Pantaxou: a Domain-Specific Language for Developing Safe Coordination Services

Julien Mercadal, Nicolas Palix, Charles Consel, Julia Lawall

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 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.

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.

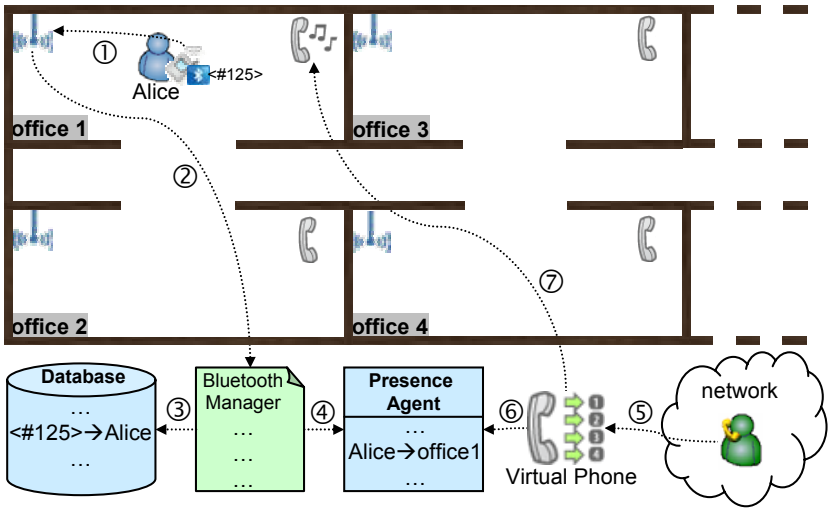
Complex interaction modes The implementation of a distributed, heterogeneous environment is typically based on a middleware that manages the interaction between a coordination service and other networked entities. Middleware typically provides some form of an interface definition language that abstracts 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 inter-

GPCE2008 October 19-23, 2008, Nashville, Tennessee, USA



1. The Bluetooth detector detects that a Bluetooth device has just entered.
2. The Bluetooth detector publishes an event to signal the presence of the Bluetooth device. This event is received by the Bluetooth Manager.
3. The Bluetooth Manager queries the database to identify the owner of the tag $\langle \#125 \rangle$.
4. The Bluetooth Manager publishes an event to signal that Alice is in office 1. This event is received by the Presence Agent.
5. When Alice is called, an audio session request is received by her Virtual Phone.
6. The Virtual Phone queries the Presence Agent to locate Alice.
7. The Virtual Phone transfers the call to the phone of office 1.

Figure 1. Secretary scenario

actions. 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,[1] 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

[1] Pantaxou means "everywhere" in Greek.

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.

```
1  #include <omg/CosEventComm.idl>
2  ...
3  interface GetRoom {
4    room getRoom(in uri user);
5  };
6  interface PresencePusher {
7    void push_presence(in uri entity, in boolean status,
8      in room r);
9  };
10 interface PresenceAgent : PresencePusher,
11   CosEventComm::PushConsumer, GetRoom { };
12 interface Phone : Device, MMDevice {
13   Phone_A create_A(in Phone_StreamCtrl the_requester,
14                    out v_Phone the_vdev,
15                    inout streamQoS the_qos,
16                    out boolean met_qos,
17                    inout string named_vdev,
18                    in flowSpec the_spec)
19     raises(streamOpFailed, streamOpDenied, notSupported,
20            QoSRequestFailed, noSuchFlow);
21
22   Phone_B create_B(...)  // similar to Phone_A
23     raises(streamOpFailed, streamOpDenied, notSupported,
24            QoSRequestFailed, noSuchFlow);
25
26   Phone_StreamCtrl bind(in Phone peer_device,
27                         inout streamQoS the_qos,
28                         out boolean is_met,
29                         in flowSpec the_spec)
30     raises (streamOpFailed, noSuchFlow, QoSRequestFailed);
31
32   Phone_StreamCtrl bind_mcast(...)  // similar to bind
33     raises (streamOpFailed, noSuchFlow, QoSRequestFailed);
34
35   string add_fdev(in F_Audio the_fdev)
36     raises(notSupported, streamOpFailed);
37 };
38 interface v_Phone : VDev { ... };
39 interface Phone_StreamCtrl : StreamCtrl { ... };
40 interface Phone_A : StreamEndPoint_A { ... };
41 interface Phone_B : StreamEndPoint_B {};
42
43 interface Audio_Producer : FlowProducer {};
44 interface Audio_Consumer : FlowConsumer { ... };
45 interface F_Audio : FDev { ... };
46 interface Audio_Connection : FlowConnection { ... };
47  ...
```

