Machine Learning for Emergent Middleware

Amel Bennaceur and **Valérie Issarny** and **Daniel Sykes**[1] and **Falk Howar** and **Bernhard Steffen**[2] and **Richard Johansson** and **Alessandro Moschitti**[3]

Abstract. Highly dynamic and heterogeneous distributed systems are challenging today's middleware technologies. Existing middleware paradigms are unable to deliver on their most central promise, which is offering interoperability. In this paper, we argue for the need to dynamically synthesise distributed system infrastructures according to the current operating environment, thereby generating "Emergent Middleware" to mediate interactions among heterogeneous networked systems that interact in an *ad hoc* way. The paper outlines the overall architecture of Enablers underlying Emergent Middleware, and in particular focuses on the key role of learning in supporting such a process, spanning statistical learning to infer the semantics of networked system functions and automata learning to extract the related behaviours of networked systems.

1 INTRODUCTION

Interoperability is a fundamental property in distributed systems, referring to the ability for two or more systems, potentially developed by different manufacturers, to work together. Interoperability has always been a challenging problem in distributed systems, and one that has been tackled in the past through a combination of middleware technologies and associated bridging solutions. However, the scope and level of ambition of distributed systems continue to expand and we now see a significant rise in complexity in the services and applications that we seek to support.

Extreme distributed systems challenge the middleware paradigm that needs to face on-the-fly connection of highly heterogeneous systems that have been developed and deployed independently of each other. In previous work, we have introduced the concept of *Emergent Middleware* to tackle the extreme levels of heterogeneity and dynamism foreseen for tomorrow's distributed systems [12, 4].

Emergent Middleware is an approach whereby the necessary middleware to achieve interoperability is not a static entity but rather is generated dynamically as required by the current context. This provides a very different perspective on middleware engineering and, in particular requires an approach that create and maintain the models of the current networked systems and exploit them to reason about the interaction of these networked system and synthesise the appropriate artefact, i.e., the emergent middleware, that enable them to interoperate. However, although the specification of system capabilities and behaviour have been acknowledged as fundamental elements of system composition in open networks, especially in the context of the Web [8, 15]), it is rather the exception than the norm to have such rich system descriptions available on the network.

[1] INRIA, France, email: first.last@inria.fr
[2] Technical University of Dortmund, email: {falk.howard, steffen}@tu-dortmund.de
[3] University of Trento, email: moschitti@disi.unitn.it

This paper focuses on the pivotal role of learning technologies in supporting Emergent Middleware, including in building the necessary semantic run-time models to support the synthesis process and also in dealing with dynamism by constantly re-evaluating the current environment and context. While learning technologies have been deployed effectively in a range of domains, including in Robotics [25], Natural Language Processing [18], Software Categorisation [24], Model-checking [21], Testing [11], and Interface Synthesis [2], and Web service matchmaking [14], this is the first attempt to apply learning technologies in middleware addressing the core problem of interoperability.

This work is part of a greater effort within the CONNECT project[4] on the synthesis of Emergent Middleware for GMES-based systems that are representative of Systems of Systems. GMES[5] (Global Monitoring for Environment and Security) is the European Programme for the establishment of a European capacity for Earth Observation started in 1998. The services provided by GMES address six main thematic areas: land monitoring, marine environment monitoring, atmosphere monitoring, emergency management, security and climate change. The emergency management service directs efforts towards a wide range of emergency situations; in particular, it covers different catastrophic circumstances: Floods, Forest fires, Landslides, Earthquakes and volcanic eruptions, Humanitarian crises.

For our experiments, we concentrate on joint forest-fire operation that involves different European organisations due to, e.g., the crossboarder location or criticality of the fire. The target GMES system involves highly heterogeneous NSs, which are connected on the fly as mobile NSs join the scene. Emergent Middleware then need to be synthesised to support such connections when they occur. In the following, we more specifically concentrate on the connection with the Weather Station NS, which may have various concrete instances, ranging from mobile stations to Internet-connected weather service. In addition, Weather Station NSs may be accessed from heterogeneous NSs, including mobile handheld devices of the various people on site and Command and Control —C2— centres (see Figure 1). We show how the learning techniques can serve complementing the base interface description of the NS with appropriate functional and behavioural semantics. It is in particular shown that the process may be fully automated, which is a key requirement of the Emergent Middleware concept.

