Automation and intelligent scheduling of distributed system functional testing

Lom Messan Hillah, Ariele-Paolo Maesano, Fabio de Rosa, Fabrice Kordon, Pierre-Henri Wuillemin, Riccardo Fontanelli, Sergio Di Bona, Davide Guerri, Libero Maesano

Automation and Intelligent Scheduling of Distributed System Functional Testing

Model-Based Functional Testing in Practice

Lom Messan Hillah[1,2], **Ariele-Paolo Maesano**[2,3], **Fabio De Rosa**[3], **Fabrice Kordon**[2], **Pierre-Henri Wuillemin**[2]
Riccardo Fontanelli[4], **Sergio Di Bona**[4], **Davide Guerri**[4], **Libero Maesano**[3]

[1] Univ. Paris Ouest Nanterre La Défense, F-92000 Nanterre, France
[2] Sorbonne Universités, UPMC Univ. Paris 06, CNRS, LIP6 UMR7606, F-75005 Paris, France
e-mail: {lom-messan.hillah, fabrice.kordon, pierre-henri.wuillemin}@lip6.fr
[3] Simple Engineering France, F-75011 Paris, France
e-mail: {ariele.maesano, libero.maesano, fabio.de-rosa}@simple-eng.com
[4] Dedalus S.p.A, 50141 Firenze, Italy
e-mail: {riccardo.fontanelli, sergio.dibona, davide.guerri}@dedalus.eu

Received: date / Revised version: date

Abstract. This paper presents the approach to functional test automation of services (black-box testing) and service architectures (grey-box testing) that has been developed within the MIDAS project and is accessible on the MIDAS SaaS. In particular, the algorithms and techniques adopted for addressing input and oracle generation, dynamic scheduling, and session planning issues supporting service functional test automation are illustrated. More specifically, the paper details: (i) the test input generation based on formal methods and temporal logic specifications, (ii) the test oracle generation based on service formal specifications, (iii) the dynamic scheduling of test cases based on probabilistic graphical reasoning, and (iv) the reactive, evidence-based planning of test sessions with on the fly generation of new test cases. Finally, the utilisation of the MIDAS prototype for the functional test of operational services and service architectures in the healthcare industry is reported and assessed. A planned evolution of the technology deals with the testing and troubleshooting of distributed systems that integrate connected objects (IoT).

1 Introduction

Service orientation is the design and implementation style most adopted in the digital economy. Cooperation between organisational entities, systems, applications and connected objects is carried out through distributed architectures of service components that: (i) handle a collection of business and/or technical functions, (ii) are accessible through Application Programming Interfaces (APIs), (iii) interact through service protocols such as REST/XML, REST/JSON [84], SOAP [62] ..., (iv) are distributed on different processes and physical/virtual machines, and (v) are deployed independently of each other. In brief, services are loosely coupled and this feature intrinsically enables agility of the engineering, delivery and deployment processes.

The Service Oriented Architecture (SOA) [39] approach has been employed for fifteen years to let heterogeneous applications cooperate [63]. More recently, systems have made available their functionalities to browsers and mobile apps through service APIs. Presently, the internal structures of applications, once monolithic, are going to be (re)designed as micro-services architectures [64] that are particularly well adapted for cloud deployment. With the service orientation applied to the Internet of Things (IoT) – things as (micro-)services [73] – there will be billions of services in the digital ecosystem (trillions in perspective).

The development, integration, delivery and deployment in the production environment of each component release can be accomplished independently from one another, if the service specifications (interfaces and semantics) are implemented consistently. This is the case for corrective maintenance – only implementation changes for bug fixing – but can also be the case of perfective maintenance, when service specifications evolve in a backward-compatible manner, by adding new operations without changing the old ones, or by implementing already planned extensions of input/output data structures.

The service integration process is a pure testing process. Integrating a service component with other upstream and downstream services means testing that the interactions are accomplished as specified and that they produce the expected effects. The service integration process is comprised of all testing activities: functional, security, robustness and performance.

Usually, testing activities are placed as stages between the service build release formation and its delivery in the production environment (the *service build integration pipeline* [1]). The transition of the service build release from one stage

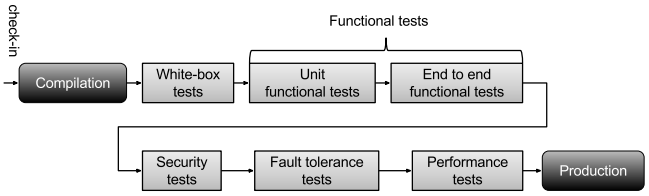

Fig. 1: Service build integration pipeline.

to the next one is permitted only when the stage tests pass, otherwise the sequence is interrupted and restarted with the check-in of the updated code. An example of service build pipeline is sketched in Fig. 1. The test tasks in each stage and the chosen sequence of the test stages should maximise the effectiveness (the fault exposing potential and the troubleshooting efficacy) and the efficiency (the fault detection rate) of the testing tasks.

A new service build release should be firstly submitted to acceptance white-box tests. All subsequent test stages target different aspects of the service external behaviour and are independent of the service implementation technology. The service build pipeline can be automated. A single pipeline stage can be fully automated, if: (i) its internal tasks can be fully automated and the stage produces automatically machine-readable outputs, and (ii) the automated tasks can be invoked through APIs by the pipeline orchestrator (for instance Jenkins [2]).

This paper illustrates the algorithms and techniques supporting automated generation and scheduling of test cases for service functional tests; they are the foreground of the EU FP7 MIDAS project [3]. The MIDAS prototype is a Software as a Service on the cloud that provides model-driven automation of test tasks, specifically – unit and end-to-end testing of services and service oriented architectures. The MIDAS prototype architecture and utilisation scheme are sketched in Fig. 2. The user, through APIs and a GUI (Graphical User Interface), supplies MIDAS with models and policies that describe the structure and the behaviours of the Distributed System Under Test (DSUT), in the specific case, of services and service architecture under test, and the test objectives, and invokes automated test generation and execution methods. MIDAS generates test suites and dynamically schedules and runs test sessions. In the test session, the MIDAS test system interacts with the DSUT, whose service components can be deployed anywhere: on premises, on private clouds and on public clouds.

Section 2 gives the research motivation on the topic and a short review of the state of the art about service functional testing. Section 3 presents the proposed techniques and algorithms supporting automated generation and scheduling that are used within the MIDAS prototype and are provided as-a-service by the MIDAS SaaS platform. Dedalus, a company specialised in healthcare systems, and a partner of the MIDAS project, has incorporated the functional test automation services of the prototype in its integration process: this experience is presented in Section 4. Section 5 relates the Dedalus team's evaluation of the experience. Finally, Section 6 discusses major advantages of the new solution and outlines future work.

2 Related work

Service testing and, in particular, end-to-end testing of large-scale service architectures is difficult, knowledge intensive, time-consuming and costly in terms of labour effort, hardware/software equipment and time-to-market. Since the usage of service oriented architectures began, service testing automation has been a critical challenge for researchers and practitioners [27][33][86][88]. In particular, tasks such as: (i) the optimised generation of test inputs [27], (ii) the generation of test oracles [19][24][74], and (iii) the optimised management of test suites for different test activities – such as progression testing, change testing, regression testing [88] – have not yet found automation solutions that can be applied to real complex service architectures such as those that are implemented in the healthcare industry [33].

Model-based testing (MBT) approaches have been proven to be suitable to address automation issues for testing. They utilise formal models (structural, functional and behavioural) of the service architecture under test to undertake the automation of the testing tasks [42]. The "first-generation" of MBT research is essentially focused on test input generation. It uses formal methods, especially SAT/SMT-based techniques [20][26][44], that allow the exhaustive exploration of the system execution traces, and efficient test input generation satisfying constraints that are formal properties expressed in temporal logic. Most MBT approaches and tools for testing automation of services are based on specific service specification languages, the most popular being WSDL – Web Service Description Language [4], and service composition languages, the most popular being WS-BPEL – Web Services Business Process Execution Language [66]. A notable approach based on WSDL is WS-TAXI [25], that combines the coverage of WS operation with data-driven test generation. It puts together SoapUI [5], the most popular Web services black-box testing tool, and TAXI, an application that automates the generation of XML instances from an XML schema. TAXI improves SoapUI by adding automated generation of test inputs starting from the operation data input specified within the WSDL file. However, it provides limited generation features (e.g. it does not handle domain-specific string generation, which is crucial for generating meaningful business data for the service under test), and only supports unit testing of services.

Many other approaches have advanced the state of the art regarding automated model-driven functional testing of Web services [28][48][52][56][87], and of composition of Web services through WS-BPEL processes (see [78] for a survey), but they are limited to unit black-box testing, combined with the "assisted automation" of either test case generation or test execution, or test case selection, as currently offered in available state-of-the-art tools such as SoapUI [5], Oracle

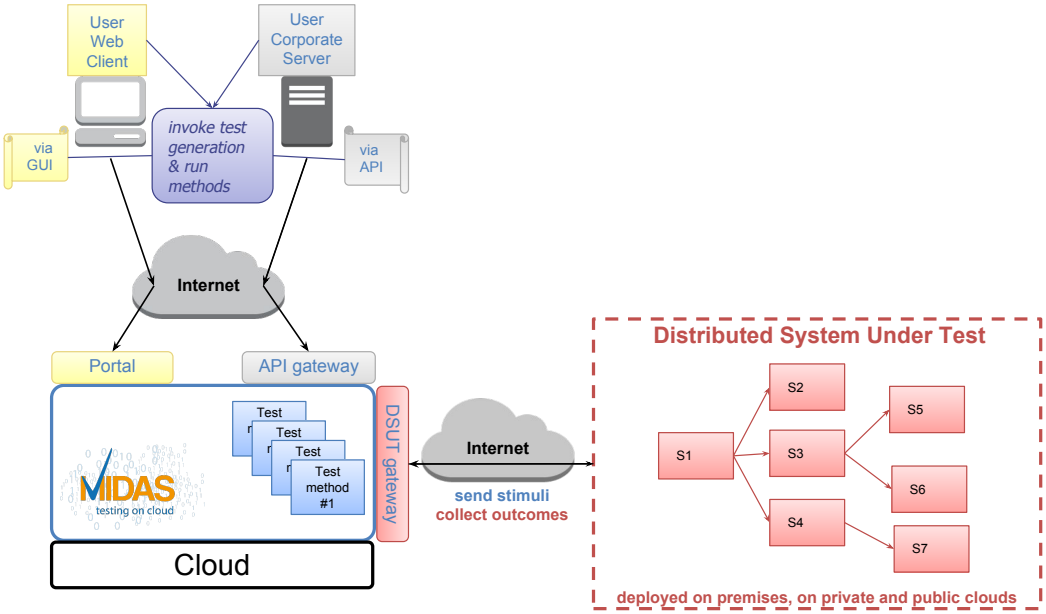

Fig. 2: MIDAS utilisation scheme.

SOA Suite [69], and Parasoft [16]. In particular, to the best of our knowledge, there is no available solution for the full automation across the spectrum of the critical tasks (test input and oracle generation, test execution/arbitration, dynamic test case prioritisation, test planning) of end-to-end testing of large-scale, multi-stakeholder services architectures.

An interesting approach to supporting testing task automation for service architectures, specifically for automated configuration and setting up of testing environments, is proposed by the Genesis project [47]. It provides developers a framework for generating service-based infrastructures, allowing them to set up customized SOA testbeds of services (that are virtual services). This approach is overall very manual and places detailed and complex system-level configuration burdens on the developers. In comparison, the MIDAS approach shields the developer from complex back-end set ups. At the front end, the developer only provides models of the distributed system under test: services, service architecture, behaviour of each component, and bindings of behaviours to the components involved in the scenarios to be tested. At the back end, the developer only needs to provide the endpoints of services under test (deployed on premises, on private and public clouds), and redirect endpoint addresses to the MIDAS DSUT Gateway (Fig. 2) so that requests and responses can be intercepted. Moreover, in contrast with the Genesis approach, there is no need of downloading, installing and configuring any framework in order to start with our solution. Modelling can be performed with classic IDEs, or even simple text editors if the developer is comfortable with the format (XML) of the input models. We are planning to further improve this step by pushing modelling on-line in a browser with wizards to assist the developer in checking structural consistency across the models, behavioural consistency across the scenarios to be tested, and many other useful modelling-level tasks that will render the whole approach very easy to use and fast to set up.

Another interesting approach to SOA testing has been proposed by the WS-Diamond project [34]. It provides a platform, to be installed in the DSUT production field, for supporting the self-healing execution of Web Services, that is, services able to self-monitor, self-diagnose the cause of a functional failure, and self-recover from those failures. This project also provides a framework, including methodologies and tools for services design that guarantee diagnosability and reparability during their execution. The WS-Diamond approach is focused on the design, implementation and execution of Web Services in a controlled environment. It is a fully integrated modus operandi and the developers are obliged to abandon their existing methodologies and tools if they want to benefit from it. The MIDAS perspective is different: it is intended to help the developers troubleshoot any kind of service architectures and requires neither special tooling or platform set up, nor specific design and development methodology.

3 Automating service functional test

In this section we present the unit and end-to-end functional test stages through an example of a real-world services architecture, depicted in Fig. 3 (The Calabria Cephalalgic Network – CCN).

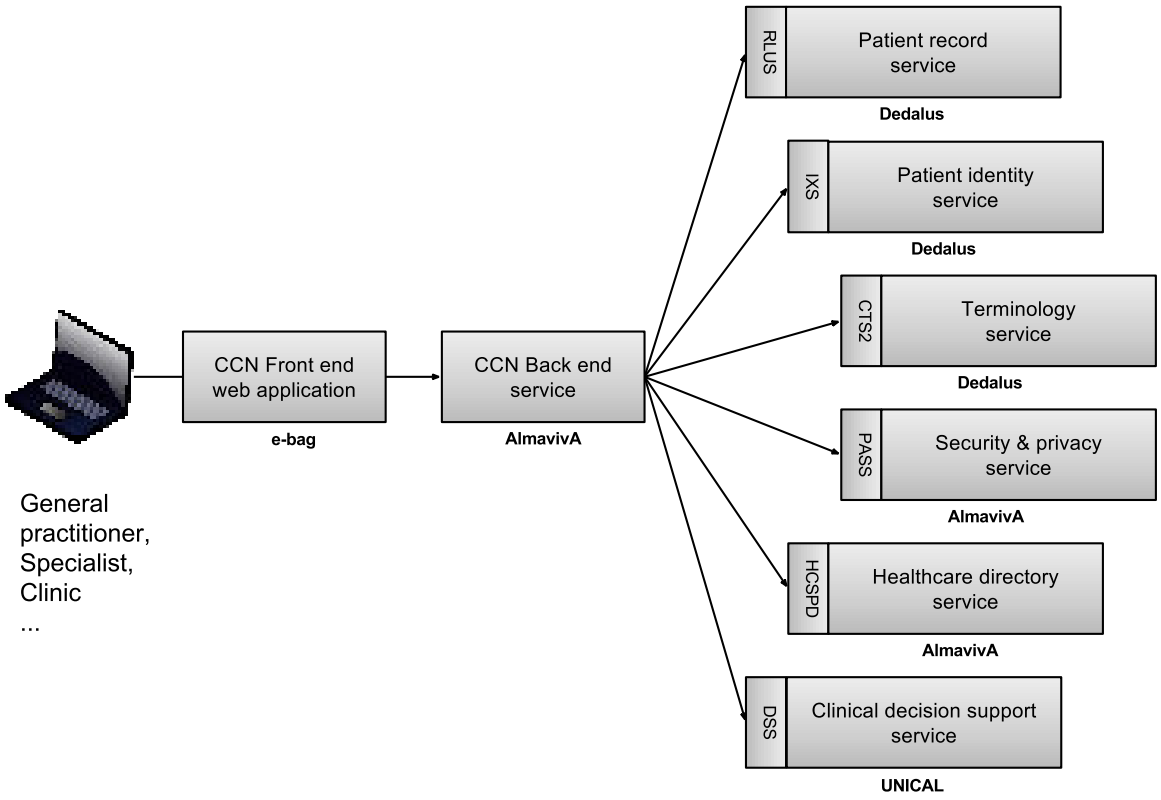

Fig. 3: Calabria Cephalalgic Network.

