# Towards an architecture for runtime interoperability

Amel Bennaceur, Gordon Blair, Franck Chauvel, Nikolaos Georgantas, Paul Grace, Falk Howar, Paola Inverardi, Valérie Issarny, Massimo Paolucci, Animesh Pathak, et al.

# Towards an Architecture for Runtime Interoperability

Amel Bennaceur[1], Gordon Blair[2], Franck Chauvel[3], Huang Gang[3], Nikolaos Georgantas[1], Paul Grace[2], Falk Howar[4], Paola Inverardi[5], Valérie Issarny[1], Massimo Paolucci[6], Animesh Pathak[1], Romina Spalazzese[5], Bernhard Steffen[4], and Bertrand Souville[6]

[1] INRIA, CRI Paris-Rocquencourt, France
[2] Lancaster University, UK
[3] School of Electronics Engineering and Computer Science, Peking University, China
[4] Technische Universitat Dortmund, Germany
[5] Universit degli Studi dell'Aquilà, Italy
[6] DOCOMO Euro-Labs, Munich, Germany

**Abstract.** Interoperability remains a fundamental challenge when connecting heterogeneous systems which encounter and spontaneously communicate with one another in pervasive computing environments. This challenge is exasperated by the highly heterogeneous technologies employed by each of the interacting parties, i.e., in terms of hardware, operating system, middleware protocols, and application protocols. This paper introduces CONNECT, a software framework which aims to resolve this interoperability challenge in a fundamentally different way. CONNECT dynamically discovers information about the running systems, uses learning to build a richer view of a system's behaviour and then uses synthesis techniques to generate a connector to achieve interoperability between heterogeneous systems. Here, we introduce the key elements of CONNECT and describe its application to a distributed marketplace application involving heterogeneous technologies.

## 1 Introduction

A fundamental requirement of distributed systems is to ensure interoperability between the communicating elements; systems that have been implemented independently of one another must be able to connect, understand and exchange data with one another. This is particularly true in highly dynamic application domains (e.g. mobile and pervasive computing) where systems typically only encounter one another at runtime. Middleware technologies have traditionally resolved many of the interoperability problems arising in these situations, such as operating system and programming language heterogeneity. Where two applications conform to a particular middleware standard, e.g. CORBA [12] and Web Services [3] [5], they are guaranteed to interoperate. However, the next generation of distributed computing applications are characterized by two important properties that force a rethink of how interoperability problems should be tackled:

T. Margaria and B. Steffen (Eds.): ISoLA 2010, Part II, LNCS 6416, pp. 206–220, 2010.

- *Extreme heterogeneity.* Complex pervasive systems are composed of technology dependent islands, i.e. domain specific systems that employ heterogeneous communication and middleware protocols. For example, Grid applications, mobile ad-hoc networks, Enterprise systems, and sensor networks all use their own protocols such that they cannot interoperate with one another.
- *Spontaneous Communication.* Connections between systems are not made until runtime (and are made between systems that were not aware of one another beforehand).

With such characteristics, requiring all applications to be developed upon a common middleware technology, e.g. CORBA or Web Services, is unsuitable in practice. Rather, new approaches are required that allow systems developed upon heterogeneous technologies to interoperate with one another at runtime. In this paper, we present the CONNECT[1] architectural framework that aims to resolve this interoperability challenge in a fundamentally different way. Rather than create a middleware solution that is destined to be yet another legacy platform that adds to the interoperability problem, we propose the novel approach of *generating the required middleware at runtime* i.e. we synthesize the necessary software to connect two end-systems. For example, if a client application developed using SOAP [13] encounters a CORBA server then the framework generates a CONNECTor that resolves the heterogeneity of the i) data exchanged, ii) application behaviour e.g. sequence of operations called, and iii) the lower level middleware and network communication protocols. In this paper we identify the requirements that need to be satisfied to guarantee interoperability, namely interoperability at the discovery, behavioral and data level. We then outline the key elements of the CONNECT framework that underpin a runtime solution to achieving such interoperability, and that are further detailed in the companion papers [14][19][2][1][10]:

- *Discovering the functionality* of networked systems and applications advertised by legacy discovery protocols e.g. Service Location Protocol (SLP) and Simple Service Discovery Protocol (SSDP). Then, transforming this to a rich intermediary description used to syntactically and semantically match heterogeneous services.
- *Using learning algorithms* to dynamically determine the interaction behaviour of a networked system from its intermediary representation and producing a model of this behaviour in the form of a labelled transition system (LTS) [14].
- *Dynamically synthesising* a software mediator using code generation techniques (from the independent LTS models of each system) that will connect and coordinate the interoperability between heterogeneous end systems [19][2].

