Noname manuscript No.
(will be inserted by the editor)

The Role of Models@run.time in Supporting On-the-fly Interoperability

N. Bencomo · A. Bennaceur · P. Grace · G. Blair · V. Issarny

Received: date / Accepted: date

Abstract Models at runtime can be defined as abstract representations of a system, including its structure and behaviour, which exist in tandem with the given system during the actual execution time of that system. Furthermore, these models should be causally connected to the system being modelled, offering a reflective capability. Significant advances have been made in recent years in applying this concept, most notably in adaptive systems. In this paper we argue that a similar approach can also be used to support the dynamic generation of software artefacts at execution time. An important area where this is relevant is the generation of software mediators to tackle the crucial problem of interoperability in distributed systems. We refer to this approach as emergent middleware, representing a fundamentally new approach to resolving interoperability problems in the complex distributed systems of today. In this context, the runtime models are used to capture meta-information about the underlying networked systems that need to interoperate, including their interfaces and additional knowledge about their associated behaviour. This is supplemented by ontological information to enable semantic reasoning. This paper focuses on this novel use of models at runtime, examining in detail the nature of such runtime models coupled with consideration of the supportive algorithms and tools that extract this knowledge and use it to synthesise the appropriate emergent middleware.

Keywords runtime models, runtime interoperability, mediators, ontology

N. Becomo, A. Bennaceur and V. Issarny
Inria, Paris-Rocquencourt, France
E-mail: firstname.lastname@inria.fr

G. Blair and P. Grace
School of Computing and Communications, Lancaster University, UK
E-mail: {gracep,gordon}@comp.lancs.ac.uk

1 Introduction

A *model@run.time* or runtime model can be defined as an abstract representation of a system, including its structure, behaviour and goals, which exists in tandem with a given system during the actual execution time of that system. Furthermore, this model should be causally connected to the system being modelled, hence offering a reflective capability. Causal connectivity allows the runtime model to provide up-to-date information about the system in order to support analysis of the system before committing to changes, and therefore avoiding potential inconsistencies in the runtime system. Significant advances have been made in the use of runtime models [9,6,3]. Architectural-based runtime models is a research topic that has generated most interest [44,43,19,53]; and mainly in the broader area of self-adaptation [43,40,25,41] where runtime models have been used to support decision making.

Crucially, it is difficult to assess accurately the impact of the changes in the environment and context before deployment and runtime due to incomplete information. Runtime models support the handling of these dynamic and to some extent unforeseen changes [52]. Importantly, runtime models can support decision making and reasoning based on knowledge unforeseen prior to the time of execution, but which emerges during execution.

Currently, there is pressure to move some activities from design-time to deployment and runtime [5]. One of the goals is to be able to insert at runtime new behaviour that was not necessarily foreseen during design time. One way to do this is to be able to synthesise the software associated with the new behaviour at runtime. As self-representations of the systems (as in the case of traditional MDE [46]), runtime models can also be used as the basis for software synthesis. However, little attention has been directed to techniques for the synthesis or generation of software using runtime models during execution. This is precisely the topic we aim to address in this paper. We argue that runtime models can support the runtime synthesis of software that will be part of the executing system and which was not necessarily conceived during design time. As a way to depict our novel ideas we use the platform provided by the CONNECT project[1], which provides us with the necessary technology to realize our vision.

In modern highly dynamic environments, networked systems appear and disappear along with the services they offer. We term a *networked system* as any system or composition of systems that expose their functionality as networked addressable services implemented and accessible using a given protocol. However, where these systems meet spontaneously, interoperability is a fundamental and challenging requirement. These systems may not know each other, but they may still try to interact in order to meet certain goals [5]. Therefore, it may be the case that for some aspects of the system, a software model needs to be conceived during runtime as it would be impossible to design it in advance. Inferring information to create runtime models [47] during

[1] `http://www.Connect-forever.eu`

execution using, for example, learning techniques [8,49] offers an interesting approach to take.

In this paper, we focus on the novel use of runtime models to support the dynamic synthesis of software. Specifically, our vision is explained using examples to synthesise emergent middleware, i.e., the synthesis of mediators that translate actions of one networked system to the actions of another networked system developed with no prior knowledge of the former in order to achieve interoperability. Using rich discovery and learning methods we are able to capture and refine the required knowledge of the context and environment. The knowledge is explicitly formulated and made available to computational manipulation in the form of a runtime model. This runtime model is based on labelled transition systems (LTSs) [32] which offer the behavioural semantics needed to model the interaction protocols, and as an established solution are supported by a number of available tools. Ontologies complement the LTSs providing semantic reasoning about the mapping between protocols. From these runtime models, mediators are synthesised.

In summary, the contribution of this paper is an approach to synthesize software, in the form of mediators, from runtime models. Crucially, the runtime models provide support to reason about interoperability issues that were unknown before execution. The core piece of this novel approach is the derivation of completely new runtime models during execution to solve the on-the-fly interoperability problem, i.e., creating a mediator from scratch. Notably, the runtime models capture not just structure and functionality, but also behaviour which is refined with the support of machine learning techniques. We also show how ontological information can support conceptual reasoning based on models.

The paper is structured as follows. In section 2, we discuss in detail the interoperability problem in complex distributed systems and the relevant role of runtime models to tackle the problem. In section 3 we present how models at runtime are used to dynamically synthesise the emergent middleware that ensure interoperation between heterogeneous networked systems. Relevant results and achievements in the scope of the CONNECT project; are also presented. In section 4 we discuss some related work. Finally, we draw conclusions in section 5.

2 Emergent Middleware to Support On-the-fly Interoperability

Interoperability is defined as the capability of two or more networked systems to exchange and understand one another's data. Where systems are designed and developed with knowledge of one another, or where systems have been developed using a common standard, the interoperability problem is largely solved. Indeed, interoperability is a primary goal of standard-based middleware solutions (e.g. CORBA and Web Services middleware). However, the increasing complexity of distributed systems introduces new problems, which existing middleware-based interoperability approaches are not suited to address. In-

deed, in environments where heterogeneity and highly dynamic behaviour are typical, e.g., pervasive computing, mobile computing, and large scale systems of systems, there are further challenges to achieving interoperability.

Heterogeneity can now be encountered in many forms. Middleware is often applied to address differences in terms of computational devices, communication networks, and operating systems. However, the design decisions taken for the development and deployment of each networked system may introduce specific interoperability challenges that must then be addressed dynamically. Firstly, using a particular middleware type means that interoperability is not possible with networked systems implemented using a different middleware due to the differences in the communication protocol of each (e.g. heterogeneous message packet formats and message sequences). Secondly, differences in the design of the application interface will hinder interoperability, hence even where a common middleware is chosen, interoperability cannot be guaranteed; differences in the syntax of interfaces, the types and data formats, the semantic meaning of data schemas, and the invocation sequence required for achieving application functionality are all potential interoperability challenges that must be addressed dynamically.

Dynamic behaviour is characterised by networked systems that come and go (often due to the increasing mobility of users); furthermore, the operating conditions in heterogeneous environments fluctuate, e.g., changing quality of service levels in mobile networks. Interactions are also spontaneous, i.e., systems wanting to interoperate, search at runtime for systems that match their requirements. Here, there can be no agreement of a common solution or standard, and the differences in application behaviour and communication protocols can only be detected and resolved at runtime.

Therefore, it is difficult to design a solution that takes into account the many dimensions of heterogeneity, and this is further exacerbated by spontaneous interactions and so no prior decisions about interoperability solutions can be assumed. Instead, a fundamental rethink is required into how interoperability can be resolved at runtime without relying on common standards or design decisions. We argue that models@runtime have an important role to play in such solutions as illustrated in this paper.

