Pantaxou
Mercadal, Julien; Palix, Nicolas; Consel, Charles; Lawall, Julia

Published in:
GPCE 08

DOI:
10.1145/1449913.1449936

Publication date:
2008

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Pantaxou: a Domain-Specific Language for Developing Safe Coordination Services

Julien Mercadal  Nicolas Palix  Charles Consel  
INRIA / LaBRI, Talence, France  
{mercadal, palix, consel}@labri.fr

Julia L. Lawall  
DIKU, University of Copenhagen, Copenhagen, Denmark  julia@diku.dk

Abstract

Coordinating entities in a networked environment has always been a significant challenge for software developers. In recent years, however, it has become even more difficult, because devices have increasingly rich capabilities, combining an ever larger range of technologies (networking, multimedia, sensors, etc.).

To address this challenge, we propose a language-based approach to covering the life-cycle of applications coordinating networked entities. Our approach covers the characterization of the networked environment, the specification of coordination applications, the verification of a networked environment and its deployment. It is carried out in practice by a domain-specific language, named Pantaxou.

This paper presents the domain-specific language Pantaxou, dedicated to the development of applications for networked heterogeneous entities. Pantaxou has been used to specify a number of coordination scenarios in areas ranging from home automation to telecommunications. The language semantics has been formally defined and a compiler has been developed. The compiler verifies the coherence of a coordination scenario and generates coordination code in Java.

Categories and Subject Descriptors  C.2.4 [Distributed Systems]: Distributed applications; D.2.2 [Design Tools and Techniques]: Modules and interfaces

General Terms  Design, Languages, Reliability, Verification

Keywords  distributed systems, coordination languages, architecture description languages, domain-specific languages, safety

1. Introduction

The commoditization of consumer electronics has created a surge in the number and kind of devices available, and the range of environments in which they are used. Examples of heterogeneous, device-rich environments include building maintenance where sensors and actuators manage activities ranging from security to energy consumption, and healthcare where a variety of devices monitor and assist subjects ranging from the ill to the elderly to world-class athletes. Most devices now have communication capabilities that enable them to combine their functionalities with that of other devices, forming a networked environment. Devices can furthermore interact via a variety of interaction modes, including commands (i.e., remote procedure calls (RPC)), events, and sessions.

The multiplicity of devices that may be present in a given environment raises the need for coordination among them. Indeed, the richness of their capabilities opens up a wide spectrum of coordination scenarios, many of which are being explored in research on pervasive computing environments [13, 14, 15, 22, 23]. Nevertheless, implementing coordination services for such environments remains a challenge. Indeed, little has been done to develop programming methodologies, support, and verifications for services coordinating heterogeneous networked devices. Let us examine the issues raised by this domain.

Complex interaction modes  The implementation of a distributed, heterogeneous environment is typically based on a middleware that manages the interaction between a coordination service and other networked entities. Middleware typically provides some form of an interface definition language that abstracts away from the details of the interaction between heterogeneous components, via automatically generated stubs [20, 25]. Existing interface definition languages, however, natively support only commands, via RPC-like mechanisms [24], deferring the implementation of more complex interaction modes such as events and streams to generic libraries or ad hoc code. The use of generic libraries, however, typically eliminates the possibility of static verification of type-safety or other correctness properties. Ad hoc implementations of complex interaction modes are difficult to construct and typically cannot be reused.

Openness of environments  Entities in a distributed environment may not always be available, either due to a failure of the entity itself or due to a failure of the network connection. Furthermore, new entities may be introduced into the environment at any time. Thus, a coordination service must use a process of service discovery in order to identify the entities with which it should interact [26]. The desired entities may be specified in terms of their types, but also in terms of their non-functional properties, as these may be relevant to the coordination activity. Existing middlewares such as CORBA typically provide some service discovery mechanisms, but these are often rudimentary, requiring desired services to be named using strings and without being able to take advantage of a subtyping hierarchy. As a result, queries may fail unpredictably because of a mismatch with the available environment entities or their functionalities.

Lack of programming support  Developing coordination services for networked entities is complicated because it involves expertise
in a number of fields such as distributed systems, networking, and operating systems. Abstraction layers such as middleware only partially shield the programmer from these aspects because of the general-purpose nature of their operations and mechanisms.

**Lack of reliability** Coordination services rely on a global understanding of the types of entities available in the environment in which they operate. General-purpose programming languages provide at best type safety, ensuring the validity of point-to-point interactions. But there is no mechanism for checking the consistency of an environment globally, from entity descriptions to the coordination logic, to an actual infrastructure.

**Our approach**

To address these issues, we propose a language-based approach, covering the complete life-cycle of an environment of networked entities: characterization of relevant entities, implementation of coordination services, and execution. To achieve reliability throughout this life-cycle, verifications are performed statically and dynamically.

