Automated Integration of Service-Oriented Software Systems

Marco Autili, Paola Inverardi, Massimo Tivoli

Automated Integration of Service-oriented Software Systems

Marco Autili, Paola Inverardi, and Massimo Tivoli

Università dell'Aquila, Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica, L'Aquila, Italy
{marco.autili,paola.inverardi,massimo.tivoli}@univaq.it

Abstract. In the near future we will be surrounded by a virtually infinite number of software applications that provide services in the digital space. This situation radically changes the way software will be produced and used: (i) software is increasingly produced according to specific goals and by integrating existing software; (ii) the focus of software production will be shifted towards reuse of third-parties software, typically black-box, that is often provided without a machine readable documentation. The evidence underlying this scenario is that the price to pay for this software availability is a lack of knowledge on the software itself, notably on its interaction behaviour. A producer will operate with software artefacts that are not completely known in terms of their functional and non-functional characteristics. The general problem is therefore directed to the ability of interacting with the artefacts to the extent the goal is reached. This is not a trivial problem given the virtually infinite interaction protocols that can be defined at application level. Different software artefacts with heterogeneous interaction protocols may need to interoperate in order to reach the goal. In this paper we focus on techniques and tools for integration code synthesis, which are able to deal with partial knowledge and automatically produce correct-by-construction service-oriented systems with respect to functional goals. The research approach we propose builds around two phases: elicit and integrate. The first concerns observation theories and techniques to elicit functional behavioural models of the interaction protocol of black-box services. The second deals with compositional theories and techniques to automatically synthesize appropriate integration means to compose the services together in order to realize a service choreography that satisfies the goal.

1 Introduction

In the near future we will be increasingly surrounded by a virtually infinite number of software services that can be composed to build new added value applications in the Digital Space. According to John Musser, founder of ProgrammableWeb[1], the production of Application Programming Interfaces (APIs) grows exponentially and some companies are accounting for billions of dollars in revenue per year via API links to their services. Moreover, the evolution of

[1] http://www.programmableweb.com.

today Internet is expected to lead to an ultra large number of available services, hence increasing their number to billions of services in the near future. This situation radically changes the way software will be produced and used: (i) software is increasingly produced according to specific goals and by integrating existing software; (ii) the focus of software production is on integration of third party and typically black-box software, that is only provided with an interface that exposes the available functionalities but does not provide the assumed interaction protocol. The first characteristic implies a **goal oriented, opportunistic use** of the software being integrated, i.e., the producer will only use a subset of the available functionalities, some of which may not even be (completely) known. The second one implies the need to (a) **extract suitable interaction models** from discoverable and accessible pieces of software, which are made available as services in the digital space, and (b) devise appropriate **integration** means (e.g., architectures, connectors, mediators, integration patterns) that ease the composition of existing services so to achieve the goal.

The aim of the proposed research is to **provide automatic support** to the production of software systems by **integrating** existing software services **according to a specified goal**.

Our proposal builds on the model-based software production paradigm while accounting for the inherent incompleteness of information about existing software. This evidence suggests the use of an experimental approach, as opposed to a creationistic one, to the production of software. Software development has been so far biased towards a creationist view: a producer is the owner of the artefact and, if needed, she can declaratively supply any needed piece of information (interfaces, behaviours, contracts, etc.). The digital space promotes a different experimental view: the knowledge of a software artefact is limited to what can be observed of it. The more powerful and extensive the observations are, the deeper the knowledge will be; the knowledge will always remain partial, though. Indeed, there is a theoretical barrier that limits, in general, the power and the extent of observations.

Beyond automation, a further big challenge underlying this scenario is therefore to live up with the fact that this immense software resources availability corresponds to a lack of knowledge about the software, notably on its behaviour. A software producer will know less and less the precise behaviour of a third-party software service, nevertheless she will try to use it to build her own application. This very same problem recognized in the software engineering domain [16] is faced in many other computer science domains, e.g., exploratory search [39] and search computing [12].

In order to face this problem and provide a producer with a supporting framework to realize software applications via **automated integration**, we envision a process that implements a radically new perspective. First results can be found in [22]. This process builds around **elicit** and **integrate** phases.

From now on, when referring to models of services we always mean models of the interaction protocols of the services, that is models of the sequences of

actions/messages exchange that need to be performed in order to consume the service, e.g., "login" and "get authorized", before "access bank account".

The elicit phase automatically produces an interaction protocol model for each service that has been discovered as a candidate to provide a desired functionality with respect to a specified system goal. This model is complete enough to allow the service to be integrated with others in order to satisfy the goal.

The integrate phase assists the producer in creating the appropriate integration means to compose the observed services together in order to produce a system that satisfies the goal.

In this paper we present a specific instance of the above reuse-based elicit-integrate development process, which is suitable for service-oriented systems.

