The Open University

Open Research Online

The Open University's repository of research publications
and other research outputs

Automated synthesis of mediators to support component interoperability

Journal Item

How to cite:

Bennaceur, Amel and Issarny, Valérie (2015). Automated synthesis of mediators to support component interoperability. IEEE Transactions on Software Engineering, 41(3) pp. 221–240.

For guidance on citations see FAQs.

© 2014 IEEE

Version: Accepted Manuscript

Link(s) to article on publisher's website:
http://dx.doi.org/doi:10.1109/TSE.2014.2364844

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online's data policy on reuse of materials please consult the policies page.

oro.open.ac.uk

Automated Synthesis of Mediators to Support Component Interoperability

Amel Bennaceur and Valérie Issarny

Abstract—Interoperability is a major concern for the software engineering field, given the increasing need to compose components dynamically and seamlessly. This dynamic composition is often hampered by differences in the interfaces and behaviours of independently-developed components. To address these differences without changing the components, mediators that systematically enforce interoperability between functionally-compatible components by mapping their interfaces and coordinating their behaviours are required. Existing approaches to mediator synthesis assume that an interface mapping is provided which specifies the correspondence between the operations and data of the components at hand. In this paper, we present an approach based on ontology reasoning and constraint programming in order to infer mappings between components' interfaces automatically. These mappings guarantee semantic compatibility between the operations and data of the interfaces. Then, we analyse the behaviours of components in order to synthesise, if possible, a mediator that coordinates the computed mappings so as to make the components interact properly. Our approach is formally-grounded to ensure the correctness of the synthesised mediator. We demonstrate the validity of our approach by implementing the MICS (Mediator synthesIs to Connect Components) tool and experimenting it with various real-world case studies.

Index Terms—Interoperability, Constraint Programming, Automated Synthesis, Mediators, Protocols

1 Introduction

Interoperability is a fundamental challenge for software engineering [1]. Today's systems are increasingly developed by reusing and integrating existing components. However, even though these components should be able to integrate since, at some high level of abstraction, they require and provide compatible functionalities, the conflicting assumptions that each of them makes about its running environment hinder their successful interoperation. Indeed, components cannot readily be reused when inconsistent semantics are buried in their interfaces or behaviours [2].

There exists a wide range of approaches to enable independent components to interoperate [3], [4]. Solutions that require performing changes to the components are usually not appropriate since the components to be integrated may be built by third parties (e.g. COTS—Commercial Off-The-Shelf—components or legacy systems); no more appropriate are approaches that prune the behaviour leading to mismatches since they also restrict the components' functionality [5]. Therefore, many solutions that aggregate the disparate components in a non-intrusive way, i.e. without changing the internal implementation of these components, have been proposed [4], [6]–[9]. These solutions use intermediary middleware entities, called *mediators* [10], [11] (also called connector wrappers [6] or mediating adapters [7]), to connect heterogeneous components despite disparities in their data and/or interaction models by performing the necessary coordination and translations while keeping them loosely-coupled.

Nevertheless, creating mediators requires a substantial development effort and a thorough knowledge of the application-domain. Moreover, the increasing complexity of today's software components, sometimes referred to as Systems of Systems [12], makes it difficult to develop 'correct' mediators manually [13]; correct mediators guarantee that the components interact without errors (e.g. deadlocks) and reach their termination successfully. Therefore, formal approaches are necessary to characterise components precisely, reason about their interaction, and generate mediators automatically [7], [14].

First, automatically generating mediators helps developers to manage the bulk of the systems they need to integrate. Indeed, developers increasingly have to incorporate to their systems convenient services such as instant messaging or social interaction. Although, most of these services provide similar functionalities, their interfaces are usually heterogeneous. For example, Google Talk and Facebook chat both have instant messaging capabilities but expose them using different interfaces. Similarly, Facebook and Google+ allow social interaction between their users using distinct interfaces. By providing an automated mediation solution, developers no longer have to struggle with heterogeneous interfaces since the generated mediators perform the necessary translations to compensate for

- *A. Bennaceur is with the Department of Computing, The Open University, United Kingdom. This work was performed during her PhD studies at Inria.*
E-mail: amel.bennaceur@open.ac.uk
- *V. Issarny is with the MiMove team, Inria, France.*
E-mail: valerie.issarny@inria.fr

the differences between these interfaces. The automated generation of mediators also enables seamless and spontaneous interaction between components in highly dynamic and extremely heterogeneous environments by computing, at runtime, a mediator that ensures their interoperation.

Existing approaches to the automatic generation of mediators assume the correspondence between the operations of the mediated components to be given in terms of an *interface mapping* [14] (also called adaptation contract [7]). Identifying such correspondence requires not only knowledge about the components but also knowledge about the domain itself.

Research on knowledge representation and artificial intelligence has now made it possible to model and automatically reason about domain information precisely, if not with the same nuanced interpretation that a developer might [15]. In particular, *ontologies* build upon a sound logical theory to provide a machine-interpretable means to reason, automatically, about the semantics of data based on the shared understanding of the domain [16]. They play a valuable role in software engineering by supporting the automated integration of knowledge among teams and project stakeholders [17]. Ontologies have also been widely used for modelling Semantic Web Services and achieving service discovery and composition [18]. OWL-S (Semantic Markup for Web Services) uses ontologies to model both the functionality of a Web Service and the associated behaviour, i.e. the protocol according to which this Web Service interacts [19]. Besides ontology-based modelling, WSMO (Web Service Modelling Ontology) relies on ontologies to support runtime mediation based on pre-defined patterns but without ensuring that such mediation does not lead to an erroneous execution (e.g. deadlock) [20]. Hence, although ontologies have long been advocated as a key enabler in the context of service mediation, no principled approach has been proposed for the automated synthesis of mediators by systematically exploiting ontologies. We argue that interoperability should not be achieved by defining yet another ontology nor yet another middleware but rather by exploiting the knowledge encoded in existing domain-specific ontologies together with the behavioural description of components and using them to generate mediators automatically. These mediators bridge the interoperability gap between heterogeneous components by delivering information when it is needed, in the right format, with the original business context intact.

This paper focuses on functionally-compatible components, i.e. components that at some high level of abstraction require and provide compatible functionalities, but are unable to interact successfully due to mismatching interfaces and behaviours. This paper concentrates on service-based components (e.g. Web Services) rather that code-based components (e.g. Java libraries). We propose an approach that combines ontology reasoning and constraint programming in order to generate a mapping between the interfaces of these components. Then, we use the generated mappings and examine the behaviours of both components to automatically synthesise a mediator that ensures their safe interaction. The mediator, if it exists, enables the components to progress synchronously and reach their final states with the guarantee that the composed system is free from deadlocks. Specifically, our contributions are:

- *Efficient interface mapping using semantic reasoning and constraint programming.* We reason about the semantics of data and operations of each component and use a domain ontology to identify the semantic correspondences between the interfaces of the components, i.e. interface mapping. Interface mapping guarantees that operations from one component can safely be performed using operations from the other component.
- *Automated synthesis of mediators.* We explore the behaviours of the two components and generate the mediator that composes the computed mappings so as to force the components to progress synchronously. The mediator, if it exists, guarantees that the mediated system is deadlock-free.
- *Tool-support for automated mediation.* We further demonstrate the feasibility of our approach through the MICS (Mediator Synthesis to Connect Components) tool and illustrate its usability using real-world case studies involving heterogeneous components. Our semantic-based approach to the automated generation of mediators leads to a considerable increase in the quality of interoperability assurance between heterogeneous components: it removes the programming effort and ensures that the components interact successfully while preserving efficient execution time.

This paper is organised as follows. Section 2 introduces the interoperable file management example that illustrates the need for mediators to facilitate interoperation between independently-developed components. It also presents the formalism we use to specify our inputs, which consists of ontologies to model domain knowledge and labelled transition systems to model the behaviour of components. Section 3 moves into the solution space, defining the formal methodology to compute interface mapping efficiently. Section 4 describes the algorithm used to generate mediators. Section 5 presents the MICS tool used to synthesise mediators and deploy them in the environment. Section 6 reports on the experiments we conducted to validate our approach. Section 7 examines related work. Section 8 discusses future work. Finally, Section 9 concludes the paper.

2 INTEROPERABILITY USING MEDIATORS

To highlight the problem of interoperability in distributed systems, we examine the case of two heterogeneous file management systems: WebDAV and Google Drive. We present the models we use to describe them and outline our approach for the automated synthesis of mediators that enable the successful interoperation of these systems despite differences in their interfaces and behaviours.

2.1 The Interoperable File Management Example

The migration from desktop applications to Web-based services is scattering user files across a myriad of remote servers (e.g. Apple iCloud, Google Drive, and Microsoft Skydrive). This dissemination poses new challenges for users, making it more difficult for them to organise, search, and manipulate their files using their preferred applications, and share them with other users. This situation, though cumbersome from a user perspective, unfortunately reflects the way file management applications—like many other existing applications—have evolved. As a result, users are forced to juggle between a plethora of applications to manage their files instead of using their favourite application regardless of the service it relies on or the standard on which it is based.

Among the protocols allowing collaborative management of files, WebDAV (Web Distributed Authoring and Versioning) is an IETF specification that extends the Hypertext Transfer Protocol (HTTP) to allow users to create, read and change documents remotely. It defines a set of properties to query and manage information about these documents, organise them using collections, and defines a locking mechanism in order to assign a unique editor of a document at any time. Another example is the Google Drive service that lets users create, store, and search Google documents and collections. It also allows users to share and collaborate online in the editing of these documents. These functionalities can be accessed using a Web browser or using the Google proprietary API.

Fig. 1. Connecting a WebDAV client to Google Drive service

Although WebDAV and Google Drive offer similar functionalities and use HTTP as the underlying transport protocol, they are unable to interoperate. For example, a user cannot access his Google Drive documents using his favourite WebDAV client (e.g. Mac Finder) as depicted in Figure 1. This is mainly due to the syntactic naming of data and operations used in each application, and the protocols according to which these operations are performed. Our goal is to synthesise a mediator automatically in order to allow these two components to interact properly. In order to reason about component interoperability and automatically synthesise the mediator, we need to model the components and their domain as described in the following.

2.2 Modelling Domains using Ontologies

Each application domain has its own vocabulary. This vocabulary has to be modelled explicitly in order to allow computers to conduct automated reasoning about the domain. Ontologies provide experts with a means to formalise the knowledge about the domain as a set of axioms that make explicit the intended meaning of a vocabulary [21]. Besides general purpose ontologies, such as dictionaries (e.g. WordNet[1]), there is an increasing number of ontologies available for various domains such as biology, geoscience, and social networks, which in turn foster the development of a multitude of search engines specially targeted for ontologies on the Web [22].

Ontologies are supported by a logic theory to reason about the properties and relations holding between the various domain entities. In particular, OWL (Web Ontology Language[2]), which is the W3C standard language to model ontologies, is based on description logics. More specifically, we focus on OWL DL, which is based on a specific description logic, $\mathcal{SHOIN}(\mathcal{D})$ [16]. In the rest of the paper, DL refers to this specific description logic.

While traditional formal specification techniques (e.g. first-order logic) might be more powerful, DL offers crucial advantages: it excels at modelling domain-specific knowledge while providing decidable and efficient reasoning algorithms [23]. DL specifies the vocabulary of a domain using concepts and relationships between these concepts. Each concept is given a definition as a set of logical axioms, which can either be atomic or defined using different operators such as disjunction ($\sqcup$), conjunction ($\sqcap$), and quantifiers ($\forall$, $\exists$). Relationships between concepts are defined using object properties. The semantics of DL is defined in terms of an interpretation $\mathcal{I}$ consisting of a non-empty set $\Delta^{\mathcal{I}}$ (the domain of the interpretation) and an interpretation function, which assigns to every atomic concept A a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to every object property R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. We provide an overview of the semantics of the basic DL operators in Table 1 and refer the interested reader to [16] for further details.

For example, consider the extract of the file management ontology depicted in Figure 2. We build this ontology by extending the NEPOMUK File Ontology.[3] The NEPOMUK File Ontology (NFO)

1. http://www.w3.org/TR/wordnet-rdf/
2. http://www.w3.org/TR/owl2-overview/
3. http://www.semanticdesktop.org/ontologies/nfo/

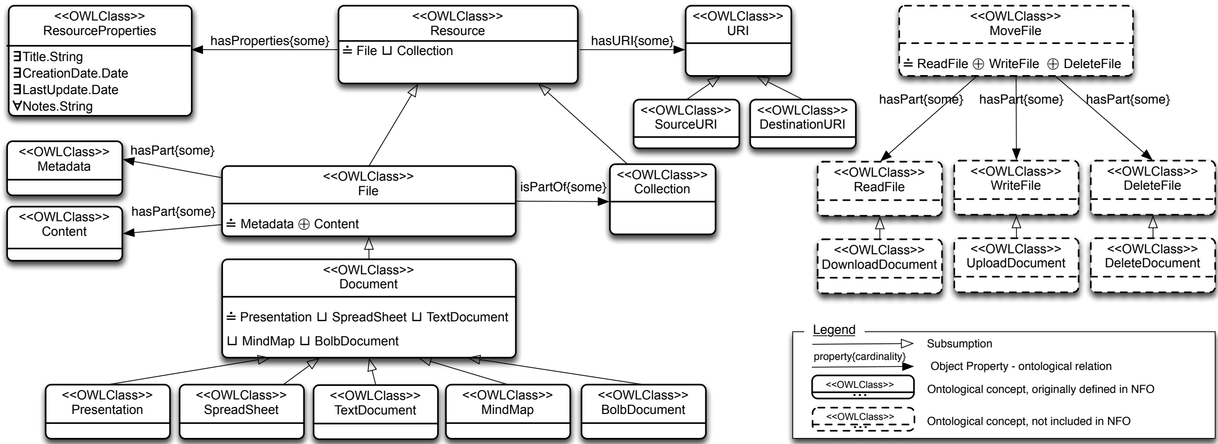

Fig. 2. The file management ontology

DL Opertor	Semantics
Conjunction	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Disjunction	$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Universal quantifier	$(\forall R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y.(x,y) \in R^{\mathcal{I}} \Rightarrow y \in C^{\mathcal{I}}\}$
Existential quantifier	$(\exists R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y.(x,y) \in R^{\mathcal{I}} \ \wedge y \in C^{\mathcal{I}}\}$
Aggregation	$(C \oplus D)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y.(x,y) \in \mathsf{hasPart}^{\mathcal{I}}$ $\wedge\, y \in C^{\mathcal{I}} \wedge \exists z.(x,z) \in \mathsf{hasPart}^{\mathcal{I}} \wedge z \in D^{\mathcal{I}}\}$

C and D are concepts and R is an object property.

