Bridging the Interoperability Gap: Overcoming Combined Application and Middleware Heterogeneity

Yérom-David Bromberg, Paul Grace, Laurent Réveillère, Gordon Blair

Bridging the Interoperability Gap: Overcoming Combined Application and Middleware Heterogeneity

Yérom-David Bromberg[1], Paul Grace[2], Laurent Réveillère[1], and Gordon Blair[2]

[1] LaBRI,University of Bordeaux, France
david.bromberg@labri.fr, laurent.reveillere@labri.fr
[2] School of Computing and Comunications, Lancaster University, UK
p.grace@lancaster.ac.uk, gordon@comp.lancs.ac.uk

Abstract. Interoperability remains a significant challenge in today's distributed systems; it is necessary to quickly compose and connect (often at runtime) previously developed and deployed systems in order to build more complex systems of systems. However, such systems are characterized by heterogeneity at both the application and middleware-level, where application differences are seen in terms of incompatible interface signatures and data content, and at the middleware level in terms of heterogeneous communication protocols. Consider a Flickr client implemented upon the XML-RPC protocol being composed with Picasa's Service; here, the Flickr and Picasa APIs differ significantly, and the underlying communication protocols are different. A number of ad-hoc solutions exist to resolve differences at either distinct level, e.g., data translation technologies, service choreography tools, or protocol bridges; however, we argue that middleware solutions to interoperability should support developers in addressing these challenges using a unified framework. For this purpose we present the Starlink framework, which allows an interoperability solution to be specified using domain specific languages that are then used to generate the necessary executable software to enable runtime interoperability. We demonstrate the effectiveness of Starlink using an application case-study and show that it successfully resolves combined application and middleware heterogeneity.

Keywords: Application, Middleware, Interoperability, Evolution, Domain Specific Languages, Automata

1 Introduction

Nowadays, complex distributed systems are composed from systems that are developed independently of one another (including legacy systems). This composition occurs either statically, or at runtime as in the case of spontaneous interactions between mobile and pervasive systems. However, existing systems are highly heterogeneous in their interaction methods making such composition challenging.

Applications and systems are developed using a multitude of incompatible middleware abstractions and protocols. For example, remote procedure call protocols such as SOAP and IIOP differ in message content, message format, and addressing meaning that they cannot directly interoperate. The range of incompatible protocols drastically limits interoperability, and thus the practical benefit of systems composition. Protocol standardization should address this issue but has been demonstrably ineffective in practice. Indeed, new competing protocols are frequently introduced to cope with the emergence of new application domains (e.g. sensors, ad-hoc networks, Grid Computing, Cloud Computing, etc.), whereas standardization is slow to complete in comparison.

Interoperability is the ability of one or more systems to exchange and understand each other's data. However, there can be significant mismatches between the interfaces of various systems that provide similar application functionalities, making interoperation impossible. Indeed, developers often implement similar application functionalities in different ways, resulting in incompatible operation signatures and data types. In addition, the behavior of the interfaces may also differ, e.g. a single operation in one case may correspond to a sequence of operations in another.

Existing solutions to these interoperability challenges have generally made assumptions about one another, e.g. that the application is fixed and the protocol heterogeneity must be resolved, or the protocol is common and application differences must be addressed. The former is the view of middleware-based solutions such as protocol bridges [1], Enterprise Service Buses, and interoperable middleware [3] [7] [16]. However, none of these solutions work when there is a difference in application functionalities. For example, in a protocol bridge even a simple difference in the operation name breaks the solution. Service Choreography and Workflow execution languages and tools underpinned by Business Processing Execution Language (BPEL) offer methods to overcome application differences but commonly assume an underlying service platform and description language, e.g. SOAP and WSDL. As a consequence, these approaches are not fit for purpose when applications rely on different middleware. Overall, there is no consistent view of how to tackle problems where both application and middleware heterogeneity is encountered in combination. This leads to the use of solutions involving ad-hoc integration of a number of different technologies.

Due to the potential differences in both middleware protocols and application behavior, a single universal bridge can not be developed to address the heterogeneity issue. Instead, a mediated solution is required in each specific case. Many protocol and application specific mediators are thus required to cover the broad solution space. Nevertheless, such a mediator needs to be dynamically generated to manage the runtime composition of services, because developing this mediator for each particular case can be a challenge for many application programmers. To address this issue, we argue that a domain-specific modelling approach can be used for describing application and protocol specificities.

This paper proposes the following contributions towards reaching this goal:

- *Application and protocol models.* We use automata to model application behaviour where a transition represents the application action and associated input and output data. Similarly, we use automata to model middleware protocols where a transition represents either a sent or received message.
- *Application-Middleware Mediators.* A merged automaton models the merge of two application automata, i.e., this states how the application states from one system are merged with the states of the other heterogeneous system. This mediator model is then used to generate a concrete application-middleware mediator that binds application transitions to physical middleware protocol messages.
- *An Interoperability Framework.* We have implemented a middleware framework to support the generation and execution of mediators. The Starlink Framework [2] interprets a concrete merged automaton to enable dynamic interoperability at both the application and protocol level. In previous work we have described how Starlink is used to achieve middleware protocol interoperability; here we expand on the approach to achieve combined application and middleware interoperability.

The remainder of the paper is structured as follows. Section 2 presents a motivating case study to highlight the interoperability challenges. In Section 3, we introduce the application and protocol models. Subsequently, in Section 4 we describe how the interoperability framework realizes and executes these models. Our case-study based evaluation is presented in Section 5 and an analysis of related work is provided in Section 6. Finally, we draw conclusions in Section 7.

2 Motivation: Flickr and Picasa Case Study

To highlight the problem of combined application and middleware heterogeneity we examine the Flickr and Picasa API services, highlight the interoperability challenges and then identify the requirements to overcome them.

2.1 Observing Application and Middleware Heterogeneity

Flickr and Picasa are both Web based services that provide similar application functionality. They allow client applications to view, search, add, edit and delete photographs. In addition, they both allow comments to be added to individual photos or sets of photographs. Although they offer similar services, clients of both can not be composed with the services of the other. In practice, interoperability between the two is hindered due to the heterogeneity at both the application level and at the protocol level.