2.1 The GMES Example

To better illustrate the interoperability problem we highlight the challenges through the use of an example from the area of the Global Monitoring for Environment and Security (GMES[2]). GMES is the European Programme for the establishment of a European capacity for Earth Observation. In particular, the emergency management thematic highlights the need to support emergency situations involving different European organisations. In emergency situations, the context is also necessarily highly dynamic and therefore provides

[2] `http://www.gmes.info/`

a strong example of the need for on-the-fly solutions to interoperability. The target GMES system therefore inevitably involves highly heterogeneous networked systems that interact in order to perform the different tasks necessary for decision making. The tasks include, among others, collecting weather information, capturing video, and getting positioning information of the different devices.

We concentrate on the particular case of a networked system connecting with another video capturing networked system. Various concrete systems are able to capture video: fixed cameras, robots with video sensing capabilities (UGV: Unmanned Ground Vehicle), or flying drones (UAV: Unmanned Aerial Vehicle). In addition, the videos may be accessed from other heterogeneous systems, including applications run on mobile handheld devices of the various actors on site, and the ones executed by the Command and Control—C2—centres (see Figure 1).

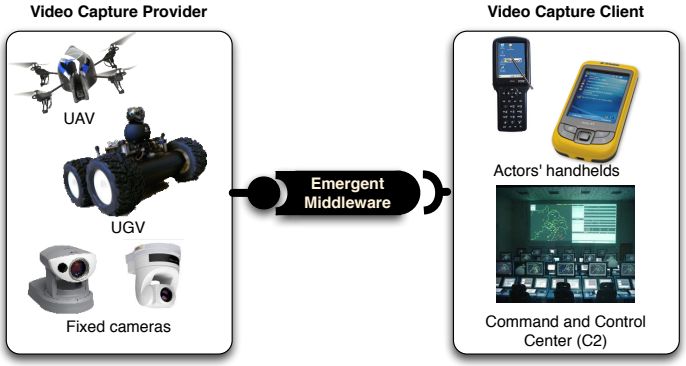

Fig. 1 The GMES case study

Specifically, we focus on two networked systems C2 and UGV. The C2 system needs to gather information from different cameras in order to analyse them and then makes decision about the appropriate emergency procedure to take. C2 has been developed to interact with a fixed camera and retrieve the videos for given periods of time; for this it uses the SOAP RPC protocol[3]. In contrast, the UGV system captures video and displays it using HTTP Live Streaming (HLS)[4]. It can also move according to some pre-defined patterns, zoom, or get a URL where the video capture can be viewed. The GMES is a substantial case study that we use in this paper to explore different aspects of the interoperability challenges that now arise, and the role of runtime models as the mean to tackle them.

[3] `http://www.w3.org/TR/soap/`

[4] `http://tools.ietf.org/html/draft-pantos-http-live-streaming`

2.2 The Case for Emergent Middleware

We support the vision of emergent middleware [30,26] to achieve interoperability. That is, where two networked systems are willing to interoperate and are mutually compatible in terms of the required and provided functionality then the middleware software to coordinate the exchange is synthesised (taking into account the respective operating context and environmental conditions of the two systems). Due to the highly heterogeneous and spontaneous nature of potential interactions, the engineering of Emergent Middleware is significantly different from traditional statically developed middleware products.

The approach to achieve such emergent software is based upon the following key requirements:

- *The creation and maintenance of runtime models of individual networked systems* (See Figure 2-❶). In order to reason about how to interoperate with a given system we need to create a runtime model of its interface and behaviour protocol. The behaviour of the system is modelled in terms of the sequence of operations that are necessary to achieve a particular service. Importantly, to underpin runtime solutions to interoperability these models must capture meaning [10]; that is, given the two networked system models it must be possible for an interoperability solution to understand and reason where systems are semantically similar. For this purpose, we use ontologies as a further extension to the model@runtime, i.e., the elements of the runtime model reference concepts defined in a domain-specific ontology (See Ontologies in Figure 2).
- *Monitoring and discovery of existing networked systems* (See Figure 2-❷). In order to build a runtime model, the operation of the networked system must be first discovered and then monitored. This requires the extraction of information about the systems using traditional resource and service discovery protocols, e.g., lookup facilities as provided by protocols such as Service Location Protocol (SLP), or Web Services Service Discovery (WS-Discovery), descriptions using languages such as Web Services Description Language (WSDL), and approaches promoting the use of ontology-based techniques to semantically match requests and advertisements [27,39].
- *Learning of networked system behaviour* (See Figure 2-❸). Using the initial discovered information as a starting point, machine learning approaches are required to learn how one must interoperate with a particular system in order to achieve particular behaviour, i.e., this will inform how to model exactly the behaviour of the networked system in terms of its middleware and application protocols.
- *Synthesis of interoperability software* (See Figure 2-❹). We require synthesis solutions that can use the runtime models of two systems to calculate a mediator that will resolve the differences between the heterogeneous protocol endpoints, and then generate the software that implements this mediator on the fly in order for it to dynamically deployed between two systems [29]. In this paper we focus on the case of mediation between a

pair of networked systems as the case of multiple systems, under the same assumptions, may be proved undecidable [21]. The synthesis is further supported by ontologies that formalise the domain knowledge [10].

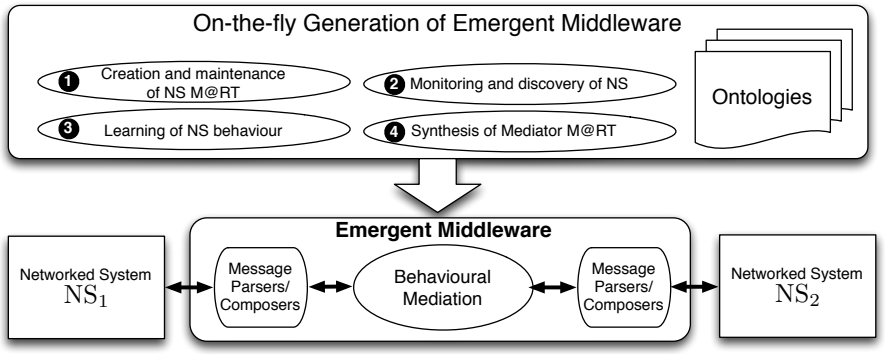

Fig. 2 Supporting Emergent Middleware

3 Models@Runtime in Action: Sustaining the Dynamic Synthesis of Emergent Middleware

Figure 3 outlines our overall approach for synthesising emergent middleware between two networked systems NS_1 and NS_2 and that addresses the requirements described above. The key philosophy of this approach is to utilise runtime models to both i) support the reasoning about what emergent middleware should be created, and ii) support the creation of this software artefact itself. Our approach is incremental as illustrated by the following steps seen in the figure:

1. Discovery and Learning of knowledge *to extract the behaviour* of the networked systems, see Figure 3-❶. The knowledge that has been discovered and learned is made explicit and available for computational treatment in the form of a runtime model.
2. Analysis of the runtime models and generation, in the appropriate cases, of the necessary mediator model that specifies the necessary translation and coordination that need to be performed in order to allows the two networked systems to interoperate. We call this step *synthesis of the mediator model*, see Figure 3-❷.
3. Concretisation of the mediator model in an artefact (i.e., the Emergent Middleware) that further deals with message-level interoperability, that is *deployment*, see Figure 3-❸.

In the remaining of this section we explain the approach in more detail.