TABLE 1. Overview of DL operators

defines the vocabulary for describing and relating information elements that are commonly used in file management applications. A Resource can be a file or a directory (i.e. a collection of files). In DL, this is written: Resource $\doteq$ File $\sqcup$ Collection. The *dot* above the "equals" symbol designates a declarative axiom, a.k.a., user-defined axioms. In addition, a Resource concept has some resource properties: Resource $\dot\sqsubseteq$ ∃hasProperties.ResourceProperties. ResourceProperties has a title of the built-in type String as well as dates for creation and last update. ResourceProperties may also have one or more notes of the the built-in type String.

We describe the aggregation of concepts, where different concepts are composed together to build a whole, using the W3C recommendation for part-whole relations—hasPart.[4] A concept E is an aggregation of concepts C and D, written $E = C \oplus D$, providing both C and D are parts of E, i.e. $E = (\exists \mathsf{hasPart}.C) \sqcap (\exists \mathsf{hasPart}.D)$. For example, the File concept is defined as the aggregation of the Metadata and Content concepts, meaning that each file instance $f \in$ File encompasses a Metadata instance ($\exists m \in$ Metadata $\wedge\ (f,m) \in$ hasPart), as well as a Content instance ($\exists c \in$ Content $\wedge\ (f,c) \in$ hasPart).

DL is used to support automatic reasoning about concepts and their relationships, in order to infer new relations that may not have been recognised by the ontology designers. Traditionally, the basic reasoning mechanism is *subsumption*.

Definition 1 (Subsumption): A concept C is subsumed by a concept D, written $C \sqsubseteq D$ iff any instance (sometimes called individual) of C also belongs to D. In addition, all the relationships in which D instances can be involved are applicable to C instances, i.e. all properties of D are also properties of C.

For example, since File $\sqsubseteq$ Resource and Resource $\dot\sqsubseteq$ ∃hasProperties.ResourceProperties, it can be inferred that a File also has some properties File $\sqsubseteq$ ∃hasProperties.ResourceProperties. Subsumption is a partial order relation, i.e. it is reflexive, antisymmetric, and transitive. As a result, the ontology can be represented as a hierarchy of concepts, which can be automatically inferred by ontology reasoners based on the axioms defining the ontological concepts.

2.3 Modelling Components by Combining Ontologies and Labelled Transition Systems

While ontologies allow domain knowledge to be defined formally, describing knowledge about the software components themselves is equally important. To enable automated reasoning about interoperability between components, we must represent explicitly all the knowledge implicitly encoded in each component. This knowledge is explicitly represented in the component model that describes the component's *capability*, *interface signature* and *behaviour*. Figure 3 depicts the model of the WebDAV client.

The *capability* gives a macro-view of the component by specifying the high-level functionality it requires from or provides to its environment [19]. For example, the WebDAV client (denoted *WDAV*) whose model is described in Figure 3 requires a file management capability.

The *interface signature*, or simply *interface*, of the component gives a finer grained description of the operations and data that the component manipulates. More precisely, the component's interface defines the

4. http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/

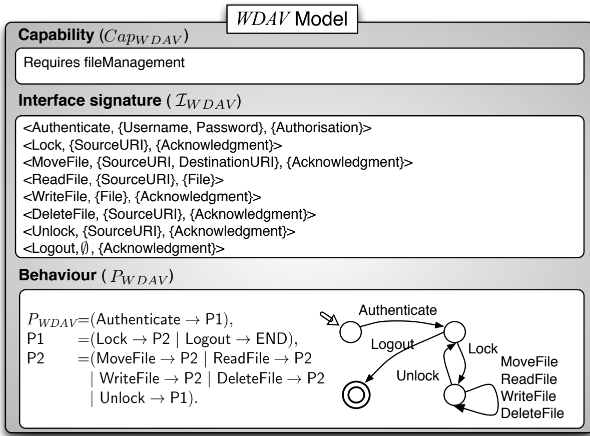

Fig. 3. Model of the WebDAV client

set of observable *actions* that the component exchanges with its running environment. It is partitioned into *required* and *provided* actions, with the understanding that required actions are received from and controlled by the environment, whereas provided actions are emitted and controlled by the component. A required action $\alpha = \langle op,\ i,\ o \rangle$, where the symbols op, i, and o are references to concepts in a domain ontology $\mathcal{O}$, represents a client-side invocation of an operation op by sending the appropriate input data i and receiving the corresponding output data o. For example, the $\langle \mathsf{ReadFile}, \{\mathsf{SourceURI}\}, \{\mathsf{File}\} \rangle$ action, belonging to *WDAV*'s interface, designates a required action that perform the ReadFile operation by taking SourceURI as input and producing File as output. ReadFile, SourceURI, and File are all concepts in the file management ontology depicted in Figure 2. In order to reason about the meaning of actions rigorously, the component model is focused on the semantic descriptions of actions, specified using references to the domain ontology, rather than their syntactic descriptions, specified using XML schemas. On the practical side, the syntactic descriptions of all operations and input/output data include semantic annotations referring to concepts in the domain ontology. These annotations can either be specified by the developer of the component or automatically inferred using learning techniques [24], [25].

The dual provided action $\overline{\beta} = \langle \overline{op},\ i,\ o \rangle$ uses the inputs and produces the corresponding output.[5] Since *WDAV* only invokes operations from the WebDAV service, its interface contains only required actions.

The *behaviour* of a component specifies its interaction with the environment and models how the actions of its interface are coordinated in order to achieve the specified capability. We build upon state-of-the-art approaches to formalise component interactions [6] using Finite State Processes (FSP). FSP [26] is a process algebra that has proven to be a convenient formalism for specifying concurrent components, analysing, and reasoning about their behaviours. The syntax of FSP is summarised in Table 2, while the interested reader is referred to [26] for further details. The behavioural specification of the WebDAV client (P_{WDAV}) in FSP is depicted in Figure 3. To simplify the presentation, the actions are represented using the operation concept alone. Note that all the concepts used to describe actions belong to the file management ontology even though they are not represented in Figure 2. WebDAV clients first login. To perform any operation, clients have to lock the resource, execute the desired operation(s) and then unlock it again. Finally, they log out to terminate.

The semantics of FSP is given in terms of Labelled Transition Systems (LTS) [27]. The LTS associated with P_{WDAV} is depicted in Figure 3. The LTS interpreting a process P can be regarded as a directed graph whose nodes represent the process states and each edge is labelled with an action belonging to the component's interface. There exists a start node from which the process begins its execution. There exists also a final state, which is associated with the END process, that indicates a successful termination of the FSP process. The expression $s \xrightarrow{a} s'$ specifies that if the process is in state s and engages in an action a, then it transits to state s'. The expression $s \stackrel{X}{\Rightarrow} s'$ where $X = \langle a_1, \ldots, a_n \rangle$, a_i being actions of the component's interface, is used as a shorthand denoting that P transits from state s to state s' after it engages in a sequence of actions X. When composed in parallel, processes synchronise on dual actions while actions that are in the alphabet of only one of the two processes can be executed independently. Hence, we assume synchronous semantics. Although the asynchronous semantics can be easily implemented, it is hard to reason about interacting processes under these semantics; in general, properties of the system such as deadlocks are undecidable [28].

FSP Syntax	
αP	P's alphabet
$a \rightarrow P$	Action prefix
$a \rightarrow P \vert b \rightarrow P$	Choice
$P; Q$	Sequential composition
$P \Vert Q$	Parallel composition
END	Built-in process, denotes successful termination

TABLE 2. FSP overview

Finally, we can reasonably assume that the semantic annotations as well as the behavioural specification of a component are either provided with or derivable from the component's interface. First, there are various approaches and standards that emphasise the need and the importance of having such a complete specification [20], [23], [29]. Second, there is an increasing number of advanced learning techniques and tools to support the inference of ontological annotations [24] as well as the extraction of behavioural models [25], [30], [31].

5. We use the overline as a convenient shorthand to denote provided actions.

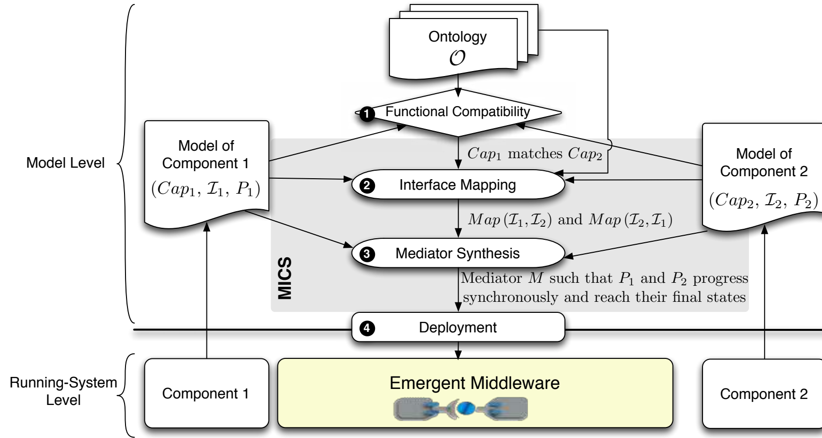

Fig. 4. Overview of our approach to the automated synthesis of mediators

2.4 Automated Synthesis of Mediators: Overview

Although the WebDAV client requires a file management functionality that may be provided by the Google Drive service, their interactions lead to an erroneous execution, namely a *mismatch*. In general, mismatches may occur due to inconsistencies in:

- *The names and types of input/output data and operations.* For example, resource containers are called *folders* in Google Drive and *collections* in WebDAV. Google Drive also distinguishes between types of documents (e.g. presentation, spreadsheet) whereas WebDAV considers them all as files.
- *The granularity of operations*. WebDAV provides an operation for moving files from one location to another whereas Google Drive does not. Still, the move operation can be realised using existing operations to achieve the same task, i.e. it can be carried out by performing the upload, download, and delete operations offered by Google Drive.
- *The ordering of operations.* WebDAV requires operations on files to be preceded by a lock operation and followed by an unlock. Google Drive does not have such a requirement. Still, it allows users to restrict or release access to a document by changing its sharing settings. Hence, although the operations of the two systems can be mapped to one another, the sequencing of actions required by WebDAV is not enforced by Google Drive.

The mediator solves the aforementioned mismatches by compensating for the differences in the data and operations of the components under consideration, and by coordinating their behaviours in order to respect the sequences of actions expected by both of them. In order to reason about component interoperation automatically and generate the appropriate mediator, we rely on the rigorous modelling of both components as defined in Section 2.3. Also, as discussed in Section 2.2, we use an off-the-shelf ontology $\mathcal{O}$ to represent domain knowledge and utilise it to reason about the relations holding between the data and operations of the two components. We assume that the models of the components are valid, i.e. they represent the actual functional (capability, interface) and behavioural semantics of the components at hand, and that all the ontological concepts referred to in the models of both components belong to the ontology $\mathcal{O}$. We recognise that this might not be the case in practice and plan to deal with partial or changing models as well as heterogeneous ontologies in future work.

We also use the ontology $\mathcal{O}$ to verify, prior to any mediation, whether it makes sense for the components to interact with each other by checking if they are *functionally compatible*, i.e. if the capability required by one component subsumes the capability provided by the other component in a way similar to capability matching in Semantic Web Services [32] (see Figure 4-❶).

A significant role of the mediator is to convert data available on one side and make it suitable and relevant to the other. This conversion can only be carried out if there exists a semantic correspondence between the actions required by one component and those provided by the other component, that is, interface mapping (see Figure 4-❷). The main idea is to use the domain-specific information embodied in the ontology $\mathcal{O}$ in order to select among all the possible combinations of the actions of the components' interfaces only those preserving the semantics of data and operations and for which automated transformations can be safely performed. Subsequently, we generate the mapping processes that receive the input/output data from one component, execute the necessary transformations, and send it back to the other component.

One important aspect of interface mapping is that it can be ambiguous, i.e. the same sequence of actions of one component may be achieved using different sequences of actions provided by the other component. It then becomes crucial to combine the mapping processes in a way that guarantees that the two components progress and reach their final

states without errors that cannot be caught by the type system alone (e.g. deadlock). The gist of the approach is then to generate a mediator M that coordinates the mapping processes and guarantees that the behaviour of the mediated system, which is represented through the parallel composition of the LTSs of both components together with that of the mediator is free from deadlocks (see Figure 4-❸). Notions such as bisimulation or refinement [33] cannot be applied since they assume the use of the same set of actions (alphabet). Neither is it possible to relabel the LTSs beforehand since the mappings are not only based on one-to-one correspondences but rather many-to-many, and also because the same action (sequence of actions) may be translated differently depending on the state in which the system is.

Finally, the mediator is deployed and enacted as an *emergent middleware* [34], which interprets the mediator model and executes the necessary transformations to allow the components to interact properly (see Figure 4-❹).

In Section 3 we explain how to compute interface mapping while in Section 4 we present how to synthesise the mediator. In Section 5, we describe MICS, the supporting tool for the automated synthesis and deployment of mediators.

3 Automated Mapping of Interfaces

Establishing the semantic correspondence between the actions of the components' interfaces is a crucial step towards the synthesis of mediators. In this section, we first specify the conditions under which such a correspondence may be established. Then, we show how to compute interface mapping efficiently using constraint programming [35].