Specifically, we introduce a domain-specific language, Pantaxou, that is geared towards identifying and modeling the building blocks relevant to a distributed heterogeneous environment and facilitating their coordination via high-level interaction modes. To address these needs, Pantaxou provides two language layers, one for creating an environment description and another for programming coordination services. The language layer for creating an environment description allows characterizing the categories of relevant entities, whether hardware (i.e., devices) or software (e.g., a database). Each description of a category amounts to an interface, defined by a set of attributes whose values identify a concrete entity, and interaction modes, consisting of signatures of commands (i.e., RPC-like operations), events, and sessions. These interfaces are organized hierarchically, allowing the relationships between them to be used during service discovery. The language layer for programming coordination services then offers high-level abstractions taking advantage of the information about environment properties contained in the environment description. Pantaxou facilitates the coordination of entities by providing an interaction model consisting of three modes that capture widely used communication abstractions, combining the standard RPC with events and sessions that are included by most libraries, middlewares or communication-oriented protocols. Because it builds upon standards-based technologies, Pantaxou’s interaction model has proven benefits in terms of re-use, portability and interoperability.

The Pantaxou environment description is used to ensure coherence throughout the entire environment life-cycle. The implementations of coordination services are guided by and must comply with the environment description. To assist in implementing coordination services, the environment description is used to tailor a programming framework, providing dedicated yet high-level operators to discover concrete entities and interact with them via their interface. Verifications based on the environment description are furthermore performed throughout the environment life-cycle. First, the environment description is analyzed to check its connectedness, i.e., that every entity interface corresponds to a source, a sink, or part of a producer-consumer pair. Second, various domain-specific properties of the implementations of the coordination services are checked, based on the information in the environment description. Finally, when deploying a concrete environment, a global verification checks whether the environment description, the coordination services, and the concrete environment match.

**Outline**

The rest of this paper is organized as follows. Section 2 introduces a working example of an advanced telecommunication scenario, relying on standard software and hardware entities. Its CORBA-based implementation is presented and key limitations are examined. Section 3 gives a tour of our DSL, Pantaxou, illustrated by our working example. Section 4 presents the main verifications enabled by the design and semantics of Pantaxou. Section 5 describes the main steps of the compilation of a Pantaxou program. Section 6 presents the related work and Section 7 provides concluding remarks.

2. **Background and working example**

Coordinating networked entities requires taking into account a number of issues regarding the capabilities and heterogeneity of the entities to be coordinated, programming support, and reliability. To illustrate these issues, we introduce a scenario that is challenging to develop with current programming languages and programming support, even though the components involved are completely
standard. We show how such a scenario can be implemented using CORBA [20], a widely used middleware for managing the interaction between networked entities.

2.1 Example scenario

Consider the problem of forwarding a secretary’s calls from her office to the office where she is currently located, to avoid losing calls. To address this situation, we may use some stationary Bluetooth device in each office (e.g., a desktop computer or a Bluetooth USB dongle) to detect the entrance of a secretary using any Bluetooth device she may wear (e.g., her personal cell phone). When the reader detects a Bluetooth device entering the office, it looks up the identity of the owner of the device in a database and stores the current office as the owner’s location. If the secretary is called while out of her office, we use her current location to forward the call to the office where she is located, indicating the name of the secretary on the phone’s display.

This scenario involves interaction between a number of components, representing both hardware and software entities: a Bluetooth detector to identify a Bluetooth device that is entering a room, a database to store the association between a Bluetooth device and its owner, and a phone to receive calls and establish a two-way audio stream. Services must be developed to coordinate these entities. A presence manager is needed to coordinate the detection of Bluetooth devices with the contents of the database containing owner identities. A presence agent is needed to map the identity of a secretary to her current office. Finally, a virtual phone is needed to forward the secretary’s calls to the phone of the office where she is currently located. The complete environment and our scenario are illustrated in Figure 1.

2.2 CORBA implementation

A CORBA implementation of the secretary scenario comprises a collection of interfaces describing how entities may interact, and application code implementing each of these interfaces. Extracts of such an implementation are shown in Figures 2 and 3. As shown in Figure 1, the secretary scenario involves a variety of kinds of entities, using different modes of interaction between them: commands are used to query the PresenceAgent to get the room where a user is located; events are used to disseminate information, such as the presence information published by the BluetoothPresenceManager, to interested parties; sessions are used to manage multimedia streams, such as audio calls.

Interfaces  Lines 3-11 of Figure 2 show some of the interfaces describing the interaction modes provided by the PresenceAgent in our secretary scenario. The GetRoom interface describes a command, implemented as the single method getRoom. The PresencePusher interface is part of the declaration of an event interaction between the BluetoothPresenceManager and the PresenceAgent. This interface describes the handler that is to be invoked when the PresenceAgent receives a presence event notification from the BluetoothPresenceManager. The PushConsumer interface, imported from the CosEventComm module, is also part of the declaration of the event interaction, allowing the PresenceAgent to receive events. Finally, the PresenceAgent interface lists all of the interfaces that describe how to interact with the PresenceAgent.