Application Heterogeneity. The APIs of Flickr[3] and Picassa[4] are large and complex (Flickr has over 100 operations available); hence we concentrate on a small subset of the behavior available. Fig. 1 illustrates how the APIs of

[3] http://www.flickr.com/services/api/
[4] http://code.google.com/apis/picasaweb/

```
flickr.photos.search(api_key, tags, text, per_page, page, ...)
flickr.photos.getInfo(api_key, photo_id)
flickr.photos.comments.getList(api_key, photo_id, min_comment_date, max_comment_date)
flickr.photos.comments.addComment(api_key, photo_id, comment_text)
```

```
PicasaBaseURL - https://picasaweb.google.com/data/feed/api
photos.search(q, max-results)   [GET PicasaBaseURL/all?q=tree&max-results=3]
getComments(kind)   [GET PhotoURL?kind=comment]
addComment(entry)  [POST PhotoURL, <entry> </entry>]
```

Fig. 1. Highlighting the Flickr and Picasa APIs

Flickr and Picasa offer a set of operations for performing similar application requirements; namely, performing a keyword search on publicly searchable photos, listing the comments that have been added to a particular photo result and then finally adding a comment to that same photograph. From these APIs it is clear that application heterogeneity exists in the following two distinct ways:

1. The interface signatures contain sets of operations that differ in operation name, and the types of input and output data of the operation. Consider the operation to perform a general keyword search of public photographs. The Flickr API provides the `search` operation with a number of parameters including the optional `text` parameter for the keyword and `page` and `per_page` parameters to restrict the returned results; alternatively, Picassa provides a search operation with input parameters: **q** for the keyword and `max-results` to restrict the results (n.b. the GET syntax is also shown).
2. The application behavior is captured in different behavior sequences. The Flickr `search` operation returns a set of identifiers. The `getInfo` operation should then be called to obtain more information about the photo, including the URL of the jpeg. Alternatively, the Picasa search operation returns the information about the photograph directly in the search results.

Middleware Heterogeneity. The Flickr and Picassa APIs also differ in the protocols they use to access the services. Picassa provides only a RESTful implementation atop HTTP with the Google Data API as an associated data model, whereas Flickr relies either on REST, SOAP, or XML-RPC. A Flickr client (e.g. a smartphone application) implemented using either SOAP or XML-RPC cannot interoperate with Picasa due to the protocol heterogeneity.

2.2 Interoperability Requirements

Heterogeneous systems that have been developed independent to one another can not interoperate. To address this issue, mediators need to be created and deployed in the network, so as to provide dynamic composition of existing systems. However, the development of such mediators must consider the following factors:

- The extreme heterogeneity in applications and middleware suggests ad-hoc, manually coded solutions will lead to significant development costs and consumption of computational resources due to the continuous redevelopment of equivalent solutions.
- When a new middleware protocol emerges, the API of a service may be migrated to it. Therefore, both existing clients and interoperability mediators for this service would no longer operate correctly.
- Similarly, when a new version of an API is released, any changes to the syntax or behavior of the API may mean that the existing clients or interoperability mediators that rely on this API no longer function.

As a consequence, to overcome the combined application and middleware heterogeneity, we propose two key requirements. First, mediators that act as interoperability enablers must be automatically generated and dynamically deployed. Second, middleware protocol migration and API evolutions must be handled with minimal development effort. In [21], Vinoski argues that interoperability is a mapping problem and that diversity and heterogeneity should be embraced rather than attempt to homogenize distributed systems. Therefore, developers should be supported in creating these mappings. Hence, in this paper we first propose a high-level, model-based specification of the application differences (independent of any middleware protocol); we subsequently propose that this be used to generate the concrete mediator by binding the solution to particular protocol-to-protocol use cases. For example, an application model of the differences between Flickr and Picasa generates an XML-RPC to REST application-specific mediator or a SOAP to REST application-specific mediator.

3 Models

Modern software development trends imply that developers implement applications through the use of reusable API operations, that, in a distributed environment, are remotely invoked through the use of an underlying middleware. For instance, Flickr, Picasa, Bing and/or Google maps API define a set of remote operations that can be invoked with different kind of middleware. The way operations are combined together by developers to perform a particular task depends on particular constraints related to the APIs used, and consequently defines an *API usage protocol*. Inherently, applications performing similar functionalities (i.e. semantically equivalent) but implemented with different APIs, behave differently, and thus have a different API usage protocol. Providing interoperability among applications based on heterogeneous APIs requires first to capture formally their respective APIs usage protocol in order to reason about their behavior.

3.1 APIs usage protocol

An API usage protocol $\mathcal{S}$ defines sequences of ordered operation invocations. Signatures of invoked operations are expressed in terms of input and/or output

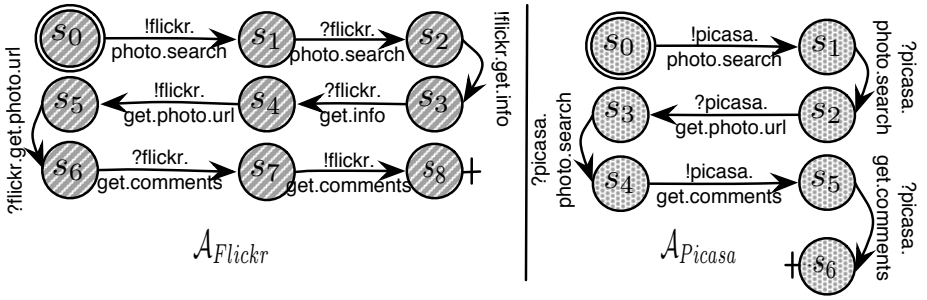


Fig. 2. Flickr and Picasa usage protocol APIs

messages, more precisely in terms of messages exchanged (as developers of Web Services are used to). Syntactical description of message data fields, including their data types are formalized through the use of *abstract messages*. An abstract message consists of a set of fields, either primitive or structured [2]. The former is composed of: (i) a label naming the field, (ii) a type describing the type of the data content, (iii) a length defining the length in bits of the field, and (iv) the value, i.e., the content of the field. A structured field is composed of multiple primitive fields. Hence, we abstract an operation invocation request, noted $rvalue\ operation(arg_1...arg_n)$, as two abstract messages. First, an abstract message named *operation* that is sent and which is composed of a set of n fields such as $field_1 = arg_1, ..., field_n = arg_n$. Second, an abstract message named $rvalue$ that is received. We note $msg \triangleright field$ the operation that selects the field $field$ from the abstract message msg.

As a result, a sequence of operation invocations $\mathcal{S}$ describing an API usage protocol is formalized as an automaton $\mathcal{A}_{\mathcal{S}}$ with edges labeled with abstract messages sent or received according to the signature of remote operations invoked. More formally, $\mathcal{A}_{\mathcal{S}}$ is defined as a 6-tuple such as $\mathcal{A}_{\mathcal{S}} = (Q, M, q_0, F, Act, \rightarrow)$ with Q a finite set of states, M a finite set of both incoming or outgoing abstract messages, $q_0 \in Q$ the starting state and $F \subset Q$ a set of accepting states. $Act = \{?, !\}$ defines two kinds of actions: ! to invoke a remote operation and ? to receive a reply from a previously invoked remote operation.

