Starlink: runtime interoperability between heterogeneous middleware protocols

Yérom-David Bromberg, Paul Grace, Laurent Réveillère

Starlink: runtime interoperability between heterogeneous middleware protocols

Yérom-David Bromberg
LaBRI,
University of Bordeaux, France
Email: david.bromberg@labri.fr

Paul Grace
School of Computing and Communications
Lancaster University, UK
Email: p.grace@lancaster.ac.uk

Laurent Réveillère
LaBRI,
University of Bordeaux, France
Email: Laurent.Reveillere@labri.fr

***Abstract*—Interoperability remains a challenging and growing problem within distributed systems. A range of heterogeneous network and middleware protocols which cannot interact with one another are now widely used; for example, the set of remote method invocation protocols, and the set of service discovery protocols. In environments where systems and services are composed dynamically, e.g. pervasive computing and systems-of-systems, the protocols used by two systems wishing to interact is unknown until runtime and hence interoperability cannot be guaranteed. In such situations, dynamic solutions are required to identify the differences between heterogeneous protocols and generate middleware connectors (or bridges) that will allow the systems to interoperate. In this paper, we present the Starlink middleware, a general framework into which runtime generated interoperability logic (in the form of higher level models) can be deployed to 'connect' two heterogeneous protocols. For this, it provides: i) an *abstract representation* of network messages with a corresponding generic parser and composer, ii) an engine to execute *coloured automata* that represent the required interoperability behaviour between protocols, and iii) *translation logic* to describe the exchange of message content from one protocol to another. We show through case-study based evaluation that Starlink can bridge heterogeneous protocol types. Starlink is also compared against base-line protocol benchmarks to show that acceptable performance can still be achieved in spite of the high-level nature of the solution.**

I. Introduction

A key requirement of distributed systems is that systems developed independently from one another must be able to interoperate with one another. Middleware and network protocols provide common interoperability standards to address this particular concern. However, with an ever growing set of heterogeneous protocols it remains a significant challenge to ensure that systems can interoperate when developed using different protocols e.g. different remote method invocation protocols, different messaging protocols, or different discovery protocols. Where systems are relatively static, they can connect via interoperability solutions such as software bridges [1] and Enterprise Service Buses (ESB) [17]. However even in this case, developing such interoperability solutions remains laborious and challenging; requiring not only knowledge of the protocols and low-level network programming involved, but also an understanding of the mapping of protocols onto one another or a common intermediary. In more dynamic environments where runtime composition of systems is a central property, e.g. pervasive computing or systems-of-systems, it is not possible to know in advance which protocols are required to interoperate; hence prior coded 'bridges' or 'buses' may not have been developed for a particular protocol pair. Consider the case where two mobile devices wish to spontaneously interact but are implemented upon heterogeneous protocols. In this paper we argue that to address these challenges it should be possible to reason about protocols at runtime, and then dynamically generate the interoperability bridge for that particular case.

The primary contribution of this paper is the Starlink framework, which utilizes high-level models of each individual protocol to generate interoperability bridges at runtime. These model: i) message format and content, ii) the behaviour of the protocol in terms of its message sequences, and iii) its use of underlying network transport protocols. Starlink provides a set of domain specific languages (DSL) to specify these models, whose content forms the overall interoperability logic. The Starlink framework, when deployed in the network, then executes this logic transparently from the protocols and ensures that two legacy systems can interoperate dynamically. To achieve this behaviour, we present three contributions:

- *Abstract message representations*. A domain specific language, Message Description Language (MDL), is used to describe protocol messages in terms of field content. This is used to dynamically generate message parsers and composers which correctly read and write messages to/from an abstract representation that is machine processable.
- *Coloured automata*. Each protocol is described as an automaton that represents the sequence of received or sent abstract messages. The correct concrete sending and receiving of messages requires knowledge of lower level network semantics (e.g. address, port, multicast?), these are added as annotations to the transitions that 'colour' the automata; hence, an automata for the execution of multiple protocols will have more than one colour.
- *Translation logic*. Starlink provides a translation language to model the translation from one protocol to another; this is described in two parts: i) a merged automata that represents the interoperability between two protocols in terms of the required exchange of messages, and ii) the translation logic for mapping the message content from messages in one protocol to the messages of the other.

We evaluate Starlink using a case study based approach and demonstrate that it achieves the main contribution of runtime interoperability between heterogeneous protocols. In the domain of discovery protocols we show that for three protocols (SLP [14], UPnP [10] and Bonjour[1]) Starlink is able to generate and execute the interoperability logic for each particular protocol pair. Further, we measure the performance of Starlink against benchmark measures of the individual protocols. The results show that while an obvious performance overhead is introduced it is not significant to the overall operation of the protocols.

The paper is structured as follows. In section II, we examine the state of the art in interoperability solutions and demonstrate that none is able to achieve runtime interoperability. In sections III and IV we describe the key features of the Starlink framework. We evaluate the framework in sections V and VI. Finally, we draw conclusions in Section VII.

II. Related Work

Prior efforts to achieving interoperability 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 based middleware [12], and by the W3C for Web Services based middleware [2] [6]. Where systems are designed and developed to interoperate based upon standards, these approaches have been very successful. However, when systems communicate spontaneously and without prior knowledge of one another they may not share this common agreement; it is likely they will utilise heterogeneous middleware. In such cases, such standards-based approaches will fail. Here we discuss three important interoperability patterns that seek to resolve the problems of heterogeneous middleware.

A. Interoperability Platforms

An interoperability platform is a client-side solution; it presents a common programming model to develop applications upon and then provides a dynamic substitution mechanism to insert the required protocol that the server-side employs. When the client encounters a service, it maps its application calls onto the newly substituted middleware. ReMMoC [11], Universal Interoperable Core [20] and WSIF [8] are examples of this pattern. For the particular use case, where you want a client application to interoperate with everyone else, interoperability platforms offer a powerful approach. However, these solutions rely upon a design time choice (by the client developer) to implement upon the interoperability platform. Therefore, they cannot generally allow applications developed upon two or more different legacy middleware to interoperate spontaneously at runtime.