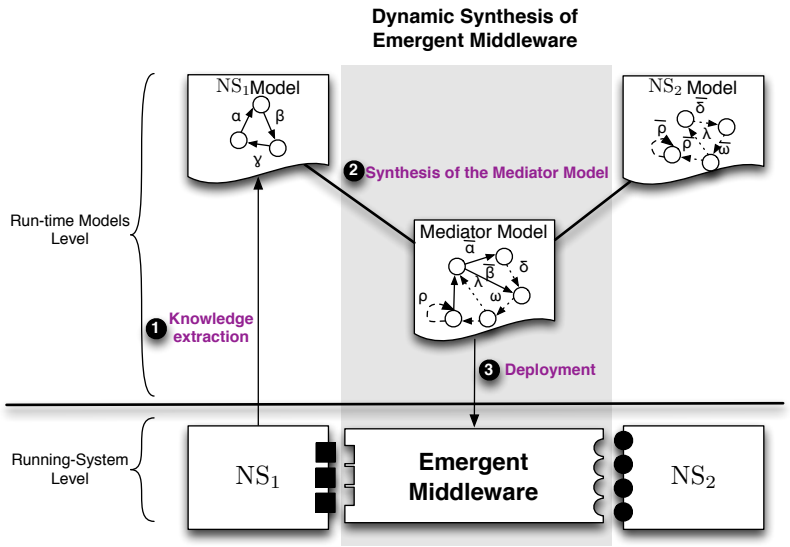

Fig. 3 Overview of the Approach

3.1 The Runtime Model Specifications

The runtime models are the central elements that allow us to reason about how to make systems interoperable. These models need to specify adequate knowledge about the networked systems from the application down to the network layers, as well as the domain knowledge using ontologies. In our approach, two distinct types of models are used to describe the different constituents of the system: the *networked system model* and the *mediator model*. These models are manipulated (constructed, transformed, refined) so as to manage the full cycle of interoperability assurance.

The Networked System Model. It captures both the functional and behavioural semantics of the networked system. The functional semantics describe the functionality of the system (i.e., what the system does), its interface described in terms of the actions it requires and/or provides, and the specific middleware platform it is implemented upon. The behavioural semantics specify how the actions of its interface are coordinated in order to achieve the system functionality. For example, Figure 4 depicts the runtime models associated with C2 and the UGV networked systems.

We rely on ontologies to describe the functional semantics of networked systems. In particular, the functionality of the each NS refers to ontological concepts. Furthermore, the actions of each NS interface are augmented with ontological annotations. We distinguish between two types of actions: *input* and *output* actions. An input action $op(in)$:*out* requires an operation *op* for which it produces some input data *in* and consumes the output data *out*. We assume that *op*, *in*, and *out* belong to the same domain ontology. For example,

the C2 interface includes the `getMPEGVideo(Camera, Period):MPEGVideo` action that specifies that C2 provides the objects `Camera` and `Period` and expects an MPEG video in return. An output action[5] $\overline{op}(in)$: *out* provides an operation *op* for which it uses the inputs *in* and produces the corresponding outputs *out*. For example, the UGV interface includes the `getVideoRTPAddress(Camera, Period):VideoAddress` action that expects the objects `Camera` and `Period` and returns an address to a video.

In addition, the binding defines the specific middleware used by a networked system to implement these actions. For example, C2 uses SOAP-RPC as the underlying middleware.

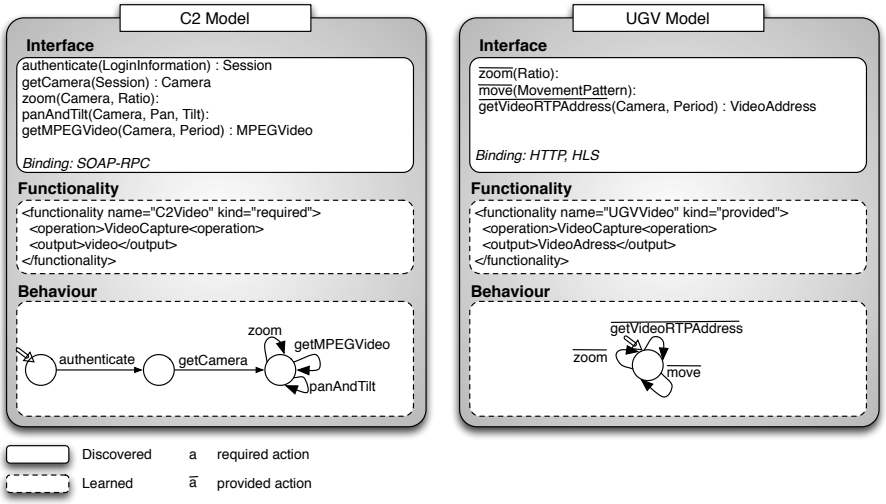

Fig. 4 The models of individual networked systems

The behaviour of the networked system is specified using Labeled Transition Systems (LTS) [32]. LTSs proved to be effective for describing, understanding and reasoning about concurrent systems. An LTS is a directed graph with labels on each edge describing the progress of the system behaviour when the action, to which the label is attached, is performed. The actions of the LTS are those specified in the interface of the networked system. For example, in Figure 4, the C2 system first performs the authentication action, it selects a camera, then it can zoom, send a pan/tilt command to the camera, or ask for a video.

The Mediator Model. The mediator is the intermediary middleware entity that stands between the two networked systems, translates their actions and coordinate their behaviours in order to guarantee their successful interoperation.

[5] Note the use of an overline as a convenient shorthand to denote output actions

In other words, the mediator is capable of receiving, sending, delaying the delivery of data, as well as reasoning about their semantics in order to generate equivalent data by transforming and composing the original ones. It cannot create data by itself.

The resultant mediator between C2 and UGV is depicted in see Figure 5. The mediator coordinates their respective actions in order to allow them to interoperate.

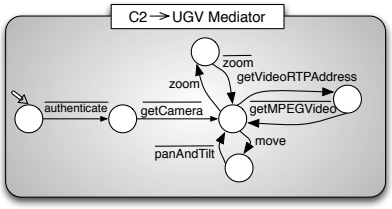

Fig. 5 The models of the C2 → UGV mediator

3.2 Building Runtime Models

Here we describe how the runtime models, which were described above, are created and completed at runtime. More specifically, we need first to determine the systems joining (or leaving) the runtime environment, this is the role of *discovery* . In particular, we build on state-of-the-art interoperable discovery methods [13,15,23] to cope with the heterogeneous discovery protocols that exist. Specifically, a *Discovery Enabler* listens on various multicast addresses used by legacy discovery protocols (e.g., Service Location Protocol[6], WS-Discovery[7], UPnP-SSDP[8], and Jini[9]); it intercepts both the advertisement messages and lookup request messages that are sent within the network environment and processes them using appropriate plug-ins. However, legacy discovery protocols provide only partial networked system models, which in most of the cases consist in the syntactic interfaces. Consequently, we rely on learning techniques to complete the model of the networked systems with further information about functional and behavioural semantics.

Learning Networked System Models. We use learning techniques to infer the functionality, the ontology-based annotations of the interface, and determine the behaviour of a networked system given the discovered interface at runtime.

[6] `http://www.openslp.org/`

[7] `http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.pdf`

[8] `http://www.upnp.org/`

[9] `http://www.jini.org/`

The functionality of a system is expressed as a semantic concept based on a domain ontology and learned using support vector machines for text categorisation, that is a machine learning technique able to categorise interface based on the terms used in their textual or XML description [7]. First, examples of interfaces and the corresponding functionality are used to train the algorithm in order to infer a *categorisation function*, that is a function that relates an interface to a semantic concept from the domain ontology with a given accuracy. More specifically, the algorithm uses natural language text to infer the likelihood between some words occurring in the interface specification and the functionality of the system. Second, the interface is analysed in order to infer the appropriate functionality. The accuracy of such inference depends on the size of the examples used during the training phase [8].