The paper is organized as follows. Section 2 describes an instance of the elicit phase, which is suitable for the automatic elicitation of the interaction protocol of a Web Service. Section 3 discusses in detail an instance of the integrate phase, which allows the producer to automatically enforce the realization of a specific form of service composition, namely a choreography. Thus, this instance is suitable for the automatic production of choreography-based service-oriented systems. Section 4 discusses related work in the domains of behavioral model elicitation techniques and of service choreography development. Section 5 discusses final remarks and future research directions.

2 The elicit phase

Given a software service S that has been discovered as a candidate to provide a desired functionality with respect to a system goal G, elicitation techniques must be defined to produce interaction protocol models that are complete enough to allow the service to be integrated with others in order to satisfy G. This means that we admit partial models of the service interaction protocols. For the integration phases to be automated, a goal G specification is a machine-readable model achieved by the producer by operationalizing the needs and preferences of the user [36].

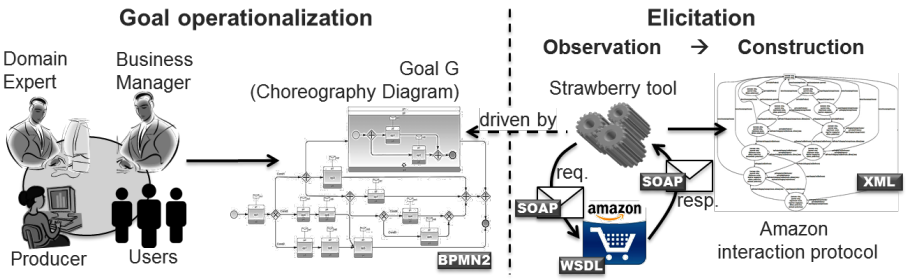

Fig. 1. Elicit phase

Referring to Figure 1, a concrete example of a goal specification can be found in [7], where domain expert and user goals are operationalized into a BPMN2[2]

[2] `http://www.omg.org/spec/BPMN/2.0`.

choreography specification, after being transformed into a CTT (ConcurTaskTrees) intermediate model [32]. The elicit phase is composed of two steps, namely *observation* and *construction* [22]. For each service to be integrated, the observation step is driven by G and collects a set of observation data. We focus on observations devoted to the identification of a set of functional behaviours, e.g., the SOAP response to a Web-Service (WS) operation invocation. Construction takes as input the set of observation data and produces a (partial) model of the observed service. This model represents the observed behaviours enriched with inferred information that is relevant for achieving G. For instance, as done in [11], the collection of the SOAP responses to WS operation invocations, enriched with the inferred partial order of the invocations, can be represented as an automaton that models the interaction protocol of the observed WS. Note that, as it is shown in Section 3, having a partial model of the interaction protocol, for each observed WS to be integrated, is sufficient to automatically synthesize the code of proxies that allow for integrating the WSs so to realize the specified BPMN2 choreography. An important aspect is that the elicit phase produces models that, although partial, are still good enough to achieve G. Goal driven elicitation can be very effective, e.g., as observed on the Amazon E-commerce WS (AEWS) where we apply the approach in [11] to elicit the AEWS interaction protocol. The experiment considered a goal-independent elicitation versus a goal-driven one [5]. Starting from the AEWS WSDL consisting of 85 XML schema type definitions and 23 WSDL operation definitions, the goal independent elicitation resulted in an interaction protocol made of 24 states and 288 transitions by using 10^6 test cases, each executed in 10^{-2} secs, e.g., few hours of testing. By considering a goal specification that the user wished to "develop a client for cart management only", the interaction protocol computed was made of 6 state and 21 transitions only. The goal driven elicitation required the generation and execution of 10^5 test cases, e.g., few seconds of testing.

As it is shown in Section 3, having a partial model of the interaction protocol, for each observed WS to be integrated, is sufficient to automatically synthesize the code of additional software entities that, proxyfing the WSs, allow for integrating them so to realize the specified BPMN2 choreography. An important aspect is that the elicit phase produces models that, although partial, are still good enough to achieve G.

Goal driven elicitation can be very effective, e.g., as observed on the Amazon E-commerce WS (AEWS) where we apply the approach in [11] to elicit the AEWS interaction protocol. The experiment considered a goal independent elicitation versus a goal driven one [5]. Starting from the AEWS WSDL consisting of 85 XML schema type definitions and 23 WSDL operation definitions, the goal independent elicitation resulted in an interaction protocol made of 24 states and 288 transitions by using 10^6 test cases, each executed in 10^{-2} secs, e.g., few hours of testing. By considering a goal specification that the user wished to "develop a client for cart management only", the interaction protocol computed was made of 6 state and 21 transitions only. The goal driven elicitation required the generation and execution of 10^5 test cases, e.g., few seconds of testing.

The following section summarizes the elicit technique that we describe in detail in [11]. It represents a specific realization of the elicit phase, which is suitable for producing the interaction protocol of a WS.