B. Bridging

Software bridges enable communication between legacy applications deployed upon different middleware. Clients from one middleware domain can interoperate with servers in another middleware domain where the bridge acts as a one-to-one mapping between the two domains; it takes messages from a client in one format and then marshal this to the format of the other middleware; the response is then mapped to the original message format. An example is the SOAP to CORBA bridge [1]. Software bridges are not a long term solution to interoperability because they are time-consuming to develop by hand, and a large number would be required given the number of protocols in use. A subsequent advancement of this approach are Enterprise Service Buses (ESB), which offer richer and more flexible patterns to broker the communication between multiple middleware domains; they specify a service-oriented middleware with a message-oriented abstraction layer atop different messaging protocols (e.g., SOAP, JMS, SMTP). Rather than providing 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 a 1-1 mapping; ESBs offer an N-1-M mapping. Example ESBs are Artix [17] and the IBM Websphere Message Broker [16]. ESBs offer a well-established solution to the problem of middleware heterogeneity; however, a strong assumption utlised is that all messaging services can be mapped to the intermediary abstraction (which is a general subset of messaging protocols).

C. Transparent Interoperability

INDISS [3], uMiddle [19], OSDA [18], and SeDiM [9] are examples of transparent interoperability solutions which attempt to go one step further than ESBs and ensure that legacy solutions can be transparently connected. Here, protocol specific messages, behaviour and data are captured (transparently) by an interoperability framework and are then translated to an intermediary representation; a subsequent mapper then translates from the intermediary to the specific legacy middleware 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). The weakness of these platforms is in the use of the transparent intermediary: i) mappers to and from this intermediary must be developed by hand for every protocol, and ii) the intermediary is a 'subset of all protocols' and as such this subset may become too small to underpin interoperability in a general fashion.

D. Modelling

An alternative approach is to model heterogeneous systems and then use these higher-level specifications to generate bridges that allow them to interoperate. The first advocate of this philosophy being MDA (Model Driven Architecture) from the OMG [13]. This separates an application's platform

[1] http://developer.apple.com/networking/bonjour/specs.html

independent model (PIM) from its platform specific model (PSM) that describes its deployment. However, MDA only re-uses existing bridges between PSMs and cannot generate new ones from the models. Contrary to this, in our previous work, we have developed the z2z language for constructing network protocol gateways [4]. Z2z is mainly composed of a compiler and an optimized run-time system. At compile time, the developer feeds the z2z compiler with a z2z specification to generate a gateway between two heterogeneous protocols. A z2z specification is a C-like language consisting of: (i) a protocol specification, describing how the protocols interact with the network, (ii) a message specification, describing the structure of message requests and responses, and (iii) a translation specification, describing how to translate messages among protocols. Although z2z is an additional step towards interoperability, it has a main constraint: z2z generated gateways are statically built, and thus are not adequate for environments where interaction protocols remain unknown until runtime. Recently, [21] proposed a runtime solution for merging application level protocol behaviour that synthesizes code from merged automata. However, this approach does not offer a solution to capturing and translating message content and behaviour common in middleware protocols.

E. Analysis

We can take away from the state of the art that there are four important requirements in order to achieve universal interoperability between any pair of heterogeneous middleware protocols (that have the potential to interoperate).

1) *Offer a runtime solution*. Ensure that interoperability can be achieved at runtime without prior implementation of protocol specific interoperability code i.e. the solution is fully generateable at runtime.
2) *Be Transparent*. The interoperability solution must be transparent to all peers, to allow legacy platforms and future developed platforms to seamlessly interoperate.
3) *Offer rich translations*. The interoperation must allow the maximum behaviour to be transfered between two protocols, and not be restricted by a global subset.
4) *Minimize development effort*. The implementation required to translate must be minimized and the approach must be easily extensible to include future protocols

III. Starlink Models

The key philosophy behind the Starlink framework is to provide dynamic protocol interoperability by raising the level of abstraction of previous contributions [4], [21]. To this end, Starlink introduces high-level models to describe protocol messages, protocol behaviour, and protocol interoperability:

- *Abstract Messages.* Messages that flow within the network are modeled as *abstract messages* in a protocol independent manner.
- *k-Colored Automata.* The behavior of a protocol is traditionally described by an automata 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 introduces the notion of k-colored automata to capture the properties of a protocol by a color k.
- *Merged Automata.* When several protocols need to interoperate, it is necessary to express the relation among them and to describe the message translation logic, which define how to translate messages from one protocol to another. Protocol interoperability is defined in Starlink by a merged automata that describe how to combine the k-colored automata of the protocols involved.

In the rest of this section, we formally define these models and show how to use them for describing existing protocols.

A. Abstract Messages

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. The Starlink framework must extract relevant elements from the received request and use them to create one or more requests according to the target protocols. Similarly, it must extract relevant elements from the received responses and ultimately create a response according to the source protocol. 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.

In Starlink, the information derived from a network message is described in a protocol independent manner by an *abstract message*. Based on this description, the Starlink framework can dynamically understand and manipulate message elements. An abstract message consists of a set of fields, either primitive or structured. A *primitive field* is composed of a label naming the field, a type describing the type of the data content, a length defining the length in bits of the field, and the value i.e. the content of the field. A *structured field* is composed of multiple primitive fields. For example, a URL field is composed of four primitive fields: the protocol, the address, the port, and the resource location. In the model provided by Starlink to describe abstract messages, we note $msg \triangleright field$ the operation to select the field $field$ from the structured message msg.

Abstract messages represent the interface between the Starlink framework and the underlying network messages. In order to achieve interoperability dynamically, network message parsers must be created at runtime to construct abstract messages from incoming messages. Similarly, network messages must be created from the data in abstract messages and sent over the network to the interoperating parties. To perform these tasks, Starlink relies on runtime generateable message composers and parsers for each protocol. We describe in Section IV-A how these software elements are generated dynamically from specifications of message format.

Importantly, maintaining the content and structure of every message enables improved reasoning and mapping between

protocols. As opposed to other approaches such as ESBs, INDISS, OSDA and uMiddle that consider the mapping of message content to a common intermediary message representation, we do not limit interoperability to the greatest subset of behaviour for all protocols. In the case of discovery protocols for example, the greatest common divisor may be service type requests only. Therefore, interoperability between two protocols such as SLP and LDAP that both support attribute-based requests is restricted.