2 EMERGENT MIDDLEWARE

Emergent Middleware is synthesised in order to overcome the interoperability issue arising from two independently-developed Networked Systems (NSs). Given two Networked Systems where one

[4] http://connect-forever.eu/
[5] http://www.gmes.info

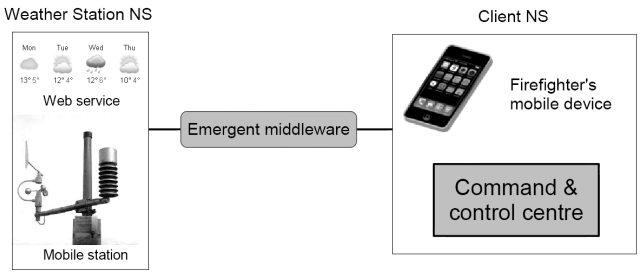

Figure 1. Heterogeneous Connections with Weather Station NSs

implements the functionality required by the other, an Emergent Middleware that mediates application- and middleware-layer protocols implemented by the two NSs is deployed in the networked environment, based on the run-time models of the two NSs and provided that a protocol mediator can indeed be computed. The following section defines the NS model we use to represent the networked systems and reason about their interoperation. Then we present the by *Enablers*, i.e., active software entities that collaborate to realise the Emergent Middleware ensuring their interoperation.

2.1 Networked System Model

The definition of NS models takes inspiration from system models elaborated by the Semantic Web community toward application-layer interoperability. As depicted on Figure 2.(a), the NS model then decomposes into:

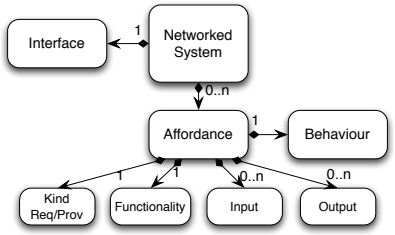

Figure 2. The Networked System (NS) Model

- *Interface*: The NS interface provides a microscopic view of the system by specifying fine-grained *actions* (or methods) that can be performed by (i.e., external action required by NS in the environment for proper functioning) and on (i.e., actions provided by the given NS in the networked environment) NS.
  There exist many interface definition languages and actually as many languages as middleware solutions. In our approach, we use a SAWSDL-like[6] XML schema. In particular, a major requirement is for interfaces to be annotated with ontology concepts so that the semantics of embedded actions and related parameters can be reasoned about.
- *Affordances*: The affordances (*a.k.a. capabilities* in OWL-S [15]) describe the high-level roles an NS plays, e.g., weather station, which are implemented as protocols over the system's observable actions (i.e., actions specified in the NS interface). The specification of an affordance decomposes into:
  - The *ontology-based semantic characterisation* of the high level *Functionality* implemented by the affordance, which is given in terms of the ontology concepts defining the given functionality and of the associated *Input* and *Output*. An affordance is further either *requested* or *provided* by the NS in the networked environment. In the former case, the NS needs to access a remote NS providing the affordance for correct operation; in the latter, the NS may be accessed for the implementation of the given affordance by a remote NS.
  - The affordance's *behaviour* describes how the actions of the interface are co-ordinated to achieve the system's given affordance. Precisely, the affordance behaviour is specified as a process over actions defined in the interface, and is represented as a Labelled Transition System (LTS).

[6] http://www.w3.org/2002/ws/sawsdl/spec/

2.2 Emergent Middleware Enablers

In order to produce an Emergent Middleware solution, an architecture of Enablers is required that executes the Emergent Middleware lifecycle. An Enabler is a software component that executes a phase of the Emergent Middleware, co-ordinating with other Enablers during the process.

