Model and Inference Driven Automated testing of Services architectures

White Paper

MIDAS Requirements for automatically testable services and services architectures
With the spreading of the digital economy, a growing number of applications, systems and devices are connected and collaborate without human intermediation, allowing the automation of business processes that support daily activities and services\textsuperscript{1}. The functional and non-functional dependability and security of such distributed architectures become more and more a critical issue. Dependability and security (functional and non-functional) are firstly the outcome of sound engineering practices.

Service Oriented Architecture (SOA)\textsuperscript{2} is a design and implementation style that allows organizations to put in practice dynamic collaborations of autonomous, heterogeneous and loosely coupled digital systems as direct service exchanges, in order to achieve flexible, reliable and secure business process automation. SOA style is currently practiced following approaches that are situated in a range between two extremities: (i) “weak” interface-based SOA and (ii) “strong” contract-based SOA\textsuperscript{3}.

---


\textsuperscript{2} OASIS Reference Model for Service Oriented Architecture 1.0. OASIS Standard, 12 October 2006. Organization for the Advancement of Structured Information Standards. URL http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

The “weak” interface-based SOA style is based merely on the design of an Application Programming Interface (API) as a primary access device to some system functionalities and on the management of this interface as a “first-class” artifact that is separate from the system internal implementation.

In the contract-based “strong” SOA style, beyond the separation between interface and implementation and the hiding of systems’ internals that are a must in any case, a service is represented by an artifact referred to as a service contract that incorporates the formal definitions not only of the service interface, but also of the service function and the external behaviors of the parties, including security and performance aspects. The service contract acts as an agreement between the parties and as a collection of requirements and a specification for the implementers.

The spread of interface-based SOA is revealed by the impressive growth of the “API economy”\(^4\). APIs are a key growth driver for hundreds of companies across a wide range of industry sectors and are going to be a primary access channel to technology-driven products and services. The spread of contract-based SOA is slower than one of interface-based SOA, but follows the diffusion of the model-driven engineering approach and concerns today a growing number of distributed system architectures in critical business sectors, where is often linked to standardization initiatives\(^5\). In several services architecture, the contract-based style often gets along with the interface-based one.

In a sense, SOA engineering, as all other engineering activities, is always model-based. We always create a mental model of the reality that we have either to design and build (prescriptive model) or to analyze and assess (descriptive model). The model always exists; the only option is about its form: it may be mental - existing only in the head of the designer, analyst ... - or explicit. An explicit model is potentially sharable between humans; an explicit and formal model of a system may be mechanically transformed until automatic generation of parts of the system. This is the objective of the OMG Model Driven Architecture


\(^5\) Healthcare Services Specification Program (HSSP) - [http://hssp.wikispaces.com/](http://hssp.wikispaces.com/)
initiative\textsuperscript{6}, and the goal of a more general trend named MDE (Model Driven Engineering).

The SOA approach is characterized by a sharp separation between the \textit{functional} (in a broad sense), black-box model and the \textit{constructional}, white-box model of a system. With respect to SOA, the inescapable limit of the MDA approach is that the white-box model of a system cannot be mechanically derived by its black-box model, even by the most detailed one. The black-box model (service model) acts as an agreement (service contract) between the supplier of the functionality (the service provider) and its user (the service consumer), while the white-box models are private and hidden. This is not an abstract issue: a published API is an agreement between the service provider and the consumers as the only means that allows interacting with the provider to coordinate the service provision/consumption.

How can the service consumer improve reasonably her/his confidence in the compliance of the actual service provision with exigencies and constraints stated in the service contract? The only answer is: \textit{by testing}. Beyond service compliance, how can the service consumer improve reasonably her/his confidence that the provider is not vulnerable to malicious attacks that can jeopardize the resources handled in behalf of the consumer? The answer is still: \textit{by testing}. Because of implementations are mutually hidden – even if they are not, it might be too complex and expensive to assess each other implementations by the deep analysis of the internals and white-box testing – \textit{black-box testing} is the only means available to businesses to improve their trust in their partners’ service provisions.

But SOA testing has the paradoxical trait that the same peculiarities that make it \textit{necessary} make also it \textit{hard} due to: lack of observability of the involved systems; lack of trust in the employed engineering methods; lack of direct control of the implementation lifecycles; late binding of systems; fundamental uncertainty of the test verdicts; organizational complexity; elastic demand of computational resources; increasing scale factor of the services architectures; high costs and, last but not least, questionable efficacy of humans in performing manually a more and more hard and complex but boring and low rewarding activity such as testing of large distributed architectures.

\textsuperscript{6} http://www.omg.org/mda/
In effects, hand-writing of test cases, manual configuration of test environments, manual scheduling of test runs and eyeball assessment of test outcomes are not only labor intensive and difficult to put in practice, but are also the least effective solution of the SOA testing problem, that cannot be overcome by mere methodological recommendations on fundamentally human-based engineering practices.

The solution of the SOA testing problem can be brought only by a disruptive innovation that drastically simplifies and routinizes the testing task by implementing and offering an automated, effective, accessible and affordable testing facility.

The goal of the MIDAS project is to design and build a test automation facility that targets SOA implementations. The MIDAS functionalities shall be available on demand, i.e. on self-provisioning, pay-per-use, elastic basis. The MIDAS facility shall be a multi-tenancy SaaS (Software as a Service) deployed on a cloud infrastructure and accessible in the Internet. The MIDAS testing approach is non-intrusive on the SUT (System Under Test): the SUT is deployed on its environment (on premise and/or on cloud) and the MIDAS facility interacts with it using the services that it publishes.

The targets of the MIDAS facility are both “weak” interface-based and “strong” contract-based SOAs. The MIDAS facility is intended to provide automation of the “core” testing activities: test generation, scheduling, execution and arbitration.