B. $k-$Colored Automata

We capture the behavior of protocols by a $k-$colored automaton $\mathcal{A}_k = (Q, M, q_0, F, Act, \rightarrow, \Rightarrow)$, where Q is a finite set of states, M are either incoming or outgoing abstract messages, $q_0 \in Q$ is the starting state and $F \subset Q$ is a set of accepting states. Act is a set of actions such that $Act = \{?, !\}$ where ? is the receive action and ! is the send action. $\rightarrow \subseteq Q \times Act \times M \times Q$ is the transition relation that can be either a *receive-transition* or a *send-transition*. The former has the following form $s_1 \xrightarrow{?m} s_2$ for $(s_1, ?, m, s_2) \in \rightarrow$ and changes the state of the automaton from s_1 to s_2 once the message m is received. The latter is noted $s_1 \xrightarrow{!m} s_2$ for $(s_1, !, m, s_2) \in \rightarrow$ and indicates the next state to which the automaton passes as soon as the message m is sent.

Moreover, each state maintains a queue to store both incoming and outgoing message instances. A sequence of stored messages is represented by a message vector noted $\overrightarrow{m}$. By either $s_i.m$ or $s_i.\overrightarrow{m}$, we denote a particular stored message or a sequence of stored messages from a specific state s_i. To further analyse at runtime the behavior of an automaton, we define a history operator as follows $\Rightarrow \subseteq Q \times Act \times \overrightarrow{m} \times Q$. Thus, let $\{s_1, s_2\} \in Q$, $s_1 \overset{!\overrightarrow{m}}{\Longrightarrow} s_2$ (resp. $s_1 \overset{?\overrightarrow{m}}{\Longrightarrow} s_2$) gives the sequence of the sent (resp. received) instances for each abstract message from the state s_1 to s_2.

Protocols may not only differ in their behavior but also in the the way they use 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. For instance, as illustrated in Fig. 1 and Fig. 2, although SLP and SSDP protocols are both asynchronous and multicasted, they differ from their multicast address and port number which are $239.255.255.253 : 427$ for SLP and $239.255.255.250 : 1900$ for SSDP. Note that, sent messages are not necessary received asynchronously, it depends of the underlaying network details. In order to capture these low level network semantics, we use automaton coloring which consists of assigning labels called colors to states of the automaton. An automaton can pass successfully from one state to another, following either a *receive-transition* or a *send-transition*, without any network issues, only if the concerned states share the same color. An automaton $\mathcal{A}_k$ is said to be $k-$colored if all its states are $k-$colored, and if there exists a function f such as $f(\langle (key_1, val_1), (key_2, val_2), ..., (key_n, val_n) \rangle) = k$. Function f is a perfect hash function that maps a list of tuples, where each tuple is a key-value pair describing low level network details, to a unique hash value k (i.e. without collisions). For instance, as described in Fig. {1, 2, 3}, according to their transport protocol, port, mode, multicast and group attributes, a specific and different color has been affected for the SLP, SSDP, and HTTP automata.

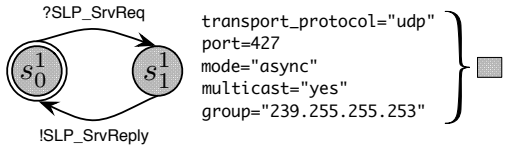

Fig. 1. SLP colored automaton

Fig. 2. SSDP colored automaton

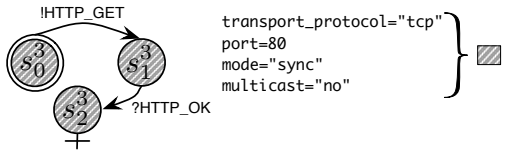

Fig. 3. HTTP colored automaton

C. Merged Automaton

Our definition of protocol interoperability requires that when one legacy system $\mathcal{L}_1$, relying on a protocol $\mathcal{P}_1$, sends a sequence of messages $\overrightarrow{m}$ to a another legacy system $\mathcal{L}_2$ that relies on a different protocol $\mathcal{P}_2$, then $\mathcal{L}_2$ must be willing to receive these messages after a set of transformations to resolve mismatches issues at both the network layer and at the message layer in terms of message format, content and sequence. However, there is a prerequisite to successfully apply these transformations: there is a need to reason about the meaning of messages that are willing to be transformed. For instance, it appears from Fig. 1, and Fig. 2 that a SSDP service should be able to understand what kind of services are requested by a SLP client if at least a `SLP_SrvReq` request is semantically equivalent to a `SSDP_Search` request. Still, a semantic equivalence is not always as simple as a one to one mapping among two messages. In the most general use case, there are several patterns of semantic mismatches hampering interoperability. A protocol $\mathcal{P}_1$ may require a single message to perform a particular task, while in another protocol $\mathcal{P}_2$, a similar task is performed by receiving several messages. Alternatively, another type of mismatch occurs if the protocol $\mathcal{P}_1$ needs to receive several messages to perform a particular task, while in the protocol $\mathcal{P}_2$ the same task is achieved with only one message. To improve interoperability among legacy systems, while ensuring simplicity in reasoning about protocols, we introduce in our model a semantic equivalence operator $\models$ such that $n \models \overrightarrow{m}$ 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 $\overrightarrow{m}$.

$$n \models \overrightarrow{m} \text{ if and only if } \forall n \triangleright field \in \mathcal{M}_{fields}(n), \quad (1)$$
$$\exists m \in \overrightarrow{m} = \langle m_1 \dots m_n \rangle |$$
$$n \triangleright field \models m \triangleright field$$

Intuitively, protocols are interoperable if there is a way to merge their respective colored automaton. Two colored automata A^1_{k1} and A^2_{k2} are mergeable if the following conditions are satisfied: (i) in A^1_{k1} there exists a state where the sequence of received messages is semantically equivalent to the required output message in the initial state of A^2_{k2}, (ii) the sequence of received messages obtained in the final state of A^2_{k2} is semantically equivalent to the required output message of one state of A^1_{k1}.