Hence, the transition relation, noted $\rightarrow \subseteq Q \times Act \times M \times Q$, can be either an *invoke-transition* or a *receive-transition*. The former is noted $s_1 \xrightarrow{!operation} s_2$ for $(s_1, !, operation, s_2) \in \rightarrow$ and indicates the next state to which the automaton passes as soon as the operation *operation* is invoked. The latter has the following form $s_1 \xrightarrow{?rvalue} s_2$ for $(s_1, ?, rvalue, s_2) \in \rightarrow$ and changes the state of the automaton from s_1 to s_2 once the invocation reply $rvalue$ is received. As a result, $\mathcal{A}_{\mathcal{S}}$ acts as a call graph of invoked operations and specifies the order in which they should be invoked.

For instance, Fig. 2 demonstrates the Picasa and Flickr API usage protocols that developers might follow to implement either a Picasa or a Flickr application with similar functionalities.

3.2 API usage Protocol mismatches

From an application perspective, two applications, $\mathcal{A}_1$ and $\mathcal{A}_2$, which follow respectively an API usage protocol $\mathcal{A}_{\mathcal{S}1}$, and an API usage protocol $\mathcal{A}_{\mathcal{S}2}$, may interact seamlessly with each other if and only if there is a way to intertwine their respective API usage protocols. Performing this kind of merging assumes to resolve different kinds of mismatches. As we express operation invocations in terms of messages exchanged, we can leverage on guidelines of possible mismatches that have already been identified for developing Web services adapters [13] and apply them to API usage protocol mismatches. For instance, the comparison of two API usage protocols such as $\mathcal{A}_{Flickr}$ and $\mathcal{A}_{Picasa}$, depicted in Fig. 2, enables us to point out the different mismatches that occur:

Ordering mismatch. When applications invoke similar remote operations in a different order, an ordering mismatch may occur. For instance, according to $\mathcal{A}_{Picasa}$, a Flickr developer should invoke a `getPhotoUrl` operation right after a `photoSearch`. However, the `getPhotoUrl` operation is called, in fact, later in the call graph.

Extra or missing message mismatch. If one application invokes a remote operation that another application never invokes, there is an extra or missing message mismatch. For instance, a Picasa developer does not invoke any operations similar to the Flickr operation `getInfo`, which is specific to the Flickr API.

One-to-many mismatch. An API can perform a particular task with only one remote operation, whereas another API may require several operations to do a similar task. For instance, obtaining a photo URL requires only one `search` operation using Picasa, whereas it requires two operations (i.e. `search` and `getInfo`) with Flickr.

In the context of an API usage protocol, the aforementioned mismatches emerge as soon as there are mismatches among operations at their signature level. As a result, there is not always a one-to-one mapping between messages. Note that resolving API usage protocol heterogeneity is theoretically similar to resolving heterogeneity at the protocol layer but acts on messages that abstract operation invocations instead of network messages. So resolving operation signatures mismatches leads us to reason about semantic equivalence among the abstract messages exchanged. To this end, we extend our model with a semantic equivalence operator $\cong$ that acts on messages, abstracting operations, as defined below.

Definition 1. *Let $\overrightarrow{m}$ a sequence of abstract messages. Further, $!m$ or $?m$ denotes a message to be sent or received, and $!s_i.m$ or $?s_i.m$ denotes a message sent or received in a specific state s_i.*

Definition 2. *Let $\cong$ a semantic equivalence operator such that $n \cong \overrightarrow{m}$ is true if and only if for every mandatory field of n, noted $\mathcal{M}_{fields}(n)$, there exists a semantically equivalent field in one message of the sequence $\overrightarrow{m}$. So $n \cong \overrightarrow{m}$ if*

and only if $\forall n \triangleright field \in \mathcal{M}_{fields}(n), \exists m \in \overrightarrow{m} = \langle m_1 \dots m_n \rangle$ *such as* $n \triangleright field \models m \triangleright field$.

3.3 k-Colored Automata: Intertwining API usage Protocol

Informally, application $\mathcal{A}_1$ may interact with application $\mathcal{A}_2$ if the following conditions are satisfied: (i) their respective API usage protocols $\mathcal{A}_{\mathcal{S}1}$ and $\mathcal{A}_{\mathcal{S}2}$ share a sufficient number of similar operations that enables them to have a successful sequence of operations to reach their respective final state, (ii) the identified semantically equivalent can be intertwined together,i.e. invoked in an alternate order when required. In other terms, invoked operations, i.e. request messages, from $\mathcal{A}_1$ must be sequentially translated into a semantically equivalent request message followed by a corresponding reply message from $\mathcal{A}_2$. Based on the previous introduced definition, we extend the model with a history and an intertwining operator to formally define the aforementioned constraints.

Definition 3. *Let* $\mathcal{I}(\mathcal{A}_{\mathcal{S}})$ *the set of initial states and* $\mathcal{ENT}(\mathcal{A}_{\mathcal{S}})$ *the set of final states of* $\mathcal{A}_{\mathcal{S}}$. *The set of all states of* $\mathcal{A}_{\mathcal{S}}$*is* $States(\mathcal{A}_{\mathcal{S}}) = \mathcal{I}(\mathcal{A}_{\mathcal{S}}) \cup \mathcal{ENT}(\mathcal{A}_{\mathcal{S}})$. *Further, let* $\mathcal{M}_{sg}$ $(\mathcal{A}_{\mathcal{S}})$ *the set of all messages and* $\mathcal{T}(A_S)$ *the set of all transitions of* $\mathcal{A}_{\mathcal{S}}$.

Definition 4. *Let* $\Rightarrow$ *the history operator defined such as* $\Rightarrow \subseteq States(A_S) \times Act \times \overrightarrow{m} \times States(A_S)$ *with* $Act = \{!, ?\}$ *and* $\overrightarrow{m} = \{m_i, ..., m_k, ..., m_n\} \in \mathcal{M}_{sg}(A_S)$ *with* $(i, k, n) \in \{1, .., n\}$. *Thus,* $s_1 \overset{!\overrightarrow{m}}{\Longrightarrow} s_2$ *(resp.* $s_1 \overset{?\overrightarrow{m}}{\Longrightarrow} s_2$*) gives the sequence of abstract messages sent (resp. received) from the state* s_1 *to* s_2.