We highlight the potential of this CONNECT framework to achieve interoperability within a case study (a distributed marketplace application) that exhibits

[1] http://connect-forever.eu/

high levels of heterogeneity. Further exploration of maintaining dependability requirements when connecting systems is provided in [10]; while further information regarding the underlying formal theory of the CONNECT framework is found in [1].

The remainder of the paper is structured as follows. In Section 2 we highlight the interoperability challenges and requirements within a distributed marketplace application. We then examine the state of the art in interoperability solutions in Section 3 to highlight their deficiencies compared to the requirements. In Section 4 we present an overview of the CONNECT architecture and its underlying principles. In section 5 we describe how CONNECT resolves interoperability within the marketplace case study, and finally in section 6 we draw conclusions and identify a roadmap for future research in this field.

## 2 Motivating Scenario: The Distributed Marketplace

Consider a stadium where fans from various countries have gathered together to watch a game. The specific application we focus on in this section is that of a distributed marketplace. Here, **merchants** publicise their wares, and **consumers** can search the market, and order from a merchant. Both merchants and consumers use mobile devices with wireless networks deployed in the stadium. Merchants publish product info which the consumers can browse through. When a consumer requests a product, the merchant gets a notification of the amount ordered and the location of the consumer, to which he can respond with a yes/no. If yes, then when he is close enough to the consumer, both of them get a proximity notification by means of their mobile device (ring/buzz).

**Table 1.** Potential Implementations of Consumers and Merchants

| Country | Discovery | Middleware | Application Data | Currency |
|---|---|---|---|---|
| Germany | SLP | Tuple Space | GetInfo | EUR |
| U.K. | SLP | SOAP | GetInfo | GBP |
| France | SSDP | SOAP | GetInfo | EUR |
| Italy | SSDP | SOAP | GetLocation+ GetPrice+ GetQuantity | EUR |
| Spain | SLP | SOAP | GetInfo | EUR |

Table 1 highlights how stadiums from different countries implement the application using heterogeneous technology. Importantly, if a client from one country attempts to dynamically interoperate with a merchant in a different country it will fail in each case. We now examine the dimensions of heterogeneity which explain why such interoperation fails:

1. **Heterogeneous discovery protocols** are used by the consumer to locate a merchant, and by the merchant to advertise his services. In Table 1, SLP and SSDP are employed; in situations where the consumer and merchant differ in this aspect, the two will be unable to discover one another and the first step fails.

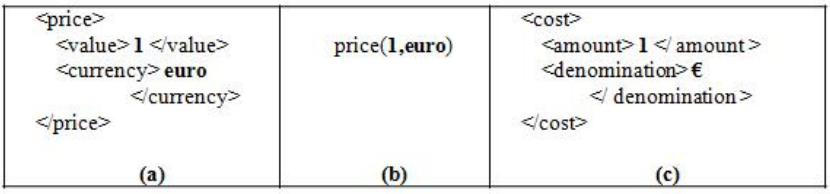


**Fig. 1.** Representing price in a) XML, b) tuple data, and c) heterogeneous XML

2. Consumers and merchants use **heterogeneous middleware protocols** to implement their functional interactions. In Table 1, a tuple space middleware and the SOAP RPC protocol [13] are used; these are different communication paradigms: the tuple space follows a shared space abstraction to write tuples to and read from, whereas RPC is a synchronous invocation of a remote operation. Hence, the two cannot interoperate directly.
3. **Application level heterogeneity.** Interoperability challenges at the application level arise due to the different ways that application developers implement the functionality. As a specific example, we assume that the merchant implements methods for the consumer to obtain information about his wares in one of two ways this would lead to different sequences of messages between the consumer and merchant: A single `GetInfo()` remote call, or three separate remote calls: `GetLocation()`, `GetPrice()`, and `GetQuantity()`.
4. **Data-representation Heterogeneity.** Implementations may represent data differently. Data representation heterogeneity is typically manifested at two levels. The simplest form of data interoperability is at the syntactic level where two different systems may use very different formats to express the same information. The French system may represent the price of the merchant's product using XML (Figure 1a), while the German tuple space may serialize a Java Object (Figure 1b). Further, even if two systems share a common language to express data, different dialects may still raise interoperability issues. Consider Figure 1c (the Spanish system) against Figure 1a; they (intuitively) carry the same meaning. Any system that recognizes the first structure will also be able to parse the second one, but it will fail to recognize the similarity between them unless it realizes that *price* $\equiv$ *cost*, that *value* $\equiv$ *amount*, that *currency* $\equiv$ *denomination* (where $\equiv$ denotes equivalence). The deeper problem of data heterogeneity is the semantic interoperability whereby all systems should have the same interpretation of data.

