the parallel composition of port and coordination processes. Similarly, a connector is defined in terms of a set of *role* processes that realize communications among components, and of a coordination process that specifies the coordination among roles. The behavior of a connector is defined as the parallel composition of role and coordination processes. Finally, a configuration is described as a set of components and connectors, which are interconnected through appropriate bindings between ports and roles.

Connector matching with respect to provided coordination protocols. In CSP-based algebras, a process describes an entity that can engage in communication events where $e!x$ and $e?x$ respectively denote input and output of data. Notation $e \rightarrow P$ denotes the execution of process P after engaging in event e, and P = *definition* denotes process naming. Given the specification of coordination protocols using a CSP-based algebra, connector matching with respect to the supported coordination protocols relies on the refinement concept, which is defined in terms of the divergences and failures model of CSP [7]. A DPE connector specification defining the set of roles $\mathcal{P}^{\mathcal{D}}_{r_j}$, $1 \leq j \leq m$, and the coordination process $\mathcal{P}^{\mathcal{D}}_{c}$, matches a connector specification required by the application which defines the set of roles $\mathcal{P}^{\mathcal{A}}_{r_i}$, $1 \leq i \leq n$, and the coordination process $\mathcal{P}^{\mathcal{A}}_{c}$, if:

$$(\forall \mathcal{P}^{\mathcal{A}}_{r_i}\colon \exists\, \mathcal{P}^{\mathcal{D}}_{r_j} \mid \mathcal{P}^{\mathcal{A}}_{r_i} \textit{ refines } \mathcal{P}^{\mathcal{D}}_{r_j}) \textit{ and } (\mathcal{P}^{\mathcal{A}}_{c} \textit{ refines } \mathcal{P}^{\mathcal{D}}_{c})$$

where the *refines* relation is similar to the compatibility relation among port and role processes that is defined in [2]. Roughly, for two processes P and Q, P *refines* Q if the behaviors of Q include those of P.

3.2. Coordination architecture examples

For illustration, let us consider the specification of the coordination protocols supported by the complex connectors depicted in Figure 1. In the following, we introduce protocol specifications for a robust use of connectors, such that an RPC request issued by a client is always served by a server and that a tuple written in the tuple space by an activity is always read by an activity. In other words, the proposed specifications define the coordination among activities and mask the intermediate communication actions using either the buffer or the tuple space. However, notice that coordination architectures with weaker requirements on activities behaviors can be specified using the same approach. For instance, in the case of the architectures of Figure 1, this would consist in specifying the behaviors of the primitive connectors from which the complex ones are built, leading to describe intermediate communications using the underlying data space.

Let us first give the specification of the coordination protocol implemented by the RPC connector of Figure 1.*a*. Coordination is achieved through the interaction between two activities that respectively act as the client and the server of an RPC request. Based on the Wright notations described in [2], we get:

Protocol RPC = {
  role Client = request ! x → result ? y(x) → Client;
  role Server = call ? x → return ! y(x) → Server;
  coordination =
    Client.request ? x → Server.call ! x →
    Server.return ? y(x) → Client.result ! y(x) →
    coordination;
  }

Let us now examine the specification of the coordination protocol implemented by the tuple space connector of Figure 1.*b*. Coordination relies on the interaction between two activities where one inserts a tuple into the tuple space (*out* role) and the other retrieves it (*in* role). We get the following specification:

Protocol TupleSpace = {
  role In = get ? y(x) → In;
  role Out = tuple ! y → Out;
  coordination =
    Out.tuple ? y(x) → In.get ! y(x) → **coordination**;
  }

The description of coordination architectures that has been discussed in this section allows to express the requirements for the supporting DPE in terms of coordination protocols. However, this is not sufficient to determine the DPE that is best suited to implement the coordination architecture required by an application. Still considering the Linda and PLinda incarnations of the tuple space connector, the latter is eligible for supporting an application having fault tolerance requirements while the former does not provide any guarantee in terms of fault-tolerance. Properties offered by a coordination architecture through the transparent implementation of distribution management mechanisms are qualified as non-functional, and their specifications are addressed in the next section.