Definition 5. *Let* $\rightsquigarrow$ *the intertwining operator such as* $!s_i.method_1 \rightsquigarrow !s_j.method_2$ *is true iff*
$\exists s_0 \in \mathcal{I}(\mathcal{A}_{\mathcal{S}1}).\exists s_i \in States(\mathcal{A}_{\mathcal{S}1}).\exists s_j \in States(\mathcal{A}_{\mathcal{S}2}).$
$\exists method_1 \in \mathcal{M}_{sg}(A_{S1}).\exists method_2 \in \mathcal{M}_{sg}(A_{S2})|$
$((?method_1 \cong ?method_2) \vee (?method_2 \cong (s_0 \overset{?\overrightarrow{m}}{\Longrightarrow} s_i, ?method_1)) \wedge$
$!s_j.method_2 \cong (s_0 \overset{?\overrightarrow{m}}{\Longrightarrow} s_i, s_0 \overset{!\overrightarrow{m}}{\Longrightarrow} s_i)$
Reciprocally, $?s_i.method_1 \rightsquigarrow ?s_j.method_2$ *is true iff* $!s_i\ .method_1 \rightsquigarrow !s_j.method_2$.

Thus, if $\mathcal{A}_{\mathcal{S}1}$ has a sequence of n intertwined operations with $\mathcal{A}_{\mathcal{S}2}$, it means that there exists n transitions, named γ-transitions, that go back and forth between $\mathcal{A}_{\mathcal{S}1}$ and $\mathcal{A}_{\mathcal{S}2}$ without sending or receiving messages but applying successful data transformation on semantically equivalent messages as described in Section 4. As a result, the resulting automaton is said to be a k-colored automaton. The k color enables one to identify states that belong to either $\mathcal{A}_{\mathcal{S}1}$ or $\mathcal{A}_{\mathcal{S}2}$ as depicted in Fig. 3. Further, states linked by a $\gamma-$transition are represented by bicolored nodes such as nodes ❶, ❷, ❸, ❹,❺, ❻. For instance, at node ❶, Flickr `photoSearch` invocation can be intertwined with the corresponding Picasa `photoSearch` as $!flickr.photoSearch \cong !picasa.photoSearch$, and a $\gamma-$transition is taken to move from the Flickr API usage protocol to the Picasa one through some translations on data fields as messages are semantically equivalent.

Definition 6. *A k-colored automaton is an automaton with all its states colored by a color k. Thus an automaton $\mathcal{A}_{\mathcal{S}}$ colored by a color k is noted ${\mathcal{A}_{\mathcal{S}}}^k$ where States(${\mathcal{A}_{\mathcal{S}}}^k$) = $\{s_0^k, ..., s_i^k, ..., s_n^k\}$.*

Definition 7. *An application $\mathcal{A}_1$ may interact with an application $\mathcal{A}_2$ iff their colored API usage protocol A_{S1}^1 and A_{S2}^2 are mergeable, and noted $A_{S1}^1 \bigoplus A_{S2}^2$, such that $\exists Seq = \{..., (s_x^1, s_y^2), ...\} \subseteq States(A_{S1}^1) \times States(A_{S2}^2) with (x, y) \in \{1, ..., n\} \wedge \exists(!m_1, !m_2) \subset \mathcal{M}_{sg}(A_{S1}^1) \times \mathcal{M}_{sg}(A_{S2}^2) \mid \{!s_x.m_1 \rightsquigarrow !s_y.m_2\} \bigwedge \exists(s_i, s_j) \subset \mathcal{ENCD}(A_{S1}^1) \times \mathcal{ENCD}(A_{S2}^2)$ with $(i, j) \in \{1, ..., n\} \mid s_i, s_j \in \mathcal{ENCD}\ (A_{S1}^1 \bigoplus A_{S2}^2)$.*

Note that all invocation operations from $\mathcal{A}_{\mathcal{S}1}$ can not be always intertwined with the ones of $\mathcal{A}_{\mathcal{S}2}$, but it does not hinder necessarily the interoperation. Hence, it is required to consider that it is possible to get two different kinds of k-colored automaton: strongly or weakly merged. The former case still arises even if some invocation operations from $\mathcal{A}_{\mathcal{S}1}$ are not intertwined, however their corresponding replies must be semantically equivalent to replies received from $\mathcal{A}_{\mathcal{S}2}$. Otherwise, if this condition is not satisfied the k-colored automaton is said to be weakly merged. For instance, in Fig. 3, the Flickr operation `getInfo` has no equivalent in the Picasa API, however, its reply is semantically equivalent to the Picasa `photoSearch` reply previously received. The flickr `getInfo` operation can be invoked (i.e through the send and receive of $flickr.getInfo$) without both being interleaved and hindering the interoperation. The depicted automaton is still strongly merged.

Definition 8. *The resulting $A_{S1}^1 \bigoplus A_{S2}^2$ is a k-colored automaton, with $k = \{1, 2\}$, defined as a $7-tuple$ $(Q, M, q0, F, Act, \rightarrow, \xrightarrow{\gamma}, P, \cong)$ where $Q = \bigcup_{k=1...2} States(A_{Sk}^k)$, $M = \bigcup_{k=1...2} \mathcal{M}_{sg}(A_{Sk}^k)$, q_0 a starting state $\in \mathcal{I}(A_{S1}^1)$, $F = \bigcup_{k=1...2} \mathcal{ENCD}(A_{Sk}^k)$, $P = \{\lambda\}$ a set of data transformations on messages semantically equivalent according to the $\cong$ relation, and $\rightarrow = \bigcup_{k=1...2} \mathcal{T}(A_{Sk}^k)$. Finally, $\xrightarrow{\gamma} \subseteq States(A_{S1}^1) \times P \times States(A_{S2}^2)$ are $\gamma-$transitions that occur when sent (or received messages) from $States(A_{S1}^1)$ can be interleaved with the ones from $States(A_{S2}^2)$ according to the $\rightsquigarrow$ operator. $\gamma-$transitions take the form $s_i \xrightarrow{\gamma(\{\lambda\})} s_j$. The operator $\cong$ is defined as previously.*

Note that a data transformation $\lambda_i \in \{\lambda\}$ is a function $\lambda_i : field_1 \times ... \times field_n$ that performs a data transformation and may require as arguments some fields extracted from previously received messages.