As mentioned earlier, the behaviour of a system is represented as an LTS. Within the CONNECT project, a dedicated learning technique based on the L^*_M algorithm [37] is used to construct the behavioural models of networked systems. It enhances Angluin's seminal L^* algorithm [2] to deal with realistic systems in minimal time with various improvements such as abstraction/refinement or dealing with data values. It is based on an iterative process by which a hypothesis behavioural model of the system is incrementally refined by actively testing interaction with the corresponding system. Hence, unlike passive learning algorithms [36,33] that only observe the interaction traces, the L^*_M chooses the sequences of actions to execute in order to learn the behaviour in minimal time.

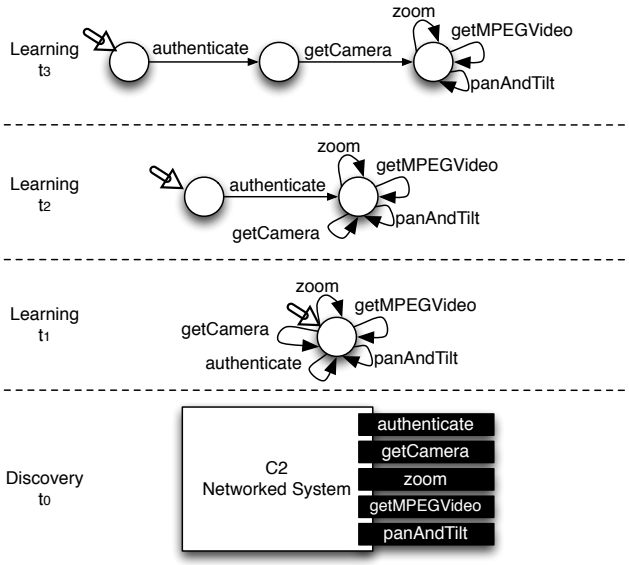

Fig. 6 Learning the behaviour of the C2 networked systems

As an illustration, consider the example of the process of learning the behavior of the system C2 shown in Figure 6. At time t_0, the interface of the C2 system is discovered. At time t_1, the learning process initiates by assuming one single state where all the actions can be performed. However, when trying to interact with the system by performing for example a `zoom` and then `authenticate`, an error (or exception) is raised. At time t_2, the model is updated so as to forbid this erroneous trace. Similarly, when authenticating and then zooming, an error is obtained, therefore the learning updates the models, which continues to be refined to obtain the model at time t_3.

Synthesising the mediator model. Networked systems can successfully interoperate only if they are *behaviourally compatible*. Existing formal notions to behavioural compatibility (e.g., bisimulation [38] or refinement [28]) assume close-world settings, i.e., the use of the same set of actions to define the behaviour of the systems. What is needed is a notion of compatibility that further takes into account the semantics of actions and relies on an intermediary system, the mediator, to perform the necessary translation between semantically-compatible actions. The main idea is that the mediator can safely compute a translation if the data received from one NS are a subtype of the data expected by the other NS. For example, in the case of mapping one action to another, this would be equivalent to Liskov substitution principle [35] with ontological subsumption instead of type subsumption. Note however that these mappings may be ambiguous (or non-deterministic), that is, the same action may correspond to different actions depending on the execution state.

The gist of the approach presented here, is to generate the mediator LTS that coordinates these mappings and synchronise with both networked systems to force them to progress synchronously. During the synthesis step, we explore the various possible mappings in order to produce a correct mediator, i.e., a mediator that guarantees that the parallel composition of the LTSs of the two networked systems together with the LTS of the mediator always reaches a final state, or determines that such mediator does not exist. We use model checking to verify that a composition of mappings verifies this condition and is hence a valid mediator. Model checking is an appealing technique to automatically verify concurrent systems by exhaustively exploring the state space. While the complete coverage of this state space may lead to state explosion, many solutions have been proposed to alleviate this issue and lead to interesting results when applied at runtime [14]. We refer the interested reader to [1] for a detailed description of the synthesis approach.

3.3 Leveraging Runtime Models to Create the Emergent Middleware

In order to produce the emergent middleware, corresponding to a particular mediator model, we need to leverage the runtime models as the only information about the requirements of the mediator are obtained at runtime. Here, an

emergent middleware is a software artefact that can be deployed in a communication network to bind two networked systems together and to make them interoperate.

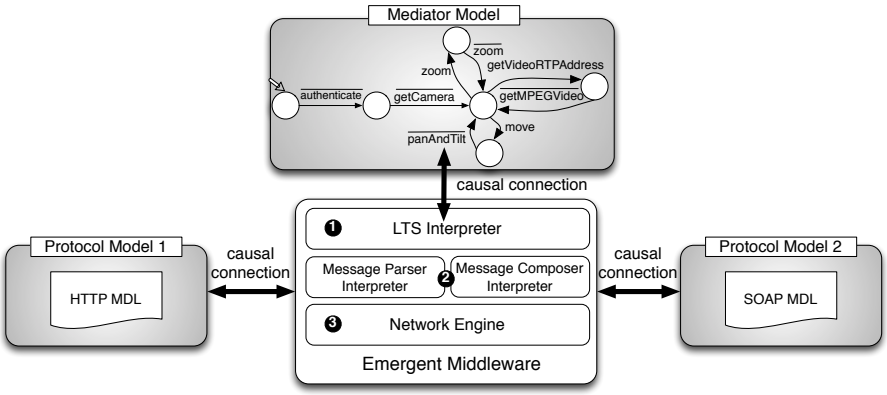

Fig. 7 Runtime models building an emergent middleware

Interpretation is the foundation of emergent middleware. It means that a middleware protocol-specific interpreter is used to execute the actions of the LTS-based models. Starlink [11] is the protocol-specific interpreter used to execute the actions of LTSs. As such, Starlink can be seen as an abstract element until it has been specialised with runtime models to describe its required behaviour. Specifically, Starlink uses an *LTS interpreter* (Figure 7-❶) to analyse and manipulate the mediator LTS, which further examines each transition action to extract the information required to generate a physical middleware-specific message. The *operation* label, the *input* parameters, and the *output* parameters form the content of this message. The interpreter then uses the appropriate middleware binding to physically execute this action content. For example, this could be an RPC invocation using the SOAP binding.

Importantly, the communication protocols themselves are also modelled; their messages are described in a Message Description Language (MDL) such that their packets can be dynamically composed and parsed based upon this description. A model of each protocol is plugged into the *message composer* and *parser* interpreters(Figure 7-❷). Hence, when Starlink executes a transition of the mediator's LTS it produces the correct concrete message. Note that concrete messages are communicated via the *network engine* (Figure 7-❸), which provides a simple subset of network transport primitive to physically connect the two networked systems. Hence, the mediator model (as exemplified in Figure 5) therefore, provides sufficient information to produce and execute an emergent middleware. Considering for example the C2 system, which uses SOAP, and the UGV system, which uses HTTP-based HLS. In order to perform the `zoom` operation (which is provided by both systems), the emergent middleware receives the SOAP request sent by C2 and use it to create an

HTTP request, which it sends to UGV. Once it receives the HTTP response from UGV, it creates the corresponding SOAP response and forwards it to C2.

The benefits of interpretation are that a causal connection is inherent in the deployment. The runtime model informs the mediator behaviour, and hence any changes made to the model on-the-fly are automatically and transparently applied. This is similarly achieved at the middleware level; if the mediator migrates to a different communication protocol, or the protocol itself changes (e.g. a version change) then only the middleware model needs to be changed.