MIDAS test automation is based on models. Like SOA engineering, SOA testing, and, in general, testing is always model-based. We always create a mental model of the system behavior in order to test it. As for engineering, the model always exists; the only option is about its form: it may be mental - existing only in the head of the tester - or explicit. An explicit model is potentially sharable between humans; an explicit and formal model of a system can be mechanically transformed until the automatic generation of test cases and oracles and the automatic configuration of the test running environment.

The underlying idea is that the testing activity shall shift from test case hand-writing, manual configuration of test environments, manual scheduling of test runs and eyeball assessment of test outcomes – all these activities being conducted by professional testers - to model authoring by designers and architects. The burden of test generation, scheduling, execution and arbitration, until the production of the test
report, is given up to the MIDAS facility. The involved models are on one side service and system models and on the other side models of the test goals, means and courses of actions. The general idea is that the deep testing knowledge of professional testers and of the research community can be embedded in the implemented automated testing methods.

What kind of testing methods, approaches and practices will be supplied by the MIDAS on demand automated test facility?

The MIDAS project shall put into operation a substantial collection of functional (unit testing, choreography, orchestration and composition testing) and security (fuzz, security policy compliance) testing methods. Moreover, the automated scheduling and even the dynamic automated test generation shall be managed by probabilistic (Bayesian) inference methods. Furthermore, the new promising usage-based testing approach that is a testing meta-method that considers the usage of the system in the field (in operation) a source of interesting data, information and knowledge (Markov models), shall be carried out. The idea is to use this knowledge, automatically concentrated in the usage profile by intelligent processing of usage data, for driving the strategy and planning of functional test. The MIDAS facility shall support usage observation on the system in the field with facilities that allow the generation and download of the usage observation software and the upload and arrangement of usage journals.

The MIDAS portfolio of automated test methods, practices and approaches is not closed. From the point of view of the test method designers and developers, the MIDAS on demand automated test facility is an open platform. MIDAS shall implement the concept of test scheme, which is an implemented testing method able to perform automated test generation, scheduling, execution and arbitration.

Hence, according to the already canonical SaaS approach, there are two categories of MIDAS “users”: the end user and the test scheme developer. The end user acts automated testing by (i) supplying models, (ii) deploying accessible systems under test (SUTs) and (iii) invoking against the SUTs the appropriate test schemes that s/he offered by the evolving MIDAS portfolio. The test scheme developer designs and builds test schemes in a format that is MIDAS-compatible, and uploads them as

---

7 The compatibility requirements and recommendations for test scheme developers are not in the scope of this document.
plug-ins on the MIDAS facility in a strictly controlled way. Test schemes are organized, built and formatted in such a way that not only they can be integrated in the MIDAS facility, but also reuse preexisting independent resources such as SUT models and test configuration models. The SUT model and the test configuration model that are already available on MIDAS are reusable by the test schemes. Eventually, the end-user shall provide only the specific information – supplied in the form of test scheme related models - needed by the testing method the invoked test scheme implements.

The MIDAS core functionalities for both end users and test scheme developers shall be presented through APIs. This access modality allows seamless integration with Integrated Development Environments (IDEs) and Application Lifecycle Management (ALM) platforms - for instance, the programmed invocation of automated non-regression test campaigns at specific milestones of a software engineering cycle - and, obviously, new engineering service compositions that are unforeseen at that time.

Not all services architectures being equally testable, in general and in particular with MIDAS, we are obliged to state a limited collection of requirements and recommendations that respectively must and should be fulfilled in order to employ the MIDAS facility for SOA testing.

MIDAS test compatibility requirements must be fulfilled by any SUT that is sought to be a MIDAS testing target. With respect to MIDAS compatibility requirements, our work has been driven by two methodological principles: (i) minimize the number of requirements and their enactment burden for the user; (ii) formulate only requirements whose fulfillment improves the general dependability, security, interoperability, conformance with standards and, last but not least, auditability and testability as a generally accepted criteria - non-specific to MIDAS - of single-node and multi-node services architectures.

MIDAS compatibility requirements are independent from any test scheme. Conversely, MIDAS compatibility recommendations are classified as general vs. test scheme specific. General recommendations are related to the use of optional features of the MIDAS facility that improve the testing process but are independent from specific test schemes. The use of these features is optional, but the fulfillment of the related recommendation is a prerequisite for the use of them.

Test scheme specific recommendations have to be adopted only if the user wants to invoke the specific test scheme. The invocation of a
specific test scheme is optional for the user, but, whether the user wants to invoke it, the fulfillment of the related recommendations will be a prerequisite. Anyway, the MIDAS facility proposes some basic test schemes that will be operational without requesting the satisfaction of any specific recommendation - only the general compatibility requirements must be fulfilled.

All compatibility requirements and recommendations can be classified in two categories: (i) those that bear on models that are needed, in general or related to specific test schemes, to drive the automated test generation, scheduling, execution and arbitration, (ii) those that bear on the SUT configuration and deployment, in order to consent basic and enhanced binding, connection and interaction of the MIDAS facility with the SUT.

General requirements on models concern “architectural” models, i.e. system models and test configuration models. The system model is descriptive model of the MIDAS target services architecture. The test configuration model is a prescriptive model that is used by the model-driven automatic generation of the test execution system.

The system model includes a service model and a SUT model. The service model is a class model, while the SUT model is an instance model of a concrete, physically deployed SUT.

According to the MDA approach, we distinguish between the service platform independent model (service PIM) and the service platform specific model (service PSM), which is a model on a specific technical interoperability platform such as SOAP or REST.

The service PIM is a standard SoaML model, compliant with the OMG Service oriented architecture Modeling Language (SoaML) Specification. The service PIM is an abstract, disembodied model of the services implemented by the SUT, which uses two kinds of stereotypes: the Service Contract and the Participant. The Service Contract describes the service abstract specification, disembodied from specific provider and consumer systems, whereas the Participant describe a class of abstract systems that realizes a number of service roles (described within the Service Contract).