**Summary of requirements.** This scenario illustrates four dimensions where systems may be heterogeneous: i) the discovery protocol, ii) the interaction protocol, iii) application behaviour, and iv) data representation and meaning. A universal interoperability solution must consider all four in order to achieve interoperability.

## 3 Beyond State of the Art Interoperability Solutions

Achieving interoperability between independently developed systems has been one of the fundamental goals of middleware researchers and developers; and prior efforts have largely concentrated on solutions where conformance to one or other standard is required e.g. as illustrated by the significant standards work produced by the OMG for CORBA middleware [12], and by the W3C for Web Services based middleware [3][5]. These attempt to make the world conform to a common standard, and this approach has been effective in many areas e.g. routing of network messages in the Internet. To some extent CORBA and Web Services have been successful in connecting systems in Enterprise applications to handle hardware platform, operating system and programming language heterogeneity. However, in the more general sense of achieving universal interoperability and dynamic interoperability between spontaneous communicating systems they have failed. Within the field of distributed software systems, any approach that assumes a common middleware or standard is destined to fail due to the following reasons:

- A one size fits all standard/middleware cannot cope with the extreme heterogeneity of distributed systems e.g. from small scale sensor applications through to large scale Internet applications.
- New distributed systems and application emerge fast, while standards development is a slow, incremental process. Hence, it is likely that new technologies will appear that will make a pre-existing interoperability standard obsolete, c.f. CORBA versus Web Services (neither can talk to the other).
- Legacy platforms remain useful. Indeed, CORBA applications remain widely in use today. However, new standards do not typically embrace this legacy issue; this in turn leads to immediate interoperability problems.

One approach to resolving the heterogeneity of middleware solutions comes in the form of interoperability platforms. ReMMoC [11], Universal Interoperable Core [22] and WSIF [6] are client side middleware which employ similar patterns to increase interoperability with heterogeneous service side protocols. First, the interoperability platform presents an API for developing applications with. Secondly, it provides a substitution mechanism where the implementation of the protocol to be translated to, is deployed locally by the middleware to allow communication directly with the legacy peers (which are simply legacy applications and their middleware). Thirdly, the API calls are translated to the substituted middleware protocol. For the particular use case, where you want a client application to interoperate with everyone else, interoperability platforms are a powerful approach. However, these solutions rely upon a design time choice to develop upon the interoperability platforms. Therefore, they are unsuited to other interoperability cases e.g. when two applications developed upon different legacy middleware want to interoperate spontaneously at runtime.

Software bridges offer another interoperability solution to enable communication between different middleware environments. Clients in one middleware

domain can interoperate with servers in another middleware domain where the bridge acts as a one-to-one mapping between domains; it will take messages from a client in one format and then marshal this to the format of the server middleware; the response is then mapped to the original message format. While a recognised solution to interoperability, bridging is infeasible in the long term as the number of middleware systems grow i.e. due to the effort required to build direct bridges between all of them. Enterprise Service Buses (ESB) can be seen as a special type of software bridge; they specify a service-oriented middleware with a message-oriented abstraction layer atop different messaging protocols (e.g., SOAP, JMS, SMTP). Rather than provide a direct one-to-one mapping between two messaging protocols, a service bus offers an intermediary message bus. Each service (e.g. a legacy database, JMS queue, Web Service etc.) maps its own message onto the bus using a piece of code, to connect and map, deployed on the peer device. The bus then transmits the intermediary messages to the corresponding endpoints that reverse the translation from the intermediary to the local message type. Hence traditional bridges offer 1-1 mapping; ESBs offer an N-1-M mapping. Example ESBs are Artix [23] and IBM Websphere Message Broker [24]. ESBs offer a solution to the problem of middleware heterogeneity; however, it focuses on the messaging abstraction only and the assumption is that all messaging services can be mapped to the intermediary abstraction (which is a general subset of messaging protocols). This decision is enacted at design or deployment time, as the endpoint must deploy code to connect to a particular message bus with an appropriate translator and hence is unsuitable for dynamic interoperation between two legacy platforms.