3.1 Introducing functional testing of services

The Calabria Cephalalgic Network (CCN) [33] is a Web application that supports the integrated care processes of the headache chronic disease, effectively coordinating different care settings (general practitioners, specialists, clinics, labs...) in a patient-centred vision. The application has been designed and developed on a multi-stakeholder services architecture (Fig. 3), and its service components are physically deployed in separate data centres and private clouds. In particular, Dedalus [6] is in charge of the provision of: (i) the patient record service – which handles the clinical and administrative data elements related to the patient; (ii) the patient identity service – which handles the patient identifiers and demographic data; (iii) the terminology service – which manages the complex terminology and the codes involved in the clinical and administrative processes. These services implement the HL7/OMG Healthcare Services Specification Program (HSSP) international standard (respectively RLUS, IXS, CTS2) [7].

The CCN distributed application is an example of microservices architecture: patient identity management, patient record management, etc., are implemented as loosely coupled services, each of them being equipped with specialised software running on different machines and whose data are stored in separate databases. Note that these constituent services have life cycles, in terms of software releases and managed data, that are independent from each other and also from the CCN application life cycle.

The CCN back end service consumes the patient record service, the patient identity service, and other services in order to provide its service. We call these downstream services that are not consumers of other services *terminal* services. Therefore, the CCN Back end service is a *non-terminal* service. The service unit test stage includes the following tasks:

- generation – asynchronously or on the fly – of operation inputs to be used as test inputs (stimuli);
- generation – asynchronously or on the fly – of test oracles (the expected outcomes of the service under test);
- deployment and initialisation of the service build release in an appropriate environment;
- configuration of the test system;
- binding of the test system with the service under test;
- execution of test cases – transmit stimuli, collect and log outcomes;
- arbitration of the test outcomes against test oracles;
- dynamic scheduling of test runs for test case dynamic prioritisation, on the basis of the arbitration verdicts;
- reporting of test sessions – building meaningful test session summaries from the bulky logs;
- planning of (new) test sessions on the basis of the results of the current one – reactive planning.

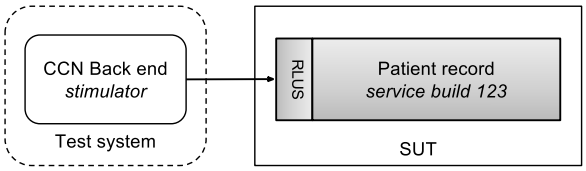

Fig. 4: Unit test environment for a terminal service.

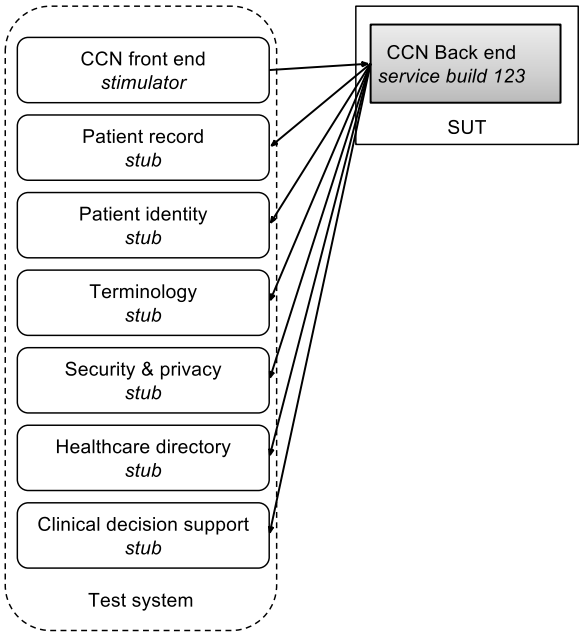

Fig. 5: Unit test environment for a non-terminal service.

An example of test environment architecture for unit test of terminal services is sketched in Fig. 4. It depicts the unit test environment of the Patient record service.

The unit test system for terminal services is configured with a stimulator test component that is able to: (i) transmit to the service under test (SUT) a test case input taken from the test suite, (ii) wait for a SUT outcome, (iii) collect the corresponding test outcome, (iv) compare the outcome with the test oracle, (v) produce test verdicts, and (vi) choose the next test case on the basis of the past test verdicts (dynamic scheduling).

For non-terminal services, such as the CCN Back end service, the typical unit test environment is depicted in Fig. 5. The test tasks involved in the stage are the same ones described for the terminal services, but the test system is configured with six stubs (virtual services) in addition to the stimulator. The stubs simulate the downstream service components. Here, the tester's objective is to test the single SUT behaviour.

When a test case includes the interaction with a downstream service, the correct behaviour of the SUT is to issue the expected input towards the service stub. If the stub receives the input in a finite time interval, it compares this input with the corresponding oracle, emits a local test verdict and, if the verdict equals *pass*, returns the canned response that is specified for the involved test case. If the input is not received in the expected time the local test verdict is set to *fail*.

The architecture of unit test of non-terminal services allows for highlighting the role of the Test Component Manager, a component of the test system that coordinates the actions of the virtualized services (stimulator, stubs) and manages the test run. The Test Component Manager collects the local verdicts (that are issued by the virtualised components), continues or halts the test run and aggregates the local test verdicts in a compound global test verdict. The standard values of the test verdicts [68] are: (i) *pass* – the test outcome is collected in a predefined interval time and matches the oracle, (ii) *fail* – either the test outcome mismatches the oracle or the related timeout event is raised that is interpreted as a failure, (iii) *error* – an error of the test system or of the SUT configuration has been detected that could also be a timeout to be attributed to an infrastructure failure, (iv) *inconc* – the verdict cannot be *pass* but the arbiter is unable to choose between *fail* and *error*, and (v) *none* – no verdict because the outcome has not yet been produced and will not be produced in the current run. After a test case run, a global compound test verdict is established that is the aggregation of the local test verdicts.

Service virtualisation allows unit testing of systems that have service dependencies. There are different degrees of virtualisation. The most advanced currently available commercial tools [15][16][17][18] offer virtualised service constructors that allow building empty stub components whose binding with the service under test is facilitated, but that must be: (i) programmed by the tester in order to provide the appropriate canned responses, and (ii) deployed by the tester in the test environment. The MIDAS solution increases the test automation by two steps: firstly, the stub canned responses are automatically produced by the test generator and part of the test cases and, secondly, at each test session, the stubs specified in the Test Configuration Model (see Sect. 3.2) are automatically created, deployed on the cloud platform, bound to the service under test and configured with the canned responses specified in the test cases.

In the end-to-end test stage of the service build, the DSUT is deployed somewhere, with well-identified endpoints. In the test system, in addition to the stimulator, interceptors are configured that are able to catch the exchanges (back and forth) between the deployed services, i.e. between the CCN back end service and the downstream services, and to arbitrate them (Fig. 6). Each interceptor waits for and catches the message issued by the CCN back end and arbitrates it: if the message is received in time and matches the oracle, the interceptor transmits it to the target downstream service. Otherwise, a local *fail* test verdict is established, the Test Component Manager halts the test case run and establishes the compound test verdict. If the test case is still running, the interceptor waits for and catches the terminal service response and arbitrates it: if the response is received in time and matches the oracle, the interceptor transmits the response back to the CCN back end service, otherwise a local *fail* test

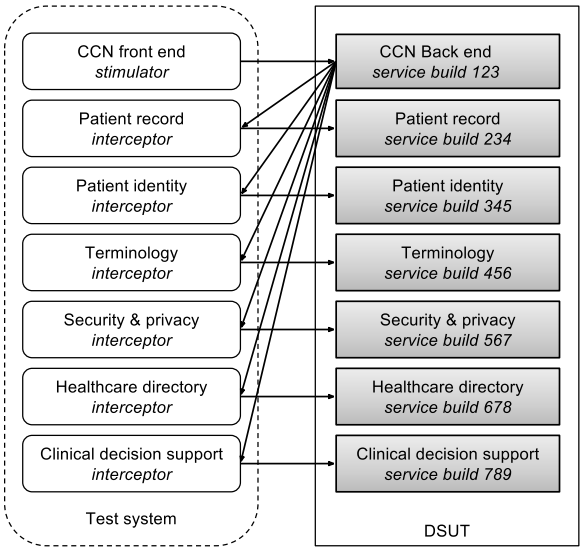

Fig. 6: End-to-end test of a distributed system.

verdict is set up, the Test Component Manager halts the test case run and establishes the global compound test verdict.

3.2 Model based automation of service functional testing

Our MBT approach to the automation of the test tasks (test generation, execution, arbitration, scheduling, reporting and planning) is based on the following models: (i) the Test Configuration Model (TCM), (ii) the Service Interface Model (SIM), and (iii) the Service Behaviour Model (SBM). Therefore, they are inputs of the functional test method implemented in the MIDAS platform, and are provided by the tester before starting any test task.

The TCM includes the service architecture (DSUT) model (the collection of services and links among them), and the Test System Model (the suite of test components needed to put in place the test scenarios). The TCM is represented by a collection of XML documents that are validated against the standard Service Component Architecture (SCA) XML schemas [8][29]. The DSUT model is a topological model of the service architecture under test: it allows for a definition of the *actual components* of the architecture, the services that each component provides and consumes and the *actual wires*, i.e. the service dependencies between components. Each actual component is typed by a participant, i.e. a definition of a component abstract type as an aggregation of provided and required service interfaces. The Test System Model defines the structure of the test system in terms of *virtual components* (stimulators, stubs), their connections with the DSUT components through *virtual* wires, and *probes* on the actual wires (interceptors). Each virtual component is typed by a participant. Thus, the TCM is a graph of nodes (actual and virtual components) and links (actual and virtual wires).

The SIM is a collection of standard definitions of the services of the DSUT, i.e.: for SOAP services, WSDL 1.1 documents [4]; for REST/XML services either WSDL 1.1 or WSDL 2.0 [9] documents, and for REST/JSON services Swagger [10] documents. The current implementation of the MIDAS platform only supports SOAP services specified by WSDL 1.1. In the next release of the platform, we plan to support REST/XML and REST/JSON services.

Each TCM virtual and actual component is equipped with a Protocol State Machine (PSM), that is a Harel statechart [40] modelling the external interaction behaviour of the component. Each PSM defines the states of the "conversation" of the component with its wired interlocutors and the transitions between these states that: (i) are triggered by events (i.e. message reception or timeout), (ii) are filtered by Boolean conditions (guards), and (iii) carry out effects (i.e. the issuance of a message). The contents of the issued messages are defined by *data-flow transfer functions*, i.e. expressions that calculate the elements of the messages to be sent as functions of the elements of the received messages and of data related to the initial state. In the actual implementation, a PSM is represented through the W3C standard SCXML [11] formalism. SCXML provides a powerful, general-purpose and declarative modelling language to describe the behaviour of timed, event-driven, state-based systems. A SCXML artefact is executable, i.e. it can be directly interpreted by a compliant SCXML engine. The SCXML standard is composed of several modules. The *core module* provides the elements of a basic Harel state machine, such as *state*, *parallel*, and *transition*. The *external communication module* provides the means of external event exchange such as *send*and *invoke*. The *script module* provides the support for ECMAScript [37] implementation of executable content (i.e. actions performed on transitions, entering and leaving states, emitting events, branching on conditions – if, else, elseif, updates on the data model – assignment of values, logging). The PSM data-flow transfer functions are written in ECMAscript. The *data module* provides the abstraction for handling named parts (e.g. arbitrary data payloads in send events, evaluation of conditional expressions on received data in events on transitions). The SBM is a collection of PSMs of the TCM actual and virtual components.

3.3 Automated generation of test cases

Figure 7 sketches the activity diagram of the automated test case generation. The input objects are:

- the TCM – the collection of XML documents, one for each participant (component abstract type), plus a description of the test environment topology (the actual and virtual component, and the actual and virtual wires that link the components);
- the SIM – the collection of documents that describe the service interfaces;
- the SBM – the collection of PSM/SCXML documents, one for each actual and virtual component;
- a collection of test input templates – optionally, templates for the test inputs (stimuli) can also be supplied by the

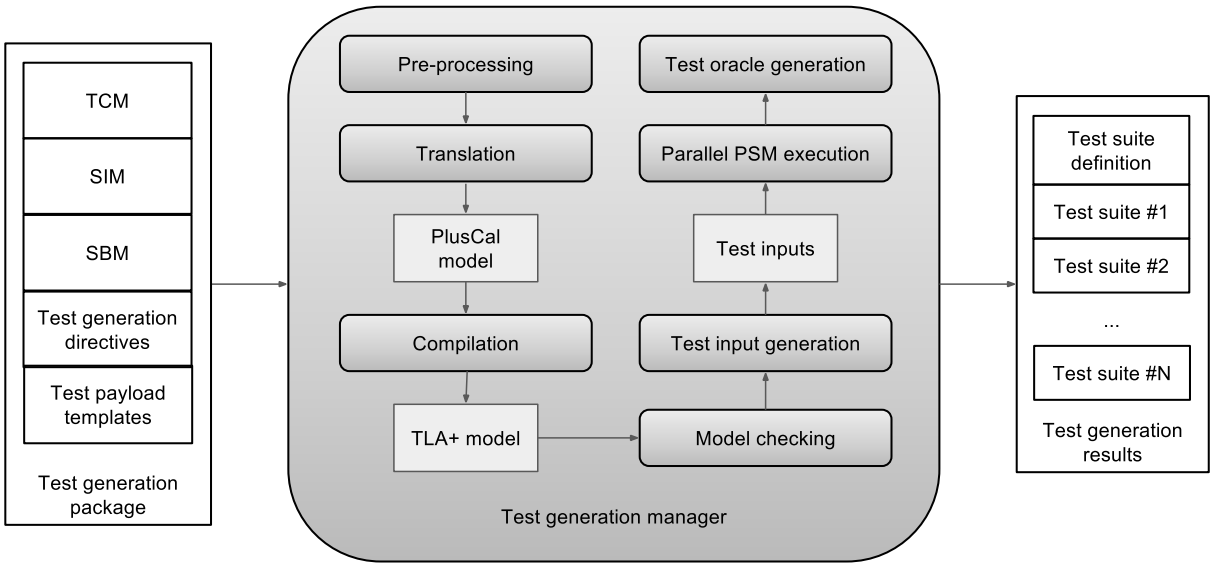

Fig. 7: Automated generation of test cases

user to guide the test case generation. This is useful for focusing on relevant ranges of business data values for the invoked operations, instead of using randomly-generated values;
– a set of test generation directives – in order to configure the behaviour of the test case generator; for instance, the test generation can be focused on specific types of messages to enable the exploration of specific logical and "physical" regions of the behaviour of the service architecture under test.

The outputs produced by the generation task are:

– Test Scenario Definition (TSD) model – the test scenario definition is an XML document that represents the abstract interaction paths that are the results of the simulated execution of the collection of PSMs.
– Test Suite (TS) files – one or more test suites – a Test Suite file is a collection of test cases; each test case instantiates a TSD interaction path. A test case is a collection of messages (input, oracles) in the partial order dictated by the interaction path.