3.4 Models@run.time in CONNECT

In order to illustrate the feasibility of the ideas presented in this paper, we now analyse its application in the broader context of the CONNECT architecture. This explores the key ideas of: i) run-time representation of mediators, ii) how runtime models are derived, and iii) how the concrete mediators are deployed.

The set of main tools developed and supported by CONNECT, and which have been successfully leveraged for the use of runtime models are: the CONNECT Discovery Enablers[10] for discovery, LearnLib[11] for learning, MICS (Mediator synthesis to CONNECT Systems)[12] for synthesis, and Starlink[13]for deployment.

Our ideas have been applied in different applications, the photosharing [29, 12], the travel agent[10], and the GMES examples. In the three applications the heterogeneity of the interfaces and behaviour prevented interoperability even when using current state-of-the-art middleware solutions. The GMES forms the most complete of the three examples (Figure 8). Let us focus on the interaction between the weather station and the C2 to illustrate the application of our approach. These two networked systems exhibit different interfaces and use heterogeneous middleware protocols to communicate. This example, although trivial if there is a developer in the loop, is not trivial to automate and has been successfully resolved using our approach based on models@runtime.

3.4.1 Weather Scenario

In the GMES example, two heterogeneous networked systems (C2 and the weather station) encounter one another and interact. C2 is a client to the weather station that provides temperature and humidity information. Specifically, we apply the tools of CONNECT to build a runtime model of a mediator to generate a concrete mediator that resolves interoperability issues between C2 and the weather station.

[10] `https://www-roc.inria.fr/arles/software/discovery`
[11] `http://www.learnlib.de/`
[12] `http://www-roc.inria.fr/arles/software/mics/`
[13] `http://starlink.sourceforge.net`

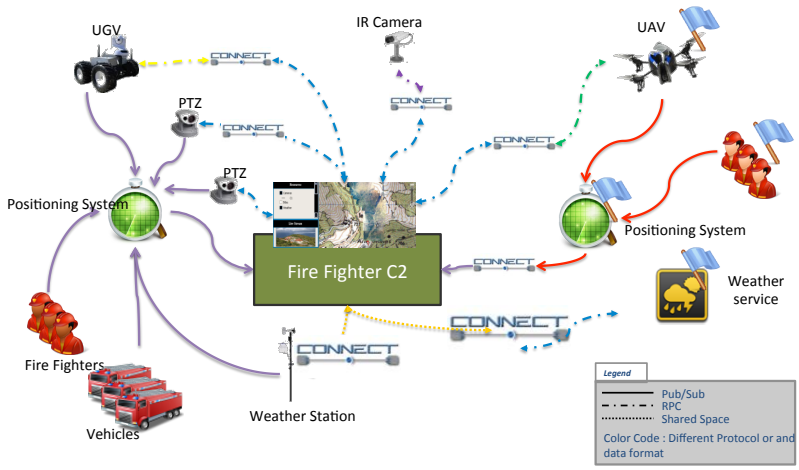

Fig. 8 The GMES scenario and its networked systems

In this section we document the outputs of the enablers to illustrate how the CONNECT architecture coordinates to enable the interaction and cooperation between the two systems. The discovery enabler first monitors the running systems, and receives lookup requests that describe the C2's requirements. It also receives the notification messages from the weather station that advertises the provided interface. The discovery enabler plug-ins transform these messages and produce a WSDL description for both networked systems. A partial view of these descriptions is given in Figures 9 and 10, which show the abstract operations provided by the weather station as well as those required by C2 respectively.

These operations are bound to concrete protocols: SOAP for C2 and CORBA for the weather station. The interfaces also serve to highlight the heterogeneity of the two interfaces; they offer/need the same functionality, but do so with different operations. The next step is to learn the behaviours of the two systems. The learning enabler receives the interface description from the discovery enabler and then interacts with deployed instances to create the behaviour models for each of them. The possible interactions in these systems are produced as LTS models and illustrated in Figure 11.

Here the weather station receives `GetTemperature` and `GetHumidity` CORBA requests and responds using CORBA reply messages, whereas C2 sends a `GetWeather` SOAP Request and expects a `weatherInfo` record con-

```
<wsdl:operation name="logToStation"
 sawsdl:modelReference="http://www.connect.com/ontology/media#Login">
  <wsdl:output message="tns:logToStationResponse" name="loginResponse">
  </wsdl:output>
  <wsdl:input message="tns:login" name="login"></wsdl:input>
</wsdl:operation>
<wsdl:operation name="getWeatherInfo"
 sawsdl:modelReference="http://www.connect.com/ontology/media#GetWeather">
  <wsdl:output message="tns:getWeatherInfoResponse"></wsdl:output>
  <wsdl:input message="tns:getWeatherInfoRequest"></wsdl:input>
</wsdl:operation>
<wsdl:operation name="quitStation"
 sawsdl:modelReference="http://www.connect.com/ontology/media#Logout">
  <wsdl:output message="tns:quitStationResponse"></wsdl:output>
  <wsdl:input message="tns:quitStationRequest"></wsdl:input>
</wsdl:operation>
```

Fig. 9 Interface description fragment for the C2

```
<operation name="login"
 sawsdl:modelReference="http://www.connect.com/ontology/media#Login">
  <input message="tns:login"></input>
  <output message="tns:loginResponse"></output>
  <fault message="tns:Exception" name="Exception"></fault>
</operation>
<operation name="getTemperature"
 sawsdl:modelReference="http://www.connect.com/ontology/media#GetTemperature">
  <input message="tns:getTemperature"></input>
  <output message="tns:getTemperatureResponse"></output>
  <fault message="tns:Exception" name="Exception"></fault>
</operation>
<operation name="getHumidity"
 sawsdl:modelReference="http://www.connect.com/ontology/media#GetHumidity">
  <input message="tns:getHumidity"></input>
  <output message="tns:getHumidityResponse"></output>
  <fault message="tns:Exception" name="Exception"></fault>
</operation>
<operation name="logout"
 sawsdl:modelReference="http://www.connect.com/ontology/media#Logout">
  <input message="tns:logout"></input>
  <output message="tns:logoutResponse"></output>
  <fault message="tns:Exception" name="Exception"></fault>
</operation>
```

Fig. 10 Interface description fragment for the weather station

taining temperature and humidity to be returned. CONNECT then creates the mediator that concretely co-ordinates the behaviours of the two systems and translates the messages to address the differences between the SOAP and CORBA communication protocols. To realise this mediator, the two LTS models (see Figure 11) are given to the synthesis enabler. The synthesis enabler reasons about the semantics of actions of the two systems given a domain ontology that states the relations holding between various concepts, e.g., that the `weatherInfo` encompasses `humidity` and `temperature`, to generate the appro-

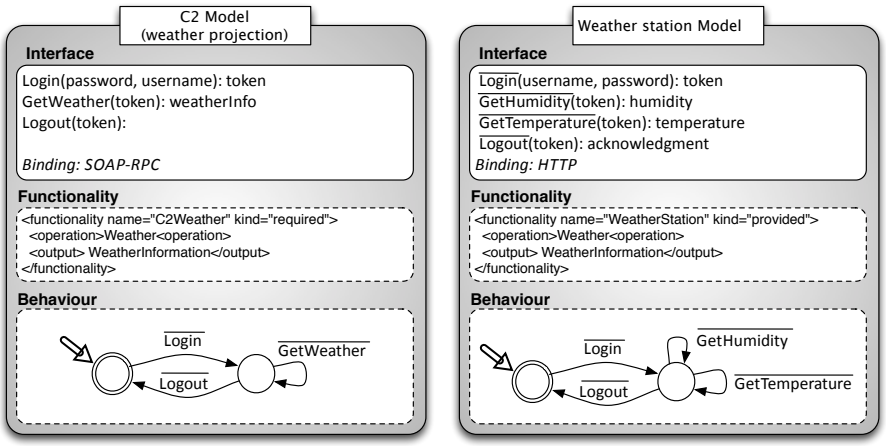

Fig. 11 Description of the weather NSs Models

priate interface mapping. In the example, there are two mappings `getWeather` $\mapsto$<`getHumidity`, `getTemperature`> and `getWeather` $\mapsto$<`getTemperature`, `getHumidity`>. The synthesis creates the LTS associated with the mapping and computes the parallel composition of the LTSs of the two networked systems. It checks that this composed LTS reaches its final state. The mappings could be ranked so as to choose the best one if many are eligible. However, in the current algorithm when many mappings are valid the selection is performed randomly. It concludes that this LTS is then a valid mediator model (see Figure 12) and ready to be deployed.