The MIDAS facility needs the availability of the essential service PIM, which includes only the minimal information that is necessary to characterize and classify the SUT nodes and ports (and also the test component elements of the test configuration model - see below). A
service PIM may include more enhanced information about protocols and choreographies, which have been produced by a model-driven, contract-first SOA engineering cycle and may be used by specific test schemes.

The service PSM is the service implementable model (WSDL, WADL ...) on the SUT interoperability platforms (SOAP, REST ...) that allows the configuration of the needed connections between the MIDAS facility and the SUT. The accuracy and consistency of the service model (PIM and PSM) is crucial.

The SUT model is a model of a concrete system but is platform independent: it is a UML Deployment model that describes the topology of the SUT and the locations of its accessible nodes/ports. The elements of the SUT model are classified by the elements of the service model. The deployed SUT node/ports (and their locations) described by the SUT are made accessible by the MIDAS facility by means of this description. Deployed node/ports that are not described by the SUT model are not accessible by the MIDAS facility. The accuracy of the SUT model is crucial for sound testing practices.

On the basis of the SUT model, the test configuration model (TC model) prescribes the architectural configuration of a test scenario in a manner that is independent by the test scheme that utilizes it. It indicates: (i) the SUT accessible nodes that shall be the targets of stimuli, responses and observations in the course of the test run, (ii) the SUT accessible places - ports and communications paths - where the stimuli, responses and observations shall be acted and (iii) the connections that shall be established between SUT and the MIDAS test execution environment.

According to the MDA approach, we distinguish between the test configuration platform independent model (TC PIM) and the test configuration platform specific model (TC PSM), which is a model on a specific test platform and environment, such as the TTCN-3 platform\(^8\).

The TC PIM is a UML Component model. TC PIM Components are stereotyped as Proxy, Emulator and Interceptor. Proxies prescribe components that represent the SUT node/ports being the targets of stimuli and responses by the MIDAS facility. Emulators prescribe components that emulate SUT nodes and virtual environment nodes – nodes that are not in the composition of the SUT and represent human or

\(^8\) http://www.ttcn-3.org/
artificial actors that interact with the system. Interceptors are able to place themselves virtually on a communication path between two SUTs (in fact between the representative Proxies) and interact with them. Emulators and Interceptors are architectural placeholders whose testing operational semantics is determined by the test schemes that utilize them.

General compatibility requirements on the SUT bear on the interoperability platforms that are compatible with the MIDAS facility and the availability for a SUT of an initialization and recovery procedure that can be invoked by the MIDAS facility. General compatibility recommendations on the SUT suggest the implementation of ancillary services and tools that make easier the SUT deployment, the check of the models’ accuracy and of the connections.

The other compatibility recommendations are related to specific test schemes, in the fields of functional, security and usage-based testing.

Functional testing, practiced as unit (single node) and integration (multi-node cooperation) testing, can be enhanced by the availability of service model elements that specify, beyond the service interface, the function (what the provider does in behalf of the consumer) and the behavior (how the parties interact to coordinate the service provision/consumption). Function specifications that are compatible with the MIDAS facility are grounded on contract-based design: a function is specified by the operation signature and its pre/post-conditions\(^9\). Function models allow the automated generation of tests and oracles.

Stateful service providers, i.e. providers that change durably the state of the resources they handle in behalf of their consumers, because of the SOA approach that forbids the direct inquiry of the aforementioned states, can be tested more effectively only by cross-checking. Cross-checking is put in practice by retrieving internal states through basic transparency services based on international standards that should be implemented by the SUT and matching this information with the SUT responses.

---

\(^9\) The MIDAS compatible formulation of operation signatures and pre/post-conditions is OCL (OMG Object Constraint Language (OCL) Version 2.3.1. formal/2012-01-01. Object Management Group. URL http://www.omg.org/spec/OCL/2.3.1/).
Choreography testing can be applied to end-to-end service exchanges on multi-node services architectures. Composition testing enhances choreography testing with the help, for each involved node, of the transfer function\textsuperscript{10}, i.e. the formal specification for the involved nodes of the correspondence between the input stimuli and the output interactions that are service composition effects of the stimuli.

Compatibility recommendations for functional testing, on models and on the SUT, are cumulative: the availability of more models (protocol, function, choreography and composition) and of more ancillary services on the SUT (state re/initialization, transparency services) allows more and more enhanced testing methods.

Security testing can be classified into two categories: (i) security-policy compliance testing and (ii) vulnerability testing. Security policies are included in the service contract and security-policy compliance testing is close to functional compliance testing. Vulnerability testing aims at seeking faults and weaknesses (the latter being not necessarily faults from the functional point of view) that have a security impact.

In order to look for such vulnerabilities, the testing perspective is moved from the system specifications to the attacker behavior. The main aspect of security testing is to stimulate the SUT with inputs that reveal vulnerabilities. Mostly, such inputs are invalid in the sense of the specification. Therefore, in contrast to functional testing, security testing is mostly negative testing and may be based on misuse cases instead of use cases.

We are going to develop two kinds of fuzzing approaches: data fuzzing and behavioral fuzzing. With data fuzzing the SUT is exercised with invalid input data while behavioral fuzzing consists of submitting invalid message sequences to the SUT.

In respect to SOAP-based web services, Representational State Transfer Interfaces (REST) and JavaScript Object Notation (JSON) are getting more popularity in the API economy. Data exchange via JSON is, in contrast to XML documents, less restrictive because of the lack of a document format description that prevents the validation of the document against this description. Hence, the general availability of even

\textsuperscript{10} Also for the transfer function the MIDAS-compatible formulation is OCL.
simple data fuzzing of JSON documents can be relevant and effective for testing purposes.