The test case generation uses model checking techniques provided by the TLA+ framework [50] that "implements" the well-known TLA formal specification language based on temporal logic. By providing a mathematics-based formal specification language with set theory and predicates, TLA+ allows for reasoning regarding the behaviour of a system from its specifications (i.e. description of the system model). Describing a system model using TLA+ enables a designer to specify all possible behaviours, or *execution traces*, of the system.

TLA stands for Temporal Logic of Action, a simple form of temporal logic that allows specifying temporal formulas F that are assertions about behaviours. When a behaviour σ satisfies F, then F is true of σ. A temporal formula that is satisfied by all behaviours is a theorem, or a *true* formula. Hence a property P of a specification S is specified as a temporal formula. A specification S is said to satisfy P iff $S \implies P$ is a theorem, meaning that P is true for all behaviours in S.

Model checking [31] is a mechanised formal verification technique that checks that a given logic formula holds on a given model. The model checking problem $M(L)$ for a temporal logic L is the problem of deciding, for any input S (a Kripke structure) and temporal formula φ, whether $S \vDash \varphi$ (S satisfies φ). A Kripke structure over a set $A = \{P_1, P_2, \ldots\}$ of atomic propositions, as defined in [77], is a tuple $S = \langle Q, R, l, I \rangle$ where:

– $Q = \{q, r, s, \ldots\}$ is a set of states representing the configurations of the system;
– $R \subseteq Q \times Q$ is a transition relation between pairs of states in Q. It is generally assumed, for simplification, that R is total, i.e. for any $q \in Q$, there exists q' such that $q\ R\ q'$;
– $l : Q \to 2^A$ labels states with propositions. $P \in l(q)$ means that P holds in state q;
– $I \subseteq Q$ is a non-empty set of initial states (or configurations).

The Kripke structure represents the state space of the system. A Kripke structure S is finite when Q is finite. According to [77], the model checking problem $M(L)$ is decidable for the great majority of propositional temporal logics, and the cost of deciding whether $S \vDash \varphi$ is a function of the sizes $|S|$ and $|\varphi|$ (both in space and run-time; we refer the interested reader to [77] for more details). $|S|$ is the sum of the number of nodes and edges of the Kripke structure, and $|\varphi|$ is the number of symbols in $|\varphi|$ (seen as a string). The size of the transition system grows exponentially in the number of concurrent components (e.g. the composition of N components, each of size k, yields k^N states), or in the number of program variables for program

graphs. This well-known challenge is called the *state explosion problem* [32]. Model checkers, like the Temporal Logic Checker (TLC) component provided by the TLA+ framework, combine different techniques to tackle this problem, such as partial order reduction (if in the order of execution the interleaving is not relevant, not all possible combinations are examined), symbolic verification (where variables of the verification algorithm denote sets of states rather than single states, using decision diagrams), abstraction techniques (e.g. partitioning the states of S into clusters, and handling those as abstract states), and bounded model checking (given a bound k, generates a formula that is satisfiable iff the property can be disproved by a counterexample of length k; if no counterexample, the bound k is incremented).

TLA+ also allows the specification of correctness properties, mainly safety and liveness properties. A safety property asserts that nothing bad happens during the execution of the system. It can be violated by a single step in the behaviour, or the first state of the running system. A liveness property asserts that something good eventually happens (e.g. an algorithm eventually terminates). The entire behaviour of the system must be observed before one can conclude that the liveness property is violated. For example, if you want to check the liveness property that eventually x equals y in a program you need to see the entire behaviour of the program to know that x is never equal to y [50].

TLA+ has been successfully used in practice, for instance in checking cache-coherence protocols [46], describing and verifying web service composition [82], and recently in particular by Amazon Web Services to strengthen the implementation and coordination of fault-tolerant distributed algorithms for their Simple Storage Service (S3) [60][61].

Each step of the test generation activity depicted in Fig. 7 is described hereafter.

Preprocessing. The Preprocessing phase enables the consistency checking across all input artefacts, the symbolic binding of the actual and virtual components through actual and virtual wires, and the construction of a parallel state machine which combines all individual component PSMs.

Translation. During the translation phase (Translation activity in Fig. 7), the PSMs are translated into a PlusCal [51] program. PlusCal is a TLA+ companion algorithmic language close to C-style or pseudo-code programming, that is supported by the TLA+ framework. PlusCal is utilised for writing formal specifications of algorithms, in a style that is more convenient for readability and easier to understand than TLA+, and it can be compiled into a TLA+ specification that can be checked with the TLA+ tools, in our case the TLC model checker. The translation follows these steps:

1. Capture all ECMAScript expressions manipulating SOAP data in the PSM, extract the manipulated fields and use their declarations from the WSDLs to declare them as variables in TLA.
2. If possible values for some fields have been refined in the message templates (for example as enumerations), refine their declaration in TLA.
3. Translate the state machine in TLA.

Data that are declared in the PSM, but not referenced in any ECMAScript expressions are considered irrelevant, so they are ignored (i.e. treated as constants).

Compilation. The model obtained in PlusCal from the translation activity is then compiled into the TLA+ core language, by using the compiler provided in the TLA+ toolkit (Compilation activity in Fig. 7). TLA+ is backed by the TLC model checker to exhaustively check correctness properties across all possible executions of the system. At this stage we generate, from the generation directives, one safety property. Through assertions, execution traces of the system satisfying the safety property – for instance the negation of "messages of some specific types are exchanged" – are requested to the model checker (Model checking activity in Fig. 7). Hence, the TLC model checker achieves the generation of the execution traces by checking the assertion and producing counterexamples if it is violated. For the use case reported in Sect. 4 regarding the prototype usage, a typical TLA+ specification for a complete scenario was 600 lines long.

Test input generation, parallel PSM execution, and oracle generation. The safety property to be checked can be indirectly specified by the user in the test generation directives, or automatically by the dynamic test scheduler (described in Sect. 3.4). The directives specify generation requirements like the requested number of test cases to generate, the timeout for the test generation, and the set of message types that should be involved in the execution traces of the system. This latter directive is translated into a safety property to be checked by the TLC model checker. For instance, the user of the test cases generator might be looking for test cases where the message type *input* of the operation *createIdentityFromEntity* exposed by the service *POCDPatientMQService* of the actual component *mpi.ixs.component* in the DSUT *standardportal.saut* can be observed. This example has been used for testing the Healthcare Pilot described in Sect. 4. All these test generation directives are specified in a simple XML configuration file alongside the files of the TCM, SIM, and SBM.

Driven by the test generation directives and the templates for the payloads provided by the tester, the test case generator can focus on interesting specific ranges of values for the test case payloads. Discrete data domains (numbers, enumerations, booleans) from the definition (WSDL file) of SOAP message fields are handled well by the TLC model checker. Although TLC also handles strings and can generate arbitrary ones, we allow regular expressions to be able to generate arbitrary meaningful business values on the fly during the execution of the PSM, like for example `2.16.840.1.113883.2.9.3.12.4.1` in the case of the CCN healthcare system.

There is no optimization for the selection of the number n of test cases from the entire solution space, which is left to the model checker. According to the generation directive about the requested message types to observe, the scheduler does not care whether there are only $m \leq n$ ($m > 0$) test cases satisfying the directive. According to its scheduling policy and the outcome of executed test cases, or if there are no test case satisfying the directive, the scheduler will request on the fly the generation of new test cases, by updating the directive. We are working on the implementation of other generation strategies such as: (i) random sampling of infinite discrete data domains, (ii) boundary values, and (iii) domain partitioning by analysing the data-flow transfer functions in the PSMs.

Input data are then extracted from the execution traces (Test input generation activity in Fig. 7). The obtained test inputs are supplied one by one to the SCXML parallel state machine built as a composition of the individual PSMs (one for each component) and the SCXML execution engine is invoked (Parallel PSM execution activity in Fig. 7). The execution of the parallel state machine, which works as an executable specification of the entire DSUT, is monitored and all generated events and messages are collected, allowing the test generator to produce: (i) one test suite definition (TSD) encoding the interaction path extracted from the execution, (ii) for each test case input the corresponding oracles, the test case being compliant with the interaction path defined in the TSD.

Our test generation approach combines both the data-centered and the logic-centered approaches, as described by Mayer et al. [56]. In the data-centered approach, fixed SOAP data is used. Incoming data is compared against predefined SOAP message, and outgoing data is also predefined. The messages are, for example, stored in XML files on disk, and referenced from the SCXML/DATAMODEL/DATA construct in the PSMs. This approach is simple, but not very flexible and expressive. In the logic-centered approach, a fully-fledged programming language is used for expressing the test logic. A program taking arbitrary steps can be applied on incoming messages to test the data. Likewise, outgoing data is created by the program. It is a very flexible and expressive approach which requires considerable implementation effort from the test developer. In our approach, it is seamlessly handled by the SCXML execution engine that supports executable content in ECMAScript expressed in the PSM, such as actions on transitions, in entering and leaving states, emitting events, branching on conditions, testing incoming data, creating outgoing data – possibly by precisely modifying specific fields in the SOAP messages skeletons or directly on incoming data (transfer functions), and logging.

The additional requirement of automation is fully met and tool support is partially met in our approach: the specification is unambiguous, machine-readable and executable, and the test logic can be as sophisticated as the test developer wishes. However, the creation of the test specification is not yet supported by wizards that will help in checking structural consistency across the different types of input models, or checking behavioural consistency across the PSMs, automatically.

Discussion on the modelling approach for the TCM and the PSMs. It is clear that TCM and SBM building is a modelling effort that requires the test designer to have knowledge of: (i) the topology of the DSUT; (ii) the external behaviour of the services under test; (iii) the capability to express this knowledge in terms of test configuration model, protocol state machines and data flow transfer functions. Conversely, this activity requires neither knowledge of the implementations of the service components nor advanced testing skills. There is no size limitation for the SBM models. Moreover, the main SCXML constructs supported in the current implementation of the MIDAS prototype are the following:

- The top-level SCXML wrapper element of a SCXML document. The actual state machine consists of its children elements.
- The elements SCXML/STATE (for atomic states), SCXML/FINAL (for final states). Atomic and final states can occur zero or more times. A state is active if it has been entered by a transition and has not subsequently been exited. The state machine must always be in only one active state. The SCXML processor must terminate processing when the state machine reaches a final state.
- The element SCXML/DATAMODEL that is a wrapper (that occurs zero or one time) encapsulating any number of SCXML/DATAMODEL/DATA children elements, each of which defines a single data object. The exact nature of the data object depends on the data model language used. Supported data model languages are ECMAScript and XML/XPATH [12]. In the MIDAS prototype, those data elements refer to (through their *src* attribute) SOAP message skeletons or predefined SOAP messages with specific data. These messages are stored in XML files.
- The element STATE/TRANSITION, for outgoing transitions from states. The *event* attribute of a transition allows for the designation of the event that enables this transition. In the MIDAS prototype, it must explicitly refer to the service interface provided or required, the operation, and operation type (input, output or fault), as in the following example: `POCDPatientMQService::findIdentitiesByTraits::input` where a request is awaited. The *cond* attribute of a transition allows for specifying any boolean expression that guards the firing of the transition. A boolean expression can reference data in the content of the event (built-in _event.data object), and the data model (i.e. SCXML/DATAMODEL/DATA). When there is no condition, an enabled transition is always fired. The *target* attribute references the target state.

  Effects are transition sub-elements that contain expressions that are evaluated when the transition is fired. Allowed effects in the MIDAS prototype are:
  - TRANSITION/ASSIGN which modifies the data model through its *location* attribute which designates the location to be filled with the result of the evaluation

of a functional expression hold in its *expr* attribute. The value of the *location* can pinpoint a specific field in the SOAP message referenced by the data model.
– TRANSITION/SEND which sends an event through its *eventexpr* attribute, and the associated data through its *namelist* attribute, to another PSM. A send operation thus creates an event specified by the *eventexpr* attribute. That event is used as a trigger by the receiving PSM transition in its *event* attribute. For example, the sent event `POCDPatientMQReference::findIdentitiesByTraits::input` matches the expected event `POCDPatientMQService::findIdentitiesByTraits::input`. The message content sent through *namelist* is referenced in the built-in _event.data object on the receiving end. Hence, the value of the *namelist* attribute is the reference to a SCXML/DATAMODEL/DATA element which references a SOAP message that could have been modified by the TRANSITION/ASSIGN beforehand.

The SBM representation approach and formalism – PSMs expressed in an easy and standard XML format, and data-flow transfer functions expressed in ECMAScript, are the most general and easy-to-use choices for service developers and testers. In fact, as a respected professional service developer states: "Whether you choose to become a REST ninja, or stick with an RPC-based mechanism like SOAP, the core concept of the service as a state machine is powerful" [64]. Developers and testers of Web services are comfortable with XML and are obliged to work with XML if they want to understand in detail the match/mismatch between the actual payloads and the oracles underlying the *pass/fail* verdicts. Moreover, JavaScript is one of the most popular and utilised programming languages that allows easy manipulation of XML structures, native access to JSON structures and whose "functional" flavour is perfectly adapted to express the data-flow transfer functions expressions [41].

3.4 Automated test execution with dynamic scheduling

The MIDAS platform provides functionalities to dynamically schedule executions and arbitrations of generated test cases (Fig 8).

Test case prioritisation, i.e. scheduling test case executions in an order that attempts to increase their effectiveness at meeting some desirable properties, is a powerful approach to maximise the value of a test suite and to minimise the testing cost. There are two general families of test prioritisation techniques: (i) coverage-based [21][22][30][57], and (ii) fault exposing potential (FEP) based [38][65][81]. Our approach to the prioritisation of test cases is entirely original [55]: it is based on the usage of probabilistic graphical models [71] to dynamically choose the next test case to run on the basis of the preceding verdicts. It accommodates both the coverage-based, the FEP-based and other approaches by simply specifying scheduling policies and beliefs. Moreover, the probabilistic inference is able to establish a dynamic relationship between test case prioritisation and the model-driven generation of new test cases, by supplying evidence-driven directives based on the previous verdicts to the generator on the fly. More specifically, the test session management component handles the schedule/execute/arbitrate cycle (see Fig. 9) as follows: (i) the scheduler either chooses a test case to perform on the basis of an inference step and communicates it to the executor, or stops the test session; (ii) the executor runs the test case completely or until some halting condition is met, then collects the outcomes and communicates these outcomes to the arbiter; (iii) the arbiter evaluates the outcomes, sets up the local verdicts, establishes the compound global verdict and communicates it to the scheduler. The schedule/execute/arbitrate cycle either continues until there are no more test cases to perform, or stops when some halting condition is met.

The scheduling component carries out the dynamic prioritisation of test cases in order to meet different objectives, such as improving the fault detection rate (the

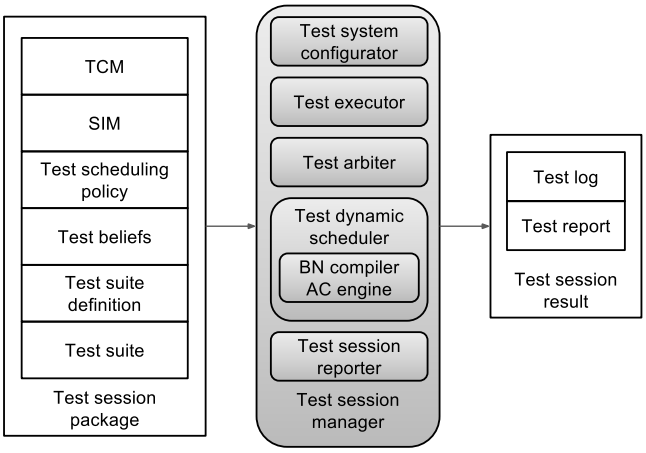

Fig. 8: Automated test run with dynamic scheduling.