INDISS [4], uMiddle [18], OSDA [15], PKUAS [9] and SeDiM [8] are examples of transparent interoperability solutions which attempt to ensure legacy solutions unaware of the heterogeneous middleware are still able to interoperate. Here, protocol specific messages, behaviour and data are captured by the interoperability framework and then translated to an intermediary representaion; a subsequent mapper then translates from the intermediary representation to the specific legacy middleware protocol to interoperate with. The use of an intermediary means that one middleware can be mapped to any other by developing these two elements only (i.e. a direct mapping to every other protocol is not required). Another difference to bridging is that the peers are unaware of the translators (and no software is required to connect to them, as opposed to connecting to bridges).

The interoperation solutions proposed above concentrate on the middleware level. They support interoperation by abstract protocols and language specifications. But, by and large they ignore the data dimension. To this extent a number of efforts, which are generically labelled as Semantic Web Services [16][21][7], attempt to enrich the Web services description languages with a description of the semantics of the data exchanged. The result of these efforts are a set of languages that describe both the orchestration of the services' operations, in the sense of the possible sequences of messages that the services can exchange as well as the meanings of these messages with respect to some reference ontology. However,

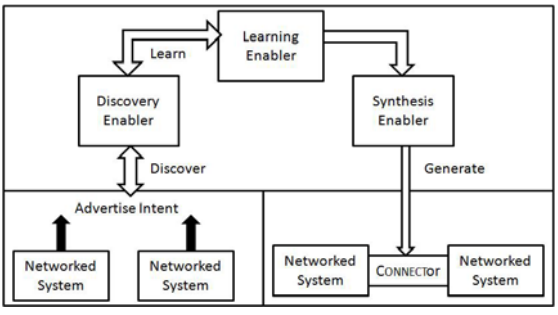


**Fig. 2.** Actors in the CONNECT architecture

such approaches assume a common middleware standard and do not address all of the heterogeneity problems previously described.

The state of the art investigation shows two important things; first, there is a clear disconnect between the main stream middleware work and the work on application, data, and semantic interoperability; second, none of the current solutions addresses all of the four requirements of dynamic pervasive systems as highlighted in the scenario in refScenario. Hence, these results show that there is significant potential for CONNECT to extend beyond the state of the art in interoperability middleware.

## 4 The Connect Architectural Framework

The CONNECT architecture provides the underlying principles and software architecture framework to enact the necessary mechanisms to achieve universal interoperability between heterogeneous systems. Figure 2 presents a high-level overview of the following actors involved within the CONNECT architecture and how they interact with one another:

- *Networked systems* are systems that manifest the will to connect to other systems for fulfilling some intent identified by their users and the applications executing upon them.
- *Enablers* are networked entities that incorporate all the intelligence and logic offered by CONNECT for enabling connection between heterogeneous networked systems. In this paper, we focus on how the discovery, learning and synthesis enablers co-ordinate to produce a CONNECTor as shown in Figure 2, while the companion papers discuss the enablers in more detail [14][19][2][10].
- CONNECTors are the synthesized software connectors produced by the action of enablers to connect networked systems.

### 4.1 Discovery and Learning of Networked Systems

Networked systems use discovery protocols to advertise their will to connect (i.e. their intent); service advertisements are used to describe the services that

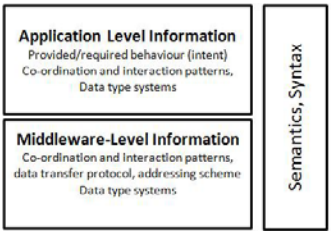


**Fig. 3.** Networked System Model

a system provides, while service lookup requests document the services that are required. It is the role of the *discovery enabler* to capture this information from the legacy network protocols in use and to create an initial picture of the network systems wishing to connect with one another.

The outputs of this enabler are models of networked system as shown in Figure 3. It is important to note that only a subset of this description is made available by the networked system; the learning enabler utilises an active learning algorithm to learn the co-ordination and interaction patterns of the application [14]. Much of the information about the middleware level is not explicit in the discovery process, but pointers within the discovery descriptions (e.g. this is a SOAP service) can be used to build the model from pre-defined, constant middleware models (e.g. a model of the SOAP protocol). The model builds upon discovery protocol descriptions that convey both syntactic information and semantic information about the externalized networked system. This semantic information is necessary in open environments, where semantics cannot be assumed to be inherently carried in a commonly agreed syntax. Typically, ontologies are used in open environments for providing a common vocabulary on which semantic descriptions of networked systems can be based.