The compatibility recommendations on models for security testing concern the availability of interaction protocol models, security policy models, encryption/decryption and signature/validation algorithms and keys utilized by the SUT. The compatibility recommendations on the SUT are the same as those for functional testing.

SOA usage-based testing is a new promising research and engineering approach about SOA testing. The first intent is to focus on highly used functions of the service and highly usual end-to-end service exchanges in services architectures. Test cases are generated such that the testing effort focuses on the highly used parts of the SUT. As a by-product of this approach, it is possible to determine the reliability (the rate of failures given a usage scenario) of the SUT with respect to the current usage and utilize this measure as a criterion within acceptance testing. Conversely, focusing on lowly used paths of interaction within the SUT is interesting for augmenting the functional test coverage. Usage-based testing is intended to support the functional testing activities of the MIDAS facility.

Usage-based testing is grounded on the concept of usage profiles. The usage profile describes the usage of a system in stochastic terms, such as the probability of the next interaction. Usage profile models are built from information extracted from usage journals that are issued from usage observation on the system in the field (SIF). The MIDAS facility shall support the user in producing easy-to-put-in-place mechanisms (observer software) that are able to perform usage observation with minimum overhead. These software components, which are placed by the SIF administrator on the chosen SIF usage observation points, are able to provide MIDAS-compatible usage journals, which shall be collected and uploaded on the MIDAS facility.

The crucial recommendation is that the usage data observation mechanisms shall engender usage observation data that should be not only compatible with the MIDAS facility but also accurate, which means that they should represent faithfully the actual usage of the SIF. The fulfillment of this recommendation is essential for sound usage profile modeling.

The first recipients of this document are the MIDAS early adopters, such as the MIDAS project partners that are in charge of the pilots
(Healthcare Generic Services pilot / Supply Chain Management pilot). This document will constitute the support that allows them to start by trying the fulfillment of the MIDAS compatibility requirements and recommendations on their SUTs and SIFs\textsuperscript{11}. Relatively to these points, the feedback from the MIDAS pilots’ experience will be precious all along the project. If some of the MIDAS compatibility requirements and recommendations will change as a consequence of design and implementation choices and of pilots’ feedbacks, updated versions of this document including these changes will be made available.

\textsuperscript{11} This document should not be considered a user guide, but only a requirement/recommendation specification.
1.1 Example: The Interbank Exchange Network

1.1.1 Introduction

The Interbank Exchange Network (IEN) use case is drawn from the section ‘Annex D – Examples’ of the OMG UML Testing Profile (UTP) specification [78] where it is described in detail. We give a concise service oriented perspective of the use case and sketch some samples of the essential service model, the SUT model and the TC model.

The picture Figure 1 (from the UML Testing Profile V1.2 specifications) gives an overview of the system. The Automated Teller Machine (ATM) interconnects to the European Union Bank (EU Bank), through the SWIFT network, which plays the role of a gateway. The example is motivated using an interbank exchange scenario in which a customer with a European Union bank account wishes to deposit money into that account from an ATM in the United States.

![Figure 1. Overview of the Interbank Exchange Network (IEN).](image)

The mappings and transformations between the models and the systems are sketched in the diagram Figure 2. From the point of view of the testing activity, and therefore of the MIDAS facility, the system model is a descriptive architectural model of the services architecture under test and the test configuration model is a prescriptive architectural model of the test execution system.
Figure 2. Architectural model mapping and transformation for testing.

The instantiated services architecture under test is composed of a number of SUT nodes that put into operation the services specified by a service PSM, for instance on the SOAP platform where it is formulated as a WSDL/XDS document. The Service PSM is a manifestation of the service PIM and its elements are deployed on the SUT.

The essential service PIM elements are used to classify the elements of the SUT model. This model is an abstraction of the SUT, i.e. a description of its topology and a documentation of the node/port locations (the locations are expressed as properties of the ports, and are not made explicit in the figures below).
The test configuration platform independent model (TC PIM) is a partial model of the test execution system. It conveys architectural information about the SUT and its testing targets, i.e. it identifies, describes and localizes the SUT nodes/ports that shall be the target of the stimulation, response and observation of the test execution system. The essential service PIM elements classify the TC PIM elements. The TC PIM is obtained by user-driven model-to-model (M2M) transformation from the SUT model.

The TC PIM is transformed into a platform specific, implementable test configuration model by an automatic model-to-text (M2T) transformation. In this example, the specific platform is a TTCN-3 execution environment, and the TC PSM is a TTCN-3 code artifact, which is a manifestation of the TC PIM. Its deployment on the MIDAS TTCN-3 execution environment puts into operation a configuration of the test execution system that takes into account the testing targets of the SUT.

All these models, including the implementable ones, are architectural models. As such, they are used by the MIDAS test schemes’ generation and running behaviors, but are independent from any testing approach, method, practice implemented by these behaviors.

The section 1.1.2 (Essential service platform independent model) presents the essential service PIM and its elements such as the Participant and the Service Contract skeletons through some samples from the IEN use case. The Essential Service PIM is a standard UML model extended by the SoaML profile [77][63]. In order to keep this presentation concise, the associated Service PSM is not detailed.

The section 1.1.3 (System under test model) sketches the SUT platform independent model and its elements, such as the Node instances and the Communication Paths through these instances, through samples from the IEN use case.

The section 1.1.4 (Test configuration model) introduces the TC platform independent model, the MIDAS Profile and Meta-model elements and some MIDAS test configuration patterns. These issues are illustrated through some samples from the IEN use case.

In the conclusion (0) we summarize and exemplify the model mapping and transformation cycle, from the system under test to the test execution system, via the descriptive system model and the prescriptive test configuration model, through the IEN example.
1.1.2 Essential service platform independent model

The essential service platform independent model of a services architecture under test is a UML/SoaML model [77][63]. It includes a SoaML Service Contract skeleton for each involved service, and a SoaML Participant skeleton for each involved participant system. The SoaML Services Architecture element is used to model the construction of a composite Participant, starting from the top level system.