4 Applying the Starlink Framework

Starlink is a runtime middleware framework which provides an engine to dynamically interpret and execute middleware models. The key design principles are based upon the knowledge that middleware technologies are built upon message-based solutions, i.e., middleware protocols consist of sending to and receiving messages from a network. We have previously documented how the framework [2]

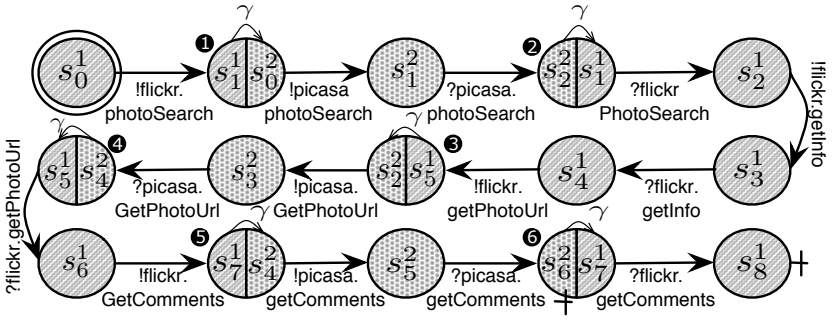


Fig. 3. Merged Flickr and Picasa usage protocol API

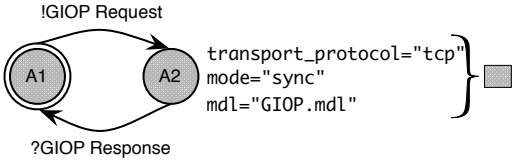
can be used to dynamically generate direct protocol bridges (i.e. connecting middleware protocols of similar types, such as service discovery and RPC). We now describe how it can be used more broadly to develop and deploy *application-middleware mediators*. We first describe the models used by Starlink and then how they are executed. Subsequently, we describe how the application models introduced in Section 3 are used to generate the Starlink executable models.

4.1 Starlink Models

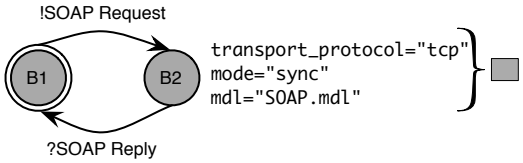
In this section we introduce the core models that are interpreted by Starlink. Firstly, how protocol message sequences are specified. Secondly, how message format is defined. Thirdly, how message translation logic is described.

Message Sequences. The behavior of a protocol is traditionally described by an automaton where transitions represent message exchanges. However, protocols vary in their interaction with the network, in terms of the transport protocol used, whether requests are sent by unicast or by multicast, and whether responses are received synchronously or asynchronously. Starlink uses the previously introduced *k-colored automata* to capture the properties of a protocol by a color k and ensure that the messages are executed using the appropriate network services [2]. Fig. 4(a) illustrates the k-colored automaton for general IIOP client behaviour, i.e. a GIOP request message is sent synchronously to an IIOP server and on the same connection it receives the GIOP reply message. Fig. 4(b) illustrates SOAP client behaviour.

Message Format and Content. A network message is organized as a sequence of text lines, or of bits, for a binary protocol, containing both fixed elements and elements specific to a given message. Extracting values from a message represented as a sequence of text or binary characters is unwieldy, and creating messages is even more complex, because the element values may become available at different times, making it difficult to predict the message size and layout. Hence, we have proposed a domain specific language approach to describe messages such that the required message parsers and composers can be generated automatically.



(a) IIOP client k-colored automaton



(b) SOAP client k-colored automaton

Fig. 4. Examples of concrete k-colored automata

```
<Message:GIOPRequest>
<Rule:MessageType=0>
<RequestID:32><Response:8>
... <ObjectKeyLength:32><ObjectKey:ObjectKeyLength>
... <OperationLength:32><Operation:OperationLength>
... <align:64><ParameterArray:eof>
<End:Message>

<Message:GIOPReply>
<Rule:MessageType=1>
<RequestID:32><ReplyStatus:32><ContextListLength:32>
...<align:64><ParameterArray:eof>
<End:Message>
```

Fig. 5. MDL specification of the GIOP message format

The Starlink framework is flexible to allow different types of language to be used to specify message formats; each language can be termed a Message Description Language (MDL). This flexibility better supports the parsing and composing of a wide range of protocols. For example, specialised languages for binary messages, text messages and XML messages can be plugged into the framework. From an MDL specification, Starlink dynamically generates parsers that transform network messages to the *abstract message representation*. Reciprocally, the generated composers do the reverse. An example of MDL specification for GIOP messages is presented in Fig. 5. Detailed discussion of the language is left from here, and further information is available in [2].

Message Translations. When several protocols need to interoperate, it is necessary to express the relation among them and to describe the *message translation logic (MTL)*, which defines how to translate messages from one protocol to another. Translation logic is used to describe the translation of data and behaviour where messages are semantically equivalent, i.e. the messages perform similar operations. This logic is executed at the bi-colored states of a colored

automata and typically consists of field transformation where a field in the message to be composed is assigned a value from a received field (there will typically be a transformation function as part of this assignment). One key operator of the MTL language is the *assignment operation*.

4.2 The Starlink Framework: dynamically interpreting middleware models

As illustrated in Fig. 6, the Starlink framework interprets the previously described middleware models at runtime in order to support the necessary middleware behaviour on demand.

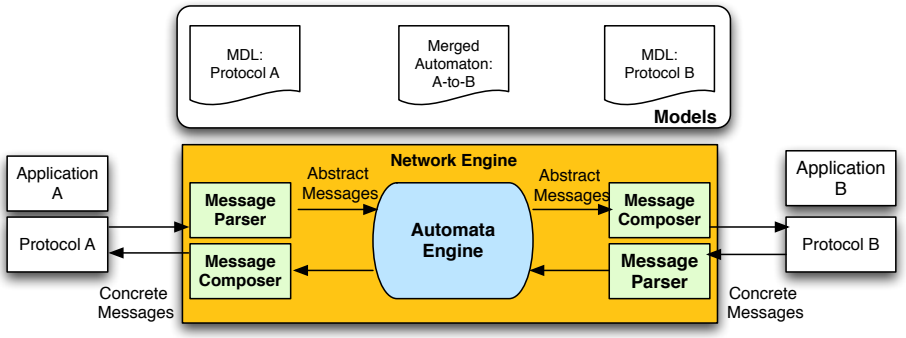


Fig. 6. Architecture of the Starlink framework

The *network engine* sends and receives physical messages (i.e. data packets) to and from the network. A transition in the k-colored automata attaches network semantics to describe the requirements of the network. The network engine then provides the services to meet these requirements, which could include different types of transport or multicast behaviour. The current implementation of the network engine provides traditional TCP and UDP services for infrastructure networks. However, the architecture is configurable so that if Starlink were to be deployed in more heterogeneous environments, e.g. ad-hoc networks, this network engine could be replaced with configurable services for ad-hoc routing [18].

The *message parsers* read the contents of a network packet and parse them into the *AbstractMessage* representation such that the data can be manipulated during the mediation process. For example, if a HTTP message is received a HTTP parser reads all the fields of the header and body. Correspondingly, *message composers* construct the data packet for a particular protocol message, e.g. constructing the content for a HTTP GET message. Importantly, the message composers and parsers are generic reusable software elements that interpret

high-level specifications of message content. The Message Description Language (MDL) specification (as described previously) specializes these generic components at runtime to create a specific protocol parser or composer.