4. Describing Non-Functional Properties

For the specification of non-functional properties, we build on our previous work [9] that addresses the specification of such properties for a specific RPC-based DPE (i.e. the OMG's OMA [12]). The undertaken approach is based on the Inscape proposal [16] that uses Hoare's predicates for describing how software components are actually used. In Inscape, a component specification is defined in terms of predicates characterizing its state. Predicate expressions are then used in verifying the correctness of component interconnections (e.g. if a server actually provides the service expected by the client) rather than proving properties of program fragments. In our framework, we are concerned with connector specification instead of component specification. Thus, non-functional properties are defined in terms of base predicates characterizing coordination actions that are carried out by connector incarnations. An additional predicate is further introduced to cope with the occurrence of failures. In our framework, a failure can occur at the level of either the communication media, processors or software. The latter type of failures relates to the robust use of connectors that is assumed in this paper (§ 3.2), i.e. a software failure occurs if the activity that must react to an initiated coordination does not participate to it.

4.1. Specifying non-functional properties

Let A_1 and A_2 be two activities and let v be the value exchanged between A_1 and A_2 for coordination (e.g. RPC request, tuple). We define the following base predicates to characterize connector behaviors from the standpoint of non-functional properties:

- $export(A_1, A_2, v)$ corresponds to the emission of v by A_1 to A_2.
- $import(A_1, A_2, v)$ states that A_2 gets v *once and only once*
- $import^+(A_1, A_2, v)$ states that A_2 gets v *at least once*.
- $failure(A_1, A_2, v)$ holds when a failure occurs during the coordination between A_1 and A_2.

Notice that the behavior that is captured by the $import$ and $import^+$ predicates depends on the coordination architecture that is considered. For instance, let us (still informally) precise the meaning of *import* in our architecture examples. In the case of the RPC-based architecture, *import* holds when A_2 processes A_1's request and issues the result of the request v. In the case of the architecture based on tuple space, the predicate holds when the value v that is exported by A_1 is read by A_2.

We further bring to the reader's attention that both coordination protocols described in subsection 3.2 imply that in the absence of failures the value exported by A_1 for coordination is eventually imported by A_2. More generally, considering the case of failures, our assumption is formally expressed as:

$$\begin{array}{l} export(A_1, A_2, v) \Rightarrow \\ \quad (import(A_1, A_2, v) \vee import^+(A_1, A_2, v) \vee \\ \quad\; failure(A_1, A_2, v)) \end{array}$$

Base behavior of coordination architectures. As stated previously, the behavior of a connector in terms of the introduced base predicates depends on the coordination achieved by the connector. The behavior of the connector protocol can be formally defined in terms of these predicates, according to Hoare's logic. For instance, let us consider the two connectors of Figure 1 whose protocols are defined in subsection 3.2. In the absence of failures we get the following specifications for the connector roles:

$$\begin{array}{c} true \\ \{\text{Client.request ! v}\} \\ export(A_{Client}, A, v) \end{array}$$

$$\begin{array}{c} true \\ \{\text{Server.return ! v}\} \\ import(A, A_{Server}, v) \end{array}$$

$$\begin{array}{c} true \\ \{\text{In.get ? u(v)}\} \\ import(A, A_{In}, v) \end{array}$$

$$\begin{array}{c} true \\ \{\text{Out.tuple ! u(v)}\} \\ export(A_{Out}, A, v) \end{array}$$

where A_x, $x \in \{Client, Server, Out, In\}$, stands for the activity executing the coordination role, A is the other participating activity, and v and v respectively stand for the sent value and its unique encoding in the set of all the exchanged values. Still considering the specification of the coordination protocols introduced in subsection 3.2, in the absence of failures we obtain the following specifications for the coordination activities, for both the RPC and the TupleSpace architectures:

$$\begin{array}{c} true \\ \{\textbf{coordination}\} \\ export(A_1, A_2, v) \wedge \\ import(A_1, A_2, v) \end{array}$$

where **coordination** denotes the connector coordination process, A_1 and A_2 denote the two activities involved in the coordination, and v is the encoding of the exchanged value.