We denote the states of a $k-$colored automaton by $States(A^1_k) = \{s^1_0, ..., s^1_i, ..., s^1_n\}$, with s^1_0 corresponding to the initial state and s^1_n being one of the possible final states. Further, a $\delta-$*transition* refers to a transition from one state to another between two automata that have different colors without sending or receiving messages. In fact, $\delta-$*transitions* model actions/transformations that are required to perform at the network layer to further exchange messages from one protocol to another. Thus, formally, two automata A^1_{k1} and A^2_{k2} are said to be mergeable and is noted $A^1_{k1} \bigotimes A^2_{k2}$ if and only if there exist $\delta-$*transitions* between them. More precisely, $\delta-$*transitions* are possible between two different states iff one of the following merge constraints is satisfied:

- $\{s^1_x, s^1_i\} \in States(A^1_{k1})$ and $\{s^2_0, s^2_y\} \in States(A^2_{k2})$ such as:

$$\exists(\tau_1, \tau_2) \subseteq A^1_{k1} \times A^2_{k2} \wedge \exists i, x, y \in \{1, n-1\}| \quad (2)$$
$$\{\tau_1 = (s^1_{x, x \neq i} \xrightarrow{?m} s^1_i) \wedge \tau_2 = (s^2_0 \xrightarrow{!n} s^2_y)$$
$$\wedge (n \models s^1_0 \overset{?\overrightarrow{m}}{\Longrightarrow} s^1_i)\}$$

- $\{s^2_x, s^2_n\} \in States(A^2_{k2})$ and $\{s^1_y, s^1_j\} \in States(A^1_{k1})$ such as:

$$\exists(\tau_3, \tau_4) \subseteq A^2_{k2} \times A^1_{k1} \wedge \exists j, x, y \in \{1, n-1\}| \quad (3)$$
$$\{\tau_3 = (s^2_{x, x \neq n} \xrightarrow{?n} s^2_n) \wedge \tau_4 = (s^1_{y, y \neq j} \xrightarrow{!m} s^1_j)$$
$$\wedge (m \models s^2_0 \overset{?\overrightarrow{n}}{\Longrightarrow} s^2_x)\}$$

Thus, $A^1_{k1} \bigotimes A^2_{k2}$ is a merged automaton iff there exists two $\delta-$*transitions* such as $s^1_i \xrightarrow{\delta(\{\lambda\})} s^2_0$, $s^2_n \xrightarrow{\delta(\{\lambda\})} s^1_j$ with $\{s^1_i, s^1_j\} \in States(A_{k1})$, $\{s^2_0, s^2_n\} \in States(A^2_{k2})$ and $\{\lambda\}$ a sequence of successful transformations performed at the network layer. Given $\bigotimes$, it is straightforward to note that n protocols $\mathcal{P}_1 \dots \mathcal{P}_n$ are interoperable if their corresponding colored automata $A^1_{k1}, \dots, A^n_{kn}$ are mergeable and is noted $A^1_{k1} \bigotimes \dots \bigotimes A^n_{kn}$. However, slight subtleties occur for merging n automata: they are either strongly or weakly merged. In the former case, it means that $A^1_{k1}, \dots, A^n_{kn}$ are mergeable two by two. This merged constraint is too strong, and some protocols can still interact following a certain sequence of exchanged messages. So, in the latter case, automata are weakly mergeable iff in their resulting merged automaton this following constraint is satisfied:

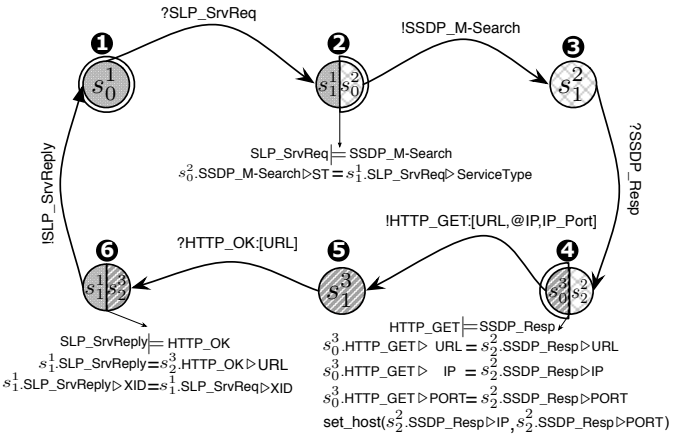

Fig. 4. A merged automaton for SLP, SSDP and HTTP protocols

$$\exists s^1_{i_1} \in States(A^1_{k1}) \dots s^n_{i_n} \in States(A^n_n) \wedge \quad (4)$$
$$\exists s \in States(A^1_{k1}) \cup States(A^n_{kn})|$$
$$\{s^1_{i_1} \xrightarrow{\delta} s^2_0, s^2_{i_2} \xrightarrow{\delta} s^3_0, \dots, s^{n-1}_{i_{n-1}} \xrightarrow{\delta} s^n_0, s^n_n \xrightarrow{\delta} s \subseteq \rightarrow\}$$

Fig. 4 illustrates a weakly merged automaton for SLP, SSDP and HTTP. States that are linked by a $\delta-$*transition* are represented by bicolored nodes such as nodes ❷, ❹ and ❻. The initial state of each automaton, which are expected to be merged, are reached by a $\delta-$*transition*. Hence, a SSDP `M_SEARCH` is sent as soon as a SLP `SrvReq` sent by a client is received. Then a HTTP `GET` request is sent at the reception of a SSDP `Resp` message to finally send back to the SLP client a SLP `SrvReply` after having received a HTTP `OK` response. In others terms, $\delta-$*transitions* enable to chain SLP, SSDP and HTTP automata through a directed path that both starts and ends in the same automaton.

An either strong or weakly merged automaton is a $\{k_1 \dots k_n\}-$colored automaton, which is an extended form of the previously introduced colored automaton (Section III-B). We note M^i the set of input or output messages, F^i the set of accepting states, and $\rightarrow^i$ the transition relation of the automaton A^i_{ki}. Hence, a merged automaton is defined as follow:

$$\mathcal{A}_{\{k_1 \dots k_n\}} = (Q, M, q_0, F, Act, \rightarrow, \Rightarrow, \xrightarrow{\delta}, \models, P),$$

where $Q = \bigcup_{i=1\dots n} States(A^i_{ki})$, $M = \bigcup_{i=1\dots n} M^i$, q_0 is a starting state from one of the merged automaton, $F \subseteq Q = \bigcup_{i=1\dots n} F^i$, and $\rightarrow = \bigcup_{i=1\dots n} \rightarrow^i$. Further, $P = \{\lambda\}$ is a set of actions/transformations to potentially apply at the network layer. $\xrightarrow{\delta} \subseteq Q_i \times P \times Q_j$, with $i \neq j \in \{1, \dots, n\}$, is the $\delta-$*transition* relation that verify either the merged