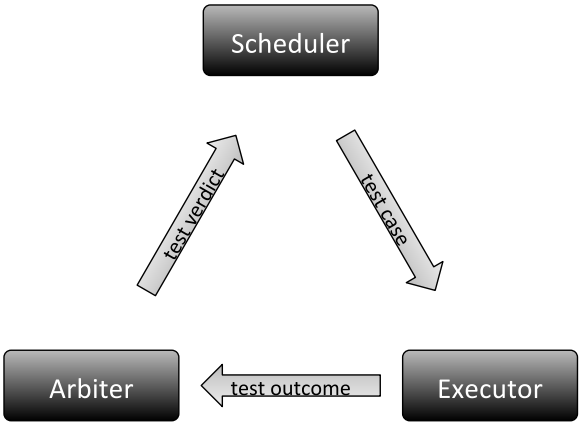

Fig. 9: Conceptual schema of the automated schedule/execute/arbitrate cycle.

discovery of the greatest number of failures in the smallest number of test runs), the quick localisation of faulty elements (troubleshooting), the test coverage of the DSUT, etc. These objectives are attained by specifying different scheduling policies .

The dynamic prioritisation of test cases for distributed system testing is typically a matter of decision under uncertainty, for which the probability theory is considered a powerful framework for representation and treatment [72].

Probability theory exhibits well-known advantages:

- modelling a complex reality, for which the exact inference is intractable, with a minimum number of parameters and the greatest accuracy,
- mathematically well-defined mechanism for representation [36],
- explicit modelling of uncertainty,
- management of multiple hypotheses about the state of the system.

In particular, the graphical approach (Bayesian Networks) [71], allows modelling conditional independences and stochastic relations between system properties and inferring the probability changes of these properties according to observations. In addition, the inference results are knowledgeable, in contrast to other approaches, such as Neural Networks, which act as black boxes. Furthermore, the Bayesian Network (BN) approach does not present the combinatorial problems of other methods such as Decision Trees. In their position paper at FSE/SDP workshop on the future of software engineering research (2011), Namin and Sridharan make the following claim: *Bayesian reasoning methods provide an ideal research paradigm for achieving reliable and efficient software testing and program analysis* [59].

Rees, Wooff and colleagues [75][85] present a seminal work about the use of probabilistic inference based on a BN framework to support input partitioning test methods that are aimed at understanding which kind of stimuli are more appropriate to reveal software failures.

Regression testing is a target of choice for prioritisation of test cases on the basis of specific criteria. Mirarab and Tahvildari [58] present an approach based on probability to prioritising test cases in order to enhance the fault detection rate. They utilise Bayesian Networks to incorporate source code changes, software fault-proneness, and test coverage data into a unified model.

The dynamic scheduler algorithm, referred in Fig. 10 as **inf4sat**, builds a Bayesian Network model from (i) the Test Configuration Model (TCM), (ii) the Test Scenario Definition (TSD) and the Test Suite (TS), and (iii) the (optional) user initial beliefs. It then compiles the Bayesian Network (BN) into an Arithmetic Circuit (AC), that is the data structure employed within the inference cycle. The objectives of the compilation and of the final representation technique are the reduction of the size and time complexity of the BN inference, in particular the size complexity of the data structure representing the graphical network and the time complexity of the inference cycle [55]. In the first version of the MIDAS prototype, the algorithm tried to compile the BN into the smallest AC. The compilation cost of the extreme size optimisation was proven later to be only partially rewarded by the added gain in inference speed. Hence, the compilation algorithm has been parametrised by adjusting the degree of size optimisation. This approach speeds the compilation phase without a significant increase of the inference time.

Figure 10 shows the inf4sat average inference time measured in a number of trials with random generated Bayesian Networks as well as its comparison with two classic inference techniques (Lazy propagation [54] and Gibbs [70]). In these trials the inf4sat compilation's degree is the most relaxed. The time improvement is measured in terms of orders of magnitude.

A schematic representation of the Bayesian Network for test scheduling as a Direct Acyclic Graph (DAG) is sketched in Fig. 12. The DAG nodes represent Boolean stochastic variables and the DAG edges the classic BN relationships *depends on* (the relationship direction is the reverse of the arrow direction, e.g. in Fig. 12 S depends on A1, A2, ...). These variables are classified in six categories:

- DSUT (S) – the S variable "represents" the DSUT and is the DAG bottom node; the intuitive meanings of the S variable values are (0 = faultless / 1 = faulty); S is instantiated to 1 (faulty) in the initialisation phase, allowing the inference process to begin with the hypothesis that there is at least one failure in the system;
- Actual Components (A) -– there is an A variable for each DSUT actual component; the intuitive meanings of the A variable values are (0 = faultless / 1 = faulty); in the initialisation phase the A variables can be affected with external values (user beliefs) in order to drive the inference;
- Issuing Interfaces (I) – there is an I variable for each component required interface and for each component provided (request/response) interface; the intuitive meanings of the I variable values are (0 = faultless / 1 = faulty); in the initialisation phase the I variables can be affected with external values (user beliefs) in order to drive the inference;
- Message Types (T) – there is a T variable for each message type whose instances can be issued by an issuing interface; the intuitive meanings of the T variable values are (0 = faultless / 1 = faulty); in the initialisation phase the T variables can be affected with external values (user beliefs) in order to drive the inference;
- Messages (M) – the M variables are the DAG top variables; there is a M variable for each message instance corresponding to an oracle in the Test Suite file; the intuitive meanings of the M variable values are (0 = pass / 1 = fail); in the initialisation phase, the M variables are affected with prior probabilities that are parametrisable; moreover, the M variables are observable: if the local verdict on the corresponding outcome is *pass* or *fail*, the corresponding

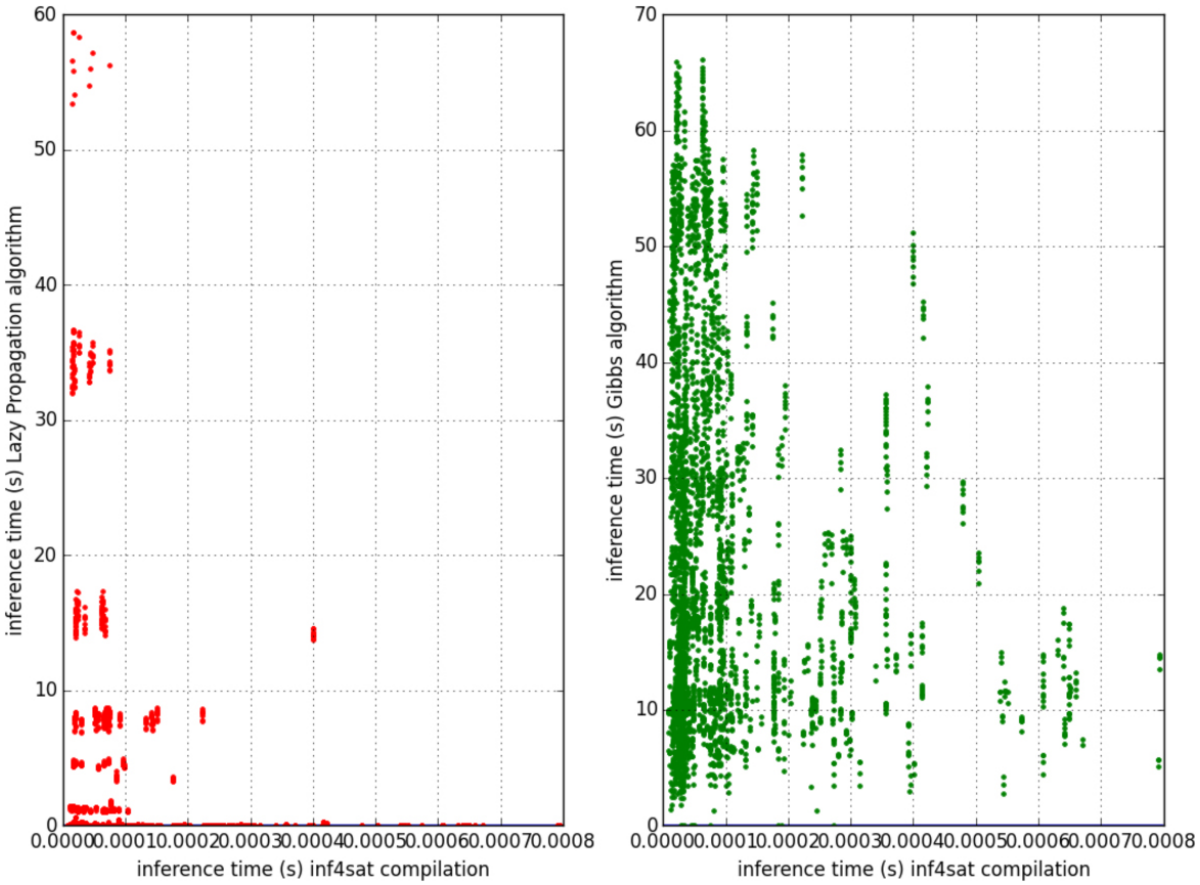

Fig. 10: Comparison of the average inference computation time between inf4sat and Lazy Propagation algorithm (left) and Gibbs sampling algorithm (right).

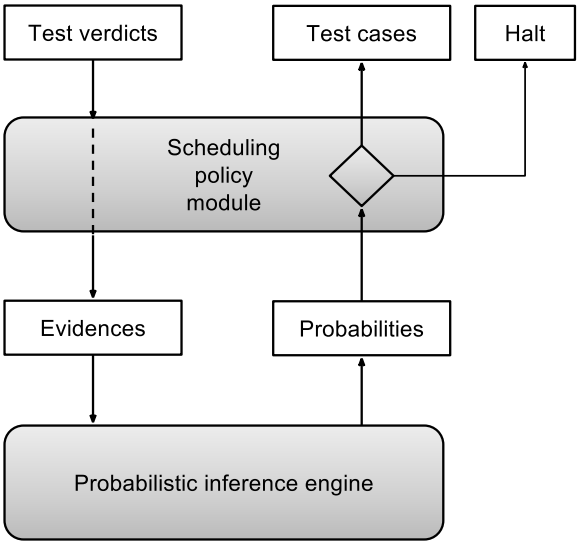

Fig. 11: Scheduling cycle.

M variable is instantiated with the corresponding value (respectively 0 or 1), otherwise – the verdict is *inconc*, *error* or *none* – the variable value is not changed;

– Test Cases (C) – there is a C variable for each test case of the Test Suite file; each C variable depends on the M variables corresponding to its oracles; the intuitive meanings of the C variable values are (0 = OK / 1 = KO); if the global compound verdict of an executed test case equals *pass* (all the local verdicts equal *pass*) the variable is set to 0 (OK), otherwise (at least one of the local verdicts does not equal *pass*) the variable is set to 1 (KO). The intuitive meaning of the C probability distribution is the failure discovery potential of the not yet executed test case, which grows with the KO probability.

At each test run, the local verdicts, when equal *pass* or *fail*, are inserted as evidences in the AC (by instantiating the corresponding M variables with the corresponding values). The subsequent inference re-calculates the BN variable values, in particular the values of the not yet observed C variables. At each test case run cycle, the scheduler re-calculates the *fitness* of the remaining test cases. The test case with the maximum fitness, or a test case randomly chosen among the test cases with the same maximum fitness, is selected for the next test run cycle. The current fitness of a test case is a function of the current probability distribution of the corresponding C stochastic variable that is defined by the scheduling policy. The two basic scheduling policies are:

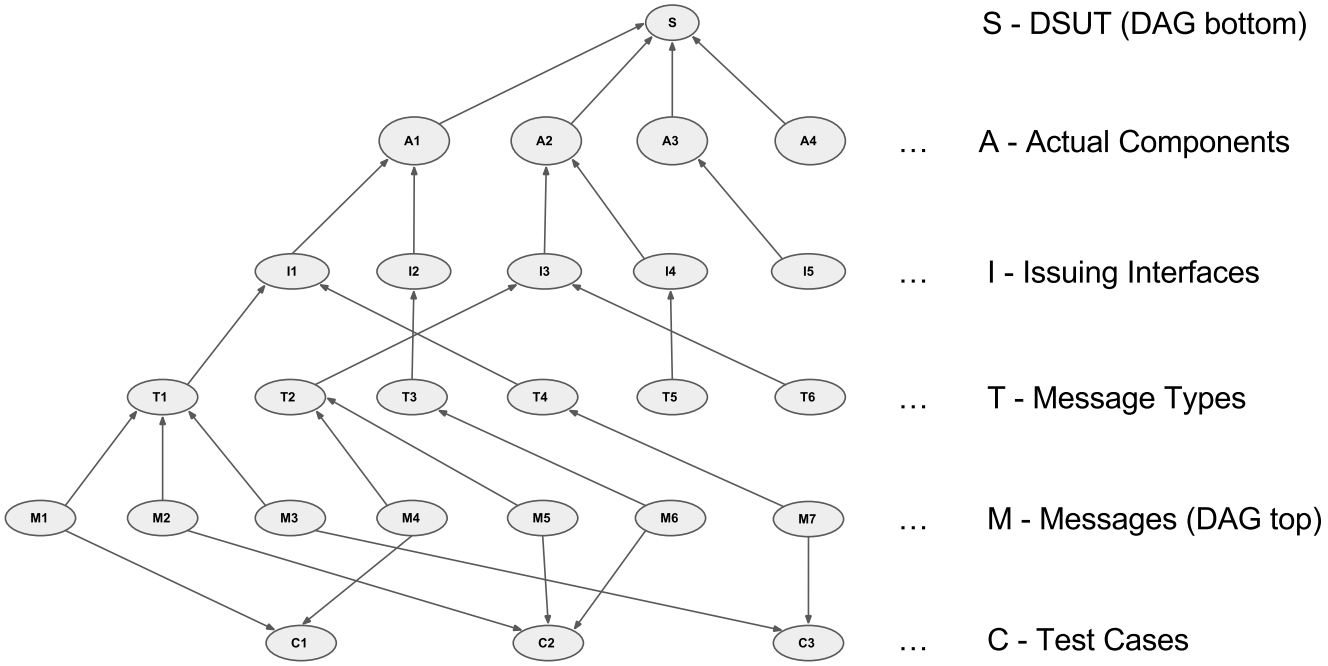

Fig. 12: Scheme of the Bayesian Network that drives the test scheduling.

- *max-entropy* policy – the max-entropy policy can be described roughly as "the least informed, the first". The test cases are ordered by the Shannon entropy [49] of the C variable, the maximum entropy being the nearest to fifty/fifty probability distribution. The max-entropy policy performs a breadth-first search in the DSUT topology that is driven by ignorance: the scheduler chooses the test case that targets the elements of the DSUT (actual components, issuing interfaces, message types) on whose "health" the test system has the minimum information;
- *max-potential* policy – the max-potential policy can be described roughly as "the max failure discovery potential, the first". The test cases are ordered by their failure discovery potential, i.e. the KO probability. In principle, the max-potential policy increases the fault detection rate. If, for example, in the past test run a message verdict equals *fail*, the probability of failure of the other messages of the same type grows and the scheduler will choose in all likelihood a test case that includes a message of the same type. This policy focuses on DSUT elements (actual components, issuing interfaces, message types) whose faulty probability is growing and looks for the maximum information about these elements that can be gathered by the execution of the test cases.