The architecture of the discovery enabler is illustrated in Figure 4. This software framework is deployed upon a third party node within the network and consists of three core elements:

- *Discovery protocol plug-ins.* Discovery protocols e.g. SLP, UPnP, LDAP, Jini, etc. are heterogeneous in terms of their behaviour and message format; further they differ in the data representation used to describe services. To resolve this, individual plug-ins for each protocol receive and send messages in the legacy format; the plug-in also translates the advertisements and requests into a common description format used by the CONNECT networked system model.
- The *Model repository* stores networked system models of all CONNECT ready systems (this is a system which advertises its intent and whose behaviour is learnable). These remains alive for the lifetime of the request-for a system advertising its services this will normally match the length of its lease as presented by the legacy protocol and, for a system's request, this is the length of time before the legacy protocol lookup request times out.

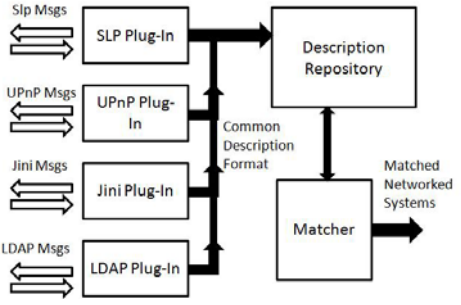


**Fig. 4.** The Discovery Enabler

– The *Functional Matcher* actively matches potential requests with advertisements i.e. matching the required and provided interface types of a network system. Simple semantic matchers can be plugged into to match descriptions of the same type, or richer semantic matchers can be employed.

Learning of networked systems is performed just after discovery and is necessary due to the fact that the retrieved descriptions of networked systems are incomplete. (as described above) CONNECT learning attempts to infer the complete interaction behaviour and employs methods based on monitoring and model-based testing of the networked systems to elicit their interaction behaviour [14]. Learning attempts to extrapolate from observed behaviour to generic behaviour. The outcome of learning is a complete, as far as possible, instantiated networked system model. The learning enabler is built upon the Learnlib tool[20]; this takes as input the interface descriptions of the networked systems (e.g. in WSDL-S) and then executes a learning algorithm which interacts directly with the service to be learned in order to infer the correct behaviour. The enabler outputs a complete LTS model to represent this behaviour of the network system.

### 4.2 Synthesis of Connectors

The CONNECTor Synthesis is a two-step process that encompasses the construction of a mediation LTS and its interpretation at runtime. The needed mediation LTS defines the behaviour that will let the networked systems synchronize and interact. It results from the analysis [19] of both the networked systems' behaviours and the ontology, and specifies all the needed message translations from one side to the other. In the following scenario in Section 5 for instance, when receiving a `getInfo` request coming from the customer side, the mediation LTS will properly request the merchant side (e.g. using `getLocation`, `getPrice`, `getQuantity`) and then aggregate and return the data to the customer. The mediation LTS resolves the application-level and data-level interoperability.

The resulting mediation LTS (see Figure 9) remains an abstract specification that does not include enough middleware-level information to be directly executed. Instead, as shown on Figure 5, the mediation LTS is seen as an orchestration of middleware invocations and is dynamically interpreted by an engine,

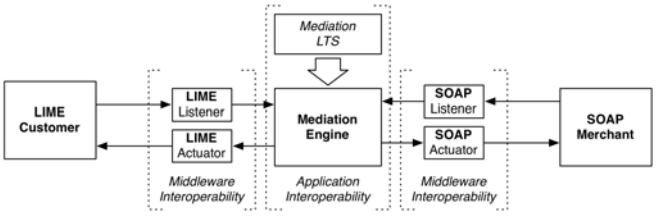


**Fig. 5.** A Software CONNECTor

which receives, translates and forwards messages from the two sides. In our example, when the mediation engine is notified of a `getInfo` tuple was released by the client, it triggers the emission of three SOAP requests and triggers the generation of one Lime tuple containing the requested information.

As shown in Figure 5, the missing middleware-level knowledge is hard-coded into reusable plug-ins denoted as *Listener* and *Actuator*. According to a given middleware protocol, a listener receives data packets and outputs application messages whereas an actuator composes network messages. In our marketplace example, the proper invocation of the LIME infrastructure and the emission and reception of SOAP messages are handled by those ad-hoc listeners and actuators. The use of such plug-ins finally ensures the middleware-level interoperability. In addition, when a new middleware is released, such plug-ins can be separately generated from the networked system models. By contrast with code-generation, the choice of interpretation eases the monitoring and dependability verification of runtime CONNECTors. Although the CONNECT framework also addresses these two issues, they are not presented here for the sake of conciseness.