2.1 StrawBerry: automated synthesis of WS interaction protocols

By taking as input a syntactical description of the WS signature, expressed by means of the WSDL notation, `StrawBerry` [11] derives in an automated way a partial ordering relation among the invocations of the different WSDL operations. This partial ordering relation is represented as an automaton that we call *Behavior Protocol automaton*. It models the interaction protocol that a client has to follow in order to correctly interact with the WS. This automaton also explicitly models the information that has to be passed to the WS operations. `StrawBerry` is a black-box and extra-procedural method. It is black-box since it takes into account only the WSDL of the WS. It is extra-procedural since it focuses on synthesizing a model of the behavior that is assumed when interacting with the WS from outside, as opposed to intra-procedural methods that synthesize a model of the implementation logic of the single WS operations [29, 38].

Figure 2 graphically represents `StrawBerry` as a process that is split in five main activities that realize its **observation** and **construction** phases.

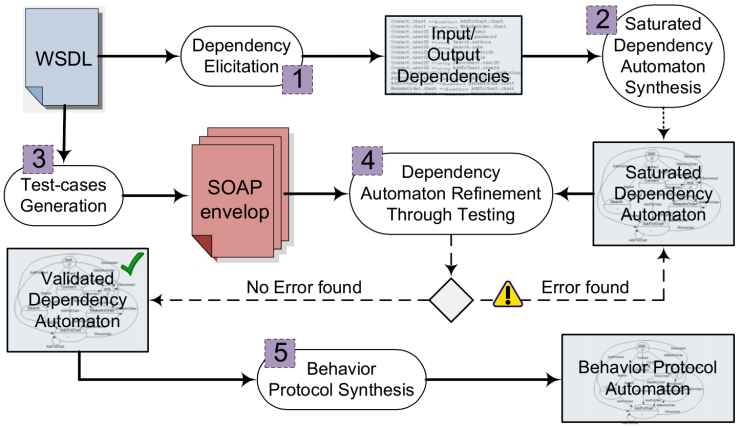

Fig. 2. Overview of the `StrawBerry` technique

Observation: the observation phase is in turn organized in two sub-phases. The first sub-phase exploits the WSDL of the WS, and performs data type analysis. The *Dependencies Elicitation* activity elicits data dependencies between the I/O parameters of the operations defined in the WSDL. A dependency is recorded whenever the type of the output of an operation (called *source*) matches with the type of the input of another operation (called *sink*). The match is syntactic. The elicited set of I/O dependencies (*Input/Output Dependencies* in Figure 2) may be optimized under some heuristics [11]. It is used for constructing a data-flow model (*Saturated Dependencies Automaton Synthesis* activity and *Saturated Dependencies Automaton* artifact) where each node stores data dependencies

that concern the output parameters of a specific operation and directed arcs are used to model syntactic matches between output parameters of an operation and input parameters of another operation. This model is completed by applying a saturation rule. This rule adds new dependencies that model the possibility for a client to invoke a WS operation by directly providing its input parameters.

The second sub-phase validates the dependencies automaton through testing against the WS to verify conformance (*Dependencies Automaton Refinement Through Testing* activity). The testing phase takes as input the SOAP messages produced by the *Test-cases Generation* activity. The latter, driven by coverage criteria, automatically derives a suite of test cases (i.e., SOAP envelop messages) for the operations to be tested, according to the WSDL of the WS. Tests are generated from the WSDL and aim at validating whether the synthesized automaton is a correct abstraction of the service implementation. Testing is used to refine the syntactic dependencies by discovering those that are semantically wrong. By construction, the inferred set of dependencies is syntactically correct. However, it might not be correct semantically since it may contain false positives. The testing activity is organized into three steps. `StrawBerry` runs positive tests in the first step and negative tests in the second step. Positive test cases reproduce the elicited data dependencies and are used to reject fake dependencies: if a positive test invocation returns an error answer, `StrawBerry` concludes that the tested dependency does not exist. Negative test cases are instead used to confirm uncertain dependencies: `StrawBerry` provides in input to the sink operation a random test case of the expected type. If this test invocation returns an error answer, then `StrawBerry` concludes that the WS was indeed expecting as input the output produced by the source operation, and it confirms the hypothesized dependency as certain. If uncertain dependencies remain after the two steps, `StrawBerry` resolves the uncertainty by assuming that the hypothesized dependencies do not exist.

Construction: the construction phase consists in a synthesis stage which aims at transforming the validated dependency automaton (a data-flow model) into an automaton defining the behavior protocol (a control-flow model), see the *Behavior Protocol Synthesis* activity in Figure 2. This automaton explicitly models also the data that has to be passed to the WS operations. More precisely, the states of the behavior protocol automaton are WS execution states and the transitions, labeled with operation names plus I/O data, model possible operation invocations from the client of the WS.