To capture the occurrence of failures, we need to enrich Hoare's formulas so as to distinguish the behavior of an operation according to its termination point, which is either normal or exceptional. Using the extension of Hoare's logic that has been proposed in [5], we get the following specifications for both the RPC and the TupleSpace coordination protocols, in the case of failures:

$$true\ \{\textbf{coordination}\}\ failure : failure(A_1, A_2, v)$$

where the notation "$failure$:" is used to state that the formula gives the operation behavior under failure occurrence. We assume the correctness of the above formulas with respect to the semantics of the connectors. In particular, this implies that connectors include mechanisms for failure detection and notification. However, notice that the last assumption is not a prerequisite and is made for simplification purposes.

Defining non-functional properties. The introduced base predicates are used to define logic formulas characterizing non-functional properties, which are conjuncted to specify behaviors of connectors. With respect to the wide range of existing coordination architectures, various non-functional properties can be defined. As an example, let us give the definition of the *exactly once* failure semantics. In the RPC coordination protocol, this semantics means that a called operation will be executed once in the absence of failures and none otherwise. In the case of the TupleSpace coordination protocol, the exactly once semantics means that a tuple written in the tuple space can be imported only once in the absence of failures (i.e. there are no duplicates) and none otherwise. We define:

$$\begin{array}{l} \textit{ExactlyOnce}(A_1,A_2,v) \equiv \\ \quad \textit{export}(A_1,A_2,v) \wedge \\ \quad ((\neg \textit{failure}(A_1,A_2,v) \Rightarrow \textit{import}(A_1,A_2,v)) \wedge \\ \quad\; (\textit{failure}(A_1,A_2,v) \Rightarrow \neg \textit{import}^+(A_1,A_2,v))) \end{array}$$

Additional examples of non-functional properties are given in the next subsection that introduces the specifications of properties relating to transaction management.

Connector matching with respect to provided non-functional properties. Given the specification of non-functional properties using the first order logic, connector matching with respect to provided non-functional properties relies on implication [9]. A DPE connector specification defining the set $\mathcal{E}^{\mathcal{D}}$ of (provided) non-functional properties $P_i^{\mathcal{D}}$, $1 \leq i \leq n$, matches a connector specification required by the application which defines the set $\mathcal{E}^{\mathcal{A}}$ of (required) non-functional properties $P_j^{\mathcal{A}}$, $1 \leq j \leq m$, if:

$$\forall P_i^{\mathcal{A}} \in \mathcal{E}^{\mathcal{A}} : \exists P_j^{\mathcal{D}} \in \mathcal{E}^{\mathcal{D}} \mid P_j^{\mathcal{D}} \Rightarrow P_i^{\mathcal{A}}$$

4.2. Coordination architecture examples

For illustration of non-functional property specifications, we consider the example of transaction management. Traditional transactions are based on the notion of atomicity, which is characterized by two properties: failure atomicity and serializability. Failure atomicity means that either all or none of the transaction's operations are performed. Serializability relates to concurrent transactions and means that they execute without any interference. These two properties usually combine the notions of Atomicity, Consistency, Integrity and Durability (ACID). However, the emergence of new complex applications has led to propose new transaction models that revise the ACID notions by introducing different definitions of the failure atomicity and serializability properties (e.g. [4]).

In the remainder,we concentrate on the failure atomicity property by considering only non-concurrent transactions. Furthermore, we assume the following operations for transaction management: *begin* that initiates a transaction, *commit* that validates the embedding transaction and *abort* that aborts the embedding transaction. Finally, we introduce the following base predicate to characterize *durability* of effects: $\mathit{validate}(A, A', v)$ holds if the effects of the computation resulting from the coordination between A and A' have been received by the latter and are made permanent through the *commit* operation, i.e.:

$$true\ \{\mathit{commit}\}\ validate(A, A', v).$$

The durability of effects implies that effects are first received, and hence the following predicate should be always verified:

$$validate(A, A', v) \Rightarrow (import(A, A', v) \vee import^{+}(A, A', v))$$

Besides the *commit* operation, we consider that an *abort* operation denotes a coordination failure that is explicitly caused by one of the transaction participants, i.e.:

$$true\ \{\mathit{abort}\}\ failure(A, A', v).$$

Failure atomicity. In a first step, we consider the simplest transaction model that consists of implementing failure atomicity only for coordinations executed by the activity initiating the transaction. This appears to be similar to the *ExactlyOnce* failure semantics except that we have to take into account the durability of effects. We have:

$$\begin{array}{l} \mathit{Atomic}(A_1, A_2, v) \equiv \\ \quad \mathit{export}(A_1, A_2, v) \wedge \\ \quad ((\neg \mathit{failure}(A_1, A_2, v) \Rightarrow \\ \qquad (\mathit{import}(A_1, A_2, v) \wedge \mathit{validate}(A_1, A_2, v)\ ) \wedge \\ \quad (\mathit{failure}(A_1, A_2, v) \Rightarrow \neg \mathit{import}^{+}\ (A_1, A_2, v))) \end{array}$$

and hence, we get: $true\ \{\mathit{commit}\}\ Atomic(A, A', v)$ for the coordinations performed by the activity that initiated the transaction using the *begin* operation. The above property does not take into account failures that might occur during the executions of nested operations, i.e. of operations executed in the scope of the coordination performed by the activity initiating the transaction.

Let us introduce a stronger failure atomicity property for transactions, which specifies the behavior of nested operations. It states that the effects of a transaction are committed if all the operations executed in the scope of the transaction have been executed successfully. On the other hand, if a failure occurs during the execution of one of the nested operations, the effects of all the operations that have been executed in the scope of the transaction are undone. We define:

$$\begin{array}{l} \mathit{FlatTransaction}(A_1, A_2, v) \equiv \\ \quad \mathit{export}(A_1, A_2, v) \wedge \\ \quad ((\mathit{ImportRecursive}(A_1, A_2, v) \Rightarrow \mathit{validate}(A_1, A_2, v)) \wedge \\ \quad\ (\neg\ \mathit{ImportRecursive}(A_1, A_2, v) \Rightarrow \mathit{Undo}(A_1, A_2, v))) \end{array}$$

$$\begin{array}{l} \mathit{ImportRecursive}(A_1, A_2, v) \equiv \\ \quad \mathit{import}(A_1, A_2, v) \wedge \neg\ \mathit{failure}(A_1, A_2, v) \\ \quad \wedge_{\forall (A,n) \in nested_v}\ \mathit{ImportRecursive}(A_2, A, n) \end{array}$$

$$\begin{array}{l} \mathit{Undo}(A_1, A_2, v) \equiv \\ \quad \neg\ \mathit{import}^{+}\ (A_1, A_2, v) \wedge_{\forall (A,n) \in nested_v}\ \mathit{Undo}(A_2, A, n) \end{array}$$

where the set $nested_v$ identifies coordinations that are dependent upon the top-level coordination between A_1 and A_2. We thus get:

$$true\ \{\mathit{commit}\}\ FlatTransaction(A, A', v)$$

for all the coordinations performed by the activity that initiated the transaction using the *begin* operation. Notice that due to the use of *validate*, the above property specifies durability of effects only for the top-level coordination. One could introduce a stronger durability property that would imply durability of nested operations. We do not detail such a property whose definition is similar to the one of *Undo*.

Let us now examine the nested transaction model, where failures occurred during the execution of nested operations do not necessarily lead the parent transaction to abort. We define:

$NestedTransaction(A_1, A_2, v) \equiv$
$export(A_1, A_2, v) \wedge$
$((NestedImport(A_1, A_2, v) \Rightarrow validate(A_1, A_2, v)) \wedge$
$(\neg\, NestedImport(A_1, A_2, v) \Rightarrow Undo(A_1, A_2, v)))$

$NestedImport(A_1, A_2, v) \equiv$
$import(A_1, A_2, v) \wedge \neg\, failure(A_1, A_2, v)$
$\wedge_{\forall (A,n) \in subt_v}\ (NestedImport(A_2, A, n) \vee Undo(A_2, A, v))$
$\wedge_{\forall (A,n) \in nested_v - subt_v}\ NestedImport(A_2, A, n)$

where the set $nested_v$ has the same meaning as before, and the set $subt_v$ identifies nested transactions. We obtain: $true\ \{commit\}\ NestedTransaction(A, A', v)$ for all the coordinations performed by the activity that initiated the transaction using the *begin* operation. As a final example of transaction models, we examine the treatment of split transactions. This model is characterized by the fact that the behavior of a split transaction has no impact on the failure atomicity of the transaction that created it. We define:

$SplitTransaction(A_1, A_2, v) \equiv$
$export(A_1, A_2, v) \wedge$
$((SplitImport(A_1, A_2, v) \Rightarrow validate(A_1, A_2, v)) \wedge$
$(\neg\, SplitImport(A_1, A_2, v) \Rightarrow SplitUndo(A_1, A_2, v)))$

$SplitImport(A_1, A_2, v) \equiv$
$import(A_1, A_2, v) \wedge \neg\, failure(A_1, A_2, v)$
$\wedge_{\forall (A,n) \in nested_v - split_v}\ SplitImport(A_2, A, n)$

$SplitUndo(A_1, A_2, v) \equiv$
$\neg\, import^+\ (A_1, A_2, v)$
$\wedge_{\forall (A,n) \in nested_v - split_v}\ SplitUndo(C_2, C, n)$

where the set $split_v$ identifies split transactions. Thus, for all the coordinations performed by the activity that initiated the transaction using the *begin* operation, we get:

$$true\ \{commit\}\ SplitTransaction(A, A', v)$$

Coordination architectures providing failure atomicity. Based on the specification of transactional properties given above, the specification of a connector providing such properties is straightforward. It amounts to declare the property name and the associated formula. For instance, the non-functional transaction property provided by the PLinda architecture [10] is the following:

NonFunctional FlatTransaction = {
Specification of the FlatTransaction property using the language of some theorem prover}

In the same way, the coordination architecture characterizing a DPE based on an implementation of the CORBA ORB [15] with an implementation of the CORBA OTS for transaction management [14] will declare the *NestedTransaction* and *FlatTransaction* properties.

Finally, in order to allow complete specification of a given system, the specification of the connector protocols should be given in terms of the base predicates used for defining non-functional properties (§ 4.1). This leads us to enrich the declaration of connectors accordingly. For instance, let us consider the PLinda architecture and let us assume that it complies to the coordination specification that was given in Subsection 4.1. We get the following PLinda connector description:

Connector PLinda = {
Protocol TupleSpace;
NonFunctional FlatTransaction, Checkpointing;
with Coordination (A, A', v) =
pre **true**;
post normal: export(A, A', v) **and** import(A, A', v),
failure: failure(A, A', v);
}

The above declaration states that the PLinda connector implements a coordination protocol based on tuple space, and provides the *FlatTransaction* and *Checkpointing* non-functional properties. It further gives the behavior of the coordination protocol in terms of the base predicates used for defining non-functional properties.

5. Conclusion

Although the tuple space is often used as the primary means to achieve coordination among activities, several alternative DPEs (e.g. RPC services, message passing environments) that support the creation, synchronization and communication of computational activities, may serve as coordination architectures. This allows the deployment of coordination-based applications over a wide range of execution platforms. In addition, these applications can be executed within different environments provided the target environments support similar coordination architectures. However, to exploit the diverse coordination architectures without sacrificing the software correctness, the application developer should be provided with means that allows to reason about the adequacy of a DPE for executing a given application.

This paper has introduced the basis for such a reasoning about DPE behaviors. The proposed approach consists of providing a twofold formal description of coordination architectures, which specifies on one hand the coordination protocols and on the other their non-functional properties. The coordination protocol specifications that are expressed in terms of a CSP-based process algebra, characterize the coordination achieved among activities by the architecture. The non-functional properties specifications that are expressed in terms of the first order logic, abstractly define the distribution management mechanisms (e.g. fault-tolerance, security mechanisms) that are transparently implemented by the architecture. The proposed twofold architecture specification can then be used to characterize the coordination architecture that is supported by an existing DPE and dually that is expected from an application. In addition, matching relations defined among the specifications of coordination protocols using the process refinement concept, and among the specifications of non-functional properties using logical implication, allow to check for architecture matching. Hence, this enables to verify that a given DPE can be safely used for executing an application based on the specifications of the provided and required coordination architectures.