## 5 Connect in Action

To demonstrate the potential of the CONNECT architecture we consider a single case within the distributed marketplace scenario where two heterogeneous end-systems encounter one another. The client consumer employs SLP as the discovery protocol and the Lime tuple space middleware [17] as the interaction protocol (the German system from Table 1). The service merchant employs SSDP as the discovery protocol and SOAP as the interaction protocol (the French system from Table 1). We apply the CONNECT architecture to build a CONNECTor that allows the consumer to interact with the client. In this section we document the outputs of the enablers to illustrate how the architecture co-ordinates to produce a CONNECTor to overcome the interaction and application heterogeneity between the two systems.

The discovery enabler first monitors the running systems, and receives SLP lookup requests that describe the German application's requirements. It also receives the notification messages from the French application in SSDP that advertise the provided interface. The discovery enabler plug-ins transform these messages and produce a WSDL description for both networked systems. A partial

```
...
<wsdl:operation name="getInfo">
    <wsdl:input message="getInformation"/>
    <wsdl:output message="InfoResponse"/>
</wsdl:operation>
<wsdl:operation name="BuyProduct">
    <wsdl:input message="BuyProduct"/>
    <wsdl:output message="BuyResponse"/>
</wsdl:operation>
<wsdl:operation name="NotifyBuzz">
    <wsdl:output message="LocatedNear"/>
</wsdl:operation>
...
```

```
...
<wsdl:operation name="SearchProducts">
    <wsdl:output message="ProductSearch"/>
    <wsdl:input message="ProductSearchResponse"/>
</wsdl:operation>
    <wsdl:operation name="BuyProduct">
    <wsdl:output message="ProductPurchaseRequest"/>
</wsdl:operation>
<wsdl:operation name="ResponseSubscribe">
    <wsdl:output message="SubscribeForResponses"/>
    <wsdl:input message="VendorNotification"/>
</wsdl:operation>
<wsdl:operation name="getVendorResponse">
    <wsdl:output message="getResponse"/>
    <wsdl:input message="getVendorResponse"/>
</wsdl:operation>
<wsdl:operation name="BuzzSubscribe">
    <wsdl:output message="SubscribeForBuzz"/>
    <wsdl:input message="BuzzNotification"/>
</wsdl:operation>
<wsdl:operation name="getBuzz">
    <wsdl:output message="getBuzz"/>
    <wsdl:input message="VendorBuzz"/>
</wsdl:operation>
...
```

**Fig. 6.** WSDL of the SOAP merchant (left) and the LIME consumer (right)

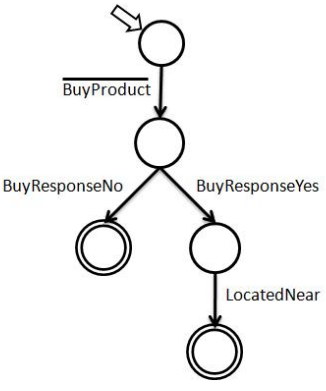


**Fig. 7.** Behaviour Model of merchant produced by learning enabler

view of these is given in Figure 6, and show the abstract operations provided by the application. In the client consumer application, these operations are bound to the concrete LIME protocol (e.g. the `SearchProducts` operation is bound to an `out` operation followed by a `rd`), and in the Merchant application the operations are bound to SOAP (e.g. the `getInfo` operation is bound to a SOAP RPC request). The WSDL also serves to highlight the heterogeneity of the two interfaces; they offer the same functionality, but do so with different behaviour. The next step in the CONNECT architecture is to learn the behaviours of the two systems.

The learning enabler receives the WSDL documents from the discovery enabler and then interacts with deployed instances of the LIME merchant and the SOAP merchant implementations in order to create the behaviour models for both the consumer and the merchant in this case. The interactions possible in

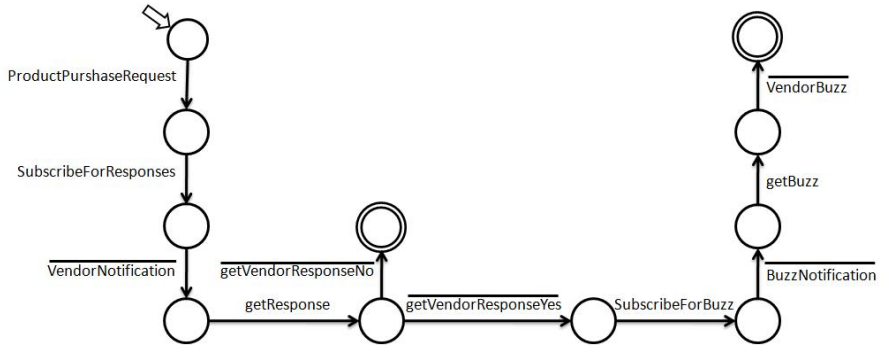


**Fig. 8.** Behaviour Model of Consumer produced by learning enabler. Messages with a bar are emitted while others are received