Other scheduling policies can be defined: for instance, a policy that mixes those mentioned above. Moreover, the inference cycle can be driven by the user beliefs that can be attributed to some A, I, T and M variables at the initialisation phase (replacing the default values). Every new session starts with the generation of a new inference engine (AC) extracted from a new BN. This new engine is untouched by previous executions. The user can influence the new session scheduling with a priori probabilities that are optionally evaluated taking into account the execution history.

The test session management is driven by the scheduler through the halting policy. There are four basic halting policies: (i) *n-KO-halt* policy – the scheduler stops the test session after the n-th C variable is set to 1 (KO); (ii) *n-OK-halt* policy – the scheduler stops the test session after the n-th C variable is set to 0 (OK); (iii) *entropy-threshold-halt* policy – the scheduler stops the test session when all the Shannon entropies of a selected group of A, I and T variables are lower than a given threshold; (iv) *no-halt* policy (default) – the scheduler stops the test session only when all the test cases have been executed.

The next steps in the development of the scheduler component will concern the improvement of the BN model and inference engine, allowing a more accurate and indicative portrait of the state beliefs over the DSUT components and the test cases. To do this, much can be adapted from existing work in the field of diagnostics and troubleshooting [35][76]. These developments will be strongly influenced by the extensive trials that will be conducted within the early adopter free trial programme (see Sect. 5.4), in which we will experience with our users the policy modules and use those results to tune current policies and eventually develop new ones driving new strategies. As of yet, it is really difficult to evaluate the multiple policies outside of multiple real-world cases scenarios.

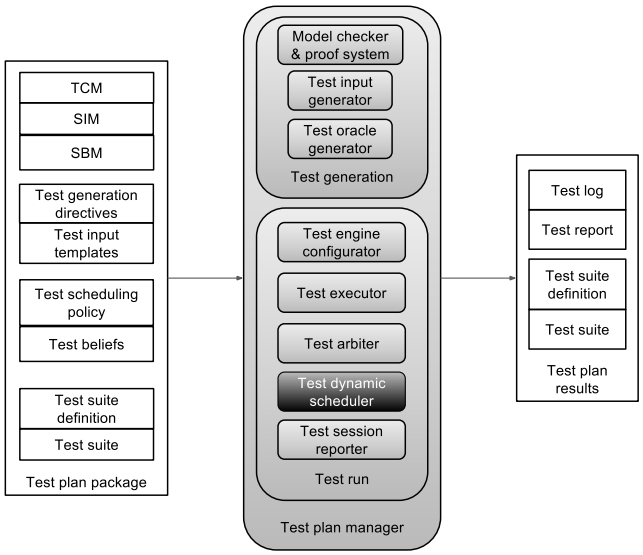

Fig. 13: Automated, evidence-based, reactive planning of test sessions.

3.5 Automated evidence-based reactive planning of test sessions

The proposed approach to automated functional testing uses the probabilistic graphical reasoning capabilities of the scheduler to drive not only the dynamic prioritisation of the set of existing test cases, but also to enable the focused generation of new test cases and their execution in a new test session. When invoking the evidence-based reactive planning test method, the user can: (i) either provide an initial (previously generated) TSD/TS file – for instance, the initial test suite can be utilised for change tests, regression tests or for smoke tests (preliminary tests aimed at revealing some failures severe enough to reject a candidate service build release); (ii) or invoke the evidence-based reactive planning test method without any test suite, with a generation policy that provokes a first generation – for instance, for progression tests (Fig. 13).

The dynamic scheduler working within the evidence-based reactive planning test method is augmented with a *generation policy module* (Fig. 14). Within the test session in progress, on the basis of evidences (verdicts) accumulated from the past test runs, the generation policy module recalculates at each cycle the degree of ignorance (Shannon entropy) on all the DSUT elements and, possibly, recommends the generation of new test cases whose execution would diminish this ignorance – for example by including test cases that trigger scenarios involving scarcely tested or untested message types, issuing interfaces, and actual components. If the recommendation is followed by the test session manager, the current test session is terminated and the test generator is invoked with appropriate generation directives. The newly generated test cases (the new TS file and, possibly, the new TSD) are taken into account with the re-initialisations of the scheduler and the executor, and a new scheduled test session is started with the new test suite.

In summary, the augmented scheduler of the evidence-based reactive planning test method drives the current test session with three request types (addressed to the test session manager):

1. Request the execution of the next test case in the current test session;
2. Request the termination of the current test session and supply a test session coverage report;
3. Request the termination of the current test session, supply the test session coverage report, and request the generation of a new test suite and the start of a new test session.

Even for service architectures with low complexity, the generation of a full coverage test suite can be considered a very difficult task and the result is certainly not scalable. Therefore, the approach that has been taken is to sequentially conduct test sessions whose search for failures is driven by the scheduler probabilistic reasoning. For this reason, the generation policy module is built to keep a broader view and a trace of the previous test sessions execution data. This module is initialised with the knowledge of all the message types, the issuing interfaces and the actual components that are described in the TCM and the SIM, not only of those involved in the test cases of the particular TS file of a specific test session. Thus, for each test session, the module keeps track of test execution information for each actual component, issuing interface and message type involved. It is able to supply the coverage report of the test session with respect to all of the testable DSUT elements. This coverage report is accompanied by a set of directives about testing message types, issuing interfaces and actual components of the DSUT to be tested. If, on the basis of the current SBM, the generator is able to satisfy to a certain degree the requested coverage, it generates the new test suite and the test session manager starts a new test session. Otherwise, the session is terminated and the user shall upgrade the SBM in order to run new testing sessions with specific coverage objectives on the basis of the previous test session coverage reports.

The evidence-based planning approach proposes a solution of the test coverage problem for a complex services architecture on the basis of troubleshooting heuristics driven by probabilistic reasoning. It pushes the test automation very far, but is still in an experimental phase. We need experience feedback on real-world case studies in order to tune, refine and improve the approach.

The general sequence diagram of the implementation of the evidence-based reactive planning is shown in Fig. 15. It shows four main phases.

The *initial request* from the front-end service follows an invocation by the end user through the MIDAS Web portal, or through direct program invocation using the front-end service API. The *initialisation* phase sets up the scheduler. The scheduler initiates the first generation of test cases by providing the generation directives to the test generator. A

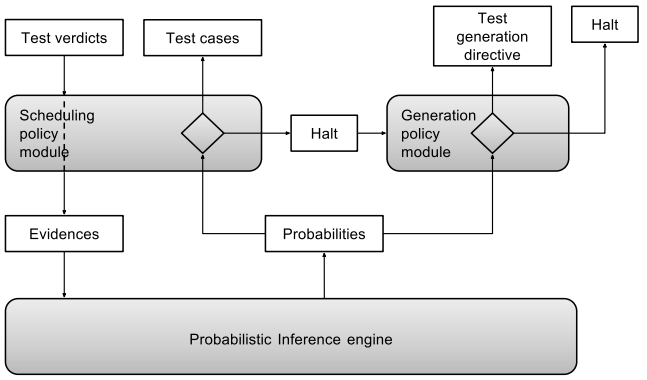

Fig. 14: Scheduling and planning cycle.

test scheduling request initiates the schedule/execute/arbitrate cycle. The scheduler returns the selected next test case to execute according to the scheduling policy and the state of the inference engine. The test executor/arbiter runs the test and returns the verdict that is notified to the scheduler through a subsequent test scheduling request. The scheduler decides whether it will return the next test case to execute, or request another generation of test cases. We call this cycle of interactions evidence-driven generation of test cases. The workflow ends when the scheduler returns an empty response to the test scheduling request – no more interesting case to test in the explored state space, or user-specified stopping conditions have been met. The *reporting* phase ends the workflow by reporting the collection of verdicts, the highlighted differences between expected and actual payloads, and the test coverage report. The report is displayed on a dashboard that is described in Sect. 4.

4 Prototype usage with real-world use cases

We have initial feedback of the usage of the functional test methods from the MIDAS pilots that have been deployed by the MIDAS partners Dedalus and ITAINNOVA. The MIDAS pilots are two real-world distributed and service-based architectures in the healthcare and logistics industries.

Dedalus has deployed standard HSSP services – RLUS, IXS, CTS2 – that had been developed and are operational in the context of different business and research projects. The Healthcare Pilot has made available a number of these services as test targets of the MIDAS prototype. In particular, these services are utilised in the Calabria Cephalalgic Network (CCN) distributed application depicted in Sect. 3, Fig. 3.

ITAINNOVA – Instituto Tecnológico de Aragón – is a public research and technological organisation (RTO) supported by the Industry and Innovation Department of the Government of Aragon. Its mission is to "help companies, technology leaders, institutions and anyone who shapes our society towards achieving a new future through innovation and technological development". The Aragon Region hosts an important Supply Chain Management industrial district and ITAINNOVA has defined a service architecture for supply chain management and logistics, has built a reference implementation of the architecture and is helping local companies and institutions to put in place services in the logistic domain and to test them with the MIDAS test methods [23].

In this paper we give some details of the usage of the MIDAS prototype for testing the Healthcare Pilot.

4.1 Testing the Healthcare Pilot

Dedalus utilised a custom-built framework for service unit testing that had already significantly shrunk the effort of manually producing and executing test cases and test suites. The major limitations of this custom-built framework have been identified as: (i) the test case overhead, (ii) the limitation to unit testing, (iii) the lack of planning and scheduling features, and (iv) reduced usability and manageability. The *test case overhead* issue relates to the necessity of creating a huge amount of test cases since the services to be tested (such as RLUS) are specified as generic and the payload structure varies according to the different instantiations of the service. In addition, the typical content transferred in the healthcare industry accommodates very complex data structures with several thousands of atomic data types.

Figure 16 depicts the test environment of the Dedalus Pilot. It consists of one virtual component and four actual components, each exposing a service as described in Sect. 3. Figure 17 shows a scenario that was tested on the presented architecture. In this scenario, the *virtual portal* looks for a patient record (*findIdentitiesByTraits*) that, if not found, is created (*createIdentityFromEntity*) and retrieved (*findIdentitiesByTraits*). The auxiliary services are used to trigger ancillary state operations such as resetState, setState, and getState.

The whole specification of the system is composed of the following documents:

- five TCM XML documents describing the components types, of length between 13 and at most 37 lines for the longest describing the virtual component. The contents of the TCM documents of the Virtual Portal and MPI-IXS components are shown in Appendices A.1 and A.2, respectively;
- one TCM XML document (47 lines) describing the topology of the test environment depicted in Fig. 16. Its content is shown in Appendix A.3;
- five SBM documents (PSM/SCXML) describing the external behaviour of each component (PSM), of length between 30 and 101 lines, the longest of which describes the virtual component which initiates the scenario depicted in Fig. 17. The contents of the SBM documents of the Virtual Portal and MPIIXS components are shown in Appendices B.1 and B.2, respectively;
- one SIM document (WSDL/XSD) for each service exposed by the components, along with the XSD files defining the data structures of the SOAP messages; there are 8 WSDL documents and 31 XSD documents;

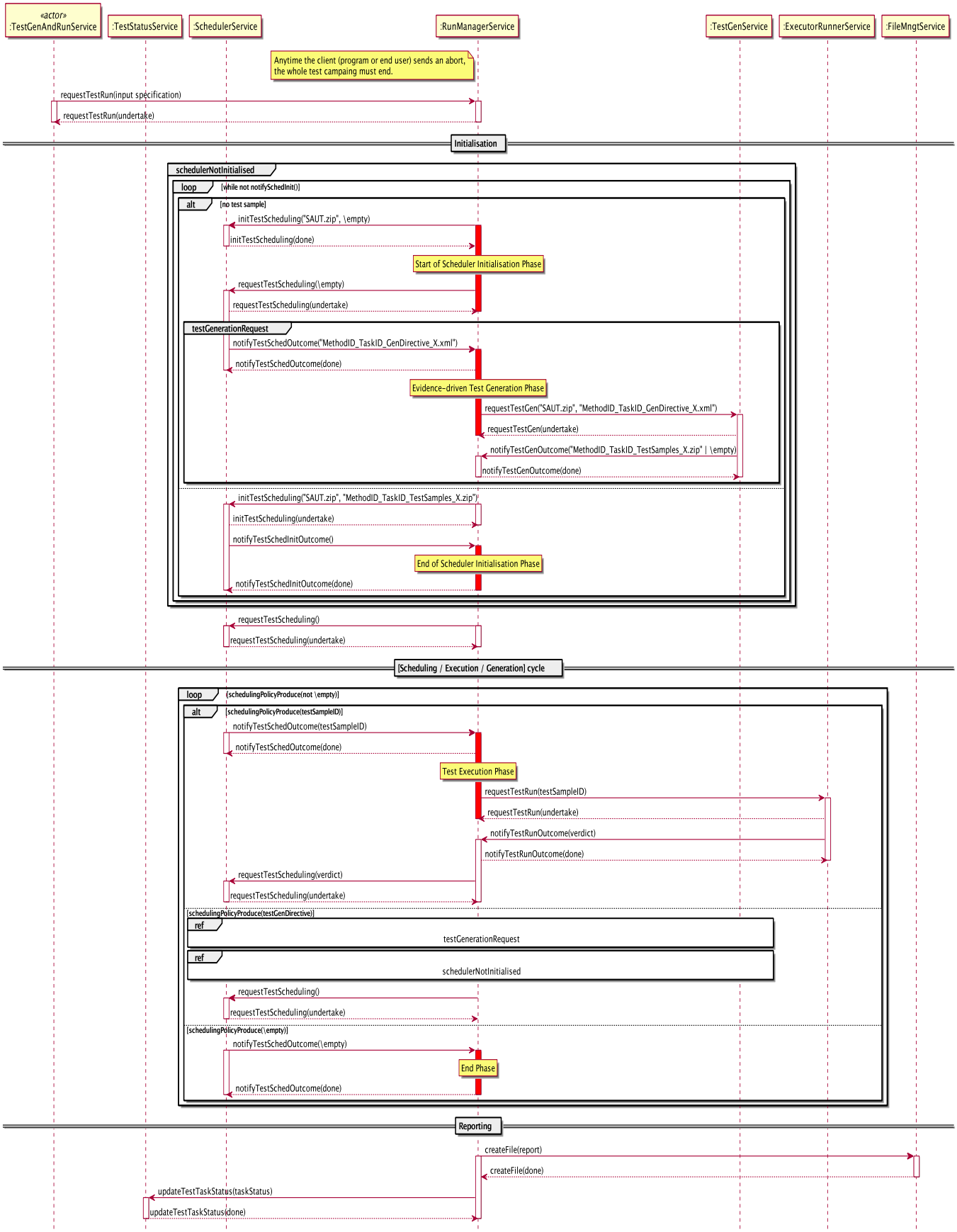

Fig. 15: Sequence diagram of the evidence-based reactive planning.

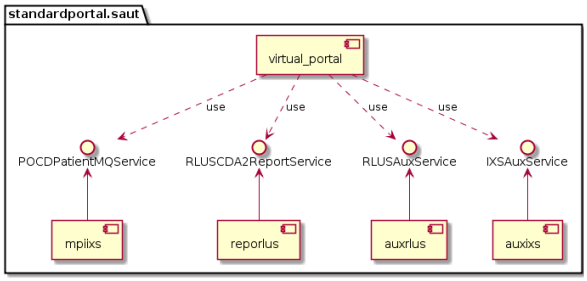

Fig. 16: TCM scheme of the Healthcare Pilot.