The Emergent Middleware Enablers are informed by *domain ontologies* that formalise the concepts associated with the application domains (i.e., the vocabulary of the application domains and their relationship) of interest. Three challenging *Enablers* must then be comprehensively elaborated to fully realise Emergent Middleware:

1. The *Discovery Enabler* is in charge of discovering the NSs operating in a given environment. The *Discovery Enabler* receives both the advertisement messages and lookup request messages that are sent within the network environment by the NSs using legacy discovery protocols (e.g., SLP[7]) thereby allowing the extraction of basic NS models based on the information exposed by NSs, i.e., identification of the NS interface together with middleware used for remote interactions. However, semantic knowledge about the NS must be learned as it is not commonly exposed by NSs directly.
2. The *Learning Enabler* specifically enhances the model of discovered NSs with the necessary functional and behavioural semantic knowledge. The *Learning Enabler* uses advanced learning algorithms to dynamically infer the ontology-based semantics of NSs' affordances and actions, as well as to determine the interaction behaviour of an NS, given the interface description exposed by the NS though some legacy discovery protocol. As detailed in subsequent sections, the Learning Enabler implements both statistical and automata learning to feed NS models with adequate semantic knowledge, i.e., functional and behavioural semantics.
3. The *Synthesis Enabler* dynamically generates the software (i.e., Emergent Middleware) that mediates interactions between two legacy NS protocols to allow them to interoperate. In more detail, once NS models are complete, initial semantic matching of two affordances, that are respectively provided and required by two given NSs, may be performed to determine whether the two NSs are candidates to have an Emergent Middleware generated between them. The semantic matching of affordances is based on the subsumption relationship possibly holding between the concepts defining the functional semantics of the compared affordances.
   Given a functional semantic match of two affordances, the affordances' behaviour may be further analysed to ultimately generate

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

a mediator in case of behavioural mismatch. It is the role of the *Synthesis Enabler* to analyse the behaviour of the two affordances and then synthesise—if applicable—the mediator component that is employed by the Emergent Middleware to enable the NSs to coordinate properly to realise the given affordance. For this, the Synthesis Enabler performs automated behavioural matching and mapping of the two models. This uses the ontology-based semantics of actions to say where two sequences of actions in the two behaviours are semantically equivalent; based upon this, the matching and mapping algorithms determine a LTS model that represents the mediator. In few words, for both affordance protocols, the mediator LTS defines the sequences of actions that serve to translate actions from one protocol to the other, further including the possible re-ordering of actions.

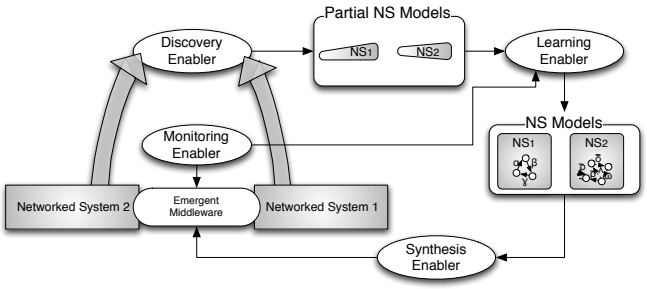

Figure 3. The Enablers supporting Emergent Middleware

The Learning phase is a continuous process where the knowledge about NSs is enriched over time, thereby implying that Emergent Middleware possibly needs to adapt as the knowledge evolves. In particular, the synthesised Emergent Middleware is equipped with monitoring probes that gather information on actual interaction between connected systems. This observed *Monitoring Data* is delivered to the Learning Enabler, where the learned hypotheses about the NSs' behaviour are compared to the observed interactions. Whenever an observation is made by the monitoring probes that is not contained in the learned behavioural models, another iteration of learning is triggered, yielding refined behavioural models. These models are then used to synthesise and deploy an evolved Emergent Middleware.

3 MACHINE LEARNING: A BRIEF TAXONOMY

Machine learning is the discipline that studies methods for automatically inducing functions (or system of functions) from data. This broad definition of course covers an endless variety of subproblems, ranging from the least-squares linear regression methods typically taught at undergraduate level [19] to advanced structured output methods that learn to associate complex objects in the input [16] with objects in the output [13] or methods that infer whole computational structures [9]. To better understand the broad range of machine learning, one must first understand the conceptual differences between learning setups in terms of their prerequisites:

- *Supervised learning* is the most archetypical problem setting in machine learning. In this setting, the learning mechanism is provided with a (typically finite) set of labelled examples: a set of pairs $T = \{(x, y)\}$. The goal is to make use of the example set T to induce a function f, such that $f(x) = y$, for future unseen instances of (x, y) pairs (see for example [19]). A major hurdle in applying supervised learning is the often enormous effort of labelling the examples.
- *Unsupervised learning* lowers the entry hurdle for application by requiring only unlabelled example sets, i.e., $T = \{x\}$. In order to be able to come up with anything useful when no supervision is provided, the learning mechanism needs a bias that guides the learning process. The most well-known example of unsupervised learning is probably k-means clustering, where the learner learns to categorise objects into broad categories even though the categories were not given a priori. Obviously, the results of unsupervised learning cannot compete with those of supervised learning.
- *Semi-supervised learning* is a pragmatic compromise. It allows one to use a combination of a small labelled example set $T_s = \{(x, y)\}$ together with a larger unlabelled example set $T_u = \{x\}$ in order to improve on both the plain supervised learner making use of T_s only and the unsupervised learner using all available examples.
- *Active learning* puts the supervisor in a feedback loop: whenever the (active) learner detects a situation where the available test set is inconclusive, the learner actively constructs complementing examples and asks the supervisor for the corresponding labelling. This learning discipline allows a much more targeted learning process, since the active learner can focus on the important/difficult cases (see for example [5]). The more structured the intended learning output is, the more successful active learning will be, as the required structural constraints are a good guide for the active construction of examples [3]. It has been successfully used in practice for inferring computational models via testing [10, 9].

Learning technology has applicability in many domains. The next sections concentrate on the learning-based techniques that we are developing to enable the automated inference of semantic knowledge about Networked Systems, both functional and behavioural. The former relies on *statistical learning* while the latter is based on *automata learning*.

4 STATISTICAL LEARNING FOR INFERRING NS FUNCTIONAL SEMANTICS

As discussed in Section 2.2, the first step in deciding whether two NSs will be able to interoperate consists in checking the compatibility of their *affordances* based on their functional semantics (i.e., ontology concepts characterising the purpose of the affordance). Then, in the successful cases, behavioural matching is performed so as to synthesise required mediator. This process highlights the central role of the functional matching of affordances in reducing the overall computation by acting as a kind of filter for the subsequent behavioural matching. Unfortunately, legacy applications do not normally provide affordance descriptions. We must therefore rely upon an engineer to provide them manually, or find some automated means to extract the probable affordance from the interface description. Note that it is not strictly necessary to have an absolutely correct affordance since falsely-identified matches will be caught in the subsequent detailed checks.

Since the interface is typically described by textual documentation, e.g., XML documents, we can capitalise on the long tradition of research in *text categorisation*. This studies approaches for automatically enriching text documents with semantic information. The

latter is typically expressed by topic categories: thus text categorisation proposes methods to assign documents (in our case, interface descriptions) to one or more categories. The main tool for implementing modern systems for automatic document classification is machine learning based on vector space document representations.

In order to be able to apply standard machine learning methods for building categorizers, we need to represent the objects we want to classify by extracting informative *features*. Such features are used as indications that an object belongs to a certain category. For categorisation of documents, the standard representation of features maps every document into a vector space using the *bag-of-words* approach [23]. In this method, every word in the vocabulary is associated with a dimension of the vector space, allowing the document to be mapped into the vector space simply by computing the occurrence frequencies of each word. For example, a document consisting of the string "get Weather, get Station" could be represented as the vector $(2, 1, 1, \ldots)$ where, e.g., 2 in the first dimension is the frequency of the "get" token. The bag-of-words representation is considered the standard representation underlying most document classification approaches. In contrast, attempts to incorporate more complex structural information have mostly been unsuccessful for the task of categorisation of single documents [20] although they have been successful for complex relational classification tasks [17].

However, the task of classifying interface descriptions is different from classifying raw textual documents. Indeed, the interface descriptions are *semi-structured* rather than unstructured, and the representation method clearly needs to take this fact into account, for instance, by separating the vector space representation into regions for the respective parts of the interface description. In addition to the text, various semi-structured identifiers should be included in the feature representation, e.g., the names of the method and input parameters defined by the interface. The inclusion of identifiers is important since: (*i*) the textual content of the identifiers is often highly informative of the functionality provided by the respective methods; and (*ii*) the free text documentation is not mandatory and may not always be present.