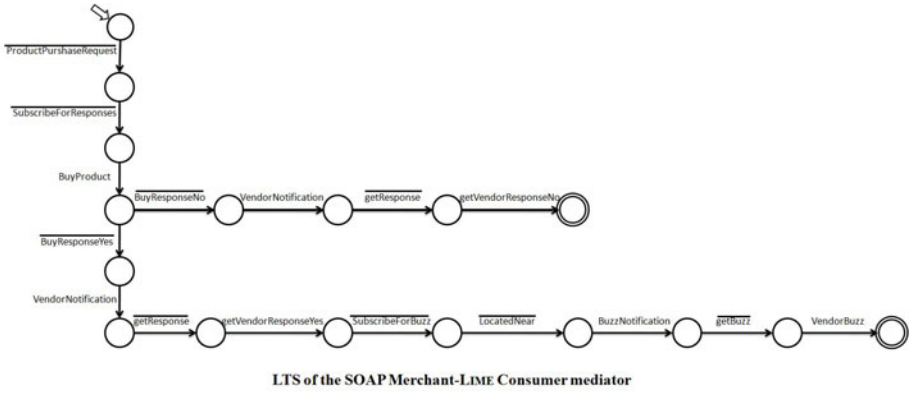


**Fig. 9.** Model of the textscConnector mediator between LIME and SOAP

these systems are produced as LTS models and are illustrated for the SOAP merchant in Figure 7 and for the LIME Consumer in Figure 8. Here we can see that a merchant receives a `BuyProduct` SOAP message and either responds with a yes or no `BuyResponse` SOAP message. If yes, the merchant moves towards the consumer and when close sends the `LocatedNear` SOAP message. In the consumer case, a `ProductPurchaseRequest` is sent as a Lime `out` message (along with a `SubscribeForResponse` reactsTo message to be informed when there is a response in the tuple space). When the merchant replies, a `VendorNotification` is received by the consumer and they read this reponse from the tuple space using a Lime `rd` message (`GetVendorResponse`). If the response is yes, then the consumer subscribes for the buzz message which is then read when the merchant is near.

The final step in the CONNECT process is to create the CONNECTor that will mediate between the consumer's request and the merchant's response. To complete this the two LTS models are passed to the synthesis enabler. This performs two tasks:

- *Behaviour matching.* An ontology is provided for the domain that states where sequences of operations are equivalent e.g. that the `ProductPurchaseRequest` and the `SubscribeForResponses` in the LIME implemented application are the same as the `BuyProduct` SOAP request. Further information about how the ontology-based behavioural matching is given in the companion paper [2].
- *Model synthesis.* The enabler produces an LTS that will mediate between the two systems; this LTS is shown in Figure 9. Here you can see how the interoperation is co-ordinated; when the LIME requests are received these then produce a `BuyProduct` SOAP request, which eventually leads to a response that is converted into a response that can be read from the tuple space. A more detailed version of the mediator (and in particular how it operates on the more detailed LTS models of this scenario) and its behaviour and outputs is again provided in the companion paper [19].

## 6 Conclusions and Future Work

In this paper we have shown that in spite of the major research and industrial efforts to solve the problem of interoperability, current solutions demonstrably fail to meet the needs of modern distributed applications especially those that embrace dynamicity and high levels of heterogeneity. An important observations is that there is a significant disconnect between middleware solutions and semantic interoperability solutions, which in turn severely hampers progress in this area. We have introduced the CONNECT architecture as a fundamentally different way to address the interoperability problem; this intrinsically supports middleware and application level interoperability and embraces learning and synthesis. The initial experiment with the architecture provides early evidence of the validity of the proposed approach and we believe that as the architecture matures it will provide further novel and rich contributions to the field of interoperability.

Future work will continue to explore a broader range of issues in the heterogeneity space. Much of this will focus on the important requirements that have been introduced in the companion papers, and their integration into the CONNECT software architecture. These include:

- *Non-functional properties.* That is creating CONNECTors that conform to the non-functional requirements of both interacting parties in the same way they meet the functional requirements currently.
- *Dynamic monitoring* of CONNECTors will be further investigated to ensure that all requirements are maintained over time. In[19] we illustrate a first integration of synthesis and monitoring.
- *Dependability.* Ensuring that the deployed CONNECTors are dependable, trustworthy and secure; this is especially important given the nature of the pervasive computing environments where these solutions will be deployed.