The *automata engine* executes the behaviour of the merged automata, i.e. it controls the sequence of sending, receiving, parsing, composing and translation of messages. In Starlink, there a three types of states: i) a *receiving* state waits to receive a message and will only follow a matching receive transition when a matching message is received; ii) a *sending* state sends a message described in the single transition; iii) a *no-action* state is a translation state that translates data from the fields on one or more of the prior messages into the message to be constructed.

4.3 Generating and Executing Application-Middleware Automata

API usage protocol automata are specified independent from particular middleware. We now describe how these are bound to a specific protocol to create a Starlink executable k-colored automaton specific to the API implementation.

As described in Section 3 the API usage protocol automaton defines the application `actions` in terms of sending actions (invocation) and receiving actions (reply response). These transitions contain the *action label* and the *abstract message* that includes the input or output data values. Actions correspond to distributed interactions. However, they cannot be executed because they do not relate to a specific communication protocol. Indeed, the labels and data are only made concrete using protocol messages. Therefore, the API usage automaton must be bound to a concrete protocol automaton in order to be executed. We term the resulting model an *application-middleware automaton*. To better illustrate this procedure, Fig. 7 shows how a simple API usage protocol automaton is bound to two heterogeneous middleware protocols, namely IIOP and SOAP. The client application performs an addition operation (Add) from a remote service. For this, it sends an `Add action`, followed by the reception of the `Add action response`. The input values consist of the `x` and `y` integer parameters to be added. The output value is the returned integer parameter `z`.

To bind to a particular protocol we require: i) the k-colored automaton of the middleware protocol (e.g. Fig. 4(a)), ii) the MDL specification of that protocol's messages (e.g. Fig. 5) and iii) the set of rules that describe how a particular protocol (e.g. GIOP) is bound to the application automata concepts (i.e. the action labels, and the parameters). The rules to bind applications to SOAP in one case and IIOP in the other are illustrated in Fig. 7. IIOP and SOAP are both RPC protocols and hence the actions correspond to the request and response messages of each protocol, as seen by the corresponding k-colored sequence. The action label then binds to specific fields within the message described by MDL: the `operation` field of the GIOP Request message, and the `methodname` field of the SOAP request envelope. Similarly, the request action parameters (the x and y integers) relate to the first two parameters in the ParameterArray field of the

GIOP Request message. The return value parameter (the z integer value) relates to the first parameter of the GIOP reply ParameterArray.

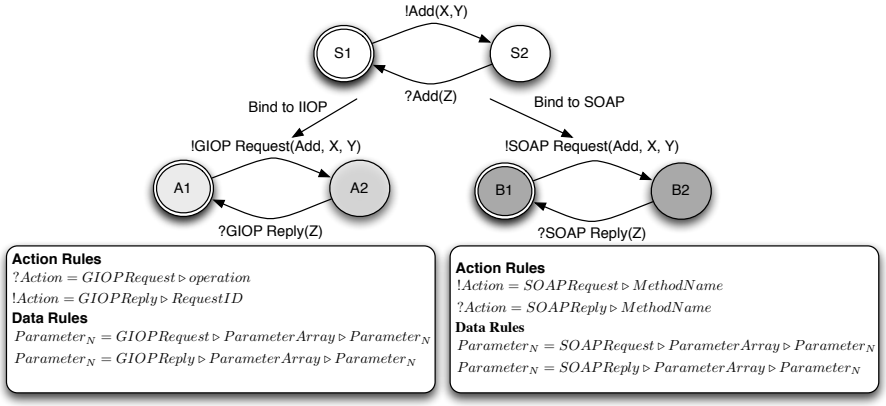


Fig. 7. Binding to concrete application-middleware automata

Starlink is then able to execute the application-middleware automaton in order to concretely achieve the application behaviour. At receiving states, the automata engine waits for middleware messages of a particular type (e.g. a SOAP Reply) and also with a particular action label (e.g. add). Subsequently, at send states the middleware message (e.g. SOAP request) is constructed placing the appropriate application labels and input values in the identified fields as described by the protocol binding rules.

4.4 Generating and Executing the Intertwining API Usage Protocol Automata

A similar binding process is carried out to generate the concrete version of an intertwining API Usage Protocol Automaton, i.e., where two heterogeneous applications are merged. For transitions, the bindings are identical to those explained in section 4.3; the difference occurs at the bi-colored states where MTL rules must be executed to translate application data from a parsed message into the composed message. In this situation we must generate the concrete MTL rules relating to the MTL definitions in the Intertwining automaton.

To illustrate this procedure we continue with the simple addition example. In this case the SOAP service provides an add operation with an `int Plus(int, int)` signature whereas the IIOP client interface signature is `int Add(int, int)`. Hence, the application difference is in the operation name. Fig. 8 shows how the merged application automaton is bound to the concrete merged automaton. On the left side of the figure is the specified application merge, with the bi-colored states representing the translation of parameters between actions. On the right

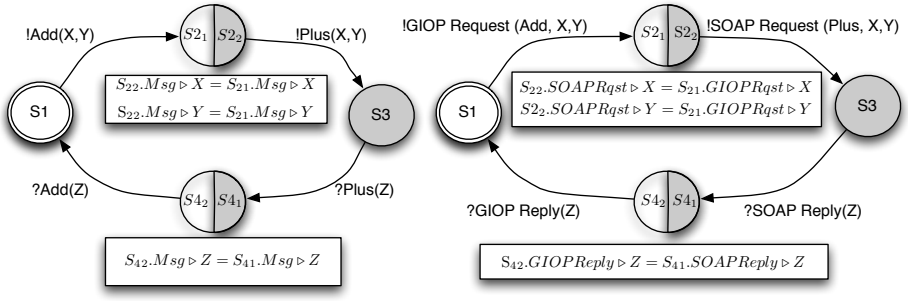


Fig. 8. Construct a concrete merged application automaton

side is the concrete merged k-colored automaton, where the action transitions are bound to specific middleware protocols (the operation name difference is overcome by this, after an Add action is received a Plus action is sent). Note, the application translations are bound to the specific MTL translations based upon the binding rules specified in Section 4.3.

5 Evaluation

To evaluate our approach for overcoming combined application and middleware heterogeneity, we use a case-study based methodology. That is, we apply Starlink to particular use cases and observe the extent to which interoperability is achieved. For this purpose we consider the application scenario described in Section 2. This application performs search and display of public photographs and requires interoperation between independently developed XML-RPC and SOAP Flickr clients and the Picasa Rest implemented API. We hypothesize the following:

1. The Starlink models can specify the application differences between Flickr and Picasa independent of SOAP, XML-RPC and HTTP messages.
2. Concrete models for both the XML-RPC and SOAP use cases can be successfully generated, deployed and executed to achieve the required interoperability with the Picasa API.
3. The use of high-level specifications simplifies the development of mediators and resolves evolution problems.