For example, if the functionality of the interface are described by an XML file written in WSDL, we would have tags and structures, as illustrated by the text fragment below, which relates to a NS implementing a weather station and is part of the GMES scenario detailed in the next section on experiments:

```
<wsdl:message name="GetWeatherByZipCodeSoapIn">
    <wsdl:part name="parameters"
        element="tns:GetWeatherByZipCode" />
</wsdl:message>
 <wsdl:message name="GetWeatherByZipCodeSoapOut">
   <wsdl:part name="parameters"
       element="tns:GetWeatherByZipCodeResponse"/>
</wsdl:message>
```

It is clear that splitting the CamelCase identifier `GetWeatherStation` into the tokens `get`, `weather`, and `station`, would provide more meaningful and generalised concepts, which the learning algorithm can use as features. Indeed, to extract useful word tokens from the identifiers, we split them into pieces based on the presence of underscores or CamelCase; all tokens are then normalised to lowercase.

Once the feature representation is available, we use it to learn several classifiers, each of them specialised to recognise if the WSDL expresses some target semantic properties. The latter can also be concepts of an ontology. Consequently, our algorithm may be used to learn classifiers that automatically assign ontology concepts to actions defined in NS interfaces. Of course, the additional use of domain (but at the same time general) ontologies facilitates the learning process by providing effective features for the interface representation. In other words, WSDL, domain ontologies and any other information contribute to defining the vector representation used for training the concept classifiers.

5 AUTOMATA LEARNING FOR INFERRING NS BEHAVIOURAL SEMANTICS

Automata learning can be considered as a key technology for dealing with *black box* systems, i.e., systems that can be observed, but for which no or little knowledge about the internal structure or even their intent is available. Active Learning (*a.k.a* regular extrapolation) attempts to construct a deterministic finite automaton that matches the behaviour of a given target system on the basis of test-based interaction with the system. The popular L^* algorithm infers Deterministic Finite Automata (DFAs) by means of *membership queries* that test whether certain strings (potential runs) are contained in the target system's language (its set of runs), and *equivalence queries* that compare intermediately constructed hypothesis automata for language equivalence with the target system.

In its basic form, L^* starts with a hypothesis automaton that treats all sequences of considered input actions alike, i.e., it has one single state, and refines this automaton on the basis of query results, iterating two main steps: (1) refining intermediate hypothesis automata using membership queries until a certain level of "consistency" is achieved (*test-based modelling*), and (2) testing hypothesis automata for equivalence with the target system via equivalence queries (*model-based testing*). This procedure successively produces state-minimal deterministic (hypothesis) automata consistent with all the encountered query results [3]. This basic pattern has been extended beyond the domain of learning DFAs to classes of automata better suited for modelling reactive systems in practice. On the basis of active learning algorithms for Mealy machines, inference algorithms for I/O-automata [1], timed automata [7], Petri Nets [6], and Register Automata [9], i.e., restricted flow graphs, have been developed.

While usually models produced by active learning are used in model-based verification or some other domain that requires complete models of the system under test (e.g., to prove absence of faults), here the inferred models serve as a basis for the interaction with the system for Emergent Middleware synthesis. This special focus poses unique requirements on the inferred models, which become apparent in the following prototypical example.

Figure 4 shows a typical interoperability scenario where two NSs are actual implementations of their specified interfaces. The NS on the right implements a weather service that provides weather forecasts for regions identified by ZIP codes. The NS on the left is a matching client. The two NSs communicate via SOAP protocol messages (1), (5), and together realise some protocol, which comprises a control part (2), and a data part (3) at both NSes. The data parts may be best described as a set of local variables or registers. The control part can be modelled as a labeled transition system with actual blocks of code labelling the transitions (4). Each code block of Fig. 4 would consist of an entry point for one interface method (e.g., `GetWeatherByZipCode`), conditions over parameters and local variables (e.g., comparing ZIP codes), assignments and operations on local variables (e.g., storing returned weather data), and a return statement.