Given the proposed twofold formal description of coordination architectures, the specifications of coordination protocols and non-functional properties can be dealt with independently and then conveniently combined to describe the architecture implemented by a given DPE. This allows the reuse of specifications for characterizing coordination architectures, hence easing the specification process. Moreover, it should be noticed that protocol specification and non-functional properties matching can be automated using respectively model checking and theorem prover technologies. As a result, this allows to provide the application developer with tools to check whether an existing DPE can be used for the execution of a given application. Furthermore, this provides the base ground for the automatic construction of DPEs customized to applications needs. Such a construction can be dealt with through the selection of a DPE that fits the application requirements in terms of coordination protocols. Then , based on the non-functional properties respectively required by the application and provided by the DPE, the DPE behavior can be enriched through the use of additional system services that implement the necessary non-functional properties.

Up to now, we have been experimenting with our approach from the standpoint of non-functional properties specification, through the design and implementation of the Aster environment for the development of distributed applications [8]. The Aster environment supports the description of the non-functional properties required by a distributed application whose activities coordinate through a DPE complying to the OMG's OMA standard [12]. Based on the application non-functional requirements, the Aster environment automatically retrieves the *object services* [13] that should be used in conjunction with the ORB to meet the application requirements. The current Aster prototype supports the description of non-functional properties relating to availability [20], security [3], and interoperability [19]. We are now studying the specification of properties relating to timeliness so as to allow the description of coordination architectures aimed at multimedia applications. In addition, we are examining the enrichment of the Aster environment so as to support the twofold description of coordination architectures discussed in this paper, hence enabling the development of various application classes over a wide range of DPEs.

References

[1] G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 1995.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 1997.

[3] C. Bidan and V. Issarny. Security benefits from software architecture. In *Proceedings of* COORDINATION*'97: Coordination Languages and Models*, pages 64–80, 1997.

[4] P. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.

[5] F. Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, SE-10(2):163–174, 1984.

[6] D. Gelertner and N. Carriero. Linda in context. *Communications of the ACM*, 32(4), 1989.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[8] V. Issarny and C. Bidan. Aster: A CORBA-based software interconnection system supporting distributed system customization. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 194–201, 1996.

[9] V. Issarny and C. Bidan. Aster: A framework for sound customization of distributed runtime systems. In *Proceedings of the Sixteenth IEEE International Conference on Distributed Computing Systems*, pages 586–593, 1996.

[10] K. Jeong and D. Shasha. PLinda 2.0: A transactional/checkpointing approach to fault tolerant linda. In *Proceedings of the Thirteenth Symposium on Reliable Distributed Systems*, pages 96–105, 1994.

[11] A. Mili, R. Mili, and R. Mittermeir. Storing and retrieving software components: A refinement based system. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 91–100, 1994.

[12] OMG. Object Management Architecture Guide (OMA Guide). Technical Report 92.11.1, OMG, 1992.

[13] OMG. Common Object Services Specification, Volume 1 – Revision 1.0. Technical Report 94.1.1, OMG Document, 1994.

[14] OMG. Object Transaction Service. Technical Report 94.8.4, OMG Document, 1994.

[15] OMG. The Common Object Request Broker: Architecture and Specification – Revision 2.0. Technical report, OMG Document, 1995.

[16] D. E. Perry. Software interconnection models. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 61–69, 1987.

[17] D. E. Perry. The Inscape environment. In *Proceedings of the Eleventh International Conference on Software Engineering*, pages 2–12, 1989.

[18] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[19] T. Saridakis, C. Bidan, and V. Issarny. A programming system for the development of TINA services. In *Proceedings of the International Conference on Open Distributed Processing*, 1997.

[20] T. Saridakis and V. Issarny. Towards Formal Reasoning on Failure Behaviors. In *Proceedings of the 2nd European Research Seminar on Advances in Distributed Systems (ERSADS)*, 1997.

[21] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, 1995.

[22] A. M. Zaremski and J. M. Wing. Specification matching of software components. In *Proceedings of the ACM SIGSOFT'95 Symposium on Foundations of Software Engineering*, pages 6–17, 1995.