5.1 Flickr-Picasa Case-Study

In this case study we develop and deploy two mediators: a Flickr-Picasa mediator for XML-RPC to Rest, and a Flickr-Picasa mediator for SOAP to Rest. In the first instance we specify the application automata describing the API usage of both the client and service, as shown Fig. 2. Although automata are written using the XML-based Starlink language for k-colored automata, we use visual

representations for clarity. Subsequently, we specified: i) the intertwined API usage automaton as shown in Fig. 3, ii) the SOAP protocol models consisting of the MDL and the k-colored protocol automaton, iii) the XML-RPC models, and iv) the Rest models.

The next step consists of generating the application-middleware mediators by binding the single intertwined automaton to the two particular use cases. We now present in the remainder of this section the results of this binding. For sake of clarity, we only describe subsets of it to highlight key results.

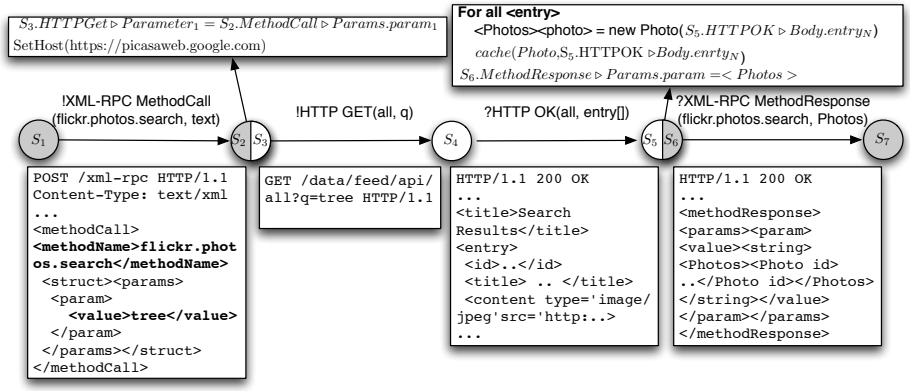


Fig. 9. XML-RPC to Rest binding for Search operation

An extract of the binding of the intertwined Flickr and Picasa search operations to an 'XML-RPC to Rest' concrete mediator is shown in Fig. 9. It illustrates how the XML-RPC Flickr message is parsed to extract the application information from transition S_1 to S_2 (e.g. the action label `flickr.photos.search` and the data parameter labelled `text`). The MTL for S_2 to S_3 then describes how the fields are translated before constructing the HTTP message to perform a Picasa search. The subsequent translation of the responses, from state S_5 to S_6, highlights a case where further functionality is required. The values that must be returned to the Flickr search operation is a list of Flickr photo identifiers in the format <`photo id id='1111' owner='1111111@NO1'`>. To handle this mapping, the bridge creates a cache of dummy identifiers for each photo result returned in the Picasa action response (the <`entry`><`id`> value from the XML data). The MTL provides a keyword operation `cache` that caches data values for arbitrary data identifiers.

An example of operation mismatch in the intertwined automaton is illustrated in Fig. 10. Indeed, when the Flickr client sends a `getInfo` action request, there is no corresponding operation in Picasa because the required action result data has already been received in Picasa's search response. Hence, when the `getInfo`

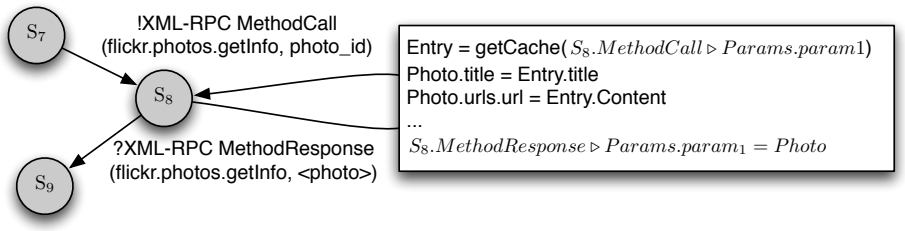


Fig. 10. MTL translation for Flickr-Picasa mismatch behaviour

XML-RPC message is received at S_8 then a data translation is performed: the *photo_id* parameter is used to extract the Picasa <`entry`> value from the cache using the `getcache` MTL keyword. The Flickr <`photo`> structure is then filled using the corresponding tags from the Picasa <`entry`> structure.

The binding of the intertwined Flickr and Picasa comment operations to a 'SOAP to Rest' concrete mediator is similar to the XML-RPC binding and uses the rules provided in section 4. In this case, the generated MTL and k-colored automata refer to SOAP message content rather than XML-RPC.

Finally, we hand developed two test standalone client applications in SOAP and XML-RPC that searched and displayed photographs from the Flickr API. We then deployed Starlink in the network and loaded the concrete models. When executed, both clients were able to search and view photographs from the Picasa API. For our experiments, we deployed a simple proxy to redirect the Flickr requests (originally directed to the Flickr servers) to the local Starlink mediator.

5.2 Analysis

The automaton that specifies the application model contains no reference to a concrete protocol, message format or network semantics. As a consequence, it is seen that the first hypothesis that application behaviour can be modelled independent of middleware is successfully achieved. The generated concrete mediators, when deployed in the network, successfully parse and compose middleware messages and bridge the heterogeneous application behaviour in both the XML-RPC and SOAP case. Hence, the hypothesis that such code can be generated for multiple specific protocols is shown to be true. Finally, it can be argued that the definition of a single application model simplifies the development of interoperability solutions. There is no need to hand code each use case, and it is similarly straightforward to handle API migrations or changes using only the models.

6 Related Work

Middleware solutions to interoperability generally focus on bridging the gap between the various middleware technologies involved. These assume a common application *standard*, i.e., that applications wishing to interoperate use the same

interface defined in the same language (e.g. Interface Description Language (IDL) or Web Services Description Language (WSDL)). In this situation the interoperability gap is between heterogeneous middleware protocols. Protocol bridges [1], Enterprise Service Buses [12, 11], and Interoperability Frameworks e.g. WSIF [6], uMiddle [16], OSDA [15], and UIC [19] are well known solutions to this problem. However, because they do not consider heterogeneity at the application-level they are not suited to the composition of complex systems-of-systems; where independently developed applications are composed dynamically it is unlikely that the application interface has been agreed in advance.

Several technologies are available to manage the differences between application service interfaces in terms of operation and message sequences. As an example, Web Services orchestration and choreography methods [17] provide languages such as Web Services-Choreography Description Language (WS-CDL) and Business Process Execution Language (BPEL) to handle such translations. These languages are similar to Starlink in that they provide high-level constructs to mediate behaviour sequences and also perform data translations. However, they assume an underlying platform (e.g. Web Services) and focus on choreography rather than on direct interoperability. As a consequence, differences in underlying protocols cannot be handled. Furthermore, they cannot manage the differences in interface languages. For example, BPEL cannot be used to generate a solution to make a CORBA IDL-based client interoperate with a SOAP WSDL-based service.