To infer the behaviour of one NS (say, the right one from Fig. 4), the role of the other NS has to be undertaken by a learning algorithm,

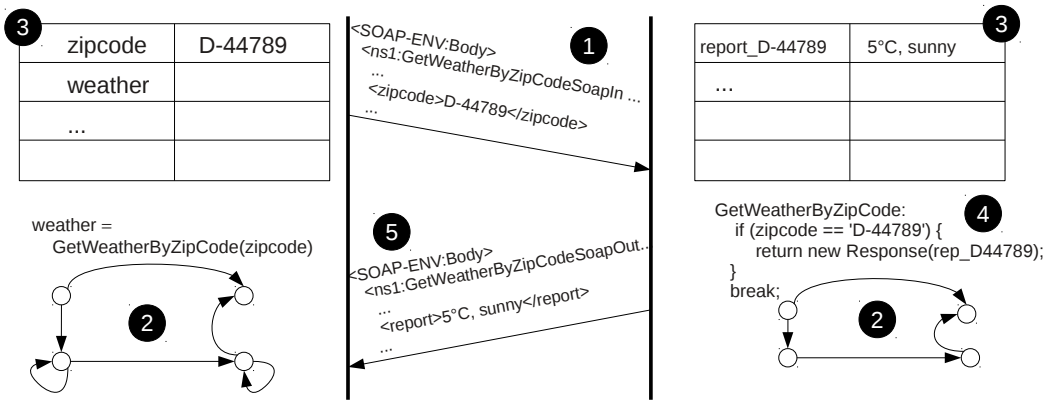

Figure 4. Communicating Components

which is aware of the interface alphabet of the NS whose affordance's behaviour is to be learned. This interface alphabet is derived automatically from the interface description of the NS under scrutiny. A test-driver is then instantiated by the Learning Enabler, translating the alphabet symbols to remote invocations of the NS to be learned.

Now, to capture the interaction of the two NSs faithfully, two phenomena have to be made explicit in the inferred models:

- *Preconditions of Primitives:* Usually real systems operate on communication primitives that contain data values relevant to the communication context and have a direct impact on the exposed behaviour. Consider as an example session identifiers or sequence numbers that are negotiated between the communication participants and included in every message. The models have to make explicit causal relations between data parameters that are used in the communication (e.g, the exact session identifier that is returned when opening a new session has to be used in subsequent calls).
- *Effects of Primitives:* The learned models will only be useful for Emergent Middleware (mediator) synthesis within a given semantic context. Most NSs have well-defined purposes as characterised by affordances (e.g., getting localised weather information). A subset of the offered communication primitives, when certain preconditions are met, will lead to successful conclusion of this purpose. This usually will not be deducible from the communication with a system: an automata learning algorithm in general cannot tell error messages and regular messages (e.g., weather information) apart. In such cases, information about effects of primitives rather has to be provided as an additional (semantic) input to the learning algorithm (e.g., in terms of ontologies [4]), as supported by the semantically annotated interface descriptions of NSes.

Summarizing, in the context of Emergent Middleware, especially dealing with parameters and value domains, and providing semantic information on the effect of communication primitives, are aspects that have to be addressed with care. We have reaffirmed this analysis in a series of experiments on actual implementations of NSs.

The automata learning technique is provided by LearnLib [**?**, 22], a component-based framework for automata learning. In the produced model, each transition consists of two parts, separated by a forward-slash symbol: on the left hand side an abstract parameterised symbol is denoted, while on the right hand side the named variable storing the invocation result is specified. Figure 5 depicts the behavioural description of the weather station, which was learned in 31 seconds on a portable computer, using 258 MQs.

The model correctly reflects the steps necessary, e.g., to read sensor data: `createProperties`, `createSession`, `getWeatherStation`, `authenticate` and `getSensor` have to be invoked before `getSensorData` can be called successfully. Additionally, the actual realisation of authentication, which cannot be deduced from the interface specification alone, is revealed in the inferred model. When simply looking at the parameter types, the action `getSensor` should be invocable directly after the `getWeatherStation` primitive. However, in reality `getSensor` is guarded by an authentication mechanism, meaning that `authenticate` has to be successfully invoked beforehand. Also, from the model, it is easily deducible that the `authenticate` action will indeed merely affect the provided station data object (and not, e.g., the whole session): requesting a new station data object will always necessitate another authentication step before `getSensor` can be invoked again, as that action requires an authenticated station data object.