The remaining 35 lines of Figure 2 show the interfaces involved in establishing an audio session for the Phone entity. For the sake of genericity, a CORBA multimedia resource is manipulated through a multimedia entity and a flow manager. The multimedia entity typically corresponds to a device, whereas the flow manager is responsible for connecting a producer and a consumer. They each require a set of interfaces (lines 12-41 for the entity and lines 43-46 for the flow manager).

In Figure 2, lines 12-37, we define the Phone interface that specializes the generic streaming interface for the Phone entity with audio type: it comprises a collection of method signatures for creating an audio flow and negotiating stream properties. For this, the CORBA documentation[17] recommends textually specializing the generic set of interfaces for streaming with respect to a dedicated stream type. For example, in line 13, the create_A method is overloaded with dedicated argument types and return type, specializing it for the Phone entity.

Application code All of these interfaces must be supported by application code that implements the interactions. Figure 3 shows an extract of the implementation of the PresenceAgent class, including both its constructor and the methods corresponding to the various interfaces it implements.

The constructor includes 8 lines of boilerplate code that register the PresenceAgent object as a recipient of typed presence events. This comprises the following steps: (1) obtaining a typed event channel (lines 6-10), (2) obtaining a typed consumer administrator object from the typed event channel (line 11), (3) obtaining a proxy push supplier from the consumer administrator (lines 12), (4) connecting the consumer to the proxy supplier (line 13). Dual
steps are required for an event producer. Because the CORBA object
resource broker (orb) is generic, and thus is not aware of the specific
names of the resources needed by the application, all of the resources
involved in these steps are requested using string representations of
their names (e.g., line 7).

The remaining methods implement the operations defined in
the interfaces. The getRoom method (line 15) implements the
GetRoom command. The disconnect_push_consumer method
(line 16) implements a method required by the PushConsumer
interface as part of declaring that the PresenceAgent is able to
receive event notifications. Finally, the methods pushPresence
and push provide callback functions for handling received events.
The pushPresence method (lines 17-19) is used for the typed
presence events expected by the PresenceAgent. However, the
generic CORBA event library also requires a handler for all types
of events, for which we define the trivial method push (line 20).

2.3 Assessment

The CORBA implementation shown in Figures 2 and 3 illustrates the
previously mentioned difficulties with implementing a networked
environment using standard middlewares.

Complex interaction modes All of the interaction modes required
in a networked environment have to be expressed using a single
kind of abstraction, the interface. CORBA interfaces are natively
designed for commands. As shown in Figure 2, expressing events
or sessions using CORBA interfaces is possible, but cumbersome.
To express that the PresenceAgent should be able to receive a
single kind of event, two separate interfaces, PresencePusher
and PushConsumer, are required. Furthermore, these interfaces
are supported in the implementation of the PresenceAgent class
by three method definitions and complex code in the constructor.
Nothing about this set of program elements indicates the relationship
between them. Similarly, 9 interfaces and 20 method signatures,
amounting to over 100 lines of declarations, were required to define
a Phone interface dedicated to the audio session type. As for the
Phone implementation, the generality of the CORBA streaming
library requires even more setup steps than events, further inflating
the application code. Overall, in addition to the volume of code
required, the CORBA declarations are not expressed at the same
level as the conceptual software architecture, making it complex to
translate between them.

Openness of environments Developing code to coordinate a net-
worked environment requires knowledge of the entities to be co-
ordinated, such as their names, capabilities and availability. In the
CORBA-based implementation in Figure 3, the coordination logic of
the PresenceAgent class makes a fixed, string-based reference to
an event supplier in line 12. The pervasive use of strings to identify
resources in the networked environment makes the code vulnerable
to spelling errors, which are not detected until run time. A more
robust solution would be to refer to an event type, defined using the
underlying programming language. An even higher level approach
would consist of first selecting an entity from which events are to
be received and then specifying the event type of interest. In do-
ning so, different entities producing the same event type could be
distinguished from an architectural point of view.

Lack of programming support As the computing power of de-
vices has increased, it has become possible to interact with them at
much higher levels of abstraction. For example, Bluecove [7] offers a
Java API to control Bluetooth devices. However, the general-
purpose nature of these libraries as well as the middleware libraries
results in coordination code that contains repetitive program patterns.
General-purpose programming languages do not provide appropriate
abstractions, leading to cumbersome code mixing distributed opera-
tions, protocol programming support, and entity management. This
situation is illustrated by the intricate steps required for a consumer
to subscribe to a typed event channel, as shown in the constructor of
the PresenceAgent class (Figure 3, lines 6-13).