3 The integrate phase

The integrate phase assists the producer in creating the appropriate *integration means* to compose the observed services together in order to produce a system that satisfies G. Multiple models may exist for each service (e.g., behavioural, interfaces, stochastic or Bayesian), each of them representing a view of the interaction protocol. Model transformation techniques ensure coherence and consistency among the different views, hence providing a systematic support to model

interoperability [14, 19]. *Model and code synthesis techniques* produce an Integration Architecture (IA), including the the corresponding code for the actual integration, out of the elicited models by suitably instantiating architectural styles [34] and integration patterns [15]. If needed, extra integration logic can be synthesized as connectors, coordinators, mediators and adapters [7, 23, 24, 30] to guarantee correctness of the IA with respect to G.

Continuing the example introduced above, Figure 3 shows a possible concrete instance of the Integrate phase [7, 8]. Here, the elicit phase has produced the interaction protocol of each participant service in the choreography specified by G (AEWS included). Starting from G and the elicited models, the Integrate phase synthesizes a set of software coordinators. The synthesis exploits model transformations implemented by means of the Atlas Transformation Language (ATL[3]). The developed ATL transformations consist of a number of rules each devoted to the management of specific BPMN2 Choreography Diagram modelling constructs. Coordinators are implemented in Java and their deployment descriptors are codified in XML. By instantiating a fully distributed architectural style, coordinators are interposed among the participant services that need to be coordinated. By exploiting a request/response delegation pattern, coordinators proxify the services and coordinate their interaction in a way that the resulting collaboration realizes G.

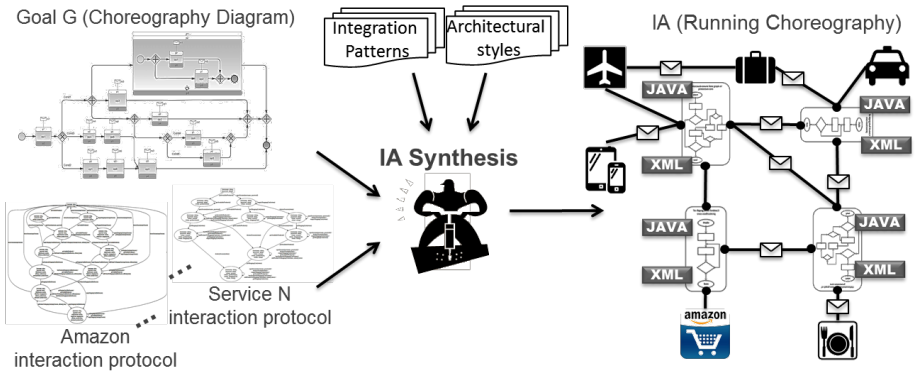

Fig. 3. Integrate phase

Next section briefly describes the integration synthesis techniques we have implemented in the CHOReOSynt tool. More details can be found in [7, 8].

3.1 CHOReOSynt: automated synthesis of service choreographies

From the BPMN2 specification of a choreography (i.e., the goal G), CHOReOSynt allows for deriving the coordinators, hereafter called Coordination Delegates (CD). CHOReOSynt offers bespoke functionalities to:

[3] ATL is a domain specific language for realizing model-to-model transformations - `www.eclipse.org/atl`

- start the synthesis process giving as input a BPMN 2.0 Choreography Diagram;
- transform the BPMN2 Choreography Diagram into an intermediate automata-based model, which is amenable to automated reasoning;
- derive a set of Coordination Models containing information that serve to coordinate the services involved in the choreography in a distributed way;
- extract the participants of the choreography and project the choreography on their behavioral role;
- simulate the behavioral role of the participants in the choreography against the interaction protocol of the services discovered by the service discovery;
- generate the Coordination Delegate artefacts and the so called "`ChorSpec`" specification to be used by the Enactment Engine component for deploying and enacting the choreography;

We have implemented these functionalities in a set of REST (Representational State Transfer) services, which are called by CHOReOSynt as shown in Figure 4 and described below.

M2M Transformator – The Model-to-Model (M2M) Transformator offers a set of model transformations. Specifically, it offers an operation `bpmn2clts()` that takes as input the BPMN2 specification of the choreography and transforms it into a model called CLTS. The latter is an extended Labeled Transition System (LTS) that allows for automatically handling complex constructs of BPMN2 Choreography Diagrams, such as gateways, loops, forks and joins.