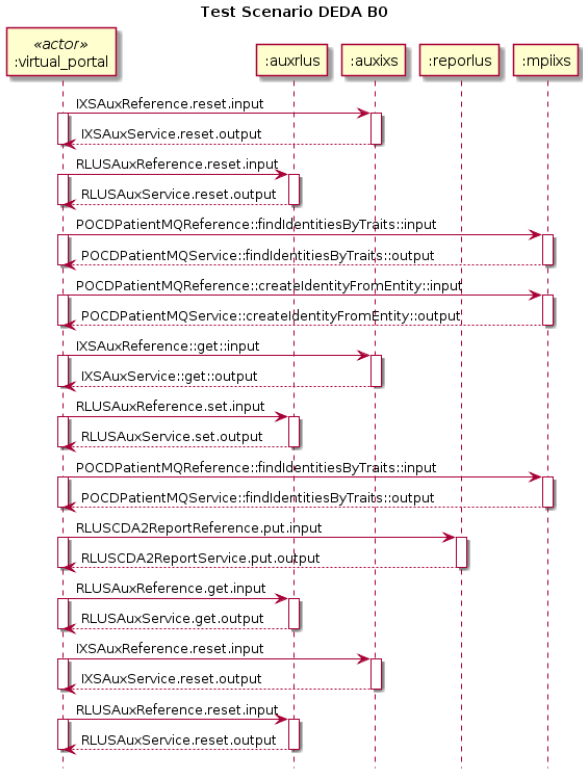

Fig. 17: Test scenario of the Healthcare Pilot.

- a few templates for the exchanged payloads; the length of the templates varies between 6 and 40 lines;
- a configuration file with parameters and policies (e.g. the stopping conditions for the functional testing such as the maximum number of failed tests). Its content is shown in Appendix C;
- an optional initial test generation directive file, that can be provided by the end user, but is not necessary since the scheduler will produce the directives as needed during the test session. It is mentioned here for the sake of completeness and to reference its content, shown in Appendix D.

The TCM and the SBM are specific to this test environment and campaign. The SIM, i.e. the collection of WSDL and XSD files, is exactly the same as that of the system in production. The templates have been generated with SoapUI [5], then configured with specific sets of values or with regular expressions. The usage of templates is vital for reducing the generation search space with the huge payloads that are typical of the healthcare industry.

Figure 18 shows an overview of the dashboard (in a browser) of the Healthcare Pilot at the end of a test session. In the following we will address the widgets by their (x, y) coordinates, with the origin of the axes at the top-left corner of the dashboard. For this execution, the stopping condition of at most one execution cycle has been met (green branch of the tree in widget(4, 3), and widget(4, 2)). Ten tests have passed (widget(5, 1)), one has failed (widget(4, 1)), and there are no inconclusive tests (widget(3, 1)). The tester can navigate through each of these three test verdict widgets to access more detailed information about each individual verdict, including the comparison between expected and actual payloads. The widget at (5, 3) displays the number of generated tests for each request from the scheduler. The widget at (1, 3) shows the coverage metric on the operations of the services that have been invoked in the scenario.

4.2 Impact of the MIDAS prototype on the Dedalus testing process

The automated generation of test cases brought by the MIDAS prototype reduces dramatically the effort that was formerly dedicated to test case handwriting. Moreover, the Dedalus custom-built test framework is able to support only service unit testing. End-to-end tests of service compositions with MIDAS require only the drafting of the appropriate TCM and SBM, which is a challenging task, but is accomplished once the models are relatively stable and the generation/run of test suites that evolves following the maintenance process (the cycle test/debug for progression tests, re-tests, regression tests) can be performed in an optimised manner by using features such as prior probabilities and beliefs and scheduling/generation policies.

With the potentially unlimited amount of large test cases that can be produced, the optimisation of the test sessions is a must. The Dedalus custom-built test framework does not have any support for test case prioritisation and optimisation of test case generation. The MIDAS intelligent scheduler and evidence-based planning facility propose solutions to the optimisation problem that are technically operational, potentially very powerful and whose usage has been experienced by the Dedalus users. We and, above all, our users shall constitute assets of experience and know-how in testing using advanced features such as the test generation based on formal methods, test prioritisation, scheduling and planning based on probabilistic graphical reasoning in different testing contexts.

Last but not least, with the Dedalus custom-built test framework, every change in the deployed DSUT (IP addresses, ports, URIs, parameterisations) requires a significant effort of reconfiguration by hand of each individual test case, practically preventing any routinisation of the test tasks and, consequently, any continuous integration approach. With the MIDAS prototype, the TCM, the SBM and the generated test suites are independent of the DSUT endpoint locations that

Fig. 18: Dashboard of the Healthcare Pilot test session.

are indicated as configuration parameters to be instantiated at test run time.

4.3 Drawbacks and limitations of the MIDAS prototype

Current known drawbacks of the MIDAS prototype concern manageability and usability issues. We plan to develop productivity tools such as wizards for the Test Configuration Model and the Service Behaviour Model, to improve the consoles and dashboards for test generation and execution and also to implement a Test Data Management System (including version management) that allows for storage and query of all models, policies, test suites, logs, reports and other artefacts (as well as all links between these artefacts) that are consumed and produced by each test generation and each test running session.

In terms of matching test outcomes with oracles, the current approach performs a complete matching over the corresponding SOAP messages (active oracles). However, it is also convenient to check only the value of specific fields in the messages, thus ignoring the other values (passive oracles). Furthermore, it is also important to allow conditional expressions in the outputs of the oracles to be computed against previous outputs in the same scenario in order to strengthen the computation of the verdict against false negatives with respect to generated data by the transfer functions in the PSMs. Passive oracles and conditional expressions are planned in the next releases.

We are also enhancing the treatment of timeouts. We will provide a collection of timer types to be instantiated in the PSMs in order to enable the user to specify: (i) *reply timers* – that are set on responses to requests that must be obtained in specified time intervals; (ii) *causality timers* – that are set on actions that have been caused by previous interactions and must be effected in specified time intervals; (iii) *delay timers* – that are set on actions that must not be taken before defined time intervals. These timers allow extensively testing time constrained service behaviour.

Quiescence [80] is not explicitly handled in the MIDAS prototype. Quiescence explicitly represents the fact that no output is provided in some system states. For example, an ATM's state after having delivered the requested amount of money should be quiescent: *it should not produce any other output until further input is given*, meaning it should not deliver the money twice. On the contrary, the state before delivering the money should not be quiescent. When a system under test has quiescent and non-quiescent states, the test system must decide whether the verdict should be pass, fail, inconclusive, error or none. In the MIDAS prototype, if a response to a request is emitted twice, it is silently ignored on the receiving end. Therefore, a state that must be quiescent is not explicitly checked for that property. A timeout is triggered when facing a quiescent state that should not be quiescent, producing a fail verdict. Quiescence testing will be provided in the next releases.

Controllability is not handled as such. We understand controllability as the ability of an external input to move the internal state of a service from an initial state to a final state in a finite time interval. In a grey-box testing approach, the observability of a service under test is limited to the exchange of messages with its interlocutors. Since its internal states are not observable, they are not controllable any more. We can

only test the view of the internal state that is provided as a resource that can be read, set and reset with the ancillary operations getState, setState and resetState anywhere in a test scenario (a PSM). The PSM author defines the resource structure and content and implements the ancillary operations on the service under test allowing testing post-conditions on the service operations defined not only on the type and content of the response, but also on the content of the read state resource. Conversely, controllability in the sense of whether a Web service has compatible partners [53] is not considered in this approach.

WS-BPEL [66] is a scripting language that allows quickly implementing *Web service orchestrators*, i.e. service components that centralise the control and data flow between the other components of a services architecture. Note that WS-BPEL scripts orchestrate only SOAP services. As such, white-box testing of WS-BPEL scripts can provide valuable feedback on the correctness of the end to end interaction scenarios between the components of an orchestrated Web services architecture. White-box test methods for WS-BPEL orchestrators are not currently implemented in the MIDAS prototype. An academic partner external to the MIDAS project has implemented a BPEL verification and white-box test method [26]. The MIDAS prototype is not only a SaaS, but also an open PaaS that enables researchers and practitioners to upload, register, install and try out on the MIDAS SaaS custom-built test methods and tools, and invoke them through the MIDAS generic API and GUI. We are working with our partner to include its WS-BPEL test method in the MIDAS test method catalogue.

5 Evaluation of the MIDAS prototype

Dedalus has carried out a study to assess the interest of the prototype after having run several test sessions. To analyse the impact of the MIDAS prototype on the efficacy and efficiency of the testing process, Dedalus has defined indicators in the following categories: (i) cost, (ii) performance, and (iii) quality.

5.1 Cost indicators

The cost indicators are related to: (i) the staff training effort, (ii) the cumulated efforts of both the building of the test campaigns and the test campaign result analysis. The effort is measured in terms of person/days spent by the R&D team to accomplish the involved tasks.

In terms of the training effort, it took approximately 6 days to train one software analyst to support the phases of test planning, design and analysis, and approximately 4 days to train one software tester/developer to put in place the test implementation, execution and reporting phases. Note that there does not yet exist packaged training courses, and the training process has been conducted mainly on-the-job with the remote support of a tutor. These costs are acceptable to Dedalus with respect to the expectation of improving the overall quality and reducing costs and delays of the software development process.

The cumulated effort spent for the test campaigns (building and analysis) amounts to 50 person/days. It is important to note that this cost has been measured starting from the availability of a running and stable version of the MIDAS prototype and does not take into account any iterations during the MIDAS project due to the deployment of alpha and beta versions of the prototype and the *beta testing* activity performed by the Dedalus team on the prototype itself. This measure of the costs in terms of person/days is objective and has supported a more general re-assessment of the costs of installing and deploying a software product of the complexity of the Healthcare Pilot, and of the expected revenue and profit, in a situation in which the MIDAS testing technology was available. This assessment can be completed only by considering the performance indicators as well.

5.2 Performance indicators

The considered key performance indicators (KPI) are:

- *Efficiency* – degree of optimization of the engineering process from the development up to the delivery and maintenance phases;
- *Effectiveness* – number of defects that can be discovered and the distribution of the discovery in the phases of the engineering cycle (the earlier in the cycle, the higher the effectiveness indicator).

The CCN project is a typical Dedalus customer project. In its study, Dedalus has compared its Economic Evaluation Sheet with other three customer provisions using similar technology (X1.V1, the Dedalus software product implementing the aforementioned services [13]) and the same economic quotation. In all of these scenarios the estimated costs for the configuration, distribution, installation and operation of the solution sum up to 50% of the total value of the provision itself and the corresponding effort amounts to about 550 person/days. More precisely, the activities that contribute to this cost are: (i) software tuning and configuration, (ii) integration with third party software (target of the MIDAS technology), (iii) internal testing, (iv) integration testing of the overall architecture (target of the MIDAS technology), (v) training, and (vi) delivery and operation. The integration testing activity evaluates to around 15% and the corresponding effort is about 80 person/days. When compared to the aforementioned cost of 50 person/days, the reduction of more than 35% for this activity is significant. Effective model driven and automated SOA testing approaches also have an impact on the costs for the integration with third party software, which is evaluated to 35% of the total cost with a corresponding effort of about 190 person/days. In a conservative assumption of reducing the costs of this phase by around 20% (corresponding to about 35 person/days), the total reduction of the cost of the overall provision would be around 13%.

On the other hand, it is important to stress that, in the evaluation of the MIDAS prototype, Dedalus has taken into account two critical aspects regarding the supply of its X1.V1 solutions:

- In the tenders, Dedalus conservatively increases, in the Economic Evaluation Sheets, the costs estimation for customisations, as well as the integration and testing of SOA solutions. The marketing department faces the usual dilemma: either increase the total fee of the provision (which for public tenders results in huge competitive disadvantages) or increase the company investment risk.
- Licensing and maintenance costs are always over-estimated because of the unpredictable cost of "in production" maintenance activities. With more effective testing, the maintenance costs could be stabilised and business models could significantly change by allowing innovative models like pay-per-use and pay-per-performance.

These two aspects are even more relevant in terms of scalability of the provision value and confidence in the return on investment. This is especially true for a platform like X1.V1 which Dedalus aims at becoming the corner-stone of complete, end-to-end SOA solutions whose volume of transactions, actors, users, accesses has the potential to increase exponentially and so does the possibility for revenues. For the Effectiveness KPI, the most important point is to find, as early as possible, the largest amount of bugs, inconsistencies, defects or issues in the software system to be deployed in the provisions. In order to evaluate the impact of MIDAS on this fundamental KPI, Dedalus has considered the following two indicators:

- number of generated tests for the testing campaign;
- number of revealed true defects (not false positives).

While in the past engineers have been able to manually write a few tens of tests for every interface using Dedalus custom-built testing framework, and considering that the focus is on very complex "service data models", the ability to increase the number of test cases by two orders of magnitude (around 5000) and achieving this automatically at a very early stage is an important advantage. During the development of the HSSP IXS and RLUS services – which preceded the MIDAS project by two years – a Dedalus partner of the CCN project conducted a manual black-box testing campaign concerning functionalities related to a few CRUD (Create, Read, Update, Delete) operations on resources and reported 10 testing issues. Out of those issues, seven were false positives and three were related to actual failures. The small amount of failures compared to the false positives can be explained by the fact that the most complex elements in the requests are query expressions that require rather good knowledge of the XML based semantic signifiers (CDA2, XDW), as well as of the XPath language [12]. These issues were related to an incorrect use of the expression language in the RLUS query operations. When using the MIDAS prototype for testing, of thousands of test cases, only one edge case which turned out to be a false positive was revealed (the result of a "hole" in the PSM specifications). This demonstrates that the proposed technology based on test massive generation and dynamic prioritisation allows for covering a high number of cases that can hardly be foreseen or even imagined with data payloads of this complexity, thus maximising testing efficacy and efficiency.

5.3 Quality indicators

The quality indicators taken into account are: (i) the confidence in software quality; (ii) the capability to reveal the majority of failures not later than the integration testing phase; (iii) the reduction of delays.

In terms of confidence in software quality, Dedalus estimates that the use of the MIDAS technology would increase the level of trustworthiness to "very confident" in about 90% of cases, considering that the "confident" level is necessary to embrace novel and less conservative business models.

In the past a great number of failures were discovered during the user acceptance phase or while the software was already in production. Dedalus estimates that the MIDAS technology has the firm potential to concentrate the discovery of defects in the early stages, and to significantly reduce the cost of late defect discovery.

In terms of reduction of delays, the critical factor is the ability to employ the technology from the start of the development cycle. Dedalus estimates that the MIDAS technology allows for the adoption of a continuous DevOps model. However, on this point experience feedback on new projects is still needed.

5.4 Licensing

As the MIDAS prototype is a SaaS, there is no specific licensing issue for the user as it would be for traditional software installed on premises. The end user basically has two access points to MIDAS: through a Web application accessible from a browser, or through APIs (today SOAP Web services). An industrial SaaS, **simplyTestify**, is being developed by Simple Engineering, a start-up partner of the MIDAS project that will run an early adopter free trial programme beginning the last quarter of 2016 [14], and the fully accessible commercial offer, on a pay-as-you-go basis including **free tier**, in the course of 2017.

6 Conclusion

The collection of functional test automation methods of the MIDAS prototype covers all the service functional test tasks, including the most "intelligent" and knowledge-intensive ones. These test methods bring solutions to tough functional test automation problems such as: (i) the configuration of the automated test execution system against large and complex services architectures, (ii) the test input generation based on formal methods and temporal logic, (iii) the test oracle

generation based on formal functional and behavioural specification of services, (iv) the intelligent dynamic scheduling of test cases, and (v) the intelligent, evidence-based, reactive planning of test sessions. Furthermore, the test automation methods are provided as services, allowing the MIDAS user to invoke them individually, to easily combine them in complex procedures and to routinise their usage in automated service integration and delivery processes. The MIDAS prototype has been utilised and evaluated by the MIDAS pilot partners on real-world use cases in the healthcare and logistics industries. New trials for assessing and refining advanced features such as model-based generation, dynamic scheduling and reactive planning for progression testing, retesting and regression testing are in progress on other real-world operational services architectures.