Figure 2. Fragments of a CORBA-based interface declarations of the secretary scenario

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

```
1  import ...
2  class PresenceAgent extends PresenceAgentPOA {
3    ...
4    public PresenceAgent(...) {
5      ...
6      ORB orb = ORB.init(...);
7      Object obj = orb.resolve_initial_references("EventService");
8      TypedEventChannelFactory m_factory = TypedEventChannelFactoryHelper.narrow(obj);
9      IntHolder id = new IntHolder();
10     TypedEventChannel tec = m_factory.create_typed_channel("TypedChannel", id);
11     org.omg.CosTypedEventChannelAdmin.TypedConsumerAdmin tca = tec.for_consumers();
12     org.omg.CosEventChannelAdmin.ProxyPushSupplier pps = tca.obtain_typed_push_supplier("IDL:PresencePusher:1.0");
13     pps.connect_push_consumer(this);
14   }
15   Room getRoom(Uri user) { ... }
16   void disconnect_push_consumer() { }
17   void push_presence(Uri entity, boolean status, Room r) {
18     ... // communication logic
19   }
20   void push(org.omg.CORBA.Any a) { }
21 }
```

Figure 3. Fragments of a CORBA-based implementation of the secretary scenario (PresenceAgent class). For simplicity, exception handling has been omitted.

receive event notifications. Finally, the methods `push_presence` and `push` provide callback functions for handling received events. The `push_presence` 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 networked environment requires knowledge of the entities to be coordinated, 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 doing 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 devices 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 operations, 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 abstractions 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 various 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 subscribed 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 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 lifecycle 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

```
domain            ::= import* datatype* commandInterface* service*
import            ::= import identifier;
datatype          ::= struct identifier (extends identifier)? { property* }
                    | enum identifier = { identifier (, identifier)+ }
commandInterface  ::= command identifier { method* }
service           ::= service identifier
                        (extends identifier)? { property* functionality* }
method            ::= type identifier ((type identifier (, type identifier)*)?);
property          ::= type identifier;
functionality     ::= requires interactionMode from identifier;
                    | provides interactionMode to identifier;
                    | bind<functionality, functionality>;
interactionMode   ::= command<identifier> | event<identifier>
                    | session<(?|!|=)identifier>
type              ::= identifier | primitiveType
primitiveType     ::= bool | int | string | uri | void
```

Figure 4. Grammar of the Pantaxou environment language layer

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.

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

```
1  import Audio;
2
3  struct Room {
4    int number;
5  }
6  struct Presence {
7    uri entity;
8    bool status;
9    Room room;
10 }
11 struct DbBluetoothInfo {
12   uri ownerName;
13 }
14 struct BluetoothDetection {
15   string bluetoothAddress;
16   int signalStrengh;
17 }
18 command DbQueryBluetooth {
19   DbBluetoothInfo
20     getInfoBT
21       (string bluetoothAddress);
22 }
23 command GetRoom {
24   Room getRoom(uri user);
25 }
```

Figure 5. Data types and signatures for the secretary scenario

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 `BluetoothPresenceManager` (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.