constraints (2) and (3) between two automata A^i_{ki} and A^j_{kj}. A $\delta-$*transition* may take one of the following forms: $s^i_{i_x} \xrightarrow{\delta\{\lambda\}} s^j_{j_0}$ or $s^j_n \xrightarrow{\delta\{\lambda\}} s^i_{i_y}$ for $\{(s^i_{i_x}, \{\lambda\}, s^j_{j_0}), (s^j_n, \{\lambda\}, s^i_{i_y})\} \in \xrightarrow{\delta}$ that passes either: (i) from a state $s^i_{i_x}$ of A^i_{ki} to the initial state $s^j_{j_0}$ of A^j_{kj}, or, (ii) from a final state s^j_n of A^j_{kj} to a state $s^i_{i_y}$ of A^i_{ki}. Transitions are taken after having applied a sequence $\{\lambda\}$ of actions at the network layer. An action $\lambda_i \in \{\lambda\}$ is the network function $\lambda_i : M \times \cdots \times M$ that may required as arguments some fields extracted from previously received messages stored in one state of an automaton. The operators $\Rightarrow$ and $\models$ are defined as previously.

D. Translation Logic

The role of the translation logic is to describe the translation of data and behaviour where messages are semantically equivalent. One key operator of the language is the *assignment operation*. Assignment allows the content of one or more fields of a particular message, to be translated to the fields of a different message. The format of the assignment operation is as follows:

$$s^1_i \in States(A^1_{k_1}), s^2_j \in States(A^2_{k_2}),$$
$$s^1_i.m_1 \triangleright field_a = s^2_j.m_2 \triangleright field_b \quad (5)$$
$$s^1_i.m_1 \triangleright field_a = \mathcal{T}(s^2_j.m_2 \triangleright field_b) \quad (6)$$

In (5) the message m_2 is retrieved from the queue at state s^2_j and the content of field $field_b$ is then assigned to the field $field_a$ of the message m_1 at the state s^1_i, if fields are of the same type. Otherwise, in situations where the content is not directly equivalent e.g. the same type, (6) defines a translation function $\mathcal{T}$ that takes the content $field_b$ and translates it, the result being assigned to the $field_a$ field. Fig. 4 shows examples of how the translation logic is applied at the bridging states between two protocols. At node ❷ (the first bridge between SLP and SSDP), as `SrvReq` $\models$ `M-Search`, we assign the `ST` field of the SSDP `M_SEARCH` message with the `ServiceType` field from the received SLP `SrvReq` message. Similarly, at node ❻, we assign the resulting URL from the received `HTTP_OK` message to the URL field of the SLP `SrvReply` message to be sent. As illustrated in Fig. 5, the overall translation logic to merged automata SLP, SSDP and HTTP is divided intro three distinct parts. One part (lines 1-3) specifies messages that are semantically equivalent according to (1). Another part (lines 4-9) gives the different fields assignment to perform in different states. The last part (lines 10-12) defines the required $\delta-$*transitions* to merge automata according to merged constraints (2) and (3). $\delta-$*transitions* are also used to define additional behaviour required by the automata that is guided by the content of messages. For example, in order to connect to a HTTP server to perform a GET message request in Fig. 4 we need to know the address and port. However, this information is only obtained from the content of the `SSDP_Resp` message. Thus, (line 11) the $\delta-$*transitions* uses as a λaction a keyword operator *setHost* that takes the fields host and port from the message and sends them to the underlying network engine to point the next tcp connection.

1 `SSDP_M-Search` $\models$ `SLP_SrvReq`
2 `HTTP_GET` $\models$ `SSDP_Resp`
3 `SLP_SrvReply` $\models$ `HTTP_OK`
4 s^2_0.`SSDP_M-Search` $\triangleright$ `ST` = s^1_1.`SLP_SrvReq` $\triangleright$ `ServiceType`
5 s^3_0.`HTTP_GET` $\triangleright$ `URL` = s^2_2.`SSDP_Resp` $\triangleright$ `URL`
6 s^3_0.`HTTP_GET` $\triangleright$ `IP` = s^2_2.`SSDP_Resp` $\triangleright$ `IP`
7 s^3_0.`HTTP_GET` $\triangleright$ `PORT` = s^2_2.`SSDP_Resp` $\triangleright$ `PORT`
8 s^1_1.`SLP_SrvReply` $\triangleright$ `URL` = s^3_2.`HTTP_OK` $\triangleright$ `URL_BASE`
9 s^1_1.`SLP_SrvReply` $\triangleright$ `XID` = s^1_1.`SLP_SrvReq` $\triangleright$ `XID`
$s^1_1 \xrightarrow{\delta} s^2_0$
10 $s^2_2 \xrightarrow{\delta\{set\_host(s^2_2.\texttt{HTTP\_GET}\triangleright\texttt{IP}, s^2_2.\texttt{HTTP\_GET}\triangleright\texttt{PORT})\}} s^3_0$
12 $s^3_2 \xrightarrow{\delta} s^1_1$

Fig. 5. Specification to merge the SLP, SSDP and HTTP automata

IV. The Starlink Framework

In this section we describe how the Starlink software framework[2] concretely applies the models from Section III in order to achieve runtime interoperability. A high level illustration of the architecture is given in Fig. 6. The framework is composed of general software elements that are specialised by models; a process that can be executed dynamically. The key elements of this architecture are:

- *Message Composers and Parsers*, which read and write messages specific to a legacy protocol. A Message Description Language (MDL) specification (as described in Section IV-A) is loaded into composers and parsers to specialise these components at runtime.
- The *Automata Engine* executes the behaviour of the merged automata i.e. it controls the sequence of sending, receiving and translation of messages.
- The *Network Engine* receives messages from the network and sends messages based upon the protocol properties provided by the Automata Engine. Further discussion of the two behaviour engines is provided in IV-B.

To illustrate this framework in action consider interoperation between two protocols, e.g. SLP and Bonjour. An SLP MDL would specialise a message composer and parser component (similarly for the Bonjour MDL at the other side of the connection). A merged automata for SLP and Bonjour would then specialise the Automata Engine. The framework is transparently deployed in the network, and neither application is aware that it is communicating with a heterogeneous protocol.