3.1 Mapping Component Interfaces

Let us consider two components' interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$. Mapping $\mathcal{I}_1$ to $\mathcal{I}_2$, written $Map\,(\mathcal{I}_1, \mathcal{I}_2)$, consists in finding all pairs (X_1, X_2) where $X_1 = \langle \alpha_1, \alpha_2, \ldots, \alpha_m \rangle, \alpha_{i=1..m} \in \mathcal{I}_1$ and $X_2 = \langle \overline{\beta_1}, \overline{\beta_2}, \ldots, \overline{\beta_n} \rangle, \overline{\beta_{j=1..n}} \in \mathcal{I}_2$ such that X_1 *maps to* X_2, denoted $X_1 \mapsto X_2$, if the required actions of X_1 can be safely performed by calling the provided actions of X_2. In addition, this pair is *minimal*, that is, any other pair of sequences of actions (X_1', X_2') such that X_1' maps to X_2' would have X_1 as a subsequence of X_1' or X_2 as a subsequence of X_2'. The interface mapping is then specified as follows:

$$
\begin{aligned}
&Map\,(\mathcal{I}_1, \mathcal{I}_2) = \\
&\{\ (X_1, X_2) |\\
&\quad X_1 = \langle \alpha_1, \alpha_2, \ldots, \alpha_m \rangle, \alpha_{i=1..m} \in \mathcal{I}_1\\
&\wedge X_2 = \langle \overline{\beta_1}, \overline{\beta_2}, \ldots, \overline{\beta_n} \rangle, \overline{\beta_{j=1..n}} \in \mathcal{I}_2\\
&\wedge X_1 \mapsto X_2\\
&\wedge \nexists (X_1', X_2') \mid\ X_1' = \langle \alpha_1, \alpha_2, \ldots, \alpha_{m'} \rangle,\ \alpha_{i=1..m'} \in \mathcal{I}_1\\
&\qquad\qquad\qquad \wedge X_2' = \langle \overline{\beta_1}, \overline{\beta_2}, \ldots, \overline{\beta_{n'}} \rangle, \overline{\beta_{j=1..n'}} \in \mathcal{I}_2\\
&\qquad\qquad\qquad \wedge (X_1' \mapsto X_2') \wedge (m' < m) \wedge (n' < n)\\
&\}
\end{aligned}
$$

Likewise, $Map\,(\mathcal{I}_2, \mathcal{I}_1)$ represents the set of all pairs (X_2, X_1), where X_2 is a sequence of required actions of $\mathcal{I}_2$ and X_1 is a sequence of provided actions of $\mathcal{I}_1$, such that X_2 maps to X_1 and this mapping is minimal.

Each mapping relation is associated with a process that specifies how the sequences of actions are coordinated. In the following, we specify the conditions that represent the safety properties which should be satisfied by the mapping relation ($\mapsto$) in order to guarantee the correct replacement of actions. We first give a formal definition in the one-to-one case ($m = 1$ and $n = 1$), which we extend to the one-to-many ($m = 1$ and $n \geq 1$) and many-to-many ($m \geq 1$ and $n \geq 1$) cases. Note that we do not distinguish between the aggregation ($\oplus$) and disjunction ($\sqcup$) constructors when computing the interface mapping as they both represent compositions of concepts. The distinction is however maintained at the ontological level, and also for data translations: in the case of disjunction $E \doteq C \sqcup D$, the translation consists in producing an instance of E by assigning to it either an instance of C or an instance of D. While in the case of aggregation $E \doteq C \oplus D$, an instance of E is produced by combining its parts, i.e. both C and D instances, in the appropriate way.

Definition 2 (One-to-One Mapping): A required action $\alpha = \langle a, I_a, O_a \rangle \in \mathcal{I}_1$ maps to a provided action $\overline{\beta} = \langle \overline{b}, I_b, O_b \rangle \in \mathcal{I}_2$ noted $\alpha \overset{1-1}{\longmapsto} \overline{\beta}$, iff:
1) $b \sqsubseteq a$ 2) $I_a \sqsubseteq I_b$ 3) $I_a \sqcup O_b \sqsubseteq O_a$

The idea behind this mapping is that a required action can be achieved using a provided one if the former supplies the required input data while the latter provides the necessary output data, and the required operation is less demanding than the provided one. This coincides with the Liskov Substitution Principle [36] where ontological subsumption can be used in ways similar to subtyping in object-oriented systems.

As a result, we generate the mapping process M_{1-1} that synchronises with each component by executing its dual action in order to let the two components progress as depicted in Figure 5. The states of the mapping process are labelled so as to reflect the progress of the components; each label is the concatenation of the label of the state of Component 1 and that of Component 2. M_{1-1} performs β (trace $0.0' \rightarrow 0.1'$) so as to synchronise with $\overline{\alpha}$ (trace $0.1' \rightarrow 1.1'$). Hence, the one-to-one mapping process corresponding to $\alpha \overset{1-1}{\longmapsto} \overline{\beta}$ is defined as follows: $M_{1-1}(\alpha, \overline{\beta}) = (\beta \rightarrow \overline{\alpha} \rightarrow \text{END})$.

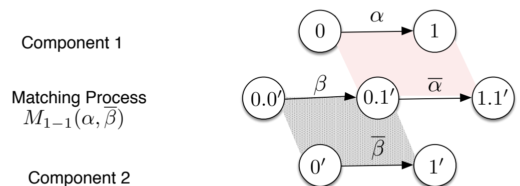

Fig. 5. One-to-one mapping process: $M_{1-1}(\alpha, \overline{\beta})$

Let us consider the ⟨ReadFile, {SourceURI}, {File}⟩ action required by *WDAV* and the $\overline{\langle\text{DownloadDocument}, \{\text{SourceURI}\}, \{\text{Document}\}\rangle}$ action provided by the Google Drive service, which we denote *GDrive*. We can verify from the file management ontology in Figure 2 that: 1) DownloadDocument $\sqsubseteq$ ReadFile, 2) SourceURI $\sqsubseteq$ SourceURI since subsumption is reflexive, and 3) Document $\sqsubseteq$ File. As a result, we generate a corresponding mapping process M_{1-1}(ReadFile, $\overline{\text{DownloadDocument}}$)=(DownloadDocument $\rightarrow$ $\overline{\text{ReadFile}}$ $\rightarrow$ END)., which requires DownloadDocument and provides ReadFile. Therefore, it needs to: (i) receive SourceURI once ReadFile is invoked, (ii) invoke DownloadDocument using SourceURI as input data and receive the associated Document, and (iii) construct and return File (since Document $\sqsubseteq$ File) as a result of the invocation of ReadFile.

***Definition** 3 (**One-to-Many Mapping**):* A required action $\alpha = \langle a, I_a, O_a\rangle \in \mathcal{I}_1$ maps to a sequence of provided actions $X_2 = \langle\overline{\beta_1}, \overline{\beta_2}, \ldots, \overline{\beta_n}\rangle$ where $\overline{\beta_{i=1..n}}=\langle\overline{b_i}, I_{b_i}, O_{b_i}\rangle \in \mathcal{I}_2$, noted $\alpha \overset{1-n}{\longmapsto} \langle\overline{\beta_1}, \ldots, \overline{\beta_n}\rangle$, iff:

1) $\bigsqcup_{i=1}^{n} b_i \sqsubseteq a$
2) $I_a \sqsubseteq I_{b_1}$
3) $I_a \bigsqcup \left(\bigsqcup_{j=1}^{i-1} O_{b_j}\right) \sqsubseteq I_{b_i}$
4) $I_a \bigsqcup \left(\bigsqcup_{j=1}^{n} O_{b_j}\right) \sqsubseteq O_a$

The first condition states that the operation a can be appropriately performed using b_i operations, that is, the disjunction of all b_i is subsumed by a. The second ensures that the sequence of provided actions can be initiated since the input data of the first action I_{b_1} can be obtained from the input data of the required action I_a. The third condition specifies that the input data of each action can be obtained from the data previously received either as input from α or as output from the preceding β_j ($j < i$). The fourth condition guarantees that the required output data O_a can be obtained from the set of data accumulated during the execution of all the provided actions.

As a result, we generate the mapping process $M_{1-n}(\alpha, X_2)$, depicted in Figure 6, which performs all the provided actions (trace $0.0' \rightarrow \cdots \rightarrow 0.n'$) so as to synchronise with the action $\overline{\alpha}$ (trace $0.n' \rightarrow 1.n'$). The mapping process corresponding to $\alpha \overset{1-n}{\longmapsto} \langle\overline{\beta_1}, \ldots, \overline{\beta_n}\rangle$ is: $M_{1-n}(\alpha, X_2) = (\beta_1 \rightarrow \beta_2 \rightarrow \cdots \rightarrow \beta_n \rightarrow \overline{\alpha} \rightarrow \text{END})$.

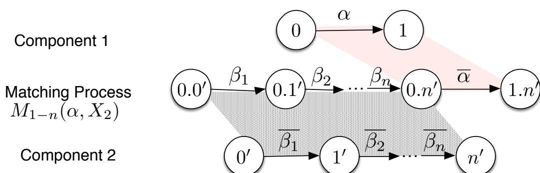

Fig. 6. One-to-many mapping process: $M_{1-n}(\alpha, X_2)$

Let us consider the ⟨MoveFile, {SourceURI, DestinationURI}, {Acknowledgment}⟩ action required by *WDAV* and the three actions: (i) $\overline{\langle\text{DownloadDocument}}$,{SourceURI},{Document}⟩, (ii) $\overline{\langle\text{UploadDocument}}$,{Metadata,Content,DestinationURI}, {Acknowledgment}⟩, and (iii) $\overline{\langle\text{DeleteDocument}}$, {SourceURI}, {Acknowledgment}⟩ provided by *GDrive*. First, the condition on the semantics of the operations is verified, that is, DownloadDocument $\oplus$ UploadDocument $\oplus$ DeleteDocument $\sqsubseteq$ MoveFile. We recall, as stated at the beginning of this section, that we do not distinguish between the aggregation ($\oplus$) and disjunction ($\sqcup$) constructors when computing the interface mapping. Then, both DownloadDocument and DeleteDocument can be performed as they only expect SourceURI as input, which is produced by MoveFile. UploadDocument requires some input data, which can only be provided by DownloadDocument since Document = Metadata $\oplus$ Content (see Figure 2) and thus must be executed after DownloadDocument. Hence, we can have the following mapping:
MoveFile $\mapsto$ ⟨$\overline{\text{DownloadDocument}}$, $\overline{\text{UploadDocument}}$, $\overline{\text{DeleteDocument}}$ ⟩

We can generate a corresponding mapping process M_{1-n}(MoveFile, ⟨$\overline{\text{DownloadDocument}}$, $\overline{\text{UploadDocument}}$, $\overline{\text{DeleteDocument}}$⟩), which: (i) receives SourceURI and DestinationURI as a result of the invocation of MoveFile, (ii) invokes DownloadDocument with SourceURI as input data, and receives the corresponding Document, (iii) invokes UploadDocument using Metadata and Content (since Document = Metadata $\oplus$ Content) and DestinationURI, which was previously received, (iv) invokes DeleteDocument using SourceURI and receives the associated Acknowledgment, and (iiv) sends back the Acknowledgment expected by MoveFile.

Since there is no data dependency between UploadDocument and DeleteDocument, there exists another mapping using these same actions in a different order: MoveFile $\mapsto$ ⟨$\overline{\text{DownloadDocument}}$, $\overline{\text{DeleteDocument}}$, $\overline{\text{UploadDocument}}$ ⟩

***Definition** 4 (**Many-to-Many Mapping**):* A sequence of required actions $X_1 = \langle\alpha_1, \alpha_2, \ldots, \alpha_m\rangle$ where $\alpha_{i=1..m}=\langle a_i, I_{a_i}, O_{a_i}\rangle \in \mathcal{I}_1$ maps to a sequence of provided actions $X_2=\langle\overline{\beta_1}, \overline{\beta_2}, \ldots, \overline{\beta_n}\rangle$ where $\overline{\beta_{j=1..n}}=\langle\overline{b_j}, I_{b_j}, O_{b_j}\rangle \in \mathcal{I}_2$, noted $\langle\alpha_1, \ldots, \alpha_l, \ldots, \alpha_m\rangle \overset{m-n}{\longmapsto} \langle\overline{\beta_1}, \ldots, \overline{\beta_n}\rangle$, iff:

1) $\bigsqcup_{j=1}^{n} b_j \sqsubseteq \bigsqcup_{i=1}^{m} a_i$
2) $\bigsqcup_{i=1}^{l} I_{a_i} \sqsubseteq I_{b_1}$
3) $\left(\bigsqcup_{j=1}^{l} I_{a_j}\right) \bigsqcup \left(\bigsqcup_{h=1}^{i-1} O_{b_h}\right) \sqsubseteq I_{b_i}$
4) $O_{a_h} = \emptyset$ for $1 \leqslant h \leqslant l-1$ (where $l \in [1, m+1]$)
5) $\left(\bigsqcup_{i=1}^{h} I_{a_i}\right) \bigsqcup \left(\bigsqcup_{k=1}^{n} O_{b_k}\right) \sqsubseteq O_{a_h}$ for $l \leqslant h \leqslant m$

The first condition states that the a_i operations can be performed using b_j operations. The second condition states that the first provided action can be performed since the necessary input data I_{b_1} can be obtained from the data previously received. The third conditions ensures that the input data of each action can be obtained from the received input data and the output data of the preceding actions. The fourth condition states that we can cache the input data of the required actions and allow them to progress if they do not necessitate any output data. We recall that a required action represents a client-side invocation of an operation by sending the appropriate input data and receiving the corresponding output data, while a provided action uses the inputs and produces the corresponding output. Let l be the index of the first required action that necessitates some output data. $l = 1$ means that the first required action necessitates some output data while $l = m + 1$ means that none of the required actions necessitates output data. Since the first $l - 1$ actions do not necessitate any output data, they can be executed before the provided actions. Finally, the last condition states that the output data of the remaining required actions, i.e. from l to m, can be obtained from the previous input data together with the output data of the provided actions.