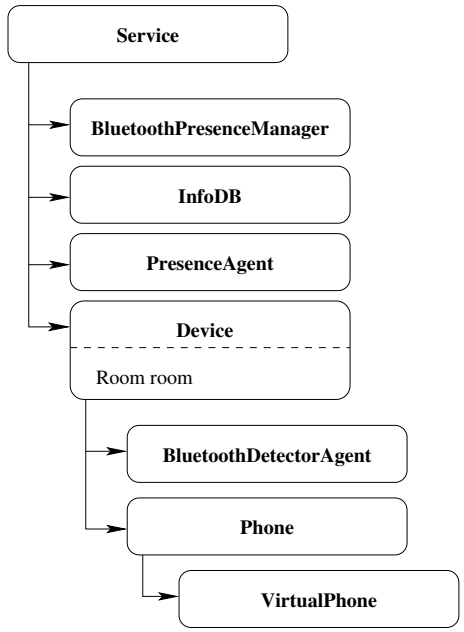
The hierarchical definition of service classes allows subclasses to inherit the attributes and interaction modes of their ancestors. This approach makes it possible to characterize classes of entities by incremental refinement. For example, in our advanced telecommunication environment, we introduce a virtual phone that behaves like a phone (its ancestor) but requires a command to get the current 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.

```
service Service {}
service BluetoothPresenceManager extends Service {
  requires event<BluetoothDetection>
    from BluetoothDetectorAgent;
  requires command<DbQueryBluetooth> from InfoDB;
  provides event<Presence> to PresenceAgent;
}
service InfoDB extends Service {
  provides command<DbQueryBluetooth>
    to BluetoothPresenceManager;
}
service PresenceAgent extends Service {
  requires event<Presence>
    from BluetoothPresenceManager;
  provides command<GetRoom> to Service;
}
service Device extends Service {
  Room room;
}
service BluetoothDetectorAgent extends Device {
  provides event<BluetoothDetection>
    to BluetoothPresenceManager;
}
service Phone extends Device {
  provides session<=Audio> to Phone;
  requires session<=Audio> from Phone;
}
service VirtualPhone extends Phone {
  requires command<GetRoom> from PresenceAgent;
}
```

(a) Service declarations



(b) Hierarchy of services

Figure 6. Model of the advanced telecommunication environment

```
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

```
import datatype Audio;
import datatype Room;
import command GetRoom;
import service Service.PresenceAgent;
import service Service.Device.Phone;

'sip:follow.me@enseirb.fr'
  instantiates
    Service.Device.Phone.VirtualPhone {

  service<Phone> { } phone;
  service<PresenceAgent> { } pa;

  session<Audio> from phone[*] { } inSess;

  var pa agent = new pa[1];

  onReceive callReception(inSess s) {
    service<Phone> {
     room =
       agent.getRoom('sip:me@enseirb.fr');
    } myPhone;
    invite(new myPhone[1], s) {
      accepted(activeSession<Audio> aas) {
        // An active audio session (aas)
        // is accessible.
      }
      rejected() {
        reject(s); // Forward the reject.
      }
    }
  }

  initial { adopt(callReception); }
}
```

(b) Call forwarding service

Figure 7. Secretary scenario implemented in Pantaxou (part 1)