The shapes of the Service Contract and Participant skeletons will be clarified below. Service Contracts and Participants will be used to classify SUT PIM elements (Node instances) and TC PIM elements (Component instances).

1.1.2.1 Definitions of system, system construction and subsystem

For the purpose of the MIDAS model-based testing approach, a system is constructed with same-meta-type elements (nodes that implement Participant roles) that are linked by same-meta-type links (communication paths that implement Service Channels). Note that, if a communication path is specified between two nodes, this means not only that the involved participant systems are potentially able to implement the appropriate service roles, but also that these roles are actually bound each other by an active service relationship, i.e. by a service relationship that is active in the end-to-end service exchange to be tested, whose interactions are conveyed precisely by the specified communication path. We consider that between two nodes we have as many communication paths as many active service relationships.

In the diagram Figure 3, a “generic” system construction is illustrated by: (i) its composition, i.e. the set of nodes that are “part of” the system inside the system composition boundary, (ii) its environment, i.e. the set of nodes that are not “part of” the system but are connected with at least one composition node and (iii) its structure, the collection of communication paths between all the nodes of the union of the composition set and the environment set. The other nodes, outside the system environment boundary, are not in the system construction.
Figure 3. Example of system construction.

A more precise formal definition of system and system construction is presented below [18]:

Let $\Pi$ a class of nodes.
Let $\bowtie$ a binary relationship between nodes; for $x, y \in \Pi$, $x \bowtie y$ represents a communication path between $x$ and $y$; $\bowtie$ is symmetric, non-reflexive and non-transitive.

Let $\Sigma$ be a class of systems.
Let $<$ a binary relationship between a node and a system; for $x \in \Pi$ and $\sigma \in \Sigma$, $x < \sigma$ means that $x$ 'is part' of $\sigma$.

The construction of $\sigma$ is a triple $< C(\sigma), E(\sigma), S(\sigma)>$ where:
the composition $C$ of $\sigma$ is defined as $C(\sigma) = \{x \in \Pi | x < \sigma\}$;
the environment $E$ of $\sigma$ is defined as $E(\sigma) = \{y \in \Pi | y \notin C(\sigma) \land \exists x: x \in C(\sigma) \land x \bowtie y\}$;
the structure $S$ of $\sigma$ is defined as $S(\sigma) = \{<x, y> | (x \bowtie y) \land (x, y \in C(\sigma) \lor (x \in C(\sigma) \land (y \in E(\sigma))))\}$.

The statements above allow a more precise definition of the often ill-defined notion of subsystem:
1.1.2.2 Top level system specification

The diagram Figure 4 presents the specification of the construction of the top level system (InterbankExchangeNetwork) through a Services Architecture. The meanings and uses of Services Architecture and of its parts (classified by Service Contracts and Participants) are defined by the SoaML Profile and Metamodel [63].

![Diagram](image_url)

Figure 4. IEN – Specification of the construction of the top level system - Services Architecture.
The Services Architecture InterbankExchangeNetwork includes four roles that are bound with four Participants – three composition roles and one environment role:

- **:HWControl** - a human or artificial agent operating as an ATM (Automatic Teller Machine) user – this is an environment role bound to Participant HWControl; in UML terms, this is a shared aggregation [63].

- **:ATM** - the Automatic Teller Machine system. This is a composition role bound to Participant ATM.

- **:Bank** - the bank system. This is a composition role bound to Participant Bank.

- **:SwiftNetwork** - the Swift gateway. This is a composition role bound to Participant SwiftNetwork.

In this example, the roles are unnamed, i.e. are indirectly named with their Participant classifiers’ names. The multiplicity of each role (the default multiplicity is 1) specifies the number of Participant parts that can be included in an instance (Collaboration Use) of the Services Architecture InterbankExchangeNetwork.

The relationships among the roles are typed by Service Contracts. This is represented by Dependency relationships between roles and Service Contract parts:

- **:ATMHandling** - regulating the service exchange and the interactions between :HWControl and the :ATM. It is a many-to-many service.

- **:BankService** - defining the service that the :Bank provides to the :ATM. It is a many-to-one service (the multiplicity 1 is default in the diagram).

- **:SwiftExchange** – regulating the service exchange interaction between the :Bank and the :SwiftNetwork. This is a many-to-one service.

In this example, also the Service Contract parts are unnamed, i.e. indirectly named by the Service Contracts that classify them.

The business process through which a customer deposits money into her/his account handled by a EU bank from an US ATM, passing through the Swift network, is choreographed by the UML Activity
DepositFrom_ATM_ThroughSwift. We will not detail this Activity that is not part of the essential service PIM.

1.1.2.3 Service contract skeletons

The “minimal” Service Contracts (skeletons) include only the information needed to build the SUT and TC models. Obviously, the result of a model-driven service engineering cycle could be a richer Service Contract, including not only the skeleton model elements, but also detailed interfaces, protocols, choreographies, operation semantics (Capabilities) and so on.

We present two Service Contract skeleton samples: of a unidirectional service (Figure 5) and of a bidirectional service (Figure 6). A unidirectional service specifies only one Interface that is required by one party and provided by the other party. A bidirectional service specifies two Interfaces that are alternately required and provided by the two parties. The Interface operations can be synchronous request/response or one-way. The meanings and uses of the Service Contract, Consumer, Provider, Service Interface, Capability elements are defined by the SoaML Profile and Metamodel [63] Interface elements are standard UML interfaces [77].
The elements of the Service Contract skeleton BankService (that are in bold lines Figure 5) are:

- The Service Contract BankService;
- the role bankuser that is bound with the Consumer BankUser;
- the role bankdesk that bound with the Provider BankDesk;
- the Provider BankDesk that realizes the Interface IBank;
- the Consumer BankUser that uses the Interface IBank;
• the Interface IBank that can be an empty element - it is pointed by (one of) its manifestations, the PSM interface IBank.wsdl on the SOAP platform.