A. Message Description Language

Starlink employs a simple mechanism to communicate with legacy protocols. Parsers read messages sent by legacy protocols (these are received as an array of bytes from the network engine) and then transform these to the *abstract message representation*. Composers do the reverse to create legacy

[2] `http://starlink.sourceforge.net`

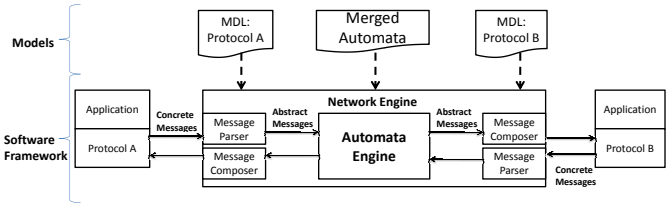

Fig. 6. Architecture of the Starlink framework

protocol messages as a byte array and transmit these using the network engine. In the framework, composers and parsers are implemented as general interpreters that execute the message description language (MDL) specifications that are loaded. For example, a parser that interprets an SLP MDL instance will parse SLP messages into the abstract message representation; concretely, this is a Java object which conforms to an XML schema of the abstract message representation (as described in Section III-A). Note, this conformance to the schema allows XPath expressions (see section IV-B to be used to read and write field values from and to these objects respectively). This process is based upon the similar idea of generating parsers for common transfer syntax approaches (e.g., Common Data Representation in GIOP from IDL). However, this is the reverse–we are modelling and generating parsers for multiple transfer syntax into a language (in this case Java) that can then execute the translations.

The Starlink framework is flexible to allow different types of language to be used to specify message formats; each language can be termed an 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. To illustrate the approach we present a language for binary messages here. Fig. 7 shows part of the MDL specification for the SLP protocol (which employs binary messages). In this specification there are two important constructs: <*Types*> and <*Header*>.

Element <*Types*> specifies the types used by message fields, and the format of each entry is <*label*>*type*</*label*> (lines 1-6); the *label* is the name for a field, and the *type* entry states the type of the data. To underpin the reading and writing of data from messages, Starlink employs pluggable marshallers and unmarshallers for each of the types. For example, Integer has a plug-in marshaller that writes Java Integers to a byte array, and a corresponding unmarshaller that transforms a byte array to a Java Integer. This mechanism allows the language to be dynamically extended to incorporate complex types (with no need to re-implement a compiler). For example the `FQDN` type is a format for domain names (Fully Qualified Domain Name); to add the `FQDN` type to this language, we simply plug-in marshallers that map `FQDN` byte arrays to a Java String. The final feature of the types are functions. Functions can be defined on types using the [f-method()] construct e.g. `[f-length(URLEntry)]` (line 5); the named f-method is executed by the marshaller when writing the type. Importantly

```
1  <Types>
2    <Version>Integer</Version>
3    ...
4    <URLEntry>String</URLEntry>
5    <URLLength>Integer[f-length(URLEntry)]</URLLength>
6  </Types>
7  ...
8  <Header type=SLP>
9    <Version>8</Version>
10   <FunctionID>8</FunctionID>
11   <MessageLength>24</MessageLength>
12   <reserved>16</reserved>
13   <NextExtOffset>24</NextExtOffset>
14   <XID>16</XID>
15   <LangTag>LangTagLen</LangTag>
16 </Header>
17 ...
18 <Message type=SLPSrvRequest>
19   <Rule>FunctionID=1></Rule>
20   <PRLength>16<PRLength>
21   <PRStringTable>PRLength</PRStringTable>
22   <SRVTypeLength>16</SRVTypeLength>
23   <SRVType>SRVTypeLength</SRVType>
24   <PredLength>16</PredLength>
25   <PredString>PredLength</PredString>
26   <SPILength>16</SPILength>
27   <SPIString>SPILength</SPIString>
28 </Message>
29 ...
```

Fig. 7. MDL specification of the SLP service

the function can take other message fields as parameters e.g. to compose the value of the `URLLength` field, you need to obtain the length of the URLEntry field. Hence, the marshaller takes the value to be written to the `URLEntry` field, calculates the length and then composes this as the `URLLength` value.

Elements <*Header*> (line 8) and <*Message*> (line 18) specify the content of the message headers and bodies. These are both composed of <*label*>*size*</*label*> entries for each field in the message. The *size* is the length of the field content in bits. There is one special label: <*rule*>*case*</*rule*>. This is used to relate the correct message body with the header. For example, the SLP `SrvRequest` message applies when the `FunctionID` field in the header equals one (line 19).

B. Automata Engine

The Automata Engine, like the message composers and parsers, interprets a loaded runtime model. In the case of the automata engine this is the behaviour model that describes how two protocols interoperate, which is composed of the i) merged automata from Section III-C,which defines the sequence of state transitions, and ii) the translation logic which specifies how the data fields are translated at particular states as stated in Section III-D. This engine is implemented to read these models from XML content (however, the notation provided in Section III offers a concise presentation, and hence we do not reproduce the XML equivalents here). To give a flavour of the concrete operation of the framework we briefly summarise the behaviour that occurs at the different state types: *receiving, sending, and no-action*.

At a **receiving state** $R1$, the automata engine listens for messages using the network engine for the protocol address and port (of the state); when a message is received it is parsed by the message parser. If the abstract message's name label

```
1  <Bridge>
2    ...
3    <TranslationLogic>
4      <Assignment>
5        <Field>
6          <Message>SSDP_Search</Message>
7          <Xpath>/field/primitiveField[label='ST']/value</Xpath>
8        </Field>
9        <Field>
10         <Message>SLPSrvRequest</Message>
11         <Xpath>/field/primitiveField[label='SRVType']/value</Xpath>
12       </Field>
13     </Assignment>
14   </TranslationLogic>
15   ...
16 </Bridge>
```

Fig. 8. Translation logic expressed in XML

matches one of the transition labels then the automata moves to the pointed to state $S1$, and then pushes the abstract message (Java object) onto the message queue at $R1$.

At a **sending state** $S1$, the automata engine reads the label of the transition and then constructs this message using a message composer before using the network engine to send it correctly with the required network transport semantics of the protocol. In the case where content has been translated by a prior state, the state $S1$ retrieves the message to be sent from the queue of a prior state before composing and sending.

A **bridge state** $B1$, represents an intermediary state from the bi-coloured states (e.g. ❷, ❹ and ❻ in Fig. 4). These states do not send and receive messages, they only translate content from one abstract message to another or perform logic required to underpin interoperability. The XML content in Fig. 8 shows an example of such a state (for the SLP `SrvReq` to SSDP `M-SEARCH` translation), presenting only the translation logic. For field assignments, the engine reads the value from the second field (as pointed to by the XPath expression), this equates to reading the value from the Java object of the abstract message, and then writes the content to the abstract message whose field is pointed to by the first field node of the XML.