Model Driven Architecture (MDA) [8] proposes a similar methodology to Starlink, which indeed is inspired by the modelling ideas put forward by MDA. Application systems are specified using an abstract model, called the Process Independent Model (PIM). The PIM is deployed atop middleware based platforms described by the Platform Specific Model (PSM). Bridges are then deployed where there are exchanges between different PSMs to ensure that the platform heterogeneity is resolved. However, MDA is characterised by ad-hoc solutions with limited support for the generation of bridges between the platform models.

Formal specifications have been proposed to generate mediators between heterogeneous systems. Yellin and Strom [22] describe a method to enhance application interfaces with sequencing constraints in addition to rules that describe how the application data can be bridged. This information is then used to generate the code of the software adapters. Similarly, [14] describes a discrete event systems method for describing a converter between disparate protocols. While closely related to our formal models of protocol and translation, our approach further investigates the concrete realities of application differences based upon heterogeneous middleware paradigms and message formats.

Currently Starlink developers construct the merged automata; however, emerging solutions have investigated how to generate the mediator automatically. [20] models protocols as labelled transition systems (LTS) and presents an algorithm to identify the merge of the two; however at present it considers only message sequence differences not data heterogeneity. Similarly, work in the CONTESSA project [9] presents a reflective approach to compose heterogneous protocol-based

services. This utilises semantic models of the transitions between the configurations of the protocols and services. While this doesn't cover the complete interoperability mappings that Starlink proposes it does offer important insights into how reasoning and composition can be performed automatically at runtime.

7 Conclusions

In this paper we have shown that the interoperability problem is characterised by differences in both application APIs and middleware protocols. Existing solutions have focused on one of these dimension while making assumptions about the common nature of the other. In complex and dynamic systems such assumptions are invalid, and new approaches are required to consider application and middleware together. For this purpose, we have presented Starlink[5]–a framework to model applications and protocols such that the code to interoperate can be generated. Specifically, this consists of application and application-middleware mediator models (specified using automata and domain specific languages) that are interpreted at runtime. We have performed evaluation of Starlink using a case study involving heterogeneous web-based services (i.e. photo sharing). Preliminary results show that Starlink can successfully address the interoperability challenges, and simplify the task of connecting disparate systems.

Starlink requires the developer to write models, however given the scale of heterogeneous applications and protocols automated generation of these models is the ultimate goal. Hence, it is necessary to *reason* about the individual application and protocol models and generate the merged automata between. To underpin this reasoning we believe that additional semantic models can be used to infer the translation logic and we are investigating the use of ontologies [5] and their associated tools. For full automation, *machine learning* is required and hence we are also investigating learning techniques to understand and model the behaviour of the individual protocols. For example, dynamic binary analysis approaches have been used to identify the field structure of network messages [4] and learning algorithms have been utilised to learn the interaction behaviour of application and middleware protocols [10].

Acknowledgments.

Part-funded unded under the EU FP7 Connect project (http://connect-forever.eu).

References

1. Soap2corba and corba2soap. http://soap2corba.sourceforge.net/.
2. Y.-D. Bromberg, P. Grace, and L. Réveillère. Starlink: runtime interoperability between heterogeneous middleware protocols. In *The 31st International Conference on Distributed Computing Systems*, Minneapolis, MN, USA, June 2011.

[5] the framework is available to download at http://starlink.sourceforge.net

3. Y.-D. Bromberg and V. Issarny. Indiss: Interoperable discovery system for networked services. In *IFIP/ACM/Usenix International Middleware Conference*, pages 164–183, 2005.
4. J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 317–329, New York, NY, USA, 2007. ACM.
5. M. Daconta, L. Obrst, and K. Smith. *The Semantic Web: A Guide to the Future of XML, Web Services and Knowledge Management*. Wiley, Indianapolis, IN, 2003.
6. M. Duftler, N. Mukhi, S. Slominski, and S. Weerawarana. Web services invocation framework (wsif). In *OOPSLA Workshop on Object Oriented Web Services*, 2001.
7. P. Grace, G. Blair, and S. Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *ACM SIGMOBILE Mobile Computing and Communications Review*, 9(1):2–14, Janaury 2005.
8. Object Management Group. Model driven architecture (mda), document number ormsc/2001-07-01. Technical report, 2001.
9. Sebastian Gutierrez-Nolasco and Nalini Venkatasubramanian. A reflective middleware framework for communication in dynamic environments. In *On the Move to Meaningful Internet Systems*, pages 791–808, London, UK, 2002. Springer-Verlag.
10. F. Howar, B. Jonsson, M. Merten, B. Steffen, and S. Cassel. On handling data in automata learning - considerations from the connect perspective. In *ISoLA (2)*, pages 221–235, 2010.
11. IBM. Websphere message broker. www.ibm.com/websphere/wbimessagebroker.
12. IONA. Artix esb. [online]. http://www.iona.com/products/artix/, 2007.
13. W. Kongdenfha, H. Motahari-Nezhad, B. Benatallah, F. Casati, and R. Saint-Paul. Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters. *IEEE Trans. Serv. Comput.*, 2:94–107, April 2009.
14. R. Kumar and S. Nelvagal, S.and Marcus. A discrete event systems approach for protocol conversion. *Discrete Event Dynamic Systems*, 7:295–315, June 1997.
15. N. Limam, J. Ziembicki, R. Ahmed, Y. Iraqi, D. Li, R. Boutaba, and F. Cuervo. Osda: Open service discovery architecture for efficient cross-domain service provisioning. *Computer Communications*, 30(3):546–563, 2007.
16. J. Nakazawa, H. Tokuda, W. Edwards, and U. Ramachandran. A bridging framework for universal interoperability in pervasive systems. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006)*, 2006.
17. C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46 – 52, oct. 2003.
18. R. Ramdhany, P. Grace, G. Coulson, and D. Hutchison. Manetkit: supporting the dynamic deployment and reconfiguration of ad-hoc routing protocols. In *Proceedings of the ACM/IFIP/USENIX 10th international conference on Middleware*, Middleware'09, pages 1–20, Berlin, Heidelberg, 2009. Springer-Verlag.
19. M. Roman, F. Kon, and R. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, 2(5), August 2001.
20. R. Spalazzese, P. Inverardi, and V. Issarny. Towards a formalization of mediating connectors for on the fly interoperability. In *The IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, 2010.
21. S. Vinoski. It's just a mapping problem [computer application adaptation]. *Internet Computing, IEEE*, 7(3):88 – 90, may-june 2003.
22. D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19:292–333, March 1997.