Consequently, we generate the mapping process $M_{m-n}(X_1, X_2)$ depicted in Figure 7. $M_{m-n}(X_1, X_2)$ first synchronises with the $l - 1$ first required actions since no output data is necessary for them to be executed (trace $0.0' \to \cdots \to (l-1).1'$). But only after performing all provided actions (trace $(l-1).1' \to \cdots \to (l-1).n'$), can $M_{m-n}(X_1, X_2)$ synchronise with the subsequent required actions, from l to m (trace $(l-1).n' \to \cdots \to m.n'$) since the output data necessary for their achievement are produced by the provided actions. The mapping process corresponding to $\langle \alpha_1, \ldots, \alpha_l, \ldots, \alpha_m \rangle \overset{m-n}{\longmapsto} \langle \overline{\beta_1}, \ldots, \overline{\beta_n} \rangle$ is as follows: $M_{m-n}(X_1, X_2) = (\overline{\alpha_1} \to \cdots \to \overline{\alpha_{l-1}} \to \beta_1 \to \cdots \to \beta_n \to \overline{\alpha_l} \to \overline{\alpha_{l+1}} \cdots \to \overline{\alpha_m} \to \text{END})$.

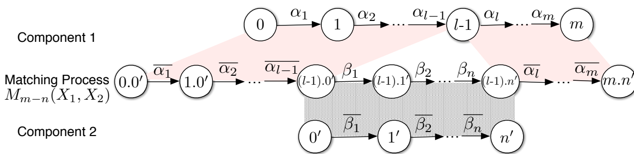

Fig. 7. Many-to-many mapping process: $M_{m-n}(X_1,X_2)$

To sum up, a sequence of required actions can be mapped to a sequence of provided actions if the following conditions are verified: (i) the functionality offered by the provided actions covers that of the required actions, (ii) the input data of each provided action can be obtained before its execution, and (iii) the output data of each required action can be obtained before its execution. Even though these mappings do not cover every possible mismatch—this would mean that we are able to prove computational equivalence—they cover a large enough set of mismatches with respect to practical and real case studies and other automated approaches.

3.2 Interface Mapping using Constraint Programming

Mapping interface $\mathcal{I}_1$ to interface $\mathcal{I}_2$ consists in searching, among all the possible pairs of sequences of required actions of $\mathcal{I}_1$ and sequences of provided actions of $\mathcal{I}_2$, those which verify the conditions of the mapping relation specified in the previous section. Furthermore, each pair of sequences of actions is minimal, that is, any other pair verifying the mapping relation would include a subsequence of the required or provided actions. As interface mapping is an NP-complete problem (see the proof at `http://www-roc.inria.fr/arles/software/mics/complexity.pdf` and in [37]), we use Constraint Programming (CP) to deal with it effectively. Indeed, CP has proved very efficient when dealing with combinatorial problems [35].

Many arithmetical and logical operators are managed by existing CP solvers. However, although there are some attempts to integrate ontologies with CP [38], none supports ontology-related operators such as subsumption or disjunction of concepts. In order to use CP to compute interface mapping, we need to enable ontology reasoning within CP solvers. Therefore, we propose to represent the ontological relations we are interested in using arithmetic operators supported by existing solvers. In particular, we devise an approach to associate a unique code to each ontological concept such that disjunction and subsumption relations amount to boolean operations.

In the following, we introduce CP. Then, we formulate the interface mapping as a constraint satisfaction problem that can be solved efficiently using CP.

3.2.1 Constraint Programming in a Nutshell

Constraint programming is the study of combinatorial problems by stating constraints (conditions, qualities) which must be satisfied by the solution(s) [35]. These problems are defined as a *constraint satisfaction problem* and modelled as a triple (X, D, C):

- *Variables*: $X = \{x_1, x_2, \ldots, x_n\}$ is the set of variables of the problem.
- *Domains*: D is a function which associates to each variable x_i its domain $D(x_i)$, i.e. the set of possible values that can be assigned to x_i.
- *Constraints*: $C = \{C_1, C_2, \ldots, C_m\}$ is the set of constraints. A constraint C_j is a mathematical relation defined over a subset $x^j = \{x_1^j, x_2^j, \ldots, x_{n^j}^j\} \subseteq X$ of variables which restricts their possible values. Constraints are used to determine unfeasible values and delete them from the domains of variables.

Solving a constraint satisfaction problem consists in finding the tuple (or tuples) $v = (v_1, \ldots, v_n)$ where $v_i \in D(x_i)$ and such that all the constraints are satisfied. Thus, CP uses constraints to state the problem declaratively without specifying a computational procedure to enforce them. The latter task is carried out by a solver. The constraint solver implements intelligent search algorithms such as backtracking and branch and bound which are exponential in time in the worst case but may be very efficient in practice. The CP solver also exploits the arithmetic properties of the operators used to express the constraints to quickly check, discredit partial solutions, and prune the search space substantially.

We represent interface mapping $Map\,(\mathcal{I}_1,\mathcal{I}_2)$ as a constraint satisfaction problem as follows:

- *Variables*: $X = \{X_1, X_2\}$ where X_1 represents a sequence of required actions of $\mathcal{I}_1$ and X_2 represents a sequence of provided actions of $\mathcal{I}_2$.
- *Domains*: $D(X_1)=\bigcup_{k=1}^{|\mathcal{I}_1|} P_k(\mathcal{I}_1)$ and $D(X_2)=\bigcup_{k=1}^{|\mathcal{I}_2|} P_k(\mathcal{I}_2)$ where $P_k(S)$ denotes the set of k-permutations of the elements of the set S. Indeed, X_1 is a sequence of actions (hence the permutations) of $\mathcal{I}_1$ of length k varying between 1 and the cardinality of $\mathcal{I}_1$, i.e. $1 < k < |\mathcal{I}_1|$. Similarly for X_2. Note that to keep the domain finite, each action can appear in the sequence at most once.
- *Constraints*: the constraints are defined by the conditions of the mapping relation ($\mapsto$) specified in Section 3.1. As for the minimality of the mapping, we eliminate redundant solutions using the *subsequence relation*, which is a partial order relation defined on the domain of each variable. Let us consider two sequences of actions $A = \langle a_1, a_2, \ldots, a_m\rangle$ and $B = \langle b_1, b_2, \ldots, b_n\rangle$; A is a subsequence of B iff $m < n$ and $\forall k \in [1..m]$, $a_k = b_k$. Exploiting this partial order relation for the branch and bound algorithm used to solve the constraint satisfaction problem allows us to keep only the minimal mappings that verify the rest of the constraints.

The solutions of the constraint satisfaction problem are the smallest, according to the subsequence relation, pairs of action sequences $(\alpha, \overline{\beta}) \in D(X_1) \times D(X_2)$ such that the required actions of α can safely be achieved using the provided actions of $\overline{\beta}$.

3.2.2 Representing Ontological Relations

Our goal is to leverage CP solvers to perform interface mapping since none of existing CP solvers deals with ontology-based operators. To this end, we define a bit vector encoding of the ontology which is correct and complete regarding the subsumption and disjunction axioms. Correctness means that if the encoding asserts that a concept subsumes another concept or that a concept is a disjunction of other concepts then these relations can be verified in the ontology. Completeness means that if the ontology states that a concept subsumes another one or that a concept is a disjunction of other concepts then this statement can also be made using the encoding.

Algorithm 1 EncodingOntology

Require: Classified ontology $\mathcal{O}$
Ensure: $Code[]$: maps each concept $C \in \mathcal{O}$ to a bit vector
1: **for all** $C \in Concepts(\mathcal{O})$ **do**
2: $\quad Set[C] \leftarrow \{NewElement()\}$
3: **end for**
4: **for all** $C \in Concepts(\mathcal{O})$ **do**
5: $\quad$ **for all** $Des \in Descendants(C)$ **do**
6: $\quad\quad Set[C] \leftarrow Set[C] \cup Set[Des]$
7: $\quad$ **end for**
8: **end for**
9: $disjunctionAxiomList = \text{Sort}(DisjunctionAxioms(\mathcal{O}))$
10: **for all** $A = \bigsqcup_{i=1}^{n} A_i \in disjunctionAxiomList$ **do**
11: $\quad \mathcal{D} \leftarrow Set[A] \setminus \bigcup_{i=1}^{n} Set[A_i]$
12: $\quad$ **for all** $d \in \mathcal{D}$ **do**
13: $\quad\quad Set[A] \leftarrow Set[A] \setminus \{d\}$
14: $\quad\quad$ **for all** A_i **do**
15: $\quad\quad\quad d_i \leftarrow \{NewElement()\}$
16: $\quad\quad\quad Set[A_i] \leftarrow Set[A_i] \cup \{d_i\}$
17: $\quad\quad\quad$ **for all** $Asc \in Ascendants(A_i)$ **do**
18: $\quad\quad\quad\quad Set[Asc] \leftarrow Set[Asc] \cup d_i$
19: $\quad\quad\quad$ **end for**
20: $\quad\quad$ **end for**
21: $\quad\quad$ **for all** $Des \in Descendants(A) \mid d \in Set[Des]$ **do**
22: $\quad\quad\quad Set[Des] \leftarrow (Set[Des] \setminus \{d\}) \cup \left(\bigcup_{i=1}^{n} d_i\right)$
23: $\quad\quad$ **end for**
24: $\quad$ **end for**
25: **end for**
26: $Code[] \leftarrow \text{SetsToBitVectors}(Set[])$
27: **return** $Code[]$

The algorithm for encoding an ontology (Algorithm 1) takes the classified ontology as its input, i.e. an ontology that also includes inferred axioms, and returns a map that associates each concept with a bit vector. We first use sets to encode the ontology concepts such that subsumption coincides with set inclusion and disjunction with set union. Then, we represent the sets using bit vectors whose size is the number of elements of all sets. Each bit is set to 1 if the corresponding element belongs to the set and to 0 otherwise. The type of elements of the sets does not matter, they are just temporary objects used to perform the encoding.

The first step of the encoding algorithm is to assign a unique element to the set that represents each concept (Lines 1–3). Then, we augment the set of each concept with the elements of the sets associated with the concepts it subsumes, i.e. its descendants (Lines 4–8) since subsumption essentially comes down to set inclusion of the instances of concepts.

We then move to disjunction axioms. We sort the axioms so that each element is made up of simple concepts or preceding concepts in the list (Line 9). For each disjunction axiom $A = \bigsqcup_{i=1}^{n} A_i$, we consider the set $\mathcal{D}$ of elements that belong to the set representing A but which are not included in any of the sets of its

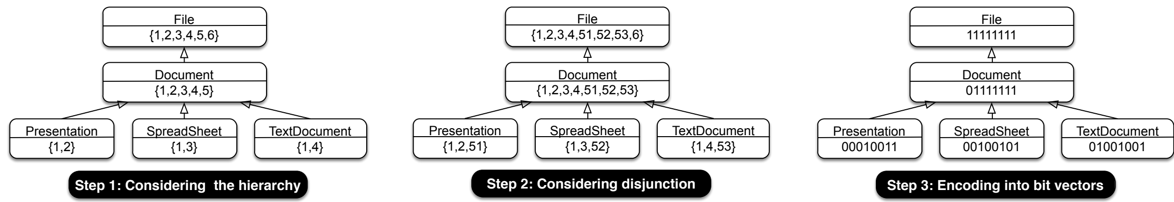

Fig. 8. Illustrating ontology encoding on an extract of the file ontology

composing classes A_i (Line 11). These elements are either the distinguishing element of A, or put into A's set by one of its sub-concepts during the previous step. The latter case represents the case where a concept is subsumed by the disjunction A but not by any of its individual concepts. To preserve the subsumption, each element $d \in \mathcal{D}$ is divided into n elements, each of which is added to one of the composing classes $A_{i=1..n}$. Hence, we first remove d from A (Line 13). Then, we create a new element d_i and add it to A_i's set as well as to the sets of its subsuming concepts, which include A (Lines 14−20). We also replace the element d in A's descendants by the new elements it was divided into (Lines 21−23). Finally, we encode the sets using bit vectors where each bit indicates whether or not an element belongs to the set (Line 26).

The size of the bit vector depends on the size of the ontology together with the number of disjunction axioms it includes. Let m be the number of concepts in the ontology. The first step of the encoding algorithm is to assign a unique element to the set that represents each concept. Hence, each element associated with a concept C will be converted into a bit in the bit vector representing the concept $Code[C]$. Consequently, we will have an m-bit vector. Then, we augment the set of each concept with the elements of its descendants, which does not involve adding new elements. As a result, we still have an m-bit vector. Let k be the number of conjunction axioms. For each axiom $A = \bigsqcup_{i=1}^{n} A_i$, we split the element belonging to A but not to any of its composing concepts A_i, if any, into a maximum of n objects, hence adding n bits. Consequently, the number of bits would be $m + k * n'$ where n' is the degree of the disjunction axioms, i.e. the maximum of all n.

As a result, subsumption can be performed using bitwise *and* as follows:

$$C \sqsubseteq D \iff Code[C] \wedge Code[D] = Code[C]$$

Disjunction is performed using bitwise *or*:

$$A = \bigsqcup_{i=1}^{n} A_i \iff Code[A] = \bigvee_{i=1}^{n} Code[A_i],$$

The proof of the correctness and completeness of this encoding can be found at `http://www-roc.inria.fr/arles/software/mics/encodingProof.pdf` and also in [37].