The mediator model has to be refined taking into account the specificities of the interaction protocols and the data syntax so as to effectively achieve interoperation of the two systems (see Figure 13). Here, the mediator model of Figure 12 is transformed into an emergent middleware by passing it to the binding procedure of the Starlink framework.

It can be seen that the produced emergent middleware depicted in Figure 13 contains the necessary information about the concrete middleware protocols (e.g. a GIOP request message for the CORBA protocol). Besides binding actions to the appropriate middleware, the translation among the input/output data is refined so as to precisely define how the fields of each message can be obtained by reusing the previously received data. This model was deployed between the two systems using Starlink; when executed, interoperability between the two systems was achieved.

3.4.2 Final Reflections

The role of runtime models is fundamental to achieving the required spontaneous interoperability in the scenario described above. In the case of the weather application, when the two systems are designed and developed independently it is impossible for them to interoperate. This is because they differ

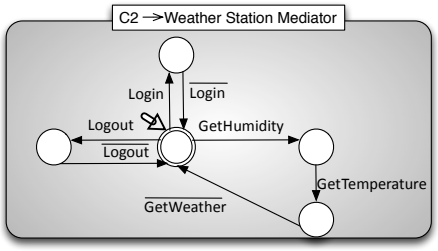

Fig. 12 Model of the mediator between C2 and the weather station

in terms of their middleware protocols and API behaviours. By leveraging runtime models in the scenario we have demonstrated that we can address this important problem. The generation of runtime models using discovery and learning provides the artefacts that underpin the runtime interoperability solution previously not possible. We have shown in the example that executable interoperability software can be synthesised and correctly deployed to allow the two heterogeneous weather systems to interact successfully with one another.

3.5 Research Challenges Ahead: Evolution of Runtime Models

We have studied the viability of the novel ideas to construct and use runtime models. We have shown both, how to build runtime models (i.e. mediator models) during execution and based on discovery and learning techniques, and how these runtime models can be used to dynamically synthesize and deploy software (i.e. emerging middleware). However, many research challenges remain. One specific challenge is that as new knowledge is being discovered or learned. Indeed, statistical learning may admit a very high accuracy of categorization but is unlikely to be perfect. Therefore, the system needs to evolve to reflect the changes perceived in the operating environment. Under those circunstances, the system should be monitored continuously to identify executions that do not conform to the learned behaviour of networked system. This verification should be carried out at runtime and the model of the mediator should be updated in accordance to the changes in the networked system model. Due to the inherent causal connection, the emergent middleware should be adapted accordingly so as to reflect the changes to the mediator model. Furthermore, ontologies may also evolve over time [42], although less frequently. In addition, as ontologies keep emerging and getting standardised, a critical issue is then matching ontologies. The logic grounding of ontologies enables a more accurate matching of concepts compared to syntactic based techniques. Nevertheless, this matching is given with a certain confidence that is never absolutely precise. Quantitative analysis and probabilistic model checking may reveal very useful when quantifying expected behaviour or the confidence on the learned concept and to adapt the overall system accordingly [14, 24].

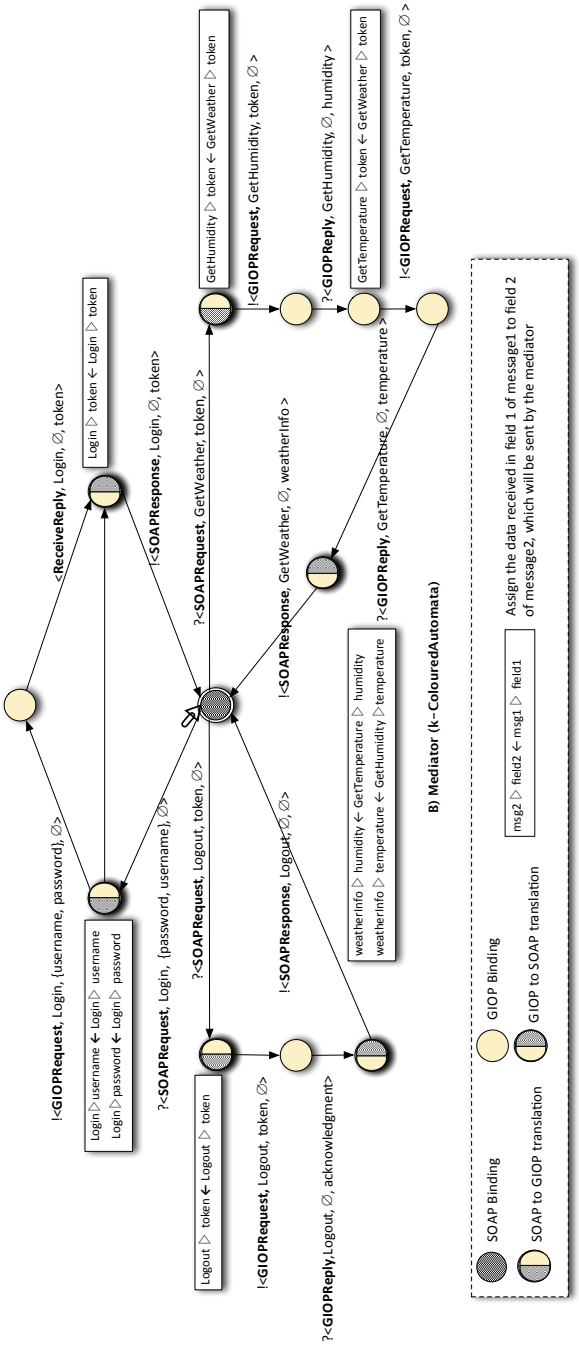

Fig. 13 Emergent middleware

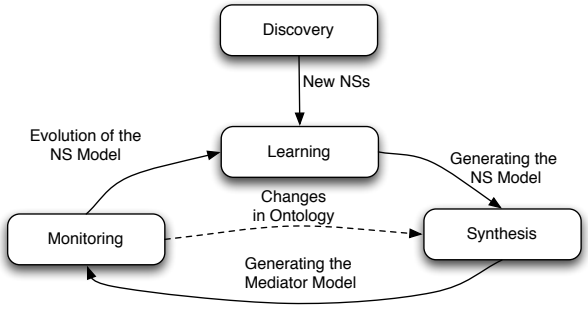

Fig. 14 Evolution of Runtime Models

As depicted in Figure 14, the system is in this case a closed-loop system to better deal with the partial knowledge it has about the environment. Indeed, closed-loop systems have been recognised as fundamental to deal with uncertainty [4, 24]. Then a major challenge is to manage efficiently the changes of the networked systems models in order to re-synthesise the mediator in an incremental way. Evolution of the runtime models is an area of high research relevance and potential. We hope that the experiences discussed and shown in this paper pave the way for new achievements.