Lack of reliability The lack of appropriate architectural abstrac-
tions in CORBA requires the programmer to manage knowledge that
crosscuts the modules managing and coordinating the environment
entities. For example, the programmer needs to keep in mind the vari-
ous suppliers and consumers of events in a distributed application,
carefully spelling the names of suppliers for subscription. Errors are
mostly detected at runtime, making debugging difficult and reliability
unpredictable. Even worse, a misspelling may lead a supplier to
produce events that will never be received, because consumers sub-
scribed to the incorrectly spelled event name. This issue is further
complicated when the code is written by multiple developers, who
may make inconsistent assumptions about the implementations of
the complex interaction modes.

Furthermore, the protocol programming support provided by a
library assumes that code conforms to the protocol specification;
any error may corrupt the corresponding underlying subsystem. In

Figure 3. Fragments of a CORBA-based implementation of the secretary scenario (PresenceAgent class). For simplicity, exception handling has been omitted.
our example, an error in the implementation of call routing may lead to lost calls, or even crash the telephony platform. More generally, to the best of our knowledge, there is no approach covering the life-cycle of a coordination application from its design to its runtime, including both programming support and verification.

3. A Tour of Pantaxou

Pantaxou has been designed around the two main stages involved in developing a networked environment: (1) specifying a model of the networked environment for a given area and (2) developing coordination applications with respect to a given model. A separate language layer is provided for each stage. This section presents Pantaxou, following these two stages. It is illustrated by our secretary scenario.

3.1 Specifying a networked environment

Pantaxou targets application areas, such as pervasive computing, in which the set of available concrete entities is unpredictable, both due to the wide range of devices currently available and the fast pace of the development of new devices. Furthermore, a goal of Pantaxou is to verify the coherence of distributed applications, thus requiring a global view of environment entities. To address these challenges, Pantaxou provides developers with a language layer to define a taxonomy of the entities relevant to a networked environment. This taxonomy declares what are the relevant kinds of entities in an abstract way, how they should interact, and how they relate to each other. Figure 4 gives a complete grammar of this language layer.

3.1.1 Data types and interfaces

Specifying a networked environment using Pantaxou first involves declaring the kinds of data the various entities exchange. Data structures are declared using standard structure declarations, as illustrated in Figure 5. For example, the data type Room (lines 3-5) is used by the GetRoom command, and the data type Presence (lines 6-10) describes the information associated with a Presence event. Data types can be imported (e.g., Audio, line 1). Pantaxou also provides some domain-specific types, such as uri (Uniform Resource Identifier).

The specification of a networked environment using Pantaxou also includes declarations of command interfaces. Examples are given on the right side of Figure 5. Such a declaration names a group of commands (e.g., DbQueryBluetooth) and lists the commands’ signatures (e.g., getInfoBT). No interface declarations are needed for events or sessions, because they always involve the same operations, e.g., publish and subscribe for events, which are parameterized over the type of the data that is exchanged.

Figure 5. Data types and signatures for the secretary scenario

Data structure declarations have essentially the same form in CORBA and Pantaxou. The command declarations of Pantaxou have essentially the same form as CORBA interface declarations, but are designated as relating to commands, and are thus explicitly distinguished from other types of declarations.

3.1.2 Taxonomy of entities

In addition to the data types and interfaces, a Pantaxou specification includes a taxonomy of the classes of services that are available in the environment. A service class groups together entities that share the same functionalities. We refer to an entity in a service class as a service. A service interacts with other entities in terms of their service classes, via a process of service discovery. It is thus shielded from irrelevant variations in the available entities. A service class declaration specifies a collection of attributes that range over the variations in non-functional service properties and declares the interaction modes that services provide to and require from other service classes. Finally, service classes are hierarchically organized, enabling the coordination logic to abstract over unnecessary service details by using less specific service classes. Unlike in CORBA, where both operations and entities are declared using a single kind of abstraction, the interface, Pantaxou uses distinct kinds of abstractions for services and for the interaction modes (commands, events, and sessions) that a service requires and provides.

Figure 6-a displays the declarations of service classes for the advanced telecommunication environment. The declared service classes form a hierarchy, as shown in Figure 6-b. Each service class is associated with a set of attributes and interaction modes. The attributes, such as room in the Device service class (line 17-19), allow a service to characterize itself in terms of the values of its non-functional properties. These attributes are then consulted during service discovery, e.g., allowing the secretary scenario to discover a phone in a specific room to be able to forward a call. The interaction modes are annotated to indicate whether the interaction mode is provided or required and the classes of services with which the interaction is allowed. For example, the PresenceAgent (lines 12-16) requires Presence events that are produced by the Bluetooth-PresenceManager (lines 13-14). A PresenceAgent must also provide the GetRoom command to any service (line 15). Finally, the Phone service (line 24-27) illustrates how to define services that use sessions. Only three lines of code are required, as compared to the over 100 lines of declarations that must be written to define a typed service interface that uses typed streams.