Let us consider the extract of the file management ontology depicted in Figure 8. File subsumes Document which is defined as the disjunction of Presentation, SpreadSheet, and TextDocument. During the first step, we associate an element, which we represent as a natural number, to each concept and put it into its ascendants. The element *'1'* represents the bottom concept $\bot$ subsumed by all concepts. Then, we consider the $\mathsf{Document} = \mathsf{Presentation} \sqcup \mathsf{SpreadSheet} \sqcup \mathsf{TextDocument}$ disjunction. The *'5'* element belongs to Document but not to any of its composing concepts, so we split it into three elements (*'51'*, *'52'*, and *'53'*) and assign each of them to a composing element in step 2. In step 3, we encode sets as bit vectors. For example, Presentation includes 1 at the position of *'1'*, *'2'*, and *'51'* elements; and 0 at all other positions. The bitwise AND between the codes of File and Document corresponds to the code of Document ($11111111 \wedge 01111111 = 01111111$), which is equivalent to stating that File subsumes Document. The bitwise OR between the codes of Presentation, SpreadSheet, and TextDocument is 01111111, which corresponds to the code of Document. Note that this extract of the file ontology contains five concepts ($m = 5$) and one disjunction axiom ($k = 1$) with three composing concepts ($n = n' = 3$), as a result the codes of the concepts are 8-bit vectors.

The encoded ontology is used by the CP solver when computing the interface mapping in order to check if a constraint is verified. For example, to map $\langle \mathsf{ReadFile}, \{\mathsf{SourceURI}\}, \{\mathsf{File}\} \rangle$ from the interface of the WebDAV client to $\langle \mathsf{DownloadDocument}, \{\mathsf{SourceURI}\}, \{\mathsf{Document}\} \rangle$ from the interface of the GoogleDocs service, the CP solver verifies the subsumption between output data, that is, $\mathsf{File} \sqsubseteq \mathsf{Document}$. This verification is performed using the code associated with each concept.

4 Automated Synthesis of Mediators

To enable functionally-compatible components to interoperate, the mediator must not only solve the differences between their interfaces but also coordinate their behaviours in order to ensure their correct interaction. Hence, given $Map\,(\mathcal{I}_1, \mathcal{I}_2)$ and $Map\,(\mathcal{I}_2, \mathcal{I}_1)$, where every required action is involved in at least one mapping, we must either generate a mediator M that composes the associated mapping processes in order to allow both components to interact correctly, or determine that no such mediator exists.

***Definition* 5 (*Mediated Behavioural Compatibility*):** Let P_1 and P_2 represent the components' behaviours. If a mediator M exists, then we say that P_1 and P_2 *are behaviourally compatible through a mediator* M, written $P_1 \leftrightarrow_M P_2$.

In order to allow the components to interact correctly, the mediator M must coordinate their behaviours so as to ensure that the parallel composition $P_1 \| M \| P_2$ successfully terminates by reaching an END state.

We incrementally build a mediator M by forcing the two processes P_1 and P_2 to progress consistently so that if one requires the sequence of actions X_1, the other process is ready to engage in a sequence of provided actions X_2 to which X_1 maps. Given that an interface mapping guarantees the semantic compatibility between the actions of the two components, then the mediator synchronises with both processes and compensates for the differences between their actions. This is formally described as follows:

if $\quad P_1 \overset{X_1}{\Rightarrow} P_1'$ and $\exists\,(X_1, X_2) \in Map\,(\mathcal{I}_1, \mathcal{I}_2)$
such that $P_2 \overset{X_2}{\Rightarrow} P_2'$ and $P_1' \leftrightarrow_{M'} P_2'$
then $\quad P_1 \leftrightarrow_M P_2$ where $M = M_{m-n}(X_1, X_2); M'$

Similarly, for the other process:

if $\quad P_2 \overset{X_2}{\Rightarrow} P_2'$ and $\exists(X_2, X_1) \in Map\,(\mathcal{I}_2, \mathcal{I}_1)$
such that $P_1 \overset{X_1}{\Rightarrow} P_1'$ and $P_1' \leftrightarrow_{M'} P_2'$
then $\quad P_1 \leftrightarrow_M P_2$ where $M = M_{m-n}(X_2, X_1); M'$

The mediator further consumes the extra provided actions $\overline{\beta}$ so as to allow the components to progress using the process $M_{\overline{\beta}} = (\beta \rightarrow \text{END})$. Extra provided actions are actions offered by one component but not required by the other and need to be invoked to allow the former component to progress. As long as the input data necessary to invoke the action has been received, the mediator can call it and ignore the output produced. However, we do not handle extra required actions since it involves offering some functionality, which the mediator cannot handle by itself. The support of extra provided actions is as follows:

if $\quad P_1 \overset{\overline{\beta}}{\rightarrow} P_1'$, and $\exists P_2$ such that $P_1' \leftrightarrow_{M'} P_2$
then $P_1 \leftrightarrow_M P_2$ where $M = M_{\overline{\beta}}\,; M'$

if $\quad P_2 \overset{\overline{\beta}}{\rightarrow} P_2'$, and $\exists P_1$ such that $P_1' \leftrightarrow_{M'} P_2$
then $P_1 \leftrightarrow_M P_2$ where $M = M_{\overline{\beta}}\,; M'$

Finally, when both processes terminate, i.e. reach an END state, then the mediator also terminates:

$$\text{END} \leftrightarrow_{\text{END}} \text{END}$$

Note that the interface mapping is not necessarily a function since the same sequence of actions can be mapped to different sequences of actions, which we considered in the definition of the recursive algorithm for mediator synthesis (see Algorithm 2).

The algorithm starts by checking the basic configuration where both processes reach their final states. In this case, the mediator is the END process (Lines 1−3). Then it considers the states of both processes and for each enabled required action a, it calculates the list of mappings that can be applied, i.e. pairs (X_1, X_2) such that X_1 starts with a and P_2 is ready to engage in X_2 (Line 6). It selects one of them and makes a recursive call to test whether it can lead to a valid mediator (Lines 8−9). The selection of the mapping to use may be motivated by some non-functional property or the length of the sequences of actions involved in the mapping but, for instance, let us assume that the algorithm simply selects the first valid mapping. The result is that a correct mediator is not unique. If the selected mapping leads to a valid mediator, the algorithm generates the associated mapping process and puts it in sequence with the returned mediator M'. Otherwise, it tries another mapping until a valid mapping is found or all the possible mappings have been tested (Lines 7−15). In the latter case, it checks whether the mediator can bypass a provided action in order to obtain a valid mediator. In this situation, the algorithm generates the appropriate process $M_{\overline{\beta}}$ and puts it in sequence with the generated mediator (Lines 16−23). If the required action cannot be mapped to any action given the states of both processes, the algorithm fails (Lines 24−26). Otherwise, it adds the new trace to the previously calculated mediator (Line 27). The algorithm explores all the outgoing transitions labelled with required actions (Lines 5−28) in order to make sure that for any required actions in which the components can engage, we are able to find a composition of mapping processes that will lead the

Algorithm 2 SynthesiseMediator

Require: P_1, P_2
Ensure: A mediator M
1: **if** $P_1 = \text{END}$ and $P_2 = \text{END}$ **then**
2:     **return** END
3: **end if**
4: $M \leftarrow \text{END}$
5: **for all** $P_i \overset{a}{\rightarrow} P'_{i=1,2}$ **do**
6:     $mappingList \leftarrow$ FindEligibleMappings(a, P_i, P_{3-i})
7:     **while** $\neg found$ and $mappingList \neq \emptyset$ **do**
8:         $Map(X_1, X_2) \leftarrow$ selectMapping$(mappingList)$
            such that $P_i \overset{X_1}{\Rightarrow} P_i''$ and $P_{3-i} \overset{X_2}{\Rightarrow} P_{3-i}'$
9:         $M' \leftarrow$ SynthesiseMediator(P_i'', P_{3-i}')
10:         **if** $M' \neq$ **fail then**
11:             $found \leftarrow true$
12:             $M_{m-n}(X_1, X_2) \leftarrow$ GenerateMapProcess(X_1, X_2)
13:             $M'' \leftarrow M_{m-n}(X_1, X_2); M'$
14:         **end if**
15:     **end while**
16:     **if** $\neg found$ and $\exists \overline{\beta} \mid P_{3-i} \overset{\overline{\beta}}{\rightarrow} P_{3-i}'$ **then**
17:         $M' \leftarrow$SynthesiseMediator(P_i, P_{3-i}')
18:         **if** $M' \neq$ **fail then**
19:             $found \leftarrow true$
20:             $M_{\overline{\beta}} \leftarrow$ GenerateExtraProvidedActionProcess$(\overline{\beta})$
21:             $M'' \leftarrow M_{\overline{\beta}}\,; M'$
22:         **end if**
23:     **end if**
24:     **if** $\neg found$ **then**
25:         **return fail**
26:     **end if**
27:     $M \leftarrow M | M''$
28: **end for**
29: **return** M

components to reach their final states. The algorithm does not systematically explore transitions labelled with provided actions because the synchronisation is triggered by required actions, which initialise the interaction by sending the input data. The mediator M hence synthesised guarantees that the parallel composition $P_1 \| M \| P_2$ is deadlock free.

Theorem 1: if $P_1 \leftrightarrow_M P_2$ then the parallel composition $P_1 \| M \| P_2$ is deadlock free.

Proof: By construction, the mediator is synthesised only if both P_1 and P_2 reach an END state. In addition, at any given state s of any of P_1 or P_2, every transition associated with a required action α such that $s \overset{\alpha}{\Rightarrow} s'$ is involved in some mapping $\alpha \mapsto \overline{\beta}$. Hence, there exists an associated mapping process $M_{m-n}(\alpha, \overline{\beta})$ that is ready to engage in the sequence of dual provided actions, i.e. $M_{m-n}(\alpha, \overline{\beta})$ is in state s_m such that $s_m \overset{\overline{\alpha}}{\Rightarrow} s'_m$. Any transition associated with a provided action $\overline{\beta}$ such that $s \overset{\overline{\beta}}{\Rightarrow} s'$: (i) synchronises with a mapping process $M_{m-n}(\alpha, \overline{\beta})$ if there exists a mapping $\alpha \mapsto \overline{\beta}$ involving it, (ii) is an extra provided action, in which case it is associated with $s \overset{\overline{\beta}}{\Rightarrow} s'$ from the $M_{\overline{\beta}}$ process, or (iii) is never triggered; we recall that the synchronisation is triggered by required actions, which initialise the interaction by sending the input data. □

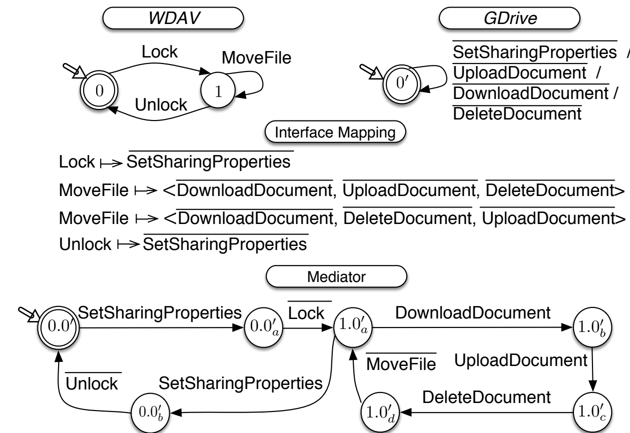

Fig. 9. Synthesis of an extract of the *WDAV*-*GDrive* mediator

Figure 9 depicts an extract of the LTSs representing the behaviour of *WDAV* and *GDrive*. As a result of the interface mapping computation, the Lock and Unlock operations map to SetSharingProperties and the MoveFile operation maps to DownloadDocument, UploadDocument, and DeleteDocument while the last two operations can be executed in any order. When both processes are at their initial states (0 and $0'$ respectively), the only applicable mapping is Lock $\mapsto$ SetSharingProperties since P_{WDAV} is only able to perform this action. After applying this mapping P_{WDAV} goes to state 1, P_{GDrive} remains in state $0'$, and a partial trace of the mediator is created from $0.0' \to 0.0'_a \to 1.0'_a$. Then, P_{WDAV} can loop on the MoveFile required action, one of the possible mappings is chosen since both are applicable as P_{GDrive} loops on DownloadDocument, UploadDocument, and DeleteDocument. P_{WDAV} stays in state 1, P_{GDrive} also remains in $0'$ while the mediator is augmented with the trace $1.0'_a \to 1.0'_b \to 1.0'_c \to 1.0'_d \to 1.0'_a$. P_{WDAV} can also branch on the Unlock operation, which maps to SetSharingProperties and results in the trace $1.0'_a \to 0.0'_b \to 0.0'$ in the mediator. Finally, both processes reach their final states and the mediator is successfully created.

5 IMPLEMENTATION: THE MICS TOOL

In order to validate our approach, we implemented the MICS (Mediator Synthesis to Connect Components) tool to generate the mediator model automatically. MICS is available at `http://www-roc.inria.fr/arles/software/mics/`.

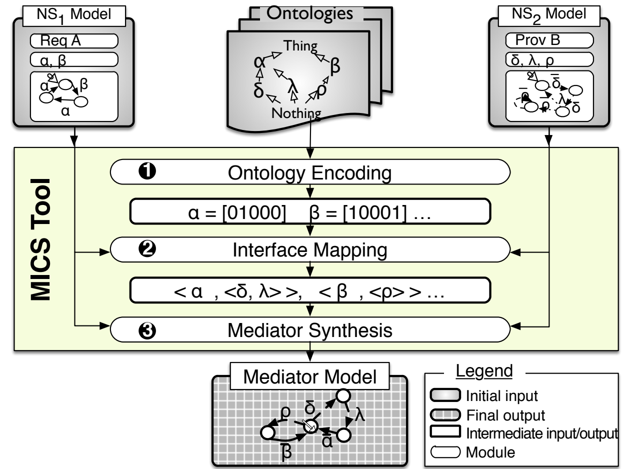

Fig. 10. Overview of MICS

MICS takes as input the models of two functionally-compatible components together with a domain ontology and produces the mediator that enables them to interoperate. MICS is made up of three modules.