## Acknowledgments

This work is done as part of the European FP7 ICT FET Connect project (http://connect-forever.eu/).

## References

1. Autili, M., Chilton, C., Inverardi, P., Kwiatkowska, M., Tivoli, M.: Towards a connector algebra. In: ISoLA 2010, Part II. LNCS, vol. 6416, pp. 278–292. Springer, Heidelberg (2010)
2. Bertolonio, A., Inverardi, P., Issarny, V., Sabetta, A., Spalazzese, R.: On-the-fly interoperability though automated mediator synthesis and monitoring. In: ISoLA 2010, Part II. LNCS, vol. 6416, pp. 251–262. Springer, Heidelberg (2010)
3. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web services architecture. In: W3C (February 2004), `http://www.w3.org/TR/sawsdl/`
4. Bromberg, Y., Issarny, V.: Indiss: Interoperable discovery system for networked services. In: Alonso, G. (ed.) Middleware 2005. LNCS, vol. 3790, pp. 164–183. Springer, Heidelberg (2005)
5. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (wsdl) 1.1. In (March 2001), `http://www.w3.org/TR/wsdl`
6. Duftler, M., Mukhi, N., Slominski, S., Weerawarana, S.: Web services invocation framework (wsif). In: OOPSLA 2001 Workshop on Object Oriented Web Services (2001)
7. Farrell, J., Lausen, H.: Semantic annotations for wsdl and xml schema (August 2007), `http://www.w3.org/TR/sawsdl/`
8. Flores, C., Blair, G., Grace, P.: An adaptive middleware to overcome service discovery heterogeneity in mobile ad hoc environments. IEEE Distributed Systems Online (2007)
9. Gang, H., Hong, M., Qian-xiang, W., Fu-qing, Y.: A systematic approach to composing heterogeneous components. Chinese Journal of Electronics 12(4), 499–505 (2003)
10. Di Giandomenico, F., Kwiatkowska, M., Martinucci, M., Masci, P., Qu, H.: Dependability analysis and verification for Connected systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 263–277. Springer, Heidelberg (2010)
11. Grace, P., Blair, G., Samuel, S.: A reflective framework for discovery and interaction in heterogeneous mobile environments. ACM SIGMOBILE Mobile Computing and Communications Review 9(1), 2–14 (2005)
12. Object Management Group. The common object request broker: Architecture and specification version 2.0 (1995)
13. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., Frystyk Nielsen, H., Karmarkar, A., Lafon, Y.: Soap version 1.2 part 1: Messaging framework (April 2001), `http://www.w3.org/TR/soap12-part1`
14. Howar, F., Jonsson, B., Merten, M., Steffen, B., Cassel, S.: On handling data in automata learning: Considerations from the Connect perspective. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 221–235. Springer, Heidelberg (2010)

15. Limam, N., Ziembicki, J., Ahmed, R., Iraqi, Y., Li, D., Boutaba, R., Cuervo, F.: Osda: Open service discovery architecture for efficient cross-domain service provisioning. Computer Communications 30(3), 546–563 (2007)
16. Martin, D., Burstein, M., Mcdermott, D., Mcilraith, S., Paolucci, M., Sycara, K., Mcguinness, D., Sirin, E., Srinivasan, N.: Bringing semantics to web services with owl-s. World Wide Web 10(3), 243–277 (2007)
17. Murphy, A., Picco, G., Roman, G.: Lime: A coordination model and middleware supporting mobility of hosts and agents. ACM Transactions on Software Engineering Methodology 15(3), 279–328 (2006)
18. Nakazawa, J., Tokuda, H., Edwards, W., Ramachandran, U.: A bridging framework for universal interoperability in pervasive systems. In: 26th IEEE International Conference on Distributed Computing Systems, ICDCS 2006 (2006)
19. Issarny, V., Inverardi, P., Spalazzese, R.: A theory of mediators for eternal CONNECTors. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 236–250. Springer, Heidelberg (2010)
20. Raffelt, H., Steffen, B.: Learnlib: A library for automata learning and experimentation. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 377–380. Springer, Heidelberg (2006)
21. Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., Fensel, D.: Web service modeling ontology. Applied Ontology Journal 1(1), 77–106 (2005)
22. Roman, M., Kon, F., Campbell, R.: Reflective middleware: From your desk to your hand. IEEE Distributed Systems Online 2(5) (August 2001)
23. Artix Enterprise Service Bus Software (2010), `http://web.progress.com/en/sonic/artix-esb.html`
24. IBM Software WebSphere, `http://www-01.ibm.com/software/websphere/`