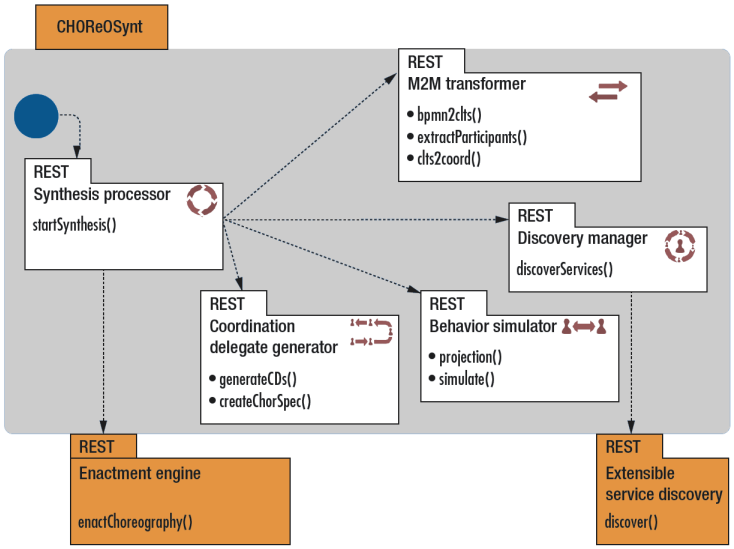

Fig. 4. CHOReOSynt REST architecture

Then, starting from the CLTS, CHOReOSynt extracts the list of the participants and, applying a further M2M transformation, automatically derives, for each participant, the CLTS model of the expected behavior with

respect to the specified choreography. To this end another operation named `extractParticipants()` is offered. The CLTS model of expected behavior is achieved by projecting (`projection()`) the choreography onto the participant, hence filtering out those transitions, and related states, that do not belong to the participant. Basically, for each participant, this CLTS specifies the interaction protocol that a candidate service (to be discovered) has to support to play the role of the participant in the choreography.

Synthesis Discovery Manager – The Synthesis process and the Discovery process interact each other to retrieve, from the service registry, those candidate services that are suitable for playing the participant roles, and hence, those services whose (offered and required) operations and protocol are compatible with the CLTS models of the expected behavior. In particular, for each participant, the call to the `discoverServices()` operation is performed. It takes the participant (abstract) CLTS as input. Then, a query is issued to the eXtensible Service Discovery (XSD) component (not in the focus of this paper). Note that, although for each choreography participant a suitable third-party service may have been discovered (and hence, its interaction protocol fits the behavior of the participant in isolation), the uncontrolled (or wrongly coordinated) composite behavior of all the discovered services may show *undesired interactions* that prevent the choreography realization. For a detailed and formal description of the notion of undesired interaction, refer to [4, 6, 9].

Behavior Simulator – Once a set of concrete candidate services has been discovered, the synthesis process has to select them by checking, for each participant, if its expected behavior can be simulated by some candidate service. Note that, for a given participant, behavioral simulation is required since, although the discovered candidate services for it are able to offer and require (at least) the operations needed to play the role of the participant, one cannot be sure that the candidate services are able to support the operations flow as expected by the choreography. Thus, in order to simulate the expected behavior of a participant with the behavior of a service, the Behavior Simulator offers an operation named `simulate()` that takes as input the projected (abstract) CLTS of the participant and the extended (concrete) LTS of the service as retrieved by the URI returned by the discovery service. It might be interesting to mention that the simulation method implemented a notion of strong simulation suitably extended to treat the CLTSs and extended the LTSs we use in CHOReOS. After simulation, if all the participant roles have been "covered" by (some of) the discovered services, the abstract CLTS is concretized with the actual names of the selected services and the actual names of the offered and requested operations. Then, the automated synthesis process distributes the coordination logic specified by the obtained CLTS into a set of Coordination Models by means of the functionality `clts2coord()`.

Coordination Delegate Generator – Once the services have been selected for all the choreography participants, and hence the CLTS has been concretized, the synthesis processor can generate the Coordination Delegates through the

operation `generateCDs()` offered by the Coordination Delegate Generator component.

Next step in the process – Once the Coordination Delegates have been generated, the Coordination Delegate Generator component can further generate a specification of the choreography (called `ChorSpec`) to be passed to the choreography Enactment Engine (not in the focus of this paper). To this end, the operation `createChorSpec()` is offered. It takes as input the selected services and the coordination delegates generated for them. The `ChorSpec` is an XML-based declarative description of the choreography that specifies the locations of the selected services and of the generated Coordination Delegate artifacts that can be deployed. Indeed, before passing the `ChorSpec` to the Enactment Engine, the Choreography Offline Testing process activity is performed to assess the quality of the choreography specification, its well formedness, etc.

4 Related work

In this section we discuss related work in the domains of behavioral model elicitation techniques and of service choreography development.

Elicitation techniques. We focus on black-box/grey-box techniques able to elicit behavioural models of the software. The reader interested on white-box techniques can refer to [3, 37, 38] and references therein.

LearnLib [21] is a framework to automatically construct a finite automaton through automata learning and experimentation. Active automata learning tries to automatically construct a finite automaton that matches the behavior of a given target automaton on the basis of active interrogation of target systems and observation of the produced behavior.