The hierarchical definition of service classes allows subclasses to inherit the attributes and interaction modes of their ancestors. This approach makes it possible to characterize classes of entities by incremental refinement. For example, in our advanced telecommunication environment, we introduce a virtual phone that behaves like a phone (its ancestor) but requires a command to get the current
```java
1 service Service {} 1
2 service BluetoothPresenceManager extends Service {} 1
3 requires event<BluetoothDetection> 1
4 from BluetoothDetectorAgent; 1
5 requires command<DbQueryBluetooth> from InfoDB; 1
6 provides event<Presence> to PresenceAgent; 1
7 } 1
8 service InfoDB extends Service {} 1
9 provides command<DbQueryBluetooth> 1
10 to BluetoothPresenceManager; 1
11 } 1
12 service PresenceAgent extends Service {} 1
13 requires event<Presence> 1
14 from BluetoothPresenceManager; 1
15 provides command<GetRoom> to Service; 1
16 } 1
17 service Device extends Service {} 1
18 Room room; 1
19 } 1
20 service BluetoothDetectorAgent extends Device {} 1
21 provides event<BluetoothDetection> 1
22 to BluetoothPresenceManager; 1
23 } 1
24 service Phone extends Device {} 1
25 provides session<Audio> to Phone; 1
26 requires session<Audio> from Phone; 1
27 } 1
28 service VirtualPhone extends Phone {} 1
29 requires command<GetRoom> from PresenceAgent; 1
30 }
```

(a) Service declarations

(b) Hierarchy of services

Figure 6. Model of the advanced telecommunication environment

```java
import datatype Room;
import datatype Presence;
import service Service.BluetoothPresenceManager;
'sip:pres.agent@enseirb.fr'
instantiates
Service.PresenceAgent {
service<BluetoothPresenceManager> {} bm;
event<Presence> from bm[*] {} presenceEvt;
var HashTable htable;
onReceive presenceReception(presenceEvt e) {
Presence p = e.data;
if (p.status) {
htable.put(p.entity, p.room);
} else {
htable.remove(p.entity);
}
}
Room getRoom(uri user) {
return htable.get(user);
}
initial {
adopt(presenceReception);
htable = new HashTable<uri, Room>();
}

(a) Presence agent service

(b) Call forwarding service

Figure 7. Secretary scenario implemented in Pantaxou (part 1)
location of a secretary to forward her calls to the phone in a given room.

3.2 Developing coordination applications

The development of an application is guided by the description of the networked environment, enabling the programmer to focus on the coordination logic. Let us illustrate this approach by examining the complete Pantaxou implementation of the coordination code for the secretary scenario, as displayed in Figures 7 and 8. In this scenario, the Presence agent service (Figure 7-a) receives Presence events from the BluetoothPresenceManager; the Presence agent service can then be queried for the current room of a secretary. The call forwarding service (Figure 7-b) coordinates the interaction between the presence agent and the phones to determine how to route calls. Finally, the Presence manager service (Figure 8) coordinates the interaction between the Bluetooth detector agents, the database, and the Presence agent to detect, identify, and publish position information about Bluetooth devices and their owners.

The implementation of a service is guided by the declarations in the environment model. We take the definition of the Presence agent service (Figure 7-a) as an example. This code should be compared with the CORBA-based implementation shown in Figure 3. This service is given the name ‘sip:pres.agent@enseirb.fr’ (line 5) and is declared to instantiate the PresenceAgent service class (lines 6-7). The declaration of the PresenceAgent service class (Figure 6, lines 12-16) indicates that this service requires interactions with BluetoothPresenceManager services, from which it will receive Presence events. The Presence agent service thus declares this service class (line 9), including the values of any desired attributes within braces (none are needed in this case), and the event that the Presence agent service should receive from members of this service class (line 11). The Presence agent service may receive events from any number of BluetoothPresenceManager services, as indicated by [*]. The declaration of the source of the Presence events has no direct counterpart in the CORBA-based implementation, as CORBA does not give a service the means to describe this information. The declaration of the required type of event on line 11 roughly corresponds to the 8 lines of code in the constructor of the PresenceAgent class shown in Figure 3. Note that in these declarations, the name of the service class of the event provider is represented as a type name, which can be checked by the Pantaxou static semantics (Section 4.2), rather than a string (Figure 3, line 12), as would be used in CORBA.

Next, the Presence agent service declares the handler, presenceReception, for the Presence event (lines 15-22). The handler presenceReception takes as argument a presenceEvt structure. This structure was previously declared on line 11 as being an instantiation of event with respect to the data type Presence (lines 6-10, Figure 5). It has as fields entity, status, and room, corresponding to the parameters of the event handler push_presence in the CORBA based implementation (lines 17-19, Figure 3). The only other data manipulated is the hash table, htable, defined on line 13. The remainder of the definition of the Presence agent service defines getRoom on lines 24-26, providing an implementation of the GetRoom command, and the initial block (lines 28-31), playing the role of a constructor. The initial block simply uses adopt (line 29) to indicate that presenceReception should be used as the current handler for its associated event, and creates the hash table htable (line 30). The simplicity of this code is in sharp contrast to that of the constructor in the CORBA-based implementation.

The remainder of the implementation of the secretary scenario consists of the Call forwarding service displayed in Figure 7-b and the Presence manager service displayed in Figure 8. The Call forwarding service illustrates how sessions are manipulated in Pantaxou. Because the virtual phone service requires a session request, it defines a handler to receive these sessions (lines 18-32). When this handler is invoked with a session request from a caller, it discovers a single phone (new myPhone[1], line 23) in the office where the secretary is located (lines 19-22). Then, the invite construct is passed the selected phone and the session request in order to create a two-way audio session. Finally, the Presence manager service follows the same basic structure as the Presence agent service. Its event handler, btEvtReception (lines 24-33) additionally creates (lines 28-31) and publishes (line 32) a Presence event.

3.3 The benefits of tightly-coupled languages

A key innovative feature of our approach is to tightly couple the environment description language and the coordination logic language. In doing so, the programming of the coordination of services is rigorously driven by the environment description. Specifically, the development of a service is driven by its service class declaration, defining what services can be discovered, whether they can interact, and if so, using what interaction modes.

To illustrate this coupling, let us examine more closely the Pantaxou coordination code for our secretary scenario (Figure 7-b). In conformance to the declaration of the virtual phone service class given in Figure 6-a, its implementation can only interact with (1) services of type PresenceAgent via the command mode of type GetRoom (line 29) and (2) services of type Phone via a session mode of type audio, as inherited from the service class Phone (lines 25-26). These declarations parameterize language constructs to guide the service implementation. In particular, consider the service discovery construct of the form