The *ontology encoding module* (see Figure 10-❶) first classifies the ontology using the Pellet reasoner.[6] Pellet is an open-source java library for OWL DL reasoning. Then, the ontology encoding module uses Algorithm 1 to associate bit vectors to the concepts of this ontology.

The *interface mapping module* (see Figure 10-❷) computes the mapping between the interfaces of the components given as input, as described in Section 3.2 using the Choco constraint solver.[7] Choco is an open-source java library for constraint solving and constraint programming. Choco does not manage ontology relations such as subsumption but thanks to the bit vector representation of concepts and the associated modelling of constraints, we are able to

6. http://clarkparsia.com/pellet/
7. http://choco.emn.fr/

specify interface mapping as a constraint satisfaction problem using operators supported by Choco.

The *mediator synthesis module* (see Figure 10-❸) relies on the generated mappings and uses Algorithm 2 to synthesise mediators.

Once the mediator model has been generated, it needs to be refined and deployed into a concrete artefact so as to realise the specified mappings and coordination. This artefact is called an *emergent middleware* [34]. The emergent middleware implements the mediator based on middleware since middleware provides reusable solutions that facilitate communication and coordination between components. To implement mediators, we must bridge the gap between the application level, which provides the abstraction necessary to reason about the meaning of the actions used by the components and analysing components' behaviours formally, and the middleware-level, which provides the techniques necessary to implement these mediators. The focus of this paper is on the synthesis of mediators at the application level rather than on their implementation at the middleware level. We refer the interested reader to [39] that formalises the relation between the application and middleware levels and to [37] for further details about the implementation of mediators.

Figure 11 depicts an example of an emergent middleware. The emergent middleware: (i) intercepts the input messages, (ii) parses them in order to abstract from the communication details and represent them in terms of actions as expected by the mediator, (iii) performs the necessary data transformations, and (iv) uses the transformed data to construct an output message in the format expected by the interacting component.

Steps i), ii) and iv) are performed using middleware-specific parsers and composers to instantiate the data structures expected by each component and their delivery according to the format expected by the component. We can either use existing middleware libraries to perform this task or rely on an interpretation framework such as Starlink [8] to generate them at runtime. In the case of the interoperable file management example, we deployed the mediator over an Apache Tomcat container.[8] The container intercepts and filters out WebDAV messages. To parse the WebDAV messages, we used Milton API.[9]

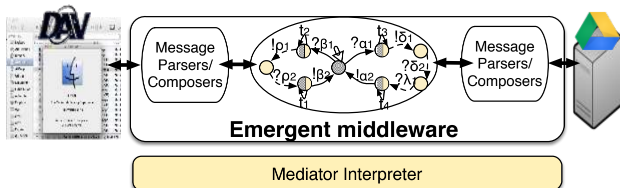

Fig. 11. Emergent middleware

8. http://tomcat.apache.org/
9. http://milton.ettrema.com/

Step iii) needs further computation. Even though subsumption guarantees the semantic compatibility between concepts, we still need to specify the necessary data transformations in order for the mediator to deal with the syntactic difference between the input/output data. Data mapping is a large and complex problem [40]. Still, we should distinguish two cases. In the case of simple types only, the translation is quite straightforward and often consists in simple cast operations. However, in most cases we need to deal with complex data types, e.g. mapping two elaborate XML Schemas. We rely on existing approaches for data mapping that devise different techniques to infer the transformations needed to translate from one XML Schema to another. We refer the interested reader to the complete survey by Shvaiko and Euzenat [40] for a thorough survey and analysis of the approaches that can be applied with this goal in mind.

6 EXPERIMENTS

A solution for mediator synthesis can only be useful if it can be applied to various real-world cases. Section 4 focuses on the theoretical aspect of the automated synthesis of mediators by proving that the synthesised mediator guarantees that the components reach their terminating states and that the composed system is deadlock-free. This section presents the practical aspect of the approach by reporting on the results of experiments using MICS to generate mediators for real-world case studies. More specifically, the case studies serve to justify the following claims:

- we automatically generate mediators that not only translate one action required by one component into an action provided by the other, but also sequences of actions between components,
- the synthesised mediators introduce an acceptable overhead, even though optimising the performance of mediators is not our main focus, and
- we can synthesise mediators at runtime provided adequate domain knowledge.

Case Studies	Highlight
Instant Messaging	one-to-one mapping
File Management	one-to-many mapping
Event Management	one-to-many mapping & loops
GMES	one-to-many mapping & extra action & runtime

TABLE 3. Summary of the case studies

Table 3 summarises the type of mappings necessary to deal with to enable components to interoperate in each case study. We begin with the instant messaging case study to illustrate one-to-one mappings. We then move to the file management case study, which we previously used to illustrate our approach, to highlight one-to-many mappings. The third case relates to interoperability between event management systems,

which are concerned with the organisation of events like conferences, seminars, concerts. This case study illustrates the case of loops, i.e. when an action is achieved by executing another action multiple times. Finally, we present the GMES (Global Monitoring of Environment and Security[10]) case to illustrate the need for mediation at runtime between components that are dynamically discovered and whose interaction is spontaneous.

In the following sections (Section 6.1 to 6.4), we provide a description of each case focusing on the performance of the synthesised mediator. Details about the ontology and the behaviours of the components can be found elsewhere [37]. The aim of these case studies is to highlight the type of mappings supported by the approach and show that the mediator introduces an acceptable overhead. In Section 6.5, we measure the time necessary to perform each step of the synthesis of mediators for each case study. The aim is to evaluate the contribution of each step in the overall synthesis time.

6.1 Instant Messaging: One-to-One Mapping

Description. The evolution of instant messaging (IM) applications provides an insight into today's communicating applications where different protocols and competing standards co-exist. Popular IM applications include Windows Live Messenger, which is based on MSNP, Yahoo! Messenger which is based on the YMSG protocol, and Google Talk which is based on the Extensible Messaging and Presence Protocol (XMPP) standard protocol.[11] These IM applications offer similar functionalities but a user of MSN Messenger would be unable to exchange instant messages with a user of Google Talk. Indeed, even though XMPP is a W3C standard, many IM systems continue to use proprietary protocols. Thus, users have to maintain multiple accounts and applications in order to interact with one another. Our aim is to let users install their favourite IM applications and synthesise a mediator that performs the necessary translations to make different IM applications interoperable.

In [41], we focus on the role of ontologies to support one-to-one mapping between heterogeneous IM applications. We started by defining an IM ontology and used it to annotate the interfaces of the IM components. Then, we used the actions of each component's interface to describe its behaviour. The mediator synthesised based on these models simply performs one-to-one mappings between the interfaces of the IM applications. We deployed the synthesised mediator on the Starlink framework [8]

Performance. To evaluate the performance of the mediator, we measured the time for exchanging a 100-character message between pairs of IM applications considering combinations of MSNP, YMSG, and XMPP. The message exchange consists in the first IM application sending a message and the other replying by sending back the message unchanged. We use the notation A/B to designate message exchange between an application using protocol A to send the message and an application using protocol B is to send back the message. We repeated the experiments 50 times and report the average time in Figure 12. The non-mediated interactions include message exchange between applications using the same protocol, MSNP/MSNP, YMSG/YMSG, and XMPP/YMSG. In addition, applications using MSNP and YMSG are able to interact using a proprietary gateway.[12] Hence, we also represent non-mediated interactions for the MSNP/YMSG case, as well as YMSG/MSNP case since IM is a peer-to-peer application. The performance of the mediator compared to interactions between non-mediated clients depends on the type of the IM protocol used: while the overhead is negligible for the XML-based XMPP system, it is significant in the case of the binary YMSG protocol. It is also worth noticing that even the gateway-based solution used in the case of MSNP/YMSG interoperation introduces an overhead of 50% compared to the non-mediated MSN interactions and more compared to the YMSG interactions. Yet the time for exchanging a message remains less than 1s. Guidelines for response time in interactive applications specify that 1s is the limit to keep the user's flow of thought seamless [42]. Hence, we can state that the mediator introduces an acceptable overhead.

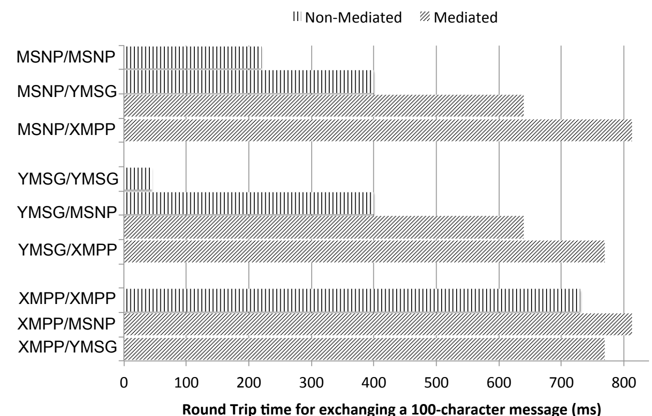

Fig. 12. Time for mediated and non-mediated interactions between IM components

6.2 File Management: One-to-Many Mapping

We used the file management case to illustrate our approach throughout this paper, in this section we evaluate the performance of the synthesised mediator.

Performance. We measured the time to perform a simple conversation, which includes authenticating,

10. http://www.gmes.info/
11. http://www.xmpp.org/
12. http://www.microsoft.com/presspass/press/2006/jul06/07-12iminteroppr.mspx

moving a file from one folder to another, and listing the content of the two folders. We used a 4KB file to lessen the network delay. We repeated each conversation 50 times and report the average execution time in Figure 13. We use the notation A/B to designate a conversation between a client A an a service B. In the case of the WebDAV client interacting with the Google Drive service (WebDAV/GoogleDrive), the overhead is negligible compared to the Google Drive interactions, while it is 75% more than the WebDAV native interactions. This is mainly due to the network communication time since the former requires 3 operations (i.e. 6 messages) while the latter requires only one operation (i.e. 2 messages). In the case of a Google Drive client interacting with the WebDAV service, the mediator introduces an 18% time overhead compared to native Google Drive interactions while it is twice the time compared to non-mediated WebDAV interactions. This time increase is due to the fact that the Google Drive actions are translated on a one-to-one basis (e.g. a DownloadDocument to a ReadFile and UploadDocument to WriteFile) instead of being merged into a single MoveFile operation. Such an optimisation is however hard to predict as the behaviour of the Google Drive client specifies that these three operations can be performed in any order.

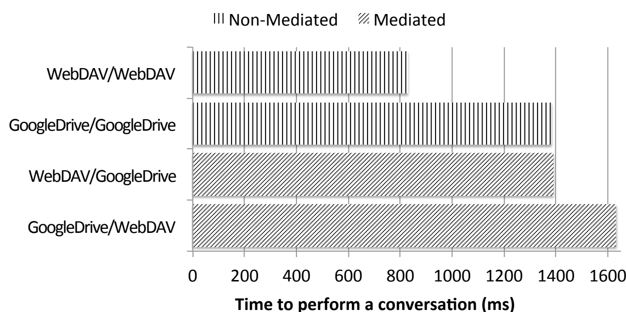

Fig. 13. Time or mediated and non-mediated interactions between WebDAV and Google Drive

6.3 Event Management: One-to-Many Mapping and Loops

Description. Event organisers usually rely on existing and specialised event management systems to prepare their events as they generally offer an economical and better quality solution compared to building their own system. In order to use an event management service, an organiser has to include within its application a client able to interact with the service provider. Depending on the event they are in charge of, organisers may have to integrate with multiple event management providers. However, event management systems often exhibit different interfaces and behaviours.

We investigated this case study with an industrial partner Ambientic.[13] Ambientic provides a suite of mobile application to facilitate the organisation of events and improve collaboration between the different stockholders. Ambientic needs solutions to interface with multiple existing event management services as required by its customers. However, developing a client for each event management service rapidly becomes fastidious. Our solution intends to simplify and automate the support of multiple services.

We considered two event management systems: Amiando[14] and RegOnline.[15] We focused on the scenario where a customer searches for events that include some keywords in their title, and then examines the information about the events found. In Amiando, customers have to send an EventFind request with the keywords to search for. The EventFind response includes a list of event identifiers. To get the information about an event, customers must invoke the EventRead operation with the event identifier as an input parameter. In RegOnline, customers have only to send a GetEvents request including the keywords to search for. As a result, the client obtains the list of events verifying the search criteria, each of which associated with the corresponding information.

We built upon the eBiquity event ontology[16], which defines the vocabulary for describing and relating information elements that are commonly used in event management systems. We used the ontology to annotate the interfaces of the components. Then we used the actions of the components' interfaces to describe their behaviours. The performance of the mediator synthesised based on the resulting models are described in the following.

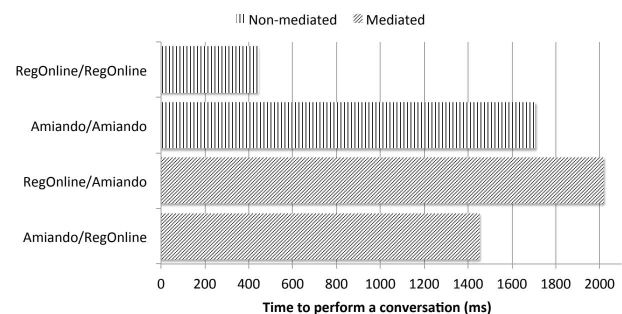

Fig. 14. Time for mediated and non-mediated interactions between Amiando and RegOnline

Performance. To evaluate the performance of the mediator, we performed a simple conversation, which consists of a search based on a substring of the title of events, then getting a list of 10 events with the corresponding description for each event. We repeated each conversation 30 times for each possible pair of Amiando and RegOnline and report the average execution time in Figure 14. Even without

13. http://www.ambientic.com/en/

14. http://developers.amiando.com/
15. http://developer.regonline.com/
16. http://ebiquity.umbc.edu/ontology/event.owl

any mediation, Amiando necessitates 3 times more time than RegOnline. This is due to the fact that it must send several messages on the network, each of which contains the information of one event whereas in RegOnline, the description of all events is sent in a single message. This is reflected in the case of RegOnline client interacting with the Amiando service as the mediator has to make many requests in order to create the message required by the client. One way to remedy this overhead is to send requests in parallel, but the synthesised mediator is not equipped for that and performs the translations only in sequence. For an Amiando client, using the RegOnline service is even more efficient than using its own service. The reason is that the mediator invokes RegOnline once and keeps the results; hence when the Amiando client sends an EventRead the response is ready and no extra processing is necessary.