The work described in [27] presents a comprehensive approach for building parametrized behaviour models of existing black-box components for performance prediction. Those parameters represent three performance-influencing factor, i.e., usage, assembly, and deployment context; this makes the models sensitive to changing load situations, connected components, and the underlying hardware. The approach makes use of static and dynamic analysis and search-based approaches, namely genetic programming. These techniques take as input monitoring data, runtime bytecode counts, and static bytecode analysis.

SPY [17] is an approach to infer a formal specification of stateful black-box components that behave as data abstractions (Java classes that behave as data containers) by observing their run-time behavior. SPY proceeds in two main stages: first, SPY infers a partial model of the considered Java class; second, through graph transformation, this partial model is generalized to deal with data values beyond the ones specified by the given instance pools. The inferred model is partial since it models the intentional behavior of the class with respect to only a set of instance pools provided as input, which are used to get values for method parameters, and an upper bound on the number of states of the model.

GK-Tail [29] is a technique to automatically generate behavioral models from (object-oriented) system execution traces. GK-Tail assumes that execution

traces are obtained by monitoring the system through message logging frameworks. For each system method, an Extended Finite State Machine (EFSM) is generated. It models the interaction between the components forming the system in terms of sequences of method invocations and data constraints on these invocations. The correctness of these data constraints depends on the completeness of the set of monitored traces with respect to all the possible system executions that might be infinite.

The work described in [10] presents an approach for inferring state machines with an infinite state space. By observing the output that the system produces when stimulated with selected inputs, they extend existing algorithms for regular inference (which infer finite state machines) to deal with infinite-state systems. This approach makes the problem of dealing with an infinite state space tractable, but may suffer a higher degree of model approximation.

The work described in [31] presents a learning-based black-box testing approach in which the problem of testing functional correctness is reduced to a constraint solving problem. Functional correctness is modeled by pre- and post-conditions that are first-order predicate formulas. A successful black-box test is an execution of the program on a set of input values satisfying the pre-condition, which terminates by retrieving a set of output values violating the post-condition. Black-box functional testing is the search for successful tests with respect to the program pre- and post-conditions. As coverage criterion the authors formulate a convergence criterion on function approximation.

The work in [13] presents an approach that, through a combination of systematic test case generation (by means of the TAUTOKO tool) and typestate mining, infers models of program behavior in the form of finite state automata describing transitions between object states. The generation of test cases permits to cover previously unobserved behavior, and systematically extends the execution space, and enriches the inferred behavior model. In this sense, it can be said this approach goes in an opposite direction with respect to `StrawBerry`.

The work in [2] concerns an application of active learning whose aim is to establish the correctness of protocol implementations relative to a given reference implementation. The work in [1] shows how to fully-automatically construct the typical abstractions needed to perform automata learning.

Choreography realization techniques. CHOReOSynt is related to several approaches developed for automated choreography enforcement.

The approach described in [18] enforces a choreography's realizability by automatically generating monitors. Each monitor acts as a local controller for its peer. Monitors are built by iterating equivalence-checking steps between two centralized models of the whole system. A monitor is similar to our coordination delegate (CD). However, our approach synthesizes CDs without producing a centralized model of the whole system, hence preventing state explosion.

The method described in [26] checks the conformance between the choreography specification and the composition of participant implementations. Their framework can model and analyze compositions in which the interactions can also be asynchronous and the messages can be stored in unbounded queues and

reordered if needed. Following this line of research, the authors of [26] provided a hierarchy of realizability notions that forms the basis for a more flexible analysis regarding classic realizability checks [25, 26]. These two approaches are novel in that they characterize relevant properties to check a certain degree of realizability. However, they statically check realizability and do not enforce it.

The ASTRO toolset [35] supports automated composition of Web services and the monitoring of their execution. It aims to compose a service starting from a business requirement and the description of the protocols defining available external services. Unlike our approach, ASTRO deals with centralized orchestration-based business processes rather than fully decentralized choreography-based ones.

The CIGAR (Concurrent and Interleaving Goal and Activity Recognition) framework aims for multigoal recognition [20]. CIGAR decomposes an observed sequence of multigoal activities into a set of action sequences, one for each goal, specifying whether a goal is active in a specific action. Although such goal decomposition somewhat recalls CHOReOSynt's choreography decentralization, goal recognition represents a fundamentally different problem regarding realizability enforcement.

Given a set of candidate services offering the desired functionalities, the TCP-Compose* algorithm [33] identifies the set of composite services that best fit the user-specified qualitative preferences over non-functional attributes. CHOReOSynt could exploit this research to extend the discovery process to enable more flexible selection of services from the registry.

The research we described in this paper is an advance over our previous research [4, 6]. Although the synthesis process described in our previous research treated most BPMN2 constructs, it considered a simplified version of their actual semantics. For instance, as in [18], the selection of conditional branches was simply abstracted as a non-deterministic choice, regardless of the run-time evaluation of their enabling conditions. Analogously, the synthesis process enforced parallel flows by non-deterministically choosing one of their linearizations obtained through interleaving, thus losing the actual degree of parallelism. To overcome these limitations, CHOReOSynt relies on a choreography model that, being more expressive than the choreography model in CIGAR and TCP-Compose*, preserves the BPMN2 constructs' actual semantics. Relying on a more expressive model led us to define a novel, more effective distributed coordination algorithm [9].