```
service<C>{...}
```

![Figure 8. Secretary scenario implemented in Pantaxou (part 2)](image-url)
When implementing a virtual phone, this service discovery con-
struct can only refer to the three possible service classes: Phone,
VirtualPhone (its sub-class) and PresenceAgent. This is illus-
trated in Figure 7-b, lines 19-22, where a service of type Phone or
VirtualPhone is looked up. Because this service lookup is based
on the service class hierarchy, the coordination code is polymorphic
over the selected service node and its sub-nodes, providing a degree
of genericity. Additionally, as exemplified by the service lookup of
phones, this service discovery construct is parameterized by values
for the attributes declared in the environment description (i.e., the
room attribute which is associated with the Phone service class by
inheritance from the service class Device).

Constructs connecting services are also parameterized by the
environment declarations. For example, when the virtual phone im-
plementation gets a call, it is required, according to the environment
declarations, to connect the incoming call to a phone – in our sce-
nario it corresponds to the phone where the secretary is located.
Accordingly, in the function callReception shown in Figure 7-b,
the invite construct can only be used with respect to audio sessions
(line 23).

As detailed in the next section, the tight coupling between the
environment description language and the coordination language en-
ables service discovery and composition to be checked statically by
the Pantaxou compiler, raising the safety level of such applications.

4. Verifications
An advantage of a domain-specific language is to make possible
domain-specific verifications by making properties apparent in the
program structure. In Pantaxou, certain critical operations, such as
interaction between entities, can only be implemented in terms
of built-in language abstractions, thus allowing verifications of
these operations that are difficult or impossible when using a more
flexible, general-purpose language such as Java. As Pantaxou covers
a range of aspects of the application life-cycle, ranging from the
development of the environment model to the execution of the
coordination service, verifications are performed at each level.

4.1 Verifying an environment model
The first verification performed by the Pantaxou compiler is to check
the consistency of the environment model. For an environment to be
consistent, every functionality provided must be required by at least
one service class and every functionality required must be provided
by at least one service class.

For this analysis, the environment model, eₙ, maps the name of
a service class into a set of functionalities that characterizes it.
Each functionality is described by the triple (d, m, idₙ) where d is
the direction (i.e., either Provides or Requires), m the interaction
mode (i.e., command, event, session), and idₙ is the identifier of the
remote service class. We write A ⊆ B to mean that A is a subclass
of B, where both A and B are either service classes or data types.
For commands, A ⊆ B if B and A are the same command, and for
sessions or events, A ⊆ B if the data type associated with A is a
subclass of the data type associated with B.

The property that every provided functionality should be required
by some service classes is formalized as follows:

∀idₙ ∈ dom(eₙ), ∀(Requires, m, idₙ) ∈ eₙ(idₙ),
∃idₙ’ ∈ dom(eₙ), ∃(Provides, m’, idₙ’),
 m ⊆ m’ ∧ idₙ’ ⊆ idₙ ∧ idₙ’ ⊆ idₙ. (1)

This property guarantees the connectedness of the model. For
instance, no event can be lost in a well-formed environment model.
This is an essential property of a critical event, such as a fire alarm.
For the command interaction mode this property corresponds to
checking for dead code.

Property (2) holds when every required functionality is provided
by at least one service class. This property ensures that every used
functionality is defined.

∀idₙ ∈ dom(eₙ), ∀(Requires, m, idₙ) ∈ eₙ(idₙ),
∃idₙ’ ∈ dom(eₙ), ∃(Provides, m’’, idₙ’’),
m’’ ⊆ m ∧ idₙ’’ ⊆ idₙ ∧ idₙ’’ ⊆ idₙ\(2\)

4.2 Verifying the type safety of a coordination service
For the definition of a coordination service, the Pantaxou compiler
checks that the use of the interaction modes corresponds to what is
declared in the environment model. Although the compiler checks
all three kinds of interaction modes, for conciseness, we only present
the analysis related to events.

The verifications are described within the static semantics of
Pantaxou. This semantics is defined using inference rules, having
a sequence of premises above a horizontal bar and a judgment
below the bar (see Equations (3) to (8)). In these rules, Γ represents
the environment model, consisting of the service classes Σ, the
commands Ξ and the data types Δ, μ is the set of interaction modes
defined for the service class of the service which is verified, and τ
represents the bindings of the variables of the service. The judgments
for statements additionally reference a type ρ, which is the expected
return type of the enclosing block. A statement that does not include
return has type Void.

The equations (3) and (4) describe the operations used by a