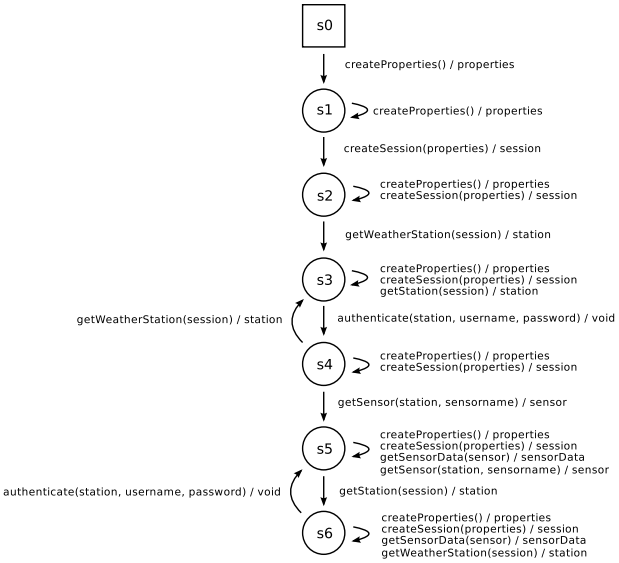

Figure 5. Behavioural Model of the Weather Station Sensor Network Service – Starting State is `s0`

6 CONCLUSIONS

This paper has presented the central role of learning in supporting the concept of Emergent Middleware, which revisits the middleware paradigm to sustain interoperability in increasingly heterogeneous and dynamic complex distributed systems. The production of Emergent Middleware raises numerous challenges, among which dealing with the *a priori* minimal knowledge about networked systems that is available to the generation process. Indeed, semantic knowledge about the interaction protocols run by the Networked Systems is needed to be able to reason and compose protocols in a way that enable NSs to collaborate properly. While making such knowledge available is increasingly common in Internet-worked environments (e.g., see effort in the Web service domain), it remains absent from the vast majority of descriptions exposed for the Networked Systems that are made available over the Internet. This paper has specifically outlined how powerful learning techniques that are being developed by the scientific community can be successfully applied to the Emergent Middleware context, thereby enabling the automated learning of both functional and behavioural semantics of NSs. In more detail, this paper has detailed how statistical and automata learning can be exploited to enable on-the-fly inference of functional and behavioural semantics of NSs, respectively.

Our experiments so far show great promise with respect to the effectiveness and efficiency of machine learning techniques applied to realistic distributed system such as in the GMES case. Our short-term future work focuses on the fine tuning of machine learning algorithms according to the specifics of the networked systems as well as enhancing the learnt models with data representations and non-functional properties, which can result in considerable gains in terms of accuracy and performance. In the mid-term, we will work on the realisation of a continuous feedback loop from real-execution observations of the networked systems to enhance the learnt models dynamically as new knowledge becomes available and to improve the synthesised emergent middleware accordingly.

ACKNOWLEDGEMENTS

This work is done as part of the European FP7 ICT FET CONNECT and FP7 ICT FET CA EternalS projects.

REFERENCES

[1] Fides Aarts and Frits Vaandrager, 'Learning I/O Automata', in *CONCUR 2010 - Concurrency Theory*, eds., Paul Gastin and Franois Laroussinie, volume 6269 of *Lecture Notes in Computer Science*, 71–85, Springer Berlin / Heidelberg, (2010).

[2] Rajeev Alur, Pavol Cerny, P. Madhusudan, and Wonhong Nam, 'Synthesis of interface specifications for Java classes', in *Proc. POPL '05*, (2005).

[3] D. Angluin, 'Learning Regular Sets from Queries and Counterexamples', *Information and Computation*, **75**(2), 87–106, (1987).

[4] Gordon Blair, Amel Bennaceur, Nikolaos Georgantas, Paul Grace, Valérie Issarny, Vatsala Nundloll, and Massimo Paolucci, 'The Role of Ontologies in Emergent Middleware: Supporting Interoperability in Complex Distributed Systems', in *Middleware 2011 - 12th International Middleware Conference*, (2011).

[5] David A. Cohn, Zoubin Ghahramani, and Michael I. Jordan, 'Active learning with statistical models', *J. Artif. Intell. Res. (JAIR)*, **4**, 129–145, (1996).