There are several improvements of the core technology that are already planned, but the most important evolution/extension is the testing of distributed systems that integrate connected objects (IoT). The challenge is to apply the MIDAS approach of "extreme" automation of functional testing to the new large-scale distributed architectures involving systems, services/APIs, mobile and stationary connected objects. We shall extend the portfolio of the observed interaction protocols, beyond SOAP, HTTP/XML, HTTP/JSON, to Web sockets [83], MQTT [67], CoAP [43] etc., and adapt the test generation, execution, scheduling and planning mechanisms of the current MIDAS prototype to these new distributed architectures. Moreover, the deployment of the MIDAS testing technology on cloud allows for facing the scalability challenge. In particular, we plan to enhance, for large scale hw/sw distributed systems, the probabilistic methods that have been proved in the past to be particularly well adapted to the testing and troubleshooting of complex systems [45][79] and are currently utilised for scheduling and planning within the MIDAS prototype.

7 Acknowledgement

This research has been conducted in the context of the MIDAS project (EC FP7 project number 318786) partially funded by the European Commission.

References

1. http://martinfowler.com/bliki/DeploymentPipeline.html.
2. https://jenkins-ci.org/.
3. http://www.midas-project.eu.
4. http://www.w3.org/TR/wsdl.
5. https://www.soapui.org/.
6. http://www.dedalus.eu/.
7. https://hssp.wikispaces.com/.
8. https://en.wikipedia.org/wiki/Service_Component_Architecture.
9. http://www.w3.org/TR/wsdl20/.
10. http://swagger.io.
11. http://www.w3.org/TR/scxml/.
12. https://www.w3.org/TR/xpath/.
13. http://www.dedalus.eu/x1v1.cfm?chg_lang=eng.
14. http://blog.simplytestify.com.
15. IBM Rational Service Tester for SOA Quality: Functional testing. http://www-03.ibm.com/software/products/fr/servicetest.
16. Parasoft: Api testing, service virtualisation, test environment and data management. https://www.parasoft.com.
17. Soasta: Load and performance testing. https://www.soasta.com/.
18. Tricentis: Risk-based testing, model-based test automation and test data management. http://www.tricentis.com.
19. Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.
20. Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Optimizing the Automatic Test Generation by SAT and SMT Solving for Boolean Expressions. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 388–391, Washington, DC, USA, 2011. IEEE Computer Society.
21. A. Askarunisa, K. A. J. Punitha, and A. M. Abirami. Black box test case prioritization techniques for semantic based composite web services using OWL-S. In *Recent Trends in Information Technology (ICRTIT), 2011 International Conference on*, pages 1215–1220. IEEE, June 2011.
22. B. Athira and P. Samuel. Web services regression test case prioritization. In *Computer Information Systems and Industrial Management Applications (CISIM), 2010 International Conference on*, pages 438–443. IEEE, October 2010.
23. M. A. Barcelona, L. García-Borgoñón, and G. López-Nicolás. Practical experiences in the usage of MIDAS in the logistics. *International Journal on Software Tools for Technology Transfer*, pages 1–15, 2016.
24. E.T. Barr, M. Harman, P. McMinn, M. Shahbaz, and Shin Yoo. The Oracle Problem in Software Testing: A Survey. *Software Engineering, IEEE Transactions on*, 41(5):507–525, May 2015.
25. Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. WS-TAXI: A WSDL-based Testing Tool for Web Services. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009*, pages 326–335. IEEE Computer Society, 2009.

26. Lina Bentakouk, Pascal Poizat, and Fatiha Zaïdi. Checking the Behavioral Conformance of Web Services with Symbolic Testing and an SMT Solver. In *TAP*, volume 6706 of *Lecture Notes in Computer Science*, pages 33–50. Springer, 2011.
27. Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. Testing and verification in service-oriented architecture: a survey. *Softw. Test. Verif. Reliab.*, 23(4):261–313, June 2013.
28. T. D. Cao, P. Felix, R. Castanet, and I. Berrada. Online Testing Framework for Web Services. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 363–372, April 2010.
29. Allen Chan. *Encyclopedia of Database Systems*, chapter Service Component Architecture (SCA), pages 2632–2633. Springer US, Boston, MA, 2009.
30. Lin Chen, Ziyuan Wang, Lei Xu, Hongmin Lu, and Baowen Xu. Test Case Prioritization for Web Service Regression Testing. In *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*, pages 173–178. IEEE, June 2010.
31. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001.
32. Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model Checking and the State Explosion Problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011.
33. D. Conforti, M. C. Groccia, B. Corasaniti, R. Guido, and R. Iannacchero. EHMTI-0172. Calabria Cephalalgic Network: innovative services and systems for the integrated clinical management of headache patients. *The Journal of Headache and Pain*, 15(Suppl 1):D12, 2014.
34. Luca Console and Mariagrazia Fugini. *WS-DIAMOND: An Approach to Web Services – DIAgnosability, MONitoring and Diagnosis*, volume 4 of *Information and Communication Technologies and the Knowledge Economy*. IOS Press, Amsterdam, October 2007.
35. Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, April 1987.
36. Rina Dechter. Bucket Elimination: a Unifying Framework for Processing Hard and Soft Constraints. *Constraints*, 2(1):51–55, April 1997.
37. ECMA International. Standard ECMA-262 - ECMAScript Language Specification 5.1 Edition. http://www.ecma-international.org/ecma-262/5.1/Ecma-262.pdf, June 2011.
38. S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test Case Prioritization: a Family of Empirical Studies. *Software Engineering, IEEE Transactions on*, 28(2):159–182, February 2002.
39. Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.
40. David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.
41. M. Haverbeke. *Eloquent JavaScript: A Modern Introduction to Programming*. No Starch Press Series. No Starch Press, 2011.
42. Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using Formal Specifications to Support Testing. *ACM Comput. Surv.*, 41(2):9:1–9:76, February 2009.
43. IETF. The Constrained Application Protocol (CoAP) - RFC 7252. https://tools.ietf.org/html/rfc7252, June 2014.
44. Seema Jehan, Ingo Pill, and Franz Wotawa. Functional SOA testing based on constraints. In *8th International Workshop on Automation of Software Test, AST 2013, San Francisco, CA, USA, May 18-19, 2013*, pages 33–39, 2013.
45. Finn V. Jensen, Uffe Kjærulff, Brian Kristiansen, Helge Langseth, Claus Skaanning, Jirí Vomlel, and Marta Vomlelová. The SACSO methodology for troubleshooting complex systems. *AI EDAM*, 15:321–333, September 2001.
46. Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark R. Tuttle, and Yuan Yu. Checking Cache-Coherence Protocols with TLA$^+$. *Formal Methods in System Design*, 22(2):125–131, 2003.
47. Lukasz Juszczyk, Hong Linh Truong, and Schahram Dustdar. GENESIS - A Framework for Automatic Generation and Steering of Testbeds of Complex Web Services. In *ICECCS*, pages 131–140. IEEE Computer Society, 2008.
48. Kathrin Kaschner and Niels Lohmann. Automatic Test Case Generation for Interacting Services. In *ICSOC Workshops*, volume 5472 of *Lecture Notes in Computer Science*, pages 66–78. Springer, 2008.
49. A. Ya. Khinchin. *Mathematical foundations of information theory*. Dover, 1957.
50. Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
51. Leslie Lamport. *Theoretical Aspects of Computing - ICTAC 2009: 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, chapter The PlusCal Algorithm Language, pages 36–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
52. Leonidas Lampropoulos and Konstantinos F. Sagonas. Automatic WSDL-guided Test Case Generation for PropEr Testing of Web Services. In *WWV*, volume 98 of *EPTCS*, pages 3–16, 2012.
53. Niels Lohmann and Karsten Wolf. Realizability Is Controllability. In *WS-FM*, volume 6194 of *Lecture Notes in Computer Science*, pages 110–127. Springer, 2009.
54. Anders L Madsen and Finn V Jensen. Lazy propagation: a junction tree inference algorithm based on lazy evaluation. *Artificial Intelligence*, 113(1):203–245, 1999.
55. Ariele-Paolo Maesano. *Bayesian dynamic scheduling for service composition testing*. Ph.D. Thesis, Université Pierre et Marie Curie - Paris VI, January 2015.
56. Philip Mayer and Daniel Lübke. Towards a BPEL unit testing framework. In *TAV-WEB*, pages 33–42. ACM, 2006.
57. Lijun Mei, W. K. Chan, T. H. Tse, and Robert G. Merkel. XML-manipulating test case prioritization for XML-manipulating services. *Journal of Systems and Software*, 84(4):603–619, April 2011.
58. Siavash Mirarab and Ladan Tahvildari. A Prioritization Approach for Software Test Cases Based on Bayesian Networks. In Matthew Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Software Engineering*, volume 4422 of *Lecture Notes in Computer Science*, pages 276–290. Springer Berlin / Heidelberg, 2007.
59. Akbar S. Namin and Mohan Sridharan. Bayesian reasoning for software testing. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 349–354, New York, NY, USA, 2010. ACM.
60. Chris Newcombe. Why Amazon Chose TLA +. In Yamine Aït Ameur and Klaus-Dieter Schewe, editors, *Abstract State Ma-*

chines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings, volume 8477 of *Lecture Notes in Computer Science*, pages 25–39. Springer, 2014.

61. Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4):66–73, March 2015.
62. E. Newcomer. *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. Independent technology guides. Addison-Wesley, 2002.
63. E. Newcomer and G. Lomow. *Understanding SOA with Web Services*. Independent technology guides. Addison-Wesley, 2005.
64. Sam Newman. *Building microservices : designing fine-grained systems*. O'Reilly, 2015.
65. C. D. Nguyen, A. Marchetto, and P. Tonella. Change Sensitivity Based Prioritization for Audit Testing of Webservice Compositions. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 357–365. IEEE, March 2011.
66. OASIS. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, April 2007.
67. OASIS. MQTT Version 3.1.1. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html, October 2014.
68. Object Management Group (OMG). Uml testing profile, version 1.2. `http://www.omg.org/spec/UTP/1.2/`.
69. Oracle. Automating Testing of SOA Composite Applications. http://bit.ly/2bhzr5F, 2016.
70. Simon Parsons. Probabilistic Graphical Models: Principles and Techniques by Daphne Koller and Nir Friedman, MIT Press, 1231 pp., ISBN 0-262-01319-3. *The Knowledge Engineering Review*, 26(02):237–238, 2011.
71. Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.
72. Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.
73. Charith Perera, Arkady B. Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Sensing as a Service Model for Smart Cities Supported by Internet of Things. *CoRR*, abs/1307.8198, 2013.
74. Mauro Pezzè and Cheng Zhang. Automated Test Oracles: A Survey. *Advances in Computers*, 95:1–48, 2015.
75. K. Rees, F. P. A. Coolen, M. Goldstein, and D. A. Wooff. Managing the uncertainties of software testing: a Bayesian approach. *Qual. Reliab. Engng. Int.*, 17(3):191–203, 2001.
76. Raymond Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32(1):57–95, 1987.
77. Philippe Schnoebelen. The Complexity of Temporal Logic Model Checking. In Philippe Balbiani, Nobu-Yuki Suzuki, Frank Wolter, and Michael Zakharyaschev, editors, *Advances in Modal Logic 4, papers from the fourth conference on "Advances in Modal logic," held in Toulouse (France) in October 2002*, pages 393–436. King's College Publications, 2002.
78. Ebrahim Shamsoddin-Motlagh. A survey of service oriented architecture systems testing. *arXiv preprint arXiv:1212.3248*, 2012.
79. Claus Skaanning, Finn V. Jensen, and Uffe Kjærulff. Printer Troubleshooting Using Bayesian Networks. In Rasiah Logananthara, Günther Palm, and Moonis Ali, editors, *Intelligent Problem Solving. Methodologies and Approaches*, volume 1821 of *Lecture Notes in Computer Science*, pages 367–380. Springer Berlin Heidelberg, 2000.
80. Gerjan Stokkink, Mark Timmer, and Mariëlle Stoelinga. Talking quiescence: a rigorous theory that supports parallel composition, action hiding and determinisation. In *MBT*, volume 80 of *EPTCS*, pages 73–87, 2012.
81. W. T. Tsai, Yinong Chen, R. Paul, H. Huang, Xinyu Zhou, and Xiao Wei. Adaptive testing, oracle generation, and test case ranking for Web services. In *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, volume 1, pages 101–106 Vol. 2. IEEE, July 2005.
82. Hongbing Wang, Qianzhao Zhou, and Yanqi Shi. Describing and Verifying Web Service Composition Using TLA Reasoning. In *2010 IEEE International Conference on Services Computing, SCC 2010, Miami, Florida, USA, July 5-10, 2010*, pages 234–241. IEEE Computer Society, 2010.
83. Web Hypertext Application Technology Working Group (WHATWG). Web sockets, in HTML Living Standard. https://html.spec.whatwg.org/multipage/comms.html#network, August 2016.
84. Erik Wilde and Cesare Pautasso, editors. *REST: From Research to Practice*. Springer, 2011.
85. D. A. Wooff, M. Goldstein, and F. P. A. Coolen. Bayesian graphical models for software testing. *Software Engineering, IEEE Transactions on*, 28(5):510–525, May 2002.
86. Franz Wotawa, Marco Schulz, Ingo Pill, Seema Jehan, Philipp Leitner, Waldemar Hummer, Stefan Schulte, Philipp Hoenisch, and Schahram Dustdar. Fifty Shades of Grey in SOA Testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*, pages 154–157. IEEE Computer Society, 2013.
87. Ching-Seh Wu and Yen-Ting Lee. Automatic SaaS test cases generation based on SOA in the cloud service. In *CloudCom*, pages 349–354. IEEE Computer Society, 2012.
88. S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

A TCM documents of Virtual Portal and MPIIXS components

This section shows the contents of the Test Configuration Model XML documents of the Virtual Portal and MPI-IXS components, and of the complete topology depicted in Fig. 16.