5 Final remarks and future perspectives

Our past experience in behavioural models elicitation and integration code synthesis gives a first evidence, yet concrete, that the proposed approach is viable once referring to specific application domains, e.g., choreography-based systems.

Our experiments with `strawberry` have shown that it is practical and realistic in that it only assumes: (i) the availability of the WSDL; and (ii) the possibility to derive a partial oracle that can distinguish between regular and error answers. Furthermore, we observed that `strawberry` nicely converges to a

realistic automaton. In future work, we intend to investigate if and how assumption (ii) could be relaxed.

Our experiments with CHOReOSynt demonstrated that considering domain-specific interaction patterns mitigates the complexity of coordination enforceability when recurrent business protocols must be enforced. Generally, choreography synthesis is difficult in that not all possible collaborations can be automatically realized. This suggests we could improve CHOReOSynt with a combination of domain-specific choreography patterns, as well as protocol interaction patterns that correspond to service collaborations that are tractable through exogenous coordination. Currently, CHOReOSynt supports pure coordination. It doesn't deal with protocol adaptation because it doesn't account for mismatches at the level of service operations and related I/O parameter types. To support data-based coordination through the elicitation and application of complex data mappings, CHOReOSynt should be enhanced to automatically infer mappings to match the data types of messages sent or received by mismatching participant services. This means effectively coping with heterogeneous service interfaces and dealing with as many Enterprise Integration Patterns [15] and protocol mediation patterns [28] as possible, in a fully automatic way. Toward that end, we achieved promising results in automated synthesis of modular mediators [24].

We want to enable the market acceptance and further enhancement of CHOReOSynt by third-party developers, especially small and medium enterprises, including development of applications for commercialization. So, we released CHOReOSynt under the umbrella of the Future Internet Software and Services Initiative (FISSi[4]). Using a market-oriented approach, FISSi aims to develop awareness of OW2 Future Internet software in both FISSi members and non-members and both open source vendors and proprietary vendors. Our primary objective, to be achieved in the near future, is to establish a community of developers and third-party market stakeholders (for example, users, application vendors, and policy makers) around CHOReOSynt.

Last but not least, an interesting future direction is the investigation of non-functional properties at the level of the elicited interaction protocols and of the synthesized choreography. For instance, this requires considering operation invocation response time, extending the choreography specification with performance or reliability attributes, and accounting for them in the CDs synthesis process.

Acknowledgments

The research work so far has been supported by the Italian MIUR, prot. 2012E47TM2 (project IDEAS - Integrated Design and Evolution of Adaptive Systems), and by the EC FP7 under grant agreement n. 257178 (project CHOReOS - Large Scale Choreographies for the Future Internet - `www.choreos.eu`). Future research efforts will be supported by the EC H2020 under grant agreement n. 644178 (project CHOReVOLUTION - Automated Synthesis of Dynamic and Secured Choreographies for the Future Internet).