6.4 GMES: Runtime Mediation

Description. To provide insight into the benefits of using the synthesis of mediators to support interoperability at runtime, we now present the experiment we conducted in the context of the GMES initiative, which was used as a demonstrator for the EU CONNECT project [43]. GMES is the European Programme for establishing a European capacity for Earth Observation. In particular, a special interest is given to the support of emergency situations (e.g. forest fire) across different European countries. Indeed, each country defines an emergency management system that encompasses multiple components that are autonomous as well as designed and implemented independently. Nonetheless, there are incentives for these components to be composed and collaborate in emergency situations. GMES makes a strong case of the need for solutions to enable multiple, and most likely heterogeneous, components to interoperate in order to perform the different tasks necessary for decision making. These tasks include collecting weather information, locating the agents involved (e.g. firemen), and monitoring the environment.

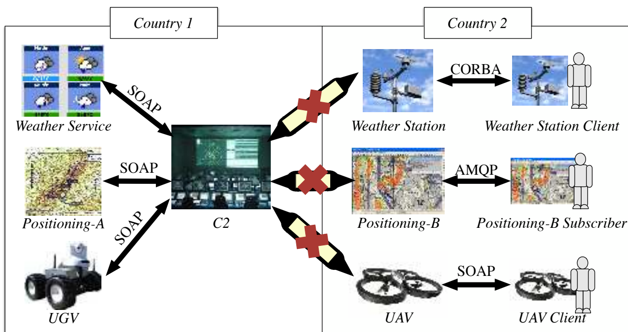

Fig. 15. Illustrating interoperability in GMES

Figure 15 depicts the case where the emergency system of *Country 1* is composed of a Command and Control centre (*C2*) which takes the necessary decisions for managing the crisis based on the information about the weather provided by the *Weather Service* component, the positions of the various agents in field given by *Positioning-A*, and the monitoring of the environment using *UGV* (Unmanned Ground Vehicle). *Country 2* assists *Country 1* by supplying components that provide *C2* with extra information. These components are *Weather Station*, *Positioning-B*, and *UAV* (Unmanned Aerial Vehicle). However, *C2* cannot use these components directly. Indeed, *Weather Station* provides specific information such as temperature or humidity whereas *Weather Service*, which is used by *C2*, returns all of this information using a single operation. The *UGV* requires the client to login, then it can explore the environment by moving forward or backward as well as turning while *UAV* is required to takeoff prior to performing any of these operations and to land before logging out. Both *UGV* and *UAV* are managed through SOAP-based controllers, hence the components' models do not include the low-level information for managing the robots. Note that GMES further illustrates differences between the components at the middleware level, e.g. SOAP and CORBA or AMQP. These differences must be handled during the implementation of the synthesised mediator. We refer the interested reader to [39] for a formal analysis of combined application-middleware differences between components while further details about application of this analysis in the GMES case can be found in [37].

In this case study, we first built a GMES ontology and asked the developers of each system, which were part of the CONNECT consortium, to use it to annotate the interface of their components. A discovery enabler, provided by the CONNECT consortium [44] was used to locate the components, extract their models, and trigger the synthesis of mediators for functionally-compatible components. We evaluate the performance of the mediator in the following.

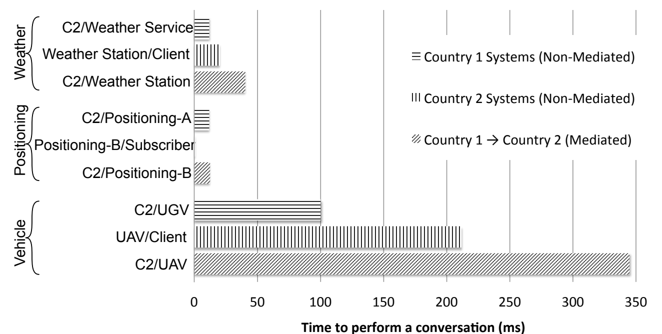

Fig. 16. Time for mediated and non-mediation interaction between GMES components

Performance. For each scenario, we measured the average time necessary to perform a meaningful conversation. In the weather scenario, the conversation

	Instant Messaging	File Management	Event Management	GMES		
				Weather	Positioning	Vehicle
Number of concepts (Disjunctions)	10 (0)	78 (7)	8 (2)	283 (2)	283 (2)	283 (2)
Time for encoding (ms)	300 ms	2502 ms	834 ms	9689 ms	9689 ms	9689 ms
$\|I_1\| \times \|I_2\|$	9 × 5	9 × 7	2 × 2	3 × 4	1 × 1	7 × 11
Time for mapping (ms)	37 ms	672 ms	247 ms	342 ms	20 ms	567 ms
$\|States(P_1)\| \times \|States(P_2)\|$	7 × 4	3 × 2	2 × 3	2 × 2	1 × 1	2 × 4
Time for synthesis (ms)	4 ms	5 ms	5 ms	2 ms	<1 ms	7 ms

TABLE 4. Processing time (in milliseconds) for each mediation step

includes authentication, obtaining weather information, and logging out. In the positioning scenario, a single operation is performed in order to locate the agents. In the vehicle control scenario, conversations consist in authentication, takeoff (in the case of *UAV* only), moving forward, turning left, moving backward, turning right, landing (in the case of *UAV* only), and logging out. We repeated each conversation 50 times and computed the average duration. The results are presented in Figure 16. In the weather scenario, the mediated interaction between *C2* and *Weather Station* takes three times the time required for *C2* to interact with the associated *Weather Service* and twice the time necessary for interaction between *Weather Station* and the associated client. This is due to the accumulation of the communication time in the mediated case since, the services being quite simple, the time to send/receive the messages on the network is much bigger that the processing time. In the positioning scenario, the overhead introduced by the mediator between *C2* and *Positioning-B* is almost non-existent. The reason is the use of a Publish/Subscribe communication paradigm (AMQP) by *Positioning-B*. Hence, the implemented mediator acts as a subscriber, receiving and transforming the data as they are published. As a result, the mediator can reply to the positioning request sent by *C2* without the need for sending a request to *Positioning-B*. In the vehicle scenario, the mediated conversations between *C2* and *UAV* require three times the time required for *C2* to interact with *UGV* and nearly twice the time necessary for interaction between *UAV* and the associated client. Similarly to the weather scenario, the mediated conversations accumulate the communication time with both systems without additional overhead due to the extra provided actions. Yet the time for mediated conversations always remains less than 1s, i.e. below the response time recommended for interactive applications to keep the user's flow of thought seamless [42].

6.5 Performance of MICS

In the previous section, we showed that automatically synthesised mediators enable heterogeneous components to interoperate while introducing an acceptable overhead. Let us now consider the time taken to synthesise mediators. Table 4 summarises the time to perform each mediation step (ontology encoding, interface mapping, and behavioural synthesis of mediators) for the aforementioned case studies.

First, the time for ontology encoding mainly depends on the size, i.e. the number of concepts, of the ontology. The greater the size of the ontology, the more time it takes to encode it. The number of disjunctions also influences the encoding, for example the ontology used for event management includes 8 concepts while the IM ontology includes 10 concepts but it takes 3 times longer to encode the former as it also includes disjunction concepts. The time to perform interface mappings depends mainly on the type of mapping encountered. In the case of one-to-one mapping, this time is minimal even with larger interfaces. In fact, computing one-to-one mappings can be performed in polynomial time, which the constraint solver is able to detect and hence calculate the mapping more efficiently. Finally, the time for the synthesis is marginal even though theoretically complex. This is because the behaviour of each component remains simple while the complexity emerges from the interaction between components.

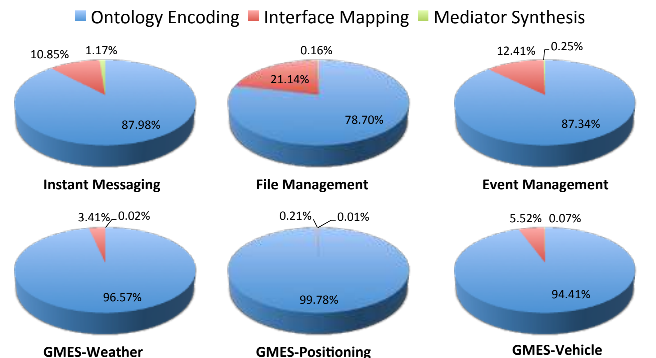

Fig. 17. Comparison of the time necessary for each mediation step

Figure 17 illustrates the time ratio for each mediation step. It can be seen that the ontology encoding is the most time-consuming step. Still, ontologies are

static entities since they represent knowledge and understanding of the application domain and are not specific to the components to be mediated. Hence, the ontology encoding can be performed beforehand and used when the need arises. The time ratio for interface mapping varies greatly between case studies according to the type of mapping: whether one-to-one or one-to-many. Finally, the time for generating the mediator based on the computed mappings represents only a small part of the overall processing time and is negligible compared to ontology processing. So, although the use of ontology allows us to reason about the semantics of data and operations and hence increases the level of automation, it comes with a cost. While standards such as OWL-S [19] or WSMO [20] amalgamate data semantics and behaviour and use ontology to represent and reason about both, our approach only uses ontologies when needed and relies on appropriate formalisms for behaviour analysis thereby making the best use of both formalisms. This separation is even more important when it comes to keeping the mediator up to date to cope with changes in the models of the components or the ontology as discussed in Section 8. When the model of a component changes, the synthesis of the mediator can be based on the previous interface mapping and ontology encoding, which are the most time-consuming phases. When one of the components changes its interface, then the synthesis has to resume from interface mapping. Finally, if the ontology evolves, the synthesis has to be restarted from the ontology encoding.

7 Related Work

Interoperability has received a great deal of interest and led to the provision of a multitude of solutions, both theoretical and practical, albeit primarily oriented toward design time. In [4], we survey the different approaches to mediation and give initial thoughts about an ontology-based approach to the dynamic synthesis of mediators. In this section, we summarise various approaches to mediation seen from the perspective of its underpinning fields: software architecture, middleware, formal methods and Semantic Web.

Software Architecture. Early work on interoperability in the software architecture field involved identifying, classifying and giving generic rules and guidelines to developers in order to increase reuse and achieve interoperability by alleviating architectural mismatches [3]. Spitznagel and Garlan [6] introduce a set of transformation patterns (e.g. data translation), which a developer can apply to basic connectors (e.g. RPC) in order to construct more complex connectors so as to solve architectural mismatches. Chang *et al.* [45] propose to use *healing connectors* in order to solve integration problems at runtime by applying healing strategies defined by the developers of the COTS components at design time. With a focus on behavioural mediation, Inverardi and Tivoli [46] define an approach to compute a mediator that composes a set of pre-defined patterns in order to guarantee that the interaction of components is deadlock-free. These patterns represent simple mechanisms that the mediator executes to solve differences between the interfaces or behaviours of components. However, the specification of the patterns to be used is left up to the developers. In our approach, the differences between the interfaces of the components are solved using mapping processes, which are automatically computed given some domain knowledge modelled using an ontology. It is the role of the domain expert to define this ontology, which is related to the application domain rather than to the software components involved.

Middleware. Middleware stands as a conceptual paradigm to connect applications effectively despite heterogeneities in the underlying hardware and software. Enterprise Service Buses [47] are open standard, message-based middleware solution that facilitates the interactions of disparate distributed applications and services. ESBs generally include built-in conversion across standard middleware technologies (e.g. SOAP, JMS) and provide a set of predefined patterns that can be used to create customised mediators. However, ESBs consider the interoperability problem from an enterprise systems perspective, where interactions are planned and long-lived and are inappropriate for cases where the component are only known at runtime. In our approach, each component is independently specified and the appropriate mediator is produced automatically, given some domain knowledge.

Formal Methods. Formal methods focus on the behaviour of software systems, which they rigorously analyse in order to reveal potential execution errors. Once potential execution errors (a.k.a. mismatches) are detected, they can be solved either by eliminating the interactions leading to the errors or by introducing a mediator that forces the components to coordinate their behaviours correctly. Existing solutions to synthesise mediators assume to be given an abstract specification of the mediator, an adaptation contract [7] or an interface mapping [14]. Nezhad *et al.* [48] propose a semi-automated approach to identity interface mapping by comparing the XML schemas describing the syntax of actions. Other approaches combine XML matching and behavioural reasoning to synthesise the mediator [49], [50]. The major difference between these approaches and ours is the consideration of the action semantics, which allows us to compute the correspondance between sequences of actions and to manage a larger range of mismatches, which cannot be directly handled considering the syntax of actions alone. Cavallaro *et al.* [51] consider the semantics of data and rely on model checking to identify mapping scripts between interaction protocols automatically. However, they propose to align the vocabulary of the processes beforehand, but many mappings may exist

and should be considered during mediator synthesis.

Semantic Web. The Semantic Web is an extension of the Web in which information is given well-defined meaning, using ontologies, in order to better enable computers and people to work in cooperation [52]. WSMO [20] defines a description language that integrates ontologies with state machines for representing Semantic Web Services. It also proposes a framework to mediate interaction between services on the basis of pre-defined mediation patterns. However, there is no guarantee that the composition of these patterns will be deadlock-free. Vaculín *et al.* [29] devise a mediation approach for OWL-S processes. They first generate all requesters paths, then find the appropriate mapping for each path by simulating the provider process. This approach does not deal with the case of one action that can be mapped to different other actions.