V. Case Study Evaluation

We use a case-study approach to evaluate the ability of the framework to achieve its primary contribution, i.e., to ensure interoperability between heterogeneous communication protocols. We hypothesize that we can create a connector between two protocols at runtime using only high-level models of these communication protocols i.e., there is no implementation or deployment of legacy code that is specific to the behaviour of an individual protocol. We concentrate on one domain of protocols, namely, service discovery (this offers an ideal first case due to the semantic similarities between the protocols) and from this we select three protocols: SLP, UPnP and Bonjour. For these, we then developed simple legacy applications to lookup a simple test service, and respond to lookup requests for the simple service. For SLP we used the OpenSLP protocol implementation[3]; for UPnP we used the Cyberlink Java implementation[4]; and for Bonjour we employed the Apple Bonjour SDK for Windows[5].

The objective of the case study is to develop Starlink models for each of the protocols such that we can take the legacy applications implemented upon the three protocols and ensure they can interoperate with one another i.e. that an SLP application's lookup request can be answered by either a UPnP service or a Bonjour Service by deploying the Starlink framework in the network. There are six particular cases i.e. SLP to UPnP and Bonjour, UPnP to SLP and Bonjour, and Bonjour to SLP and UPnP. For each case, the legacy lookup application received a response to the lookup request from the heterogeneous protocol. For conciseness, we discuss only two cases in detail:

- *SLP to Bonjour*. These two protocols are both binary protocols and their message sequences are similar. They differ in message content and network addresses.
- *SLP to UPnP*. In this case, there is heterogeneity of the protocol messages and the behaviour message sequence. SLP employs binary messages, while UPnP uses text-based messages. SLP is a simple request response, whereas UPnP involves multiple requests to the service.

A. SLP to Bonjour

To provide a system enabling interoperability from SLP to Bonjour, we create five different specifications that are loaded into the Starlink framework: i) an MDL specification of SLP messages as previously illustrated in Fig. 7, ii) an MDL specification of Bonjour messages (Bonjour uses DNS messages so this MDL describes DNS questions and responses), iii) a coloured automaton of SLP (see Fig. 1), iv) a coloured automaton of Bonjour as shown in Fig. 9, and v) a merged automaton as shown in Fig. 10.

Once the aforementioned specifications have been loaded into Starlink, SLP `SrvReq` messages generated from the client, to perform a service lookup, are captured on address $239.255.255.253 : 427$ by Starlink that dynamically applies the translation logic. Particularly, it translates captured messages to abstract messages to extract thereafter their service type field `SrvType` in the aim of assigning their value into the `DomainName` field of DNS questions (See Fig. 10,❷) that are then multicasted to address $224.0.0.251 : 5353$ (See Fig. 10,❸). Handlers for outgoing messages are suspended until their corresponding responses are received. Received responses are in turn translated to abstract messages enabling Starlink to compose new SLP `SRVReply` abstract messages (See Fig. 10,❹). Two of their fields are mandatory: (i) the URL reply of the service (this was transfered from the `RDATA` value of the DNS Response), and (ii) the `XID` field (which is the `XID` field of the original request message, this was retrieved from the message at state 2's queue). At the end, a `SRVReply` is generated from its abstract representation and sent back to the SLP address that legacy client is listening on.

[3]http://www.openslp.org/

[4]http://www.cybergarage.org

[5]http://developer.apple.com/opensource/

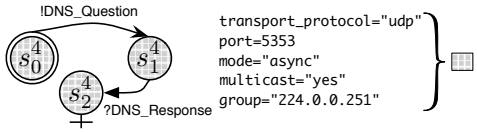

Fig. 9. mDNS colored automaton

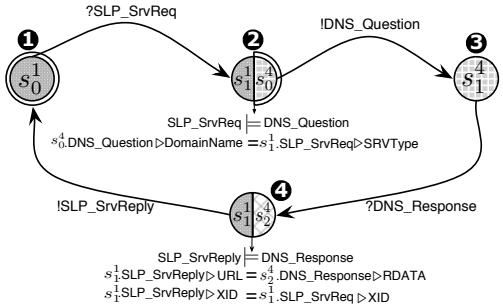

Fig. 10. A merged automaton for SLP and mDNS protocols

```
1  <Types>
2    <Method>String<Method>
3    <URI>String<URI>
4    <Version>String<Version>
5    <ST>String<ST>
6    <MX>Integer<MX>
7    ...
8  </Types>
9
10 <Header type=SSDP>
11   <Method>32</Method>
12   <URI>32</URI>
13   <Version>13,10</Version>
14   <Fields>13,10:58</Fields>
15 </Header>
16
17 <Message type=SSDP_M−Search>
18   <Rule>Method=M-SEARCH</Rule>
19 </Message>
20
21 <Message type=SSDP_Resp>
22   <Rule>Method=HTTP/1.1</Rule>
23 </Message>
```

Fig. 11. MDL specification of SSDP

B. SLP to UPnP

UPnP uses two protocols to perform discovery: SSDP multicasts a lookup request and receives an SSDP response. A further HTTP request is then needed to retrieve the URL of the service from this device. Hence, in this case seven models were loaded into the framework: i) the SLP MDL, ii) the SLP coloured automaton, iii) the SSDP MDL, iv) the SSDP coloured automaton, v) the HTTP MDL, vi) the HTTP coloured automaton, and vii) the merged automaton for the three protocols. An important difference here is that SSDP and HTTP are text messages and as such require a different MDL and corresponding parser and composer. Fig. 11 shows the SSDP MDL, this identifies the general boundaries of fields "e.g.\r\n" (chars 13,10) because there is no fixed layout or ordering of fields. The inner field boundary (e.g. the ':' split - char 58) then takes the field label from the left and the field value from the right to build a field in the abstract message. When the SLP client was executed, the merged automaton successfully sent an SLP `SrvReply` message composed of the content from the SSDP and HTTP fields.