[4] `http://www.ow2.org/view/Future_Internet/CHOReOS`

References

1. F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. Vaandrager. Automata learning through counterexample guided abstraction refinement. In *FM 2012: Formal Methods*, volume LNCS 7436, pages 10–27. Springer Berlin Heidelberg, 2012.
2. F. Aarts, H. Kuppens, J. Tretmans, F. Vaandrager, and S. Verwer. Improving active mealy machine learning for protocol conformance testing. *Machine Learning*, 96(1-2):189–224, 2014.
3. R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. *SIGPLAN Not.*, 40(1), 2005.
4. M. Autili, D. Di Ruscio, A. Di Salle, P. Inverardi, and M. Tivoli. A model-based synthesis process for choreography realizability enforcement. In *Proc. of FASE'13*, pages 37–52, LNCS 7793. 2013.
5. M. Autili, D. Di Ruscio, P. Inverardi, P. Pelliccione, and M. Tivoli. Modelland: Where do models come from? In *Models@run.time*, page LNCS 8378. 2014.
6. M. Autili, A. Di Salle, and M. Tivoli. Synthesis of resilient choreographies. In *Software Engineering for Resilient Systems*, pages 94–108, LNCS 8166. 2013.
7. M. Autili, P. Inverardi, and M. Tivoli. Automated synthesis of service choreographies. *IEEE Software*, (99), 2015.
8. M. Autili, D. D. Ruscio, A. D. Salle, and A. Perucci. Choreosynt: enforcing choreography realizability in the future internet. In *Proc. of FSE'14*, 2014.
9. M. Autili and M. Tivoli. Distributed enforcement of service choreographies. In *Proc. of FOCLASA'14*, 2014.
10. T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines using domains with equality tests. In *Proc. of FASE'08*, page LNCS 4961. 2008.
11. A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proc. of ESEC/FSE*, 2009.
12. S. Ceri, D. Braga, F. Corcoglioniti, M. Grossniklaus, and S. Vadacca. Search computing challenges and directions. In *Objects and Databases*, volume 6348 of *LNCS*, pages 1–5. 2010.
13. V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller. Automatically generating test cases for specification mining. *Software Engineering, IEEE Transactions on*, 38(2):243–257, 2012.
14. D. Di Ruscio, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio. Model-driven techniques to enhance architectural languages interoperability. In *Proc. of FASE'12*, pages 26–42, LNCS 7212. 2012.
15. B. W. G. Hohpe. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, pages 1–480. Addison-Wesley, 2004.
16. D. Garlan. Software engineering in an uncertain world. In *Proc. of FoSER '10*, pages 125–128, 2010.
17. C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *Proc. of ICSE'09*, pages 430–440, 2009.
18. M. Güdemann, G. Salaün, and M. Ouederni. Counterexample guided synthesis of monitors for realizability enforcement. In S. Chakraborty and M. Mukund, editors, *Automated Technology for Verification and Analysis*, LNCS, pages 238–253. 2012.
19. R. Hilliard, I. Malavolta, H. Muccini, and P. Pelliccione. On the composition and reuse of viewpoints across architecture frameworks. In *Proc. of WICSA-ECSA'12*, pages 131–140. IEEE Computer Society, 2012.
20. D. H. Hu and Q. Yang. CIGAR: Concurrent and Interleaving Goal and Activity Recognition. In *Proc. of AAAI'08*, pages 1363–1368, 2008.

21. H. Hungar, T. Margaria, and B. Steffen. Test-based model generation for legacy systems. In *Proc. of ITC'03*, volume 2, pages 150–159, 2003.
22. P. Inverardi, M. Autili, D. Di Ruscio, P. Pelliccione, and M. Tivoli. Producing software by integration: Challenges and research directions (keynote). In *Proc. of ESEC/FSE'13*, pages 2–12, 2013.
23. P. Inverardi, R. Spalazzese, and M. Tivoli. Application-layer connector synthesis. In *SFM*, 2011.
24. P. Inverardi and M. Tivoli. Automatic synthesis of modular connectors via composition of protocol mediation patterns. In *Proceedings of ICSE'13*, 2013.
25. R. Kazhamiakin and M. Pistore. Analysis of realizability conditions for web service choreographies. In *In Proc. of FORTE'06*, pages 61–76, LNCS 4229. 2006.
26. R. Kazhamiakin and M. Pistore. Choreography conformance analysis: Asynchronous communications and information alignment. In *Web Services and Formal Methods*, volume 4184 of *LNCS*, pages 227–241. 2006.
27. K. Krogmann, M. Kuperberg, and R. Reussner. Using genetic search for reverse engineering of parametric behavior models for performance prediction. *Software Engineering, IEEE Transactions on*, 36(6):865–877, Nov 2010.
28. X. Li, Y. Fan, J. Wang, L. Wang, and F. Jiang. A pattern-based approach to development of service mediators for protocol mediation. In *proceedings of WICSA '08*, pages 137–146. IEEE Computer Society, 2008.
29. D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proc. of ICSE08*, 2008.
30. R. Mateescu, P. Poizat, and G. Salaün. Adaptation of service protocols using process algebra and on-the-fly reduction techniques. *Software Engineering, IEEE Transactions on*, 38(4):755–777, 2012.
31. K. Meinke. Automated black-box testing of functional correctness using function approximation. In *Proc. of ISSTA'04*, pages 143–153, 2004.
32. F. Paternó and C. Santoro. Preventing user errors by systematic analysis of deviations from the system task model. *International Journal of Human-Computer Studies*, 56(2):225 – 245, 2002.
33. G. Santhanam, S. Basu, and V. Honavar. Tcpcompose a tcp-net based algorithm for efficient composition of web services using qualitative preferences. In *Service-Oriented Computing ICSOC 2008*, volume 5364 of *LNCS*, pages 453–467. 2008.
34. R. N. Taylor, N. Medvidović, , and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*, pages 1–736. Wiley and Sons, 2009.
35. M. Trainotti, M. Pistore, G. Calabrese, G. Zacco, G. Lucchese, F. Barbon, P. Bertoli, and P. Traverso. Astro: Supporting composition and execution of web services. In *ICSOC'05*, pages 495–501, LNCS 3826. 2005.
36. A. van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.
37. A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *Proc. of ASE'09*, pages 295–306, 2009.
38. A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proc. of ESEC-FSE'07*, pages 35–44, 2007.
39. R. W. White and R. A. Roth. *Exploratory Search: Beyond the Query-Response Paradigm*. Synthesis Lectures on Information Concepts, Retrieval, and Services. 2009.