Other elements that are not in the ServiceContract BankService skeleton could be:

• the Protocol State Machine BankDesk Protocol that defines the interaction protocol with the Provider role bankdesk;

• the Capability BankDesk Capability exposed by the Provider BankDesk that encapsulates the semantics of the service operations.

The diagram of the bidirectional Service Contract skeleton ATMHandling is illustrated Figure 6. Instead of the Provider and the Consumer that classify the roles of a unidirectional Service Contract, the roles atmuser and atmcashier are classified respectively by the Service Interfaces ATM_User and ATM_Cashier.
1.1.2.4 Participants

Participants are classes of abstract systems that are used as classifiers of the roles of a Services Architecture. Each Participant is equipped with a port for each Service Contract role that it fulfills, these ports being typed by the service interface elements such as Providers, Consumers (for unidirectional services) and Service Interfaces (for bidirectional services) that are used as classifiers of the roles defined in the Service Contracts that the Participant fulfills. These ports expose the appropriate provided and required interfaces specified by their service interface types.
The collection of Participants involved in the Services Architecture that realizes the construction of the top level InterbankExchangeNetwork composite Participant is presented Figure 7.

**Figure 7. IEN – InterbankExchangeNetwork construction Participants.**

Bank is in turn a composite Participant whose construction is realized by the Services Architecture Bank (Figure 8) that includes two composition roles classified by:

- **Participant BankCounter** – that operates as a “front office”,
- **Participant AccountHandler** – the “back office” that handles the bank accounts\(^{12}\).

---

\(^{12}\) This internal architecture is introduced here for explanation purposes and does not correspond exactly with that described in the Annex D of the OMG UML Testing Profile (UTP) specification \([78]\).
The other roles (:ATM, :SwiftNetwork) are environment roles, i.e. placeholders for systems that are not in the composition of Bank but in its environment.

![Diagram of Bank services architecture](image)

**Figure 8. IEN – Participant Bank construction Services Architecture.**

Participants BankCounter and AccountHandler are presented Figure 9.

![Diagram of participants](image)

**Figure 9. IEN – Participants BankCounter and AccountHandler that fulfil the composition roles of Bank.**
The composite aggregation structure of the Participant Bank is illustrated Figure 10. The composite parts :BankCounter and :AccountHandler are bound with the corresponding roles defined in the Services Architecture Bank (Figure 8). The delegation connectors link the ports of the composite class to the ports of their component parts. So, the port Bank/bankdesk delegates to the port BankCounter/bankdesk and the port Bank/bankentry delegates to (provided interface) and is delegated from (required interface) the port BankCounter/bankentry. The ports linked by the delegation connectors must be classified by the same service interface (Provider, Consumer and Service Interface) elements. The Service Channel Connector links the component Participants’ ports bound with the roles defined in the Service Contract AccountHandling.

![Figure 10. IEN – Composite Participant Bank.](image)

All the IEN Participants, with their composite aggregation and active service relationships are presented Figure 11.
Figure 11. IEN - Participant composite aggregation hierarchy and active service relationships.

Figure 12. IEN - Relay Participants.
The essential service model includes other Participants that model service relays. For each Service Contract, a relay Participant is derived that is able to enact all and only the Service Contract roles. Conventionally, we name each relay Participant by the Service Contract from which is derived. The relay Participants model elements can be mechanically generated in a seamless way from the related Service Contracts model elements. Relay Participants will be used to classify TC PIM Interceptor instances (see below). The relay Participants for the IEN use case are presented Figure 12.

1.1.3 System under test model

The system under test model (SUT model) describes the nodes/ports of the deployed SUT that are potentially accessible by the test execution system. It models only the composition of the SUT, not its environment. This model is completely independent from the test schemes that can be applied to the SUT. It describes only its architectural configuration, i.e. its topology, and documents the locations of the deployed and accessible nodes/ports. It does not include any information about test approaches, methods and practices implemented by specific test schemes. It is formulated as a UML Deployment and each deployed and accessible SUT node is represented as a UML Node instance.

A SUT model is always grounded upon an essential service model. The service model Participants are used as classifiers of the SUT model Node instances. The classifiers of the Node instance usbank are presented Figure 13.

The SUT nodes/ports that are described in the SUT model are accessible by the test execution system that can interact directly with them (see below). Conversely, deployed SUT nodes that are not described in the SUT model are not accessible: they cannot interact directly with the test execution system.

Each Node instance defines its accessible ports that are included in the collection of ports that are defined by its Participant classifier. The port locations are documented as port Property values (that are not represented in the figure below).
Accessible connections between SUT nodes are represented by Communication Paths between the Node instances’ ports. Conversely, implemented connections between SUT nodes that are not represented as Communication Paths between the SUT model Node instances are not accessible by the test execution environment.

The SUT model of the IEN use case - in which a customer with a European Union bank account deposits money into that account from an Automated Teller Machine of a United States bank - is illustrated with the deployment diagram Figure 14. It shows five Node instances and four Communication Paths. The Node instances are:

- **usatm:**ATM models the US ATM where the user deposits money – it exhibits two accessible ports (atmcashier and bankuser);
- **usbank:**Bank models the US bank system – it exhibits two accessible ports (bankdesk and bankentry);
- **swift:**SwiftNetwork models the Swift exchange system – it exhibits the accessible port swift;
- **eubankco:**BankCounter models the EU bank system counter (gateway) component – it exhibits two accessible ports (bankentry and frontoffice);
• **euaccnth:AccountHandler** models the EU bank system account handler component – it exhibits the accessible *port backoffice.*

![Diagram: IEN - SUT Model](image)

**Figure 14. IEN - SUT model.**

The *ports* are connected by four *Communication Paths* (*cp01*, *cp02*, *cp03*, *cp04*), as indicated in the diagram Figure 14. The *port atmcashier* of *usatm* is not bound to any *Communication Path*. This means that if the test execution system wants to trigger or capture some behavior at this *port* must stimulate or observe it directly.