4 Related Work

The related work is presented in three different domains: (1) Mediation at Runtime, (2) Self-representation and Reflection, and (3) Software Synthesis at Runtime. The first is about the area of application where we have used our approach, while the other two are related to specific runtime models issues. We also talk about (4) Future Trends in the area of models@run,time.

(1) Mediation at Runtime. Mediation has deserved a large amount of work in various domains, e.g., database integration [31], communication-protocol conversion [34], component adaptation [54], control-system supervision [50], connector wrapping [48], and Semantic Web Services composition [51]. In particular, Yellin and Strom [54] devise an approach to the semi-automated generation of mediators based upon declarative interface mapping. They devise an algorithm to check behavioural compatibility between systems without relying on model checking but assuming that the interface mapping is not ambiguous. While the generation of mediators was mainly a design-time concern, the increasing openness of today's highly dynamic and complex systems makes the mediator generation shifting towards runtime. Denaro *et al.* [20] propose a solution to interoperability across different implementations of the same standard interface in two phases. At design time, developers define common misuse cases of the interface and define the corresponding healing strategies. At runtime, the system is tested against these case and the appropriate healing strategy is applied. At the architectural-level, Chang *et al.* [17]

propose to use *healing connectors* to solve integration problem at runtime by applying healing strategies defined by the developers of the Commercial off-the-shelf (COTS) components.

Another direction is to rely on ontologies to automate the generation of the mediator. Hence, WSMO [18] proposes a runtime framework, the Web Service Execution Environment (WSMX), to mediate interaction between heterogeneous services by inspecting their individual protocols and performing the necessary translation on basis of pre-defined mediation patterns. However, there is no guarantee that the composition of these patterns will not lead to a deadlock. To ensure the correctness of mediation, Cavallaro *et al.* [16] consider the semantics of data and relies on model checking to automatically identify mapping scripts between interaction protocols. Nevertheless, they propose to perform the interface mapping beforehand so as to align the actions of both systems. However many mappings may exist and should be considered during the mediator generation. Indeed, the interface and behavioural descriptions are inter-related and should be considered in conjunction. Moreover, they focus on the mediation at the application layer assuming the use of Web services for the underlying middleware.

The aforementioned research initiatives have made excellent contributions. However, in environments where there is little or no knowledge about the systems that are going to meet and interact, the generation of suitable mediators must happen at runtime whereas in all these approaches, the mediator models or some mediation strategies and patterns are known *a priori* and applied at runtime. In our approach, the construction of the individual models as well as the generation of the appropriate mediator are performed at runtime.

(2) Self-representation and Reflection. Several research approaches have used architectural-based models as the runtime representations of the system to support the treatment of runtime phenomena [53, 19, 22, 40]. In contrast to runtime models that represent directly behavior as in our work, architectural runtime models represent structural views of the running system. Different from [53, 19, 22], and as in [40], we maintain runtime models causally connected with the running system (i.e. we use reflection). Causal connection means that if the model is modified, the running system will change correspondingly. Finally, and different from those approaches that use architectural runtime models to represent structural views of the running system, our approach deals directly with behaviour semantics based on the LTS-based models and not just with architectural notions. To the best of our knowledge, our approach is the first to pursue this.

(3) Software Synthesis at Runtime. Morin *et al* [40] and Welsh *et al.* [52] also synthesize software artifacts supported by the runtime models. The authors of [40] generate the adaptation logic (i.e. reconfiguration scripts) to reconfigure the system by comparing the current configuration of the running system with the model which represents the target configuration. In [52], Welsh *et al.* also generate the adaptation logic, but different from [40], they use runtime requirement models. None of them infer information from the runtime system to create the runtime models, i.e. in their cases, the runtime

model is defined before execution. In our case, we novelly use machine learning techniques to infer knowledge that will be used to create the runtime model during execution.

(4) Future trends in Models@run.time. As seen above and in [9,6], models@run.time so far has been used in different areas e.g. dynamic architectures, self-adaptation, and requirements-aware systems [45] among others. These research initiatives have focused on runtime models which specification is designed before the system's execution. However, recently, researchers have started to envision the use of runtime models in providing intelligent support to software at runtime [5] as the line between development models and runtime models gets blur [9]. Also, runtime models look useful when tackling the uncertainty [52] common in the modern and future software systems [24]. To be able to design these future software systems, inferring the knowledge necessary to conceived runtime models during execution is crucial. In this paper, we have shown early results to conceived runtime models, based on information about the running system and inferred using learning techniques during runtime. Finally, we highlight the need of efficient formal methods at runtime to provide effective support in producing high-integrity systems [14].

5 Conclusions

In this paper, we have demonstrated the novel and unique use of runtime models to support the dynamic synthesis of software. Our way of using runtime models captures syntax and also semantics of behaviour and supports runtime reasoning. Prior models@runtime approaches have generally concentrated on architectural-based runtime models and self-adaptation of existing software artifacts. However, such artefacts cannot always be produced in advance, and we believe that models@runtime have a fundamental role to play in the production of dynamic, adaptive, and on-the-fly software. To support such a vision, two important methods underpin our approach:

- the realisation of runtime models during the execution of the system. Crucially, we have illustrated how the required runtime models are automatically inferred during execution and refined by exploiting learning and synthesis techniques.
- the use of these runtime models to support the dynamic synthesis of software. This dynamic software synthesis approach relies on a formal foundation; during runtime, mediators are formally characterized to allow the runtime synthesis of software. To do that, LTS based models were used to define the matching and mapping relationships between mismatching protocols. Such relationships allow the formal definition of the algorithm that synthesized mediators.

As a first step in illustrating the potential of our solution, we have applied our approach in the specific area of emergent middleware to support on-the-fly interoperability. In particular cases requiring spontaneous interaction, the

use of models@runtime provided the necessary technologies to produce the required middleware software on-the-fly. Here, we leveraged the tools available from the CONNECT project to realize our vision. From this initial work, we have shown how systems can infer information to build runtime models during execution. Importantly, ontologies were exploited to enrich the runtime models and facilitated the mutual understanding required to perform the matching and mapping between the networked heterogeneous systems. Such reasoning about information that was not necessarily known before execution, is in contrast to the traditional use of runtime models.

Finally, we argue that to support future-proof software systems, the focus of software development should shift away from a traditional approach where environmental conditions are foreseen and behaviour of the system is coded accordingly. Dynamic approaches are required where components and/or services are dynamically discovered and then composed together to recreate the system according to the current requirements and environmental contexts. This dynamic composition requires the synthesis of software on-the fly as shown in this paper. We believe that the use of runtime models will play an important role and hope that the approach we have presented provides the foundations for further advances in the area.

Acknowledgements This work is carried out as part of the European FP7 ICT FET CONNECT (`http://connect-forever.eu/`) project and the Marie Curie Fellowship Requirements@run.time. The authors would like to thank Antoine Leger from Thales for valuable input about the GMES case study.