[6] Javier Esparza, Martin Leucker, and Maximilian Schlund, 'Learning Workflow Petri Nets', in *Proceedings of the 31st International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (Petri Nets'10)*, Lecture Notes in Computer Science. Springer, (2010). to appear.

[7] Olga Grinchtein, Bengt Jonsson, and Paul Pettersson, 'Inference of Event-Recording Automata Using Timed Decision Trees', in *Proc. CONCUR 2006,* 17^{th} *Int. Conf. on Concurrency Theory*, pp. 435–449, (2006).

[8] Andreas Heß and Nicholas Kushmerick, 'Learning to attach semantic metadata to web services', in *International Semantic Web Conference*, pp. 258–273, (2003).

[9] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel, 'Inferring Canonical Register Automata', in *VMCAI 2012*, (to appear).

[10] Falk Howar, Bernhard Steffen, and Maik Merten, 'Automata Learning with Automated Alphabet Abstraction Refinement', in *Twelfth International Conference on Verification, Model Checking, and Abstract Interpretation*, (2011).

[11] H. Hungar, T. Margaria, and B. Steffen, 'Test-based model generation for legacy systems', in *Test Conference, 2003. Proceedings. ITC 2003. International*, volume 1, pp. 971–980, (30-Oct. 2, 2003).

[12] Valérie Issarny, Bernhard Steffen, Bengt Jonsson, Gordon Blair, Paul Grace, Marta Kwiatkowska, Radu Calinescu, Paola Inverardi, Massimo Tivoli, Antonia Bertolino, and Antonino Sabetta, 'CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems', in *14th IEEE International Conference on Engineering of Complex Computer Systems*, (2009).

[13] Thorsten Joachims, Thomas Hofmann, Yisong Yue, and Chun-Nam John Yu, 'Predicting structured objects with support vector machines', *Commun. ACM*, **52**(11), 97–104, (2009).

[14] Ioannis Katakis, Georgios Meditskos, Grigorios Tsoumakas, Nick Bassiliades, and Ioannis P. Vlahavas, 'On the combination of textual and semantic descriptions for automated semantic web service classification', in *AIAI*, pp. 95–104, (2009).

[15] David L. Martin, Mark H. Burstein, Drew V. McDermott, Sheila A. McIlraith, Massimo Paolucci, Katia P. Sycara, Deborah L. McGuinness, Evren Sirin, and Naveen Srinivasan, 'Bringing semantics to web services with OWL-S', in *World Wide Web*, pp. 243–277, (2007).

[16] Alessandro Moschitti, 'Efficient convolution kernels for dependency and constituent syntactic trees', in *ECML*, pp. 318–329, (2006).

[17] Alessandro Moschitti, 'Kernel methods, syntax and semantics for relational text categorization', in *Proceedings of ACM 17th Conference on Information and Knowledge Management (CIKM)*, Napa Valley, United States, (2008).

[18] Alessandro Moschitti, 'Kernel-based machines for abstract and easy modeling of automatic learning', in *11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-11*, (2011).

[19] Alessandro Moschitti, 'Kernel-based machines for abstract and easy modeling of automatic learning', in *11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-11*, pp. 458–503, (2011).

[20] Alessandro Moschitti and Roberto Basili, 'Complex linguistic features for text classification: A comprehensive study', in *Proceedings of the 26th European Conference on Information Retrieval Research (ECIR 2004)*, pp. 181–196, Sunderland, United Kingdom, (2004).

[21] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis, 'Black box checking', in *FORTE*, pp. 225–240, (1999).

[22] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria, 'LearnLib: a framework for extrapolating behavioral models', *Int. J. Softw. Tools Technol. Transf.*, **11**(5), 393–407, (2009).

[23] Gerard Salton, A. Wong, and C. S. Yang, 'A vector space model for automatic indexing', Technical Report TR74-218, Department of Computer Science, Cornell University, Ithaca, New York, (1974).

[24] R.W. Selby and A.A. Porter, 'Learning from examples: generation and evaluation of decision trees for software resource analysis', *Software Engineering, IEEE Transactions on*, **14**(12), (1988).

[25] Peter Stone and Manuela Veloso, 'Multiagent systems: A survey from a machine learning perspective', *Autonomous Robots*, **8**, (2000).