```
1  import datatype BluetoothDetection;
2  import datatype DbBluetoothInfo;
3  import datatype Presence;
4  import datatype Room;
5  import command DbQueryBluetooth;
6  import service Service.InfoDB;
7  import service
8            Service.BlueToothPresenceManager;
9  'sip:bt.manager@enseirb.fr'
10   instantiates
11     Service.BluetoothPresenceManager {
12
13   service<BluetoothDetectorAgent> { }
14     bluetoothDetectorAgent;
15   service<InfoDB> { } database;
16
17   event<BluetoothDetection>
18     from bluetoothDetectorAgent[*] {
19     signalStrengh > 50;
20   } btDetectionEvt;
21
22   var database db = new database[1];
23
24   onReceive btEvtReception(btDetectionEvt e)
25   {
26     DbBluetoothInfo i =
27       db.getInfoBT(e.data.bluetoothAddress);
28     event<Presence> {
29       entity = i.ownerName;
30       room = e.src.room;
31     } p;
32     publish(new p);
33   }
34
35   initial { adopt(btEvtReception); }
36 }
```

Presence manager service

Figure 8. Secretary scenario implemented in Pantaxou (part 2)

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 `Service.PresenceAgent` service class (lines 6-7). The declaration of the `Service.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*`>{ ... }`

When implementing a virtual phone, this service discovery construct can only refer to the three possible service classes: `Phone`, `VirtualPhone` (its sub-class) and `PresenceAgent`. This is illustrated 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 implementation gets a call, it is required, according to the environment declarations, to connect the incoming call to a phone – in our scenario 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 enables 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, e_s, maps the name of a service class into a set of functionalities that characterizes it. Each functionality is described by the triple (d, m, id) where d is the direction (*i.e.*, either Provides or Requires), m the interaction mode (*i.e.*, command, event, session), and id is the identifier of the remote service class. We write $A \sqsubseteq B$ to mean that A is a subclass of B, where both A and B are either service classes or data types. For commands, $A \sqsubseteq B$ if A and B are the same command, and for events or sessions, $A \sqsubseteq B$ 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:

$$\begin{array}{l}\forall id_s \in \mathsf{dom}(e_s),\ \forall(\mathsf{Provides}, m, id_t) \in e_s(id_s),\\ \quad \exists id_{t'} \in \mathsf{dom}(e_s),\ \exists(\mathsf{Requires}, m', id_{s'}) \in e_s(id_{t'}),\\ \quad\quad m \sqsubseteq m' \wedge id_{t'} \sqsubseteq id_t \wedge id_{s'} \sqsubseteq id_s\end{array} \quad (1)$$

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.

$$\begin{array}{l}\forall id_s \in \mathsf{dom}(e_s),\ \forall(\mathsf{Requires}, m, id_t) \in e_s(id_s),\\ \quad \exists id_{t'} \in \mathsf{dom}(e_s),\ \exists(\mathsf{Provides}, m', id_{s'}) \in e_s(id_{t'}),\\ \quad\quad m' \sqsubseteq m \wedge id_t \sqsubseteq id_{t'} \wedge id_s \sqsubseteq id_{s'}\end{array} \quad (2)$$

Note that the connectedness property cannot be verified at the interface level when using CORBA, because CORBA does not provide a means for an interface to describe the resources it requires.

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 t, 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 declaration of an event, yielding an extended type environment $\tau[id \mapsto \mathsf{ProvidedEventType}(\mathit{userTypeLabel})]$ 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 declared 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.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{userTypePath} : \mathit{userType}\\ \mathit{userType} = \mathbf{UserType}(\mathit{userTypeLabel}, \tau_{\mathsf{prop}})\\ \Gamma,\ \mu,\ \tau, \tau_{\mathsf{prop}} \vdash \mathit{properties}\\ \exists \mathit{userTypeLabel}',\ \mathsf{ProvidedEventType}(\mathit{userTypeLabel}') \in \mu\\ \Gamma = (\_, \_, \Delta) \qquad \Delta(\mathit{userTypeLabel}) \sqsubseteq \Delta(\mathit{userTypeLabel}')\end{array}}{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash \texttt{event<}\mathit{userTypePath}\texttt{>}\ \texttt{\{}\mathit{properties}\texttt{\}}\ \mathit{id};\\ : \tau[\mathit{id} \mapsto \mathsf{ProvidedEventType}(\mathit{userTypeLabel})]\end{array}} \quad (3)$$

After the event declaration, the service may publish the corresponding 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).

$$\frac{\Gamma,\ \mu,\ \tau \vdash \mathit{exp} : \mathsf{ProvidedEventType}(\_)}{\Gamma,\ \mu,\ \tau,\ t \vdash \texttt{publish (}\mathit{exp}\texttt{);} : \mathsf{Void}} \quad (4)$$