Even though a lot of progress has been made both in understanding and achieving interoperability, it remains an open issue as we can unfortunately observe it in our everyday life. We believe that the automation of mediator synthesis has a great potential and show how it can be achieved in this paper.

8 Future Work

In the previous sections, we presented an approach for the synthesis of correct mediators that enable heterogeneous components to interoperate. In this section, we discuss some possible enhancements of this approach.

Ontology Heterogeneity and Quality. It is crucial to think about ontologies as a means to interoperability rather than universality. It is often the case that many ontologies co-exist and need to be matched with one another. Ontology matching techniques primarily exploit knowledge explicitly encoded in the ontology rather than trying to guess the meaning encoded in the schemas, as is the case with XML schemas for example. More specifically, while XML schema matching techniques rely on the use of statistics measures of syntactic similarity, ontologies deal with axioms and how they can be put together [40]. In the future, we aim to extend our model so as to consider the heterogeneity of the ontologies themselves and reason about interface mapping under imprecise information. In addition, empirical studies, which involve users with different expertise in ontology languages and knowledge of the application domain, would allow us to evaluate the effort necessary for developing ontologies as well as to assess the quality of the resulting ontologies and theirs potential heterogeneity.

Goal-driven Mediation. In our approach, we postulate that all actions required by one component must have a semantically compatible action or sequence of actions provided by the other component. This requirement allows us to prove that the mediated system is free from deadlocks. The user may be interested in achieving a specific task only and we can permit interaction between components if we can mediate their behaviour to perform this specific task [53]. To paraphrase, instead of generating the mediator process M such that P_1 and P_2 reach their final states and $P_1 \| M \| P_2$ is deadlock free, we generate M such that the composition satisfies a given goal G, i.e. $P_1 \| M \| P_2 \models G$. However, this general case of mediator synthesis is known to be computationally expensive: when G is expressed as an LTL formula, it may reach complexity of double exponent in the size of G [54]. Yet there exist subclass of formulas for which the synthesis problem can be solved in polynomial time [55]. Further investigation is necessary to evaluate how relevant these cases are.

Multi-Party Mediators. With the advent of social-based interactions and the increased emphasis on collaboration, interoperability between multiple (more than two) components is gaining momentum. One simple solution to handle this case is by combining assembly methods (e.g., [56]) with pairwise mediators. The former consider the structural constraints and specify a coarse-grained composition of components based on their capabilities, while the latter take care of enforcing this composition despite the interface and behavioural differences that may exist between each pair of components. Another direction is to perform the interface mapping to find the correspondence between the required actions of one components and sequences of actions from all other components. These mappings then should be composed in order to enable each component to reach its terminating state as well as to ensure that the system composed of all components is deadlock-free. While the interface mapping is similar to the case of two components, further investigation is necessary to evaluate the complexity of the simultaneous exploration of multiple components' behaviours while composing the interface mappings.

Coping with Changes. Our ultimate goal is to make the mediator evolve gracefully as additional knowledge becomes available, components change, or ontology evolves. In addition, we may use machine learning techniques to infer the model of the system. While machine learning improves automation by inferring the model of the component from its implementation, it also induces some inaccuracy that we must handle. Therefore, we have to keep monitoring the system and the environment to detect changes and update the mediator accordingly. In this context, incremental re-synthesis would be essential to cope with both the dynamic aspect and partial knowledge about the environment.

9 Conclusion

Interoperability is a key challenge in software engineering whether expressed in terms of the compatibility of different components and protocols, in terms of compliance to industry standards or increasingly in terms of the ability to share and reuse data gathered from different systems. The possibility of achieving

interoperability between components without actually modifying their interfaces or behaviour is desirable and often necessary in today's open environments [57]. Mediators promote the seamless interconnection of heterogeneous components by performing the necessary translations between their messages and coordinating their behaviour. Our core contribution stems from the principled automated synthesis of mediators. In this paper, we presented an approach to infer mappings between component interfaces by reasoning about the semantics of their data and operations. We then use these mappings to automatically synthesise a correct-by-construction mediator. An important aspect of our approach is the use of ontologies to capture the semantic knowledge about the communicating components. This rigorous approach to generating mediators removes the need to develop *ad hoc* bridging solutions and fosters future-proof interoperability. We believe that this work holds great promise for the future.

ACKNOWLEDGMENTS

We gratefully acknowledge Daniel Sykes, Animesh Pathak and Thiago Teixeira for their insightful comments. We thank Thierry Martinez for his help on MICS as well as Emil Andriescu and Roberto Speicys Cardoso for their collaboration in the Event Management case study. The authors would like to thank the referees for their thorough reviews, helpful comments, and corrections that have improved this paper. They would like to acknowledge that this research was supported by the FP7 CONNECT project and ERC Advanced Grant no. 291652 (ASAP).

REFERENCES

[1] D. Konstantas, J.-P. Bourrires, M. Lonard, and N. Boudjlida, *Interoperability of enterprise software and applications*. Springer-Verlag, 2006.
[2] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: Why reuse is still so hard," *IEEE Software*, 2009.
[3] M. Shaw, "Architectural issues in software reuse: It's not just the functionality, it's the packaging," in *Proc. SSR*, 1995.
[4] V. Issarny, A. Bennaceur, and Y.-D. Bromberg, "Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability," in *SFM-11*, 2011.
[5] M. Autili, P. Inverardi, A. Navarra, and M. Tivoli, "SYNTHESIS: A tool for automatically assembling correct and distributed component-based systems," in *Proc. ICSE*, 2007.
[6] B. Spitznagel and D. Garlan, "A compositional formalization of connector wrappers," in *Proc. ICSE*, 2003.
[7] R. Mateescu, P. Poizat, and G. Salaun, "Adaptation of service protocols using process algebra and on-the-fly reduction techniques," *IEEE Trans. on Soft. Eng.*, 2011.
[8] Y.-D. Bromberg, P. Grace, L. Réveillère, and G. S. Blair, "Bridging the interoperability gap: Overcoming combined application and middleware heterogeneity," in *Proc. Middleware*, 2011.
[9] C. Gierds, A. J. Mooij, and K. Wolf, "Reducing adapter synthesis to controller synthesis," *IEEE T. Services Computing*, 2012.
[10] G. Wiederhold, "Mediators in the architecture of future information systems," *IEEE Computer*, vol. 25, no. 3, pp. 38–49, 1992.
[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
[12] M. W. Maier, "Integrated modeling: A unified approach to system engineering," *Journal of Syst. and Softw.*, 1996.
[13] E. Morris, L. Levine, C. Meyers, P. Place, and D. Plakosh, "System of systems interoperability (sosi): final report," DTIC Document, CMU/SEI, Tech. Rep., 2004.
[14] D. M. Yellin and R. E. Strom, "Protocol specifications and component adaptors," *ACM Trans. Program. Lang. Syst.*, 1997.
[15] N. Shadbolt, T. Berners-Lee, and W. Hall, "The semantic web revisited," *IEEE Intelligent Systems*, 2006.
[16] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, *The Description Logic Handbook*. Cambridge University Press, 2003.
[17] C. Calero, F. Ruiz, and M. Piattini, *Ontologies for Software Engineering and Software Technology*. Springer-Verlag, 2006.
[18] S. B. Mokhtar, N. Georgantas, and V. Issarny, "COCOA: Conversation-based service composition in pervasive computing environments with qos support," *J. of Syst. and Soft.*., 2007.
[19] D. L. Martin, M. H. Burstein, D. V. McDermott, S. A. McIlraith, M. Paolucci, K. P. Sycara, D. L. McGuinness, E. Sirin, and N. Srinivasan, "Bringing semantics to web services with OWL-S," in *Proc. WWW*, 2007.
[20] E. Cimpian and A. Mocan, "WSMX process mediation based on choreographies," in *Proc. Business Process Mgmt Work.*, 2005.
[21] N. Guarino, "Helping people (and machines) understanding each other: The role of formal ontology," in *Proc. CoopIS*, 2004.
[22] M. d'Aquin and N. F. Noy, "Where to publish and find ontologies? a survey of ontology libraries," *J. Web Sem.*, 2012.
[23] A. Borgida and P. T. Devanbu, "Adding more "DL" to IDL: Towards more knowledgeable component inter-operability," in *Proc. ICSE*, 1999.
[24] N. Oldham, C. Thomas, A. P. Sheth, and K. Verma, "METEOR-S web service annotation framework with machine learning classification," in *Proc. SWSWPC*, 2004.
[25] A. Bennaceur, V. Issarny, D. Sykes, F. Howar, M. Isberner, B. Steffen, R. Johansson, and A. Moschitti, "Machine learning for emergent middleware," in *Proc. of the Joint workshop on Intelligent Methods for Soft. System Eng., JIMSE*, 2012.
[26] J. Magee and J. Kramer, *Concurrency : State models and Java programs*. Hoboken (N.J.) : Wiley, 2006.
[27] R. M. Keller, "Formal verification of parallel programs," *Commun. ACM*, 1976.
[28] D. Brand and P. Zafiropulo, "On communicating finite-state machines," *Journal of the ACM*, 1983.
[29] R. Vaculín and K. P. Sycara, "Towards automatic mediation of OWL-S process models," in *Proc. ICWS*, 2007.
[30] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proc. ICSE*, 2008.
[31] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic, "Using dynamic execution traces and program invariants to enhance behavioral model inference," in *Proc. ICSE*, 2010.
[32] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara, "Semantic matching of WS capabilities," in *Proc. ISWC*, 2002.
[33] C. A. R. Hoare, "Process algebra: A unifying approach," in *25 Years Comm. Seq. Processes*, 2004.
[34] G. S. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci, "The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems," in *Proc. Middleware*, 2011.
[35] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming*. Elsevier Science, 2006.
[36] B. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Trans. Program. Lang. Syst.*, 1994.
[37] A. Bennaceur, "Dynamic synthesis of mediators in ubiquitous environments," Ph.D. dissertation, Université Paris VI, 2013. [Online]. Available: http://hal.inria.fr/tel-00849402/en
[38] F. Laburthe, "Constraints over ontologies," in *Proc. CP*, 2003.
[39] A. Bennaceur and V. Issarny, "Layered Connectors: Revisiting the Formal Basis of Architectural Connection for Complex Distributed Systems," in *Proc. ECSA*, 2014, to appear. [Online]. Available: http://hal.inria.fr/hal-01015897
[40] P. Shvaiko and J. Euzenat, "A survey of schema-based matching approaches," *J. Data Semantics IV*, 2005.
[41] A. Bennaceur, V. Issarny, R. Spalazzese, and S. Tyagi, "Achieving Interoperability through Semantics-based Technologies: The Instant Messaging Case," in *Proc. ISWC*, 2012.
[42] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[43] V. Issarny, B. Steffen, B. Jonsson, G. S. Blair, P. Grace, M. Z. Kwiatkowska, R. Calinescu, P. Inverardi, M. Tivoli, A. Bertolino, and A. Sabetta, "CONNECT challenges: towards emergent connectors for eternal networked systems," in *Proc. ICECCS*, 2009.
[44] N. Bencomo, A. Bennaceur, P. Grace, G. S. Blair, and V. Issarny, "The role of models@run.time in supporting on-the-fly interoperability," *Computing*, vol. 95, no. 3, pp. 167–190, 2013.
[45] H. Chang, L. Mariani, and M. Pezzè, "In-field healing of integration problems with cots components," in *Proc. ICSE*, 2009.
[46] P. Inverardi and M. Tivoli, "Automatic synthesis of modular connectors via composition of protocol mediation patterns," in *Proc. ICSE*, 2013.
[47] F. Menge, "Enterprise Service Bus," in *Proc. of the Free and open source soft. conf.*, 2007.
[48] H. R. M. Nezhad, G. Y. Xu, and B. Benatallah, "Protocol-aware matching of web service interfaces for adapter development," in *Proc. WWW*, 2010.
[49] A. Brogi and R. Popescu, "Automated generation of bpel adapters," in *ICSOC*, 2006, pp. 27–39.
[50] A. Algergawy, R. Nayak, N. Siegmund, V. Köppen, and G. Saake, "Combining schema and level-based matching for web service discovery," in *Proc. ICWE*, 2010, pp. 114–128.
[51] L. Cavallaro, E. D. Nitto, and M. Pradella, "An automatic approach to enable replacement of conversational services," in *Proc. ICSOC/ServiceWave*, 2009.
[52] T. Berners-Lee, J. Hendler, O. Lassila *et al.*, "The semantic web," *Scientific american*, 2001.
[53] L. Cavallaro, P. Sawyer, D. Sykes, N. Bencomo, and V. Issarny, "Satisfying requirements for pervasive service compositions," in *Proc. Models@run.time*, 2012.
[54] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proc. POPL*, 1989.
[55] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *J. Comput. Syst. Sci.*, vol. 78, no. 3, pp. 911–938, 2012.
[56] D. Sykes, J. Magee, and J. Kramer, "Flashmob: distributed adaptive self-assembly," in *Proc. SEAMS*, 2011.
[57] L. Baresi, E. Di Nitto, and C. Ghezzi, "Toward open-world software: Issue and challenges," *Computer*, 2006.

Amel Bennaceur received the PhD degree in Computer Science from the University of Paris VI in July 2013. She is now a Research Associate at the Open University, UK. Her research interests include dynamic mediator synthesis for interoperability and collaborative security. See http://amel.me for further details and a list of publications.

Valérie Issarny is Directrice de Recherche at Inria, France. Her research interests relate to distributed systems, software engineering, middleware, pervasive computing, and service-oriented computing. She has been involved in a number of European and French projects and initiatives in the above areas. She has further been member of organisation and programme committees of leading events in those fields. Since 2013, she is the scientific manager of the Inria@Silicon Valley program and coordinate the Inria CityLab initiative on smart cities promoting citizen engagement, and that is developed in close collaboration with researchers of CITRIS, the Center for Information Technology Research in the Interest of Society, at University of California, Berkeley. Further information about Valérie's research interests and her publications can be obtained at: https://www.rocq.inria.fr/arles/issarny.