Note that the *Node* instance *usatm* whose classifier is the composite *Participant Bank* is modeled as a *black-box*, i.e. the test environment cannot access its internal interactions. Conversely, the composite architecture of *Bank* is flattened for the EU bank, and the *Node* instances *eubankco:BankCounter* and *euaccnth:AccountHandler* are explicitly represented with the *Communication Path* between them.

The SUT model *must* be a *flat model*: the nested composites *must* always be flattened to the chosen/allowed local aggregation level. It represents a deployment of the instantiated *Services Architecture* that is presented in the diagram Figure 15. This is a *system construction* model as defined paragraph ¡Error! No se encuentra el origen de la referencia..
The end-to-end service exchange interactions that are choreographed by **DepositFrom_ATM_ThroughSwift** are accessible on the SUT whose model is presented in the diagram Figure 14. If appropriately configured, the test execution environment is able to send a stimulus through the interface **IATM** of the port **atmcashier** and to access the SUT response at the other accessible ports and **Communication Paths**.

### 1.1.4 Test configuration model

The Test Configuration model (TC model) describes the configuration of the architectural components of a UTP Test Context [78]. It represents: (i) the SUT nodes and ports that are the targets of stimuli, responses and observations in the course of the test run, (ii) the “sensors/actuators” that are virtually placed on the SUT nodes/connections in order to exercise the aforementioned stimuli, responses and observations and (iii) the connections between SUT and the test execution environment that have to be established. The TC model must be associated to a SUT model and must be consistent with its topology, but is not aware of the test approaches, methods and practices that are implemented by the test schemes invoked on the SUT: a TC model may be reused by several test schemes.
The TC model is formulated as UML Component model extended by the MIDAS UML Profile and Meta-model, that is an extension of the UTP Profile and Meta-model [78]. The elements of the TC model are stereotyped Component instances.

### 1.1.4.1 TC UML profile

MIDAS introduces three UML Component stereotypes (Figure 16) as a specialization of the UTP profile and meta-model [78]. The MIDAS Proxy, Emulator and Interceptor specialize the UTP TestComponent as an extension of Component\(^\text{13}\).

![Figure 16. Test configuration profile.](image)

A Proxy Component models a test execution system component that operates as a proxy of an accessible SUT node - described by a SUT model Node instance - that is the potential target of stimuli, responses and observations of the test execution environment.

An Emulator Component models a test execution system component that is able to emulate either a SUT node (represented by a UML Node instance in the SUT model), which is a composition element, or a “virtual” environment node. For instance, an environment node for the SUT deployment represented Figure 14 could be an implementation of the Participant HWControl, i.e. a system that implements the Service Interface ATM_User on a port such as atmuser.

\(^{13}\) This is in fact a restricted specialization of the UTP profile, in which TestComponent is an extension of the more general Class classifier.
The test execution system component modeled by an *Emulator* instance shall be able to interact with the SUT nodes through the test execution system component modeled by the corresponding *Proxy* instance, and the allowed interactions between the test execution environment and the SUTs nodes must be formally compliant with the service interfaces specified by the *Participants* that classifies respectively the *Emulator* instance and the *Proxy* instance.

An *Interceptor Component* models a test execution system component that is able to act as if it were virtually placed on an accessible communication path between SUT nodes that is represented by a *Communication Path* between ports in the SUT model.

The test execution system component modeled by an *Interceptor* shall be able to interact with the SUT nodes at the connection end-points, and the resulting interactions shall be compliant with the service interfaces specified by the *Participants* that classifies the end-point SUTs.

The TC model elements instantiate the essential service model *Participants*. *Proxy* instances *must* be classified only by *Participants* that classifies SUT model *Node* instances. The classifier of an *Emulator* instance *must* be one of the *Participants* that classify composition and environment roles in the corresponding instantiated *Services Architecture*. *Interceptor* instances *must* be classified by the *relay Participants* derived by the Service Contracts involved in the aforementioned *Services Architecture*.

*Proxy* instances model test execution system components that must act as mere proxies of the SUT nodes and shall not implement any particular testing operational semantics. The *Emulator* and *Interceptor* instances are *per se mere skeletons* without testing operational semantics. They are used to configure the capability of the test execution system to interact with the SUT in compliance with the service interface specifications documented in the service model. Their particular operational semantics in a *test scenario*, i.e. the interaction control flow, the interaction contents and the actions at test run time, is provided by the invoked test scheme and represented by a UTP *Test Case*.

### 1.1.4.2 Test configuration patterns

We introduce three *test configuration pattern* families:

- *unit test configuration* patterns,
- *unit test with stubs configuration* patterns,
- **integration test configuration** patterns.

   Figure 17 sketches an example of the application of the *unit test configuration pattern* for bidirectional services to the SUT whose model is illustrated by the diagram Figure 14.

   **Figure 17. IEN - TC model - unit test.**

   The *Emulator* instance and the *Proxy* instance that are specified in the TC model Figure 17 are classified by the service model *Participants* as shown Figure 18 and Figure 19.

   **Figure 18. IEN - Emulator hwcntrl classifiers.**
Figure 19. IEN - Proxy euatm classifiers.

The TC model Figure 17, produced by the application of the unit test pattern, specifies the stimulation and the observation of the behavior of the SUT usatm through the exchange regulated by the bidirectional service ATMHandling through the accessible port atmcashier classified by the Provider ATM_Cashier.