References

1. CONNECT consortium. CONNECT Deliverable D3.3: Dynamic connector synthesis: revised prototype implementation. FET IP CONNECT EU project. http://hal.inria.fr/hal-00695592/.
2. Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
3. Uwe Aßmann, Nelly Bencomo, Betty H. C. Cheng, and Robert B. France. Models@run.time (dagstuhl seminar 11481). *Dagstuhl Reports*, 1(11):91–123, 2011.
4. Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward open-world software: Issue and challenges. *IEEE Computer*, 39(10):36–43, 2006.
5. Luciano Baresi and Carlo Ghezzi. The disappearing boundary between development-time and run-time. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 17–22. ACM, 2010.
6. Nelly Bencomo, Gordon S. Blair, Franck Fleurey, and Cédric Jeanneret. Summary of the 5th international workshop on models@run.time. In *MoDELS Workshops*, pages 204–208, 2010.
7. Amel Bennaceur, Valérie Issarny, Johansson Richard, Moschitti Alessandro, Spalazzese Romina, and Daniel Sykes. Automatic Service Categorisation through Machine Learning in Emergent Middleware. In *Software Technologies Concertation on Formal Methods for Components and Objects (FMCO'11)*, 2011.
8. Amel Bennaceur, Johansson Richard, Moschitti Alessandro, Spalazzese Romina, Daniel Sykes, Rachid Saadi, and Valérie Issarny. Inferring Affordances Using Learning Techniques. In *International Workshop on Eternal Systems (EternalS'11)*, 2011.
9. Gordon Blair, Nelly Bencomo, and Robert B. France. Models@ run.time. *Computer*, 42(10):22–27, 2009.

10. Gordon S. 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'11*, pages 410–430, 2011.
11. Yérom-David Bromberg, Paul Grace, and Laurent Réveillère. Starlink: runtime interoperability between heterogeneous middleware protocols. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 446–455. IEEE, 2011.
12. Yérom-David Bromberg, Paul Grace, Laurent Réveillère, and Gordon S. Blair. Bridging the interoperability gap: Overcoming combined application and middleware heterogeneity. In *Middleware*, pages 390–409, 2011.
13. Yérom-David Bromberg and Valérie Issarny. INDISS: Interoperable discovery system for networked services. In *Middleware*, pages 164–183, 2005.
14. Radu Calinescu and Shinji Kikuchi. Formal methods @ runtime. In *Monterey Workshop*, pages 122–135, 2010.
15. Mauro Caporuscio, Pierre-Guillaume Raverdy, Hassine Moungla, and Valérie Issarny. ubisoap: A service oriented middleware for seamless networking. In *ICSOC*, pages 195–209, 2008.
16. Luca Cavallaro, Elisabetta Di Nitto, and Matteo Pradella. An automatic approach to enable replacement of conversational services. In *Proc. ICSOC/ServiceWave*, pages 159–174. Springer, 2009.
17. Hervé Chang, Leonardo Mariani, and Mauro Pezzè. In-field healing of integration problems with cots components. In *ICSE*, pages 166–176, 2009.
18. Emilia Cimpian and Adrian Mocan. WSMX process mediation based on choreographies. In *Proceedings of Business Process Management Workshop*, pages 130–143, 2005.
19. Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 21–26, New York, NY, USA, 2002. ACM.
20. Giovanni Denaro, Mauro Pezzè, and Davide Tosi. Ensuring interoperable service-oriented systems through engineered self-healing. In *ESEC/SIGSOFT FSE*, pages 253–262, 2009.
21. Ting Deng, Wenfei Fan, Leonid Libkin, and Yinghui Wu. On the aggregation problem for synthesized web services. In *Proc. of the 13th International Conference on Database Theory, ICDT*, pages 242–251, 2010.
22. Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *Software IEEE*, 23(2):62–70, 2006.
23. Carlos A. Flores-Cortés, Paul Grace, and Gordon S. Blair. Sedim: A middleware framework for interoperable service discovery in heterogeneous networks. *TAAS*, 6(1):6, 2011.
24. David Garlan. Software engineering in an uncertain world. In *FoSER*, pages 125–128, 2010.
25. J.C. Georgas, A. van der Hoek, and R.N. Taylor. Using architectural models to manage and visualize runtime adaptation. *Computer*, 42(10):52 –60, oct. 2009.
26. Paul Grace, Gordon S. Blair, and Valérie Issarny. Emergent middleware. *ERCIM News*, 2012(88), 2012.
27. Armin Haller, Emilia Cimpian, Adrian Mocan, Eyal Oren, and Christoph Bussler. Wsmx - a semantic service-oriented architecture. In *ICWS*, pages 321–328, 2005.
28. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
29. Valérie Issarny, Amel Bennaceur, and Yerom-David Bromberg. Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In *SFM-11*, volume 6659 of *Lecture Notes in Computer Science*, pages 217–255. Springer, 2011.
30. Valérie Issarny, Bernhard Steffen, Bengt Jonsson, Gordon S. Blair, Paul Grace, Marta Z. Kwiatkowska, Radu Calinescu, Paola Inverardi, Massimo Tivoli, Antonia Bertolino, and Antonino Sabetta. Connect challenges: Towards emergent connectors for eternal networked systems. In *ICECCS*, pages 154–161, 2009.
31. Vanja Josifovski and Tore Risch. Integrating heterogenous overlapping databases through object-oriented transformations. In *VLDB*, pages 435–446, 1999.
32. Robert M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

33. Ivo Krka, Yuriy Brun, Daniel Popescu, Joshua Garcia, and Nenad Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *ICSE (2)*, pages 179–182, 2010.
34. Simon S. Lam. Protocol conversion. *IEEE Trans. Software Eng.*, 14(3):353–362, 1988.
35. Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
36. Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *ICSE*, pages 501–510, 2008.
37. Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next generation learnlib. In *TACAS*, pages 220–223, 2011.
38. Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.
39. Sonia Ben Mokhtar, Pierre-Guillaume Raverdy, Aitor Urbieta, and Roberto Speicys Cardoso. Interoperable semantic and syntactic service discovery for ambient computing environments. *IJACI*, 2(4):13–32, 2010.
40. Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models at runtime to support dynamic adaptation. *IEEE Computer*, pages 46–53, October 2009.
41. Brice Morin, Olivier Barais, Grégory Nain, and Jean-Marc Jézéquel. Taming dynamically adaptive systems using models and aspects. In *ICSE*, pages 122–132, 2009.
42. Natalya Fridman Noy, Abhita Chugh, William Liu, and Mark A. Musen. A framework for ontology evolution in collaborative environments. In *International Semantic Web Conference*, pages 544–558, 2006.
43. Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications*, 14(3):54–62, 1999.
44. Peyman Oreizy, David S. Rosenblum, and Richard N. Taylor. On the role of connectors in modeling and implementing software architectures. Technical Report 98-04, University of California, Irvine, 1998.
45. Pete Sawyer, Nelly Bencomo, Jon Whittle, Emmanuel Letier, and Anthony Finkelstein. Requirements-aware systems: A research agenda for re for self-adaptive systems. *Requirements Engineering, IEEE International Conference on*, 0:95–103, 2010.
46. Douglas C. Schmidt. Model driven engineering. *IEEE Computer*, pages 25–31, 2006.
47. Hui Song, Gang Huang, Yingfei Xiong, Franck Chauvel, Yanchun Sun, and Hong Mei. Inferring meta-models for runtime system data from the clients of management apis. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part II*, MODELS'10, pages 168–182, Berlin, Heidelberg, 2010. Springer-Verlag.
48. Bridget Spitznagel and David Garlan. A compositional formalization of connector wrappers. In *ICSE*, pages 374–384, 2003.
49. Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In *SFM*, pages 256–296, 2011.
50. Andreas Tolk and J. Mark Pullen. Using web services and data mediation/storage services to enable command and control to simulation interoperability. In *DS-RT*, pages 27–34, 2005.
51. R. Vaculin, R. Neruda, and K. Sycara. The process mediation framework for semantic web services. volume 3, pages 27–58. Inderscience, 2009.
52. Kristopher Welsh, Pete Sawyer, and Nelly Bencomo. Towards requirements aware systems: Run-time resolution of design-time assumptions. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 560–563, 2011.
53. Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. Discotect: A system for discovering architectures from running systems. In *In Proc. 26th International Conference on Software Engineering*, pages 470–479, 2004.
54. D.M. Yellin and R.E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):292–333, 1997.