C. Analysis

Originally we stated that we required the following from the Starlink framework:

- We require transparent interoperability. In the case study, the legacy protocols are implemented and deployed independently of the Starlink, they are never aware of the framework and hence the case studies show that transparent interoperability has been achieved.
- Offer rich translations. The case studies show that we can translate correctly between three different protocols that are heterogeneous in terms of protocol sequences, e.g. SLP compared to UPnP, and heterogeneous in terms of message content e.g. binary to text messages.
- Minimise development effort. In the cases we only need to provide high-level models, there is no low-level programming. Further, we are able to reuse the models across the cases i.e. we only need to model SLP once and then write only the merged automata for the particular case (typically, these automata are around 100 lines of XML, but this depends on the complexity of the translation).

VI. Performance Evaluation

Starlink performances are evaluated by investigating the time taken to perform interoperability translation. We then compare this to the typical responsiveness of discovery protocols in terms of the time taken to return a service reply to a lookup request. Fig. 12(a) shows the results of the bench measures of the individual protocols (these are measures of the legacy applications implemented using OpenSLP for SLP, the Apple Windows SDK for Bonjour and Cyberlink for UPnP). To obtain the measures, we calculated the time from when the client sent the message until the response was received. For each case, we repeated the experiment 100 times and took the min, max, median of these results. All experiments were performed with the client and the service on the same machine (3 Ghz CPU, 2Gb memory running Windows Vista Operating System, the Java VM was version 1.6.2) to avoid measuring additional network latency, which may not be constant.

Subsequently, we measured the time taken to translate from one protocol to another within the Starlink framework. This measures the time from when the message was first received by the framework until the translated output response was sent on the output socket. Fig 12(b) shows these measures. We can see from the results that there is a significant but varied expense to additional translation: in case 6 it is approximately a 600 percentage increase in response time, while in case 1 it is 5 percent. This is because the cost of translation is bounded by the response of the legacy protocols; if SLP takes 6 seconds to respond that is added to the translation. However, in the domain of service discovery protocols the timeout of the

Response time measures for legacy discovery protocols

Protocol	Min (ms)	Median (ms)	Max (ms)
SLP	5982	6022	6053
Bonjour	687	710	726
UPnP	945	1014	1079

(a)

Translation times of Starlink connectors

Case	Min(ms)	Median (ms)	Max (ms)
1. SLP to UPnP	319	337	343
2. SLP to Bonjour	255	271	287
3. UPnP to SLP	6208	6311	6450
4. UPnP to Bonjour	253	289	311
5. Bonjour to UPnP	334	359	379
6. Bonjour to SLP	6168	6190	6244

(b)

Fig. 12. Native service discovery vs. Starlink (ms)

request response is generally in terms of seconds (OpenSLP sets the default timeout to 15 seconds, while Cyberlink does not bound the response time); all of the results are within this range and the solution is both possible and acceptable.

VII. Concluding Remarks and Future Work

In this paper we have presented the Starlink framework whose key contribution is the ability to create and deploy runtime solutions to interoperability using only high-level models of protocol behaviour. We have shown that in the domain of service discovery, Starlink is able to successfully ensure that a range of protocol types can interoperate with one another in a transparent fashion. This is done so with no need to implement or compile interoperability solutions in advance, or implement code specific to individual protocols. Finally, we have shown that the framework adds an overhead to the general communication cost; while this cost can be minimized it can never be removed because additional behaviour (translations, extra protocol messages etc.) must be executed; however, this cost remains within the bounds of expected performance from the legacy applications in the use case.

Future work aims to increase automation. At present, the merged automata with the corresponding translation logic is modelled by a developer; however, in order for it to be a true runtime solution this model should be generated by the framework itself. That is, the framework will reason about the MDLs and the coloured automata of multiple protocols and then generate the correct merge to achieve interoperability. For this we see two important directions:

- *Ontology integration*. To reason about and generate the translation of field content between messages we believe that additional semantic models can be used to infer the translation logic. For this we are investigating the use of ontologies [7] and their associated tools. Here ontologies describing two protocols would be reasoned upon and the semantic matches would be inferred, i.e., the fields where data can be translated.
- *Learning*. 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 [5] and learning algorithms have been utilised to learn the interaction behaviour of protocols [15]. We hope to build upon these techniques in order to learn both MDLs and coloured automata for protocols.

Acknowledgments.

This work was in part funded under the European FP7 ICT FET Connect project (http://connect-forever.eu/).

References

[1] Soap2corba and corba2soap. http://soap2corba.sourceforge.net/.
[2] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture, February 2004.
[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] Y.-D. Bromberg, L. Réveillère, J. L. Lawall, and G. Muller. Automatic generation of network protocol gateways. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, pages 21–41, Urbana Champaign, IL, USA, 2009. Springer-Verlag New York, Inc.
[5] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn 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.
[6] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. In *http://www.w3.org/TR/wsdl*, March 2001.
[7] Michael C. Daconta, Leo J. Obrst, and Kevin T. Smith. *The Semantic Web: A Guide to the Future of XML, Web Services and Knowledge Management*. Wiley, Indianapolis, IN, 2003.
[8] M. Duftler, N. Mukhi, S. Slominski, and S. Weerawarana. Web services invocation framework (wsif). In *OOPSLA 2001 Workshop on Object Oriented Web Services*, 2001.
[9] C. Flores, G. Blair, and P. Grace. An adaptive middleware to overcome service discovery heterogeneity in mobile ad hoc environments. *IEEE Distributed Systems Online*, 2007.
[10] UPnP Forum. Upnp device architecture version 1.0. http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0.pdf, October 2008.
[11] 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.
[12] Object Management Group. The common object request broker: Architecture and specification version 2.0. Technical report, 1995.
[13] Object Management Group. Model driven architecture (mda), document number ormsc/2001-07-01. Technical report, 2001.
[14] E. Guttman, C. Perkins, and J. Veizades. Service location protocol version 2, IETF RFC 2608. http://www.ietf.org/rfc/rfc2608.txt, June 1999.
[15] Falk Howar, Bengt Jonsson, Maik Merten, Bernhard Steffen, and Sofia Cassel. On handling data in automata learning - considerations from the connect perspective. In *ISoLA (2)*, pages 221–235, 2010.
[16] IBM. Websphere message broker. www.ibm.com/websphere/wbimessagebroker.
[17] IONA. Artix esb. [online]. http://www.iona.com/products/artix/, 2007.
[18] 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.
[19] 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.
[20] M. Roman, F. Kon, and R. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, 2(5), August 2001.
[21] 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 (WICSA/ECSA)*, 2010.