The diagram Figure 20 documents the result of the application of a unit test with stubs configuration pattern to the IEN SUT (Figure 14). The behavior of the node represented by the Proxy usatm:ATM is stimulated and observed at the ports atmcashier by the Emulator instance hwcntrl:HWControl and bankuser by the Emulator instance usbank:Bank.

This pattern is well suited for unit testing within service compositions. In order to provide the service ATMHandling to hwcntrl - for instance to effect a synchronous withdrawal from a US account through a US ATM - usatm needs to consume the service BankService provided by usbank. Hence, an interaction at the port atmcashier can cause an interaction at the port bankuser. If the service model of the Participant ATM includes the definition of transfer functions that specifies the correspondence between IBank operations’ input variables with the corresponding (in the service composition) IATM operations’ input
variables\(^{14}\), advanced testing methods allow an appropriated Emulator \texttt{usbank:Bank} to get static or dynamic oracles for \texttt{IBank} interactions.

![Diagram](image)

**Figure 20. IEN – TC model – unit test with stub.**

The diagram Figure 21 documents a TC model produced by the application of an integration test configuration pattern to the IEN SUT (Figure 14).

![Diagram](image)

**Figure 21. IEN – TC model – integration test.**

\(^{14}\) For instance, the transfer function can be formulated in OCL.
The diagram Figure 22 shows the classifiers of the Interceptor instance cp01 that is employed in this model.

![Figure 22. IEN - Interceptor cp01 classifiers.](image-url)

The Interceptor instance cp01 allows specifying the observation of and the actions on the interactions between the port usatm/bankuser and the port usbank/bankdesk, i.e. the integration test between usatm and usbank.

1.1.4.3 The Interbank Exchange Network TC model

The combination of the patterns presented in the preceding section should allow the definition of almost any test configuration. The end-to-end service exchange that realizes business process of the deposit of money in an EU bank account through a US ATM, can be stimulated and observed by the appropriate combination of the test configuration patterns on the SUT.

The TC model is the result of a M2M mechanical transformation from the SUT model that is driven by the tester specification. The tester specifies: (i) the SUT subsystem, i.e. the collection of the SUT Nodes (the Proxy instances) that are the targets of stimuli, responses and observations (and, implicitly, the connections between SUT and the test execution environment); (ii) the Emulator instances that either replace SUT Nodes or implement “virtual” environment nodes; and (iii) the Interceptor instances that are virtually placed on the SUT Connection Paths.
A TC model for partial integration test on the SUT whose model diagram is presented Figure 14, is illustrated Figure 23. This TC model specifies a test configuration in which only the US system is tested (the Swift network is replaced by an Emulator instance).

![Figure 23. IEN – Partial test integration configuration model.](image)

A TC model for end-to-end service exchange on the SUT whose model diagram is presented Figure 14, is illustrated Figure 24. This TC model includes five Proxy instances, one Emulator instance and four Interceptor instances. It specifies a complete integration test execution system in which a ‘virtual’ environment node is implemented by an emulator and all the other SUT nodes are observed/stimulated by interceptors through the proxies.
Each test scheme that conforms to UTP should be able to build the Test Context by M2M transformation of the TC model, i.e. by filling the TC model with a specific Test Case, and subsequently to generate the test execution environment specific code artifact (in this example the TTCN-3 code) by M2T transformation.

1.1.5 Conclusion

An IEN example of model mapping and transformation is sketched in the diagram Figure 25. The implemented TC model is that illustrated by the diagram Figure 23.
Note that: (i) the notations that are used Figure 25 to sketch the SUT and the test execution system are the same that are employed for the formulations of respectively the SUT model and the TC model; (ii) the communication path cp01 is actually cut by the test configuration, because the interactions between port usatm/bankuser and port usbank/bankdesk nodes are intercepted by the test execution system, in compliance with the TC model illustrated Figure 23; (iii) the ports of proxies usatm and usbank are bound to the corresponding ports of the namesake SUT nodes.

Figure 25. IEN – Model mapping and transformation cycle.
With the IEN use case illustrated Figure 25 we exemplify the model mapping and transformation cycle that conducts to the deployment of the architectural code of a test execution system in the MIDAS facility.

In the example the system model includes an essential service PIM and a service PSM (a collection of WSDL artifacts). These service PIM/PSM are the results of: (i) either a contract-based, model-driven engineering cycle, (ii) or a reverse modeling activity, (iii) or both. On the basis of these models, the MIDAS user (for instance, the testee) is able to author the SUT model and upload it on the MIDAS facility in a seamless way.

From the descriptive system model (PIM, PSM and SUT models) it is possible to obtain, by M2M transformation driven by the tester specification, the prescriptive TC PIM with a very limited effort of the user.

The M2T transformation from the TC PIM to the executable TC PSM on the TTCN-3 execution environment, with automated generation of the test execution system skeleton code (its “architectural” part), is performed directly by the MIDAS facility. Each test scheme generation behavior should be able to: (i) transform the TC model in a full Test Context model by combining it with the Test Case that it realizes; (ii) generate from the Test Context the platform specific test execution system (architecture and operational semantics), on one or more test execution platforms such as the TTCN-3 test execution environment, by applying its own methods and by using the MIDAS test execution system configuration API.

In terms of reuse, starting from the essential service PIM and the service PSM, the user can author several SUT models, corresponding to different SUT deployments. Starting from a SUT model and the associated service PIM/PSM the user can specify and generate several TC models by user-driven M2M transformation. Moreover, each TC model, that is an architectural model independent by any test method and approach, can be reused by each MIDAS compliant test scheme that is able to build a Test Context by M2M transformation. Last but not least, the generation of the test execution system by M2T transformation of the Test Context is proposed by MIDAS for the TTCN-3 platform, but is not a priori limited to that platform.

---

15 When the service model that is issued from the design phase lacks some features of the essential service model it must be completed by the user.
REFERENCES


MIDAS • Project Number 318786 • White paper • Page 48 of 50