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 service class, τ_{prop}, are compatible with the properties, *properties*, requested by the developer. Finally, an extended type environment is returned that binds *id* to the service information.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{servicePath} : t \qquad t = \mathbf{ServiceType}(\_, \_, \tau_{\mathsf{prop}})\\ \Gamma,\ \mu,\ \tau, \tau_{\mathsf{prop}} \vdash \mathit{properties}\end{array}}{\Gamma,\ \mu,\ \tau \vdash \texttt{service<}\mathit{servicePath}\texttt{>}\ \texttt{\{}\mathit{properties}\texttt{\}}\ \mathit{id}; : \tau[\mathit{id} \mapsto t]} \quad (5)$$

The second step is to declare the event to which the service wants to subscribe. This declaration indicates from which service classes (*service*) the event must be received and from how many services (*i*). Equation (6) checks that the subscribed service class publishes

events that have a subtype of the event requested, and that the type of the event requested is a subtype of an event required by the service evaluated.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{userTypePath} : \mathit{userType} \\ \mathit{userType} = \mathbf{UserType}(\mathit{userTypeLabel}_1, \tau_{\mathsf{prop}}) \\ \Gamma,\ \mu,\ \tau, \tau_{\mathsf{prop}} \vdash \mathit{properties} \qquad \tau(\mathit{service}) = \mathbf{ServiceType}(\_, \mu', \_) \\ \exists \mathit{userTypeLabel}_2,\ \mathsf{ProvidedEventType}(\mathit{userTypeLabel}_2) \in \mu' \\ \exists \mathit{userTypeLabel}_3,\ \mathsf{RequiredEventType}(\mathit{userTypeLabel}_3) \in \mu \\ \Gamma = (\_, \_, \Delta) \\ \Delta(\mathit{userTypeLabel}_2) \sqsubseteq \Delta(\mathit{userTypeLabel}_1) \sqsubseteq \Delta(\mathit{userTypeLabel}_3)\end{array}}{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash \texttt{event<}\mathit{userTypePath}\texttt{> from } \mathit{service}\texttt{[}i\texttt{] \{}\mathit{properties}\texttt{\} } \mathit{id}\texttt{;} \\ : \tau[\mathit{id} \mapsto \mathsf{RequiredEventType}(\mathit{userTypeLabel})]\end{array}} \quad (6)$$

The next step is to define the callback handler that is executed when an event is received. Equation (7) retrieves the event declaration used and checks its type, *i.e.*, that it is a RequiredEventType event declaration. The event type is then used in evaluating the handler compound.

$$\frac{\begin{array}{c}\tau(imType) = \mathsf{RequiredEventType}(\_) \\ \Gamma,\ \mu,\ \tau[\mathit{im} \mapsto \tau(imType)],\ \mathsf{Void} \vdash \mathit{cmpd} : \mathsf{Void}\end{array}}{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash \texttt{onReceive } \mathit{id}\ \texttt{(}\mathit{imType}\ \mathit{im}\texttt{)}\ \mathit{cmpd} \\ : \tau[\mathit{id} \mapsto \mathsf{EventReceptionType}(\mathit{imType})]\end{array}} \quad (7)$$

Finally, the handler that should be responsible for a given kind of event is selected by an `adopt` statement. This statement is also used to select the handler for an incoming session. Equation (8) checks that the expression used refers to a reception declaration, *i.e.*, either a EventReceptionType or SessionReceptionType.

$$\frac{\Gamma,\ \mu,\ \tau \vdash \mathit{exp} : \mathsf{EventReceptionType}(\_) + \mathsf{SessionReceptionType}(\_)}{\Gamma,\ \mu,\ \tau,\ t \vdash \texttt{adopt(}\mathit{exp}\texttt{);} : \mathsf{Void}} \quad (8)$$

4.3 Verifying a target environment

The above verifications concern static properties, and are performed by the compiler. We now present the verifications of dynamic properties performed by the underlying framework.

4.3.1 At the service level

When deploying a service, the framework checks that every functionality and service required is available in the system. To enable this verification, the Pantaxou compiler extracts the set of dependencies of a service from the application source code. At deployment time, the framework then checks that the service classes that are indicated as required are available in the environment. Attribute values used in service discovery, however, are dynamically chosen, and thus not taken into account in this analysis. They are instead taken into account when service discovery is requested. Still, the basic feasibility of the possible service discovery requests is ensured.

4.3.2 At the system level

To ensure the preservation of the consistency of the system, the framework checks at every operation in the life-cycle of a service (*e.g.*, registration, subscription and removal) that every provided 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.

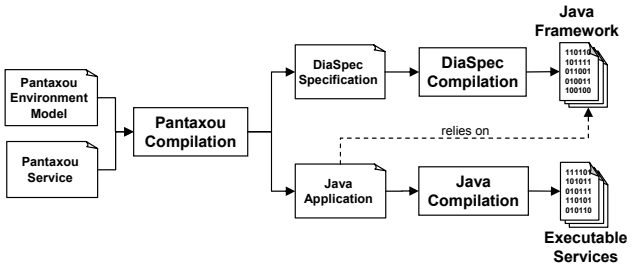


Figure 9. Complete compilation process

6. Related work