service to publish an event. Equation (3) describes the declara-
tion of an event, yielding an extended type environment τ |id →
ProvidedEventType[userTypeLabel]] that binds an identifier to the
event information. It first retrieves the event data type userTypePath
from the environment model Γ and then checks the properties de-
clared by the developer against the ones declared in the environment.
Finally, the rule checks that the event data type declared is a subtype
of an event data type that must be published by this service.

Γ ⊢ userTypePath : userType
userType = UserTypeError(userTypeLabel, τprop)
Γ, µ, τ, τprop ⊢ userTypeLabel, ProvidedEventType[userTypeLabel] ∈ μ
Γ = (τ |id → ProvidedEventType[userTypeLabel])

(3)

After the event declaration, the service may publish the correspond-
ing events. Equation (4) checks that the publish statement is only
used with an expression that evaluates to a publishable event, i.e.,
of type ProvidedEventType as defined by Equation (3).

Γ, µ, τ, ℓ ⊢ publish (exp) : Void

(4)

Equations (5) to (8) check the operations related to received
events. A service must take four steps to be able to receive an event:
(1) declare the kind of service class that publishes the requested
type of event (e.g., line 9 in Figure 7-a), (2) declare the desired
event type (e.g., line 11 in Figure 7-a), (3) define the handler to
execute when an event is received (e.g., lines 15-22 in Figure 7-a),
and (4) use adopt to select and activate one of these handlers (e.g.,
line 29 in Figure 7-a). The compiler checks at each of these steps
the consistency between the declarations, as well as between the
declarations and the environment model.

Equation (5) defines the semantics of a service declaration. It
first retrieves the information about the named service class from
the environment model, and then checks that the properties of the
When deploying a service, the framework checks that every function-
ality and service required is available in the system. To enable this
functionality is used by at least one service, and that every required
functionality is provided by at least one service. If one of these
operations leads to an inconsistent state of the system, services may
be deactivated or a human operator may be notified according to the
system’s policy and the criticity of the services involved.

5. Compilation

Figure 9 shows the main steps in the compilation of a Pantaxou
application. First, a Pantaxou application is passed to the Pantaxou
compiler, which (1) verifies the consistency of the model of the
networked environment and the conformance of the Pantaxou
service with this model, as described in Section 4, (2) transforms the
model into a specification expressed in the DiaSpec ADL [9, 10, 11],
which is dedicated to specifying the interactions that are allowed
within a distributed application and has been developed in parallel by
some of the authors, and (3) transforms the coordination logic into
a Java application. The DiaSpec specification is then passed to the
DiaSpec compiler, which generates a Java programming framework
on which to build distributed applications. This programming
framework is dedicated to the given environment model and provides
boilerplate functionalities, including service discovery and methods
for publishing typed events. The Java application generated by the
Pantaxou compiler relies on this dedicated programming framework.
Finally, a standard Java compiler is invoked to produce executable
code.

6. Related work

Following Arbab and Papadopoulos’ classification of coordination
languages, Pantaxou can be considered as an event-oriented lan-
guage [21], belonging to the language family that includes Mani-
fold [3], MICADO [6], STL [12] and COOL [4]. In contrast with
these languages, Pantaxou is designed to cope with entities that are
heterogeneous in terms of functionalities, capabilities and availabil-
ity. Furthermore, our language is not limited to a single interaction
mode but instead offers an interaction model consisting of com-
mands, events and sessions, covering a large spectrum of coordi-
nation activities. For example, our notion of session goes beyond
existing stream management mechanisms because it is integrated
into our programming paradigm. Finally, in contrast with other lan-
guages, Pantaxou covers the complete life-cycle of an application,
from the environment description to its runtime, including domain-
specific verifications.

CORBA component model (CCM) [18, 19] provides a higher-
level IDL than the one of CORBA, described in Section 2 (CORBA
IDLv2). CCM provides extensions to define components, events
and streams. However, the CCM IDL is based on the CORBA IDL
and thus inherits some of its low-level features. As a consequence,
more information must be made explicit in a CCM environment
model compared to Pantaxou (e.g., detailed, low-level declaration
of events and streams). In CCM, verification is limited to the de-
clarated environment model and mainly targets the correct interaction