A.1 TCM document of Virtual Portal

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<sca:composite xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200903" xmlns:wsdli="http://www.w3.org/ns/wsdl-instance" name="portal.composite" targetNamespace="urn:dedalus:sca4saut:portal">

 <sca:component name="portal.component">

  <sca:reference name="POCDPatientMQReference">
   <sca:interface.wsdl interface="http://www.omg.org/spec/IXS/201212#wsdl.porttype(IXSMgmtAndQueryInterface)"/>
   <sca:binding.ws wsdlElement="http://www.omg.org/spec/IXS/201212#wsdl.port(POCDPatientMQService)" wsdli:wsdlLocation="http://www.omg.org/spec/IXS/201212 POCDPatientMQService/midasixsmqpocdpatient.wsdl"/>
  </sca:reference>

  <sca:reference name="RLUSCDA2ReportReference">
   <sca:interface.wsdl interface="urn:dedalus:rlus:cda2report#wsdl.porttype(RLUSPortType)"/>
   <sca:binding.ws wsdlElement="urn:dedalus:rlus:cda2report#wsdl.port(RLUSService)" wsdli:wsdlLocation="urn:dedalus:rlus:cda2report RLUSCDA2ReportService/midasrluscda2report.wsdl"/>
  </sca:reference>

  <sca:reference name="RLUSAuxReference">
   <sca:interface.wsdl
    interface="urn:dedalus:rlus:aux#wsdl.porttype(RLUSAuxInterface)" />
   <sca:binding.ws wsdlElement="urn:dedalus:rlus:aux#wsdl.port(RLUSService)"
    wsdli:wsdlLocation="urn:dedalus:rlus:aux RLUSAuxService/midasaux_rlus.wsdl" />
  </sca:reference>

  <sca:reference name="IXSAuxReference">
   <sca:interface.wsdl
    interface="http://www.omg.org/spec/IXS/201212#wsdl.porttype(IXSAuxInterface)" />
   <sca:binding.ws
    wsdlElement="http://www.omg.org/spec/IXS/201212#wsdl.port(IXSAuxService)"
    wsdli:wsdlLocation="http://www.omg.org/spec/IXS/201212 IXSAuxService/midasaux_ixs.wsdl" />
  </sca:reference>
 </sca:component>

 <sca:reference multiplicity="0..1" name="POCDPatientMQReference" promote="portal.component/POCDPatientMQReference"/>
 <sca:reference multiplicity="0..1" name="RLUSCDA2ReportReference" promote="portal.component/RLUSCDA2ReportReference"/>
 <sca:reference multiplicity="0..1" name="RLUSAuxReference" promote="portal.component/RLUSAuxReference" />
 <sca:reference multiplicity="0..1" name="IXSAuxReference" promote="portal.component/IXSAuxReference" />

</sca:composite>
```

A.2 TCM document of MPIIXS

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<sca:composite xmlns:s4s="http://www.midas-project.eu/xmlns/sca4saut" xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200903" xmlns:wsdli="http://www.w3.org/ns/wsdl-instance" name="mpi.ixs.composite" targetNamespace="urn:dedalus:sca4saut:mpiixs">

 <sca:component name="mpi.ixs.component">

  <sca:service name="POCDPatientAdminService">
   <sca:interface.wsdl interface="http://www.omg.org/spec/IXS/201212#wsdl.porttype(IXSAdminEditorInterface)"/>
   <sca:binding.ws wsdlElement="http://www.omg.org/spec/IXS/201212#wsdl.port(POCDPatientAdminService)" wsdli:wsdlLocation="http://www.omg.org/spec/IXS/201212 POCDPatientAdminService/midasixsadminpocdpatient.wsdl"/>
  </sca:service>

  <sca:service name="POCDPatientMQService">
   <sca:interface.wsdl interface="http://www.omg.org/spec/IXS/201212#wsdl.porttype(IXSMgmtAndQueryInterface)"/>
   <sca:binding.ws wsdlElement="http://www.omg.org/spec/IXS/201212#wsdl.port(POCDPatientMQService)" wsdli:wsdlLocation="http://www.omg.org/spec/IXS/201212 POCDPatientMQService/midasixsmqpocdpatient.wsdl"/>
  </sca:service>

  <sca:service name="IXSMetadataService">
   <sca:interface.wsdl interface="http://www.omg.org/spec/IXS/201212#wsdl.porttype(IXSMetaDataInterface)"/>
   <sca:binding.ws wsdlElement="http://www.omg.org/spec/IXS/201212#wsdl.port(IXSMetadata)" wsdli:wsdlLocation="http://www.omg.org/spec/IXS/201212 IXSMetadataService/midasixsmeta.wsdl"/>
  </sca:service>

 </sca:component>

 <sca:service name="POCDPatientMQService" promote="mpi.ixs.component/POCDPatientMQService"/>

</sca:composite>
```

A.3 TCM document of the complete topology

```
<?xml version="1.0" encoding="UTF-8" standalone="
    no"?>
<sca:composite
 xmlns:mpi="urn:dedalus:sca4saut:mpiixs"
 xmlns:auxixs="urn:dedalus:sca4saut:auxixs"
 xmlns:repo="urn:dedalus:sca4saut:repositoryrlus"
 xmlns:auxrlus="urn:dedalus:sca4saut:auxrlus"
 xmlns:portal="urn:dedalus:sca4saut:portal"
 xmlns:sca="http://docs.oasis-open.org/ns/opencsa/
     sca/200903"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:wsdli="http://www.w3.org/ns/wsdl-instance"
 xmlns:s4s="http://www.midas-project.eu/xmlns/
     sca4saut"
 name="standardportal.saut"
 targetNamespace="urn:dedalus:sca4saut:
     standardportal.saut">

 <sca:property name="saut" />

 <sca:component name="virtual_portal">
  <sca:property name="virtual" />
  <sca:property name="init" />
  <sca:implementation.composite name="portal:
      portal.composite" />
  <sca:reference name="POCDPatientMQReference"
      target="mpiixs/POCDPatientMQService" />
  <sca:reference name="RLUSCDA2ReportReference"
      target="reporlus/RLUSCDA2ReportService" />
  <sca:reference name="RLUSAuxReference" target="
      auxrlus/RLUSAuxService" />
  <sca:reference name="IXSAuxReference" target="
      auxixs/IXSAuxService" />
 </sca:component>

 <sca:component name="mpiixs">
  <sca:implementation.composite name="mpi:mpi.ixs
      .composite" />
  <sca:service name="POCDPatientMQService"/>
 </sca:component>

 <sca:component name="auxixs">
  <sca:implementation.composite name="auxixs:
      auxiliary.ixs.composite" />
  <sca:service name="IXSAuxService"/>
 </sca:component>

  <sca:component name="reporlus">
  <sca:implementation.composite name="repo:
      repository.rlus.composite" />
  <sca:service name="RLUSCDA2ReportService"/>
 </sca:component>

 <sca:component name="auxrlus">
  <sca:implementation.composite name="auxixs:
      auxiliary.rlus.composite" />
  <sca:service name="RLUSAuxService"/>
 </sca:component>

</sca:composite>
```

B SBM documents of Virtual Portal and MPIIXS components

This section shows the contents of the Service Behaviour Model XML documents of the the Virtual Portal and MPIIXS components, depicted in Fig. 17.

B.1 SBM document of Virtual Portal

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns:xsi="http://www.w3.org/2001/XMLSchema
    -instance" xmlns="http://www.w3.org/2005/07/
    scxml" xsi:schemaLocation="http://www.w3.org
    /2005/07/scxml http://www.w3.org/2011/04/SCXML
    /scxml.xsd" initial="initial" name="b0portal.
    psm" version="1.0">

 <datamodel>
  <data id="stimulus_payload" src="
      resetRequestIXS.xml"/>
  <data id="aux_get_request_ixs" src="
      getRequestIXS.xml"/>
  <data id="aux_reset_request_rlus" src="
      resetRequestRLUS.xml"/>
  <data id="aux_reset_request_ixs" src="
      resetRequestIXS.xml"/>
  <data id="aux_set_request_rlus" src="
      setRequestMarcoRLUS.xml"/>
  <data id="aux_get_request_rlus" src="
      getRequestRLUS.xml"/>
  <data id="rlus_put_request" src="
      putRequestMarco.xml"/>
  <data id="findIdentitiesByTraits_input" src="
      findIdentitiesByTraitsMarcoReq.xml"/>
  <data id="createIdentityFromEntity_input" src="
      createIdentityFromEntityMarcoReq.xml"/>
 </datamodel>

 <state id="initial">
  <transition target="ixs_reset">
   <log expr="'Initial stimulus for reset of IXS
       '"/>
   <send eventexpr="'IXSAuxReference::reset::
       input'" namelist="stimulus_payload"/>
  </transition>
 </state>

 <state id="ixs_reset">
  <transition target="rlus_reset" event="
      IXSAuxReference::reset::output">
   <log expr="'Reset of IXS'"/>
   <send eventexpr="'RLUSAuxReference::reset::
       input'" namelist="aux_reset_request_rlus
       "/>
  </transition>
 </state>

 <state id="rlus_reset">
  <transition target="patient_not_found" event="
      RLUSAuxReference::reset::output">
   <log expr="'Reset of RLUS'"/>
   <send eventexpr="'POCDPatientMQReference::
       findIdentitiesByTraits::input'" namelist="
       findIdentitiesByTraits_input"/>
  </transition>
 </state>

 <state id="patient_not_found">
  <transition target="patient_created" event="
      POCDPatientMQReference::
      findIdentitiesByTraits::output">
   <log expr="'Patient not found'"/>
   <send eventexpr="'POCDPatientMQReference::
       createIdentityFromEntity::input'" namelist
       ="createIdentityFromEntity_input"/>
  </transition>
 </state>

 <state id="patient_created">
```

```
  <transition target="ixs_checked" event="
      POCDPatientMQReference::
      createIdentityFromEntity::output">
    <log expr="'Patient created'"/>
    <send eventexpr="'IXSAuxReference::get::input
        '" namelist="aux_get_request_ixs"/>
  </transition>
</state>

<state id="ixs_checked">
  <transition target="rlus_initialized" event="
      IXSAuxReference::get::output">
    <log expr="'IXS checked'"/>
    <send eventexpr="'RLUSAuxReference::set::input
        '" namelist="aux_set_request_rlus"/>
  </transition>
</state>

<state id="rlus_initialized">
  <transition target="patient_found" event="
      RLUSAuxReference::set::output">
    <log expr="'RLUS initialized'"/>
    <send eventexpr="'POCDPatientMQReference::
        findIdentitiesByTraits::input'" namelist="
        findIdentitiesByTraits_input"/>
  </transition>
</state>

<state id="patient_found">
  <transition target="report_stored" event="
      POCDPatientMQReference::
      findIdentitiesByTraits::output">
    <log expr="'Patient found"/>
    <send eventexpr="'RLUSCDA2ReportReference::put
        ::input'" namelist="rlus_put_request"/>
  </transition>
</state>

<state id="report_stored">
  <transition target="rlus_checked" event="
      RLUSCDA2ReportReference::put::output">
    <log expr="'Report stored'"/>
    <send eventexpr="'RLUSAuxReference::get::input
        '" namelist="aux_get_request_rlus"/>
  </transition>
</state>

<state id="rlus_checked">
  <transition target="ixs_final_reset" event="
      RLUSAuxReference::get::output">
    <log expr="'RLUS checked'"/>
    <send eventexpr="'IXSAuxReference::reset::
        input'" namelist="stimulus_payload"/>
  </transition>
</state>

<state id="ixs_final_reset">
  <transition target="rlus_final_reset" event="
      IXSAuxReference::reset::output">
    <log expr="'Report stored, IXS reset.'"/>
    <send eventexpr="'RLUSAuxReference::reset::
        input'" namelist="aux_reset_request_rlus
        "/>
  </transition>
</state>

<state id="rlus_final_reset">
  <transition target="final" event="
      RLUSAuxReference::reset::output">
    <log expr="'RLUS finally reset'"/>
  </transition>
</state>

<final id="final"/>

</scxml>
```

B.2 SBM document of MPIIXS

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml initial="initial" name="mpi.ixs.psm"
    version="1.0"
 xmlns="http://www.w3.org/2005/07/scxml"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-
     instance"
 xsi:schemaLocation="http://www.w3.org/2005/07/
     scxml http://www.w3.org/2011/04/SCXML/scxml.
     xsd">

 <datamodel>
  <data id="findIdentitiesByTraits_input" src="
      findIdentitiesByTraitsMarcoReq.xml"/>
  <data id="
      findIdentitiesByTraits_output_NOTFOUND" src
      ="findIdentitiesByTraitsMarcoResp_NOTFOUND.
      xml"/>
  <data id="findIdentitiesByTraits_output" src="
      findIdentitiesByTraitsMarcoResp.xml"/>
  <data id="createIdentityFromEntity_input" src="
      createIdentityFromEntityMarcoReq.xml"/>
  <data id="createIdentityFromEntity_output" src
      ="createIdentityFromEntityMarcoResp.xml"/>
 </datamodel>

 <state id="initial">
  <transition event="POCDPatientMQService::
      findIdentitiesByTraits::input" target="
      creation">
       <log expr="'[IXS] First lookup of patient
            failed'" />
       <send eventexpr="'POCDPatientMQService::
           findIdentitiesByTraits::output'"
           namelist="
           findIdentitiesByTraits_output_NOTFOUND
           "/>
     </transition>
 </state>

 <state id="creation">
  <transition event="POCDPatientMQService::
      createIdentityFromEntity::input" target="
      find">
    <log expr="'[IXS] Creation of the patient
        record'"/>
    <send eventexpr="'POCDPatientMQService::
        createIdentityFromEntity::output'"
        namelist="createIdentityFromEntity_output
        "/>
  </transition>
 </state>

 <state id="find">
  <transition event="POCDPatientMQService::
      findIdentitiesByTraits::input" target="
      final">
    <log expr="'[IXS] Second lookup of patient
        succeeded'"/>
    <send eventexpr="'POCDPatientMQService::
        findIdentitiesByTraits::output'" namelist
        ="findIdentitiesByTraits_output"/>
  </transition>
 </state>
```

```
35
36  <final id="final"/>
37 </scxml>
```

C Configuration file

This section shows the content of the configuration file for the Healthcare Pilot experiment described in Sect. 4.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Service WSDLs are in subdirectories inside
    the wsdl directory, with their skeletons -->
<!-- Stimulus payloads are in the same directory
    as the WSDL of the called service -->
<!-- Each binding associates a unique psm file to
    a unique composite file -->
<!-- Resources files for the PSMs data models (
    SOAP message skeletons declared in data) may
    not have a path (only their file names). In
    this case, the preprocessor will bind those
    resources to their concrete path. -->

<sauts>
  <saut name="standardportal.saut.composite">
      <sautDirectory>sauts</sautDirectory>
    <psmDirectory>psms</psmDirectory>
    <compositeDirectory>atomicparticipants</
        compositeDirectory>
    <wsdlDirectory>services</wsdlDirectory>
    <binds>
      <bind psm="mpi.ixs.psm.scxml" composite="mpi.
          ixs.composite" />
      <bind psm="auxiliary.ixs.psm.scxml" composite
          ="auxiliary.ixs.composite" />
      <bind psm="auxiliary.rlus.psm.scxml" composite
          ="auxiliary.rlus.composite" />
      <bind psm="b0portal.psm.scxml" composite="
          portal.composite" />
          <bind psm="repository.rlus.psm.scxml"
              composite="repository.rlus.composite"
               />
    </binds>
    <stimuli>
      <stimulus id="stimulus_payload" action="
          stimulus_action == reset" payload="
          resetRequestIXS.xml" />
    </stimuli>
    <stopOn>
          <or>
            <nbMaxExec>0</nbMaxExec>
            <nbMaxFail>5</nbMaxFail>
          </or>
      </stopOn>
  </saut>
</sauts>
```

D Test Generation Directive

This section shows the content of a typical test generation directive, as can be issued by the dynamic scheduler or the end user upon first upload of the archive containing all the DSUT modelling artefacts onto the MIDAS platform, before invoking the functional test method.

```
<?xml version="1.0" encoding="UTF-8"?>
<TestGenSampleInputInfoset>
  <generationDirective id="
      MethodID_TaskID_GenDirective_0">
    <!-- Pluscal designates the use of model
        checking through TLA+ -->
    <genStrategy>Pluscal</genStrategy>
    <defaultTestCaseNumber>3</defaultTestCaseNumber
        >
    <!-- units allowed are : sec, min, hour, day-->
    <timeout unit="sec">600</timeout>
    <selectionDirective>
      <messagetypes>
      <!-- One or more -->
      <!-- sautID/participantID/sendingPortID/
          operationName/[input|output|fault/
          faultType] -->
      <messagetype>standardportal.saut/mpi.ixs.
          component/POCDPatientMQService/
          createIdentityFromEntity/input</
          messagetype>
    </messagetypes>
    </selectionDirective>
  </generationDirective>
</TestGenSampleInputInfoset>
```