Following Arbab and Papadopoulos' classification of coordination languages, Pantaxou can be considered as an event-oriented language [21], belonging to the language family that includes Manifold [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 availability. Furthermore, our language is not limited to a single interaction mode but instead offers an interaction model consisting of commands, events and sessions, covering a large spectrum of coordination 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 languages, 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 declared 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 DiaSpec, 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 `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

[1] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, Orlando, FL, USA, 2002. ACM.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[3] F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.

[4] M. Barbuceanu and M. S. Fox. Cool: A language for describing coordination in multiagent systems. In V. Lesser and L. Gasser, editors, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 17–24, San Francisco, CA, USA, 1995. AAAI Press.

[5] P. Barron and V. Cahill. YABS: a domain-specific language for pervasive computing based on stigmergy. In *GPCE'06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 285–294, Portland, OR, USA, 2006. ACM.

[6] L. Berger, A.-M. Dery, and M. Blay-Fornarino. Interactions between objects: An aspect of object-oriented languages. In *ECOOP '98: Workshop on Object-Oriented Technology*, pages 422–423, Brussels, Belgium, 1998. Springer-Verlag.

[7] BlueCove group, http://code.google.com/p/bluecove/. *BlueCove: Java library for Bluetooth.*

[8] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented Programming in AmbientTalk. *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)*, pages 230–254, 2006.

[9] W. Jouve, D. Cassou, J. Mercadal, C. Consel, and J. Lawall. Architecturing distributed applications. Technical report, INRIA/LaBRI, 2008.

[10] W. Jouve, J. Lancia, N. Palix, C. Consel, and J. Lawall. High-level programming support for robust pervasive computing applications. In *Proceedings of the 6th IEEE Conference on Pervasive Computing and Communications (PERCOM'08)*, pages 252–255, Hong Kong, China, mar 2008.

[11] W. Jouve, N. Palix, C. Consel, and P. Kadionik. A SIP-based programming framework for advanced telephony applications. In *The 2nd Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm'08)*, page 11, Heidelberg, Germany, jul 2008.

[12] O. Krone, F. Chantemargue, T. Dagaeff, and M. Schumacher. Coordinating autonomous entities with STL. *SIGAPP Appl. Comput. Rev.*, 6(2):18–32, 1998.

[13] M. Kumar, B. A. Shirazi, S. K. Das, B. Y. Sung, D. Levine, and M. Singhal. PICO: A middleware framework for pervasive computing. *IEEE Pervasive Computing*, 02(3):72–79, 2003.

[14] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *PERCOM'04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*, page 263, Orlando, FL, USA, 2004. IEEE Computer Society.

[15] C. Mascolo, S. Hailes, L. Lymberopoulos, G. P. Picco, P. Costa, G. Blair, P. Okanda, T. Sivaharan, W. Fritsche, M. Karl, M. A. Rónai, K. Fodor, and A. Boulis. Survey of middleware for networked embedded systems. Technical report, FP6 IP "RUNES", 2005.

[16] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A language and environment for architecture-based software development and evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 44–53, Los Angeles, California, United States, 1999. IEEE Computer Society Press.

[17] Object Management Group (OMG), Framingham, USA. *Audio/Video Stream Specification*, Jan 2000.

[18] Object Management Group (OMG). *Streams for CCM Specification*, jul 2005.

[19] Object Management Group (OMG). *CORBA Component Model Specification*, apr 2006.

[20] OMG. *CORBA: The Common Object Request Broker: Architecture and Specification*. Object Management Group, 1995.

[21] G. Papadopoulos and F. Arbab. Coordination models and languages. Technical Report SEN-R9834, Centrum voor Wiskunde en Informatica, Dec. 1998.

[22] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas. Olympus: A high-level programming model for pervasive computing environments. In *PERCOM'05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, pages 7–16, Kauai, Hawaii, 2005. IEEE Computer Society.

[23] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: a middleware platform for active spaces. *SIGMOBILE Mobile Computing and Communications Review*, 6(4):65–67, 2002.

[24] Sun Microsystem. RPC: Remote procedure call protocol specification, version 2. Technical report, Sun Microsystem, 1988.

[25] Sun Microsystems, Inc, http://java.sun.com/javase/technologies/core/basic/rmi/. *Java Remote Method Invocation*.

[26] F. Zhu, M. Mutka, and L. Ni. Service discovery in pervasive computing environments. *Pervasive Computing, IEEE*, 4(4):81–90, Oct.-Dec. 2005.