between ports with respect to their types. In contrast, Pantaxou not only performs this verification at the environment level, but also at service implementation level. Finally, because CCM, like CORBA, is language independent, the implementation of components is out of scope. As a consequence, it cannot provide language support for key activities in the distributed setting, such as service discovery. Unlike CCM, Pantaxou consists of two tightly-coupled languages for hierarchical environment description and coordination logic that provide guidance and high-level support for programming services and raise the level of safety of the resulting applications.

Architecture description languages (ADLs) provide a language to define an architecture, whether or not distributed. However, an ADL is often viewed as a specification language, defining architectures generically, without targeting a specific domain. Verification tools have been developed to check ADL specifications, often targeting deadlock detection [2]. Traditionally, ADLs are independent of a given programming language [16], although this decoupling has recently been challenged by ArchJava: an ADL that extends Java with architecture declarations [1].

Pantaxou differs from ADLs in that it tightly integrates two language layers: one to describe a distributed environment and another one to program the coordination logic. The environment description language is an ADL dedicated to distributed environments, which are not addressed by ArchJava. Because our ADL is dedicated, descriptions can be verified more thoroughly, enabling for instance connector protocols to be checked (i.e., the implementation of a service declared as consuming an event is required to adopt/subscribe to the appropriate event). Because we target distributed environments, the programmer is provided with language support for service discovery and compiler support to manage distributed and networking aspects.

A lightweight ADL for distributed applications, named Dia-Spec, has been proposed by Jouve et al. [9] to describe entities of a distributed system. A DiaSpec description is passed to a generator that produces a dedicated framework, facilitating the development of an application written in Java. The layer of Pantaxou dedicated to describing environments is similar to DiaSpec and is compiled into it. The key difference is the fact that Pantaxou covers the entire lifecycle of an application, including the coordination logic, enabling more properties to be checked. Specifically, Pantaxou provides syntactic constructs that guarantee the correctness of coordination. For example, calls are guaranteed to be appropriately forwarded thanks to the \texttt{invite} construct.

DSLs for ubiquitous computing include YABS [5] and AmbientTalk [8]. YABS follows a tuple-based paradigm, whereas AmbientTalk is control-based like Pantaxou. Both YABS and AmbientTalk make assumptions regarding the kind of distributed environments targeted that are different from Pantaxou’s. For example, they assume a peer-to-peer model with an unreliable network, including nomadic scenarios. In contrast, we focus on a centralized, reliable network as can be found in homes, buildings and campuses. Also we propose an interaction model based on a client-server model, building on widespread existing technologies. Furthermore, Pantaxou offers a domain-specific architecture description language and extended verifications.

7. Assessment and Conclusions

In this paper, we have presented Pantaxou, a domain-specific language for developing coordination services for distributed networked environments. The key advantage of Pantaxou is the high level at which the coordination code can express the needed interactions with services, according to various widely used interaction modes. Another benefit of Pantaxou is the verifications of these interactions and the overall consistency of the environment provided by the compiler and the generated framework.

Pantaxou has proved to be easy to use, as the secretary scenario developed here was implemented in a few hours by a group of graduate students who had been presented the language and the underlying development methodology. They also implemented a comparable application directly in Java, for comparison. The coordination logic written in Java was about 5 times longer than the Pantaxou implementation. In fact, what struck them most was the conciseness of Pantaxou programs, making concrete for them the notion of high-level abstraction. They also praised the safety of Pantaxou, as compared to the low-level debugging they had to perform in Java. Finally, they appreciated the seamless integration of service discovery in Pantaxou that facilitated the overall programming process.

In addition to the secretary scenario, we have used Pantaxou to implement scenarios in a number of areas of building management. The language has thus shown itself to be usable for a wide range of tasks.

We are now exploring how to express some non-functional aspects such as security and QoS in Pantaxou. These aspects open up opportunities for new verifications at both the environment model level and the coordination logic level. Another research perspective is to embed Pantaxou coordination logic constructs in Java, which will provide general-purpose computing, while preserving the safety provided by our verifications.

References


