Bayesian dynamic scheduling for service composition testing

Ariele-Paolo Maesano

THESE DE DOCTORAT DE

L'UNIVERSITE PIERRE ET MARIE CURIE

Spécialité

Intelligence Artificielle et Décision

(Ecole doctorale)

EDITE

Présentée par

M. Ariele-Paolo Maesano

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse:

Bayesian dynamic scheduling for service composition testing

Soutenue le

Devant le jury composé de:

M. Benoit Iung - Rapporteur

M. Philippe Leray - Rapporteur

M. Fabrice Kordon - Examinateur

M. Sebastien Wieczorek - Examinateur

M. Christophe Gonzales - Directeur de these

M. Pierre-Henri Wuillemin - Encadrant

M. Fabio De Rosa - Encadrant

Abstract

In the digital economy, the service/API approach of the collaboration of heterogeneous distributed systems, applications and devices spreads rapidly. Effective and efficient testing of services and services architectures is one of the few concrete means for improving the stakeholders' trust and a critical challenge for researches and practitioners. Service testing is difficult, hard to manage and expensive in terms of hardware and software equipment, labour effort and time to market. Automation of test generation and of test run, accessible as a service on cloud can help researchers and practitioners. The objective of this research, that is conducted today in the context of a European project (MIDAS, Model and inference Driven – Automated testing of Services architecture), is a method and tool for intelligent dynamic scheduling of test sessions on services architectures that is deployed on the MIDAS platform as a service on cloud. Dynamic scheduling of test runs is the dynamic choice of the next test case to run on the basis of the past test verdicts, for precocious detection of failures and localisation of faulty elements (troubleshooting). Dynamic test scheduling is a complex task to be effected in an uncertain environment whose purpose is to enable shorter and more frequent test sessions, in the context of Test-Driven Development and Continuous Integration Testing. The research reported in this manuscript aims to design and implement a test scheduling service that utilises a Bayesian Network model of the test environment and algorithms of probabilistic inference. In order to cope with the high computational complexity problems to which these methods and algorithms are confronted, an original inference by compilation algorithm (inf4sat) has been developed that compiles complex Bayesian Networks into compact Arithmetic Circuits. Compared to its "competitors" ("classic" Bayesian network and Arithmetic circuit inference algorithm), inf4sat is more robust, less consuming in memory size, exhibits better inference speed and pushes further the practical tractability limits of complex models. Conversely, it is not as good as its competitors in compilation speed, but this trade-off (compilation speed vs. inference speed) is completely consistent with the proposed *modus operandi* of the algorithm, which allows compiling once, saving the AC image and running fast (without any compilation time) a large number of frequent test sessions on the same AC image, particularly for re-testing and regression testing. The scheduler is utilised on the MIDAS platform to schedule automated test sessions on real world case studies that are services architectures in the health and logistic domains. The feedback from the experience will allow refining and tuning the architectural choices, the modus operandi and the scheduler policies. Future work includes the enrichment of the test report with the results of probabilistic inference, the investigation of scheduling policies dedicated to re-testing and regression testing, the Noisy-OR approach of Bayesian Reasoning (that can be easily integrated in the current implementation) and more ambitious projects about the dynamic management of the automated test generation activity by the scheduler probabilistic inference and a technique of layered partitioning troubleshooting, with dynamic aggregation/disaggregation of regions of very large services architecture.

Table of Contents

1. INTRODUCTION

GENERAL OVERVIEW OF THE FIELD OF APPLICATION: SOA TESTING

The SOA/API *design and implementation style* [SoaML 1 0 1 2012] is particularly suitable for the digital economy, where entities such as applications, systems, devices are connected and collaborate without human intermediation (see section 2 - § 'Services architecture'). The collaboration between these entities is based on the *exchange of service provisions through remote Application Programming Interfaces* (APIs) and on the *mutual information hiding of the implementations*, which is not only a confidentiality issue, but also a way of mastering the complexity [Parnas et al. 1983].

Generally speaking, the service interface description (the APIs) is, by definition, well-defined, *formal* and *machine readable*, while the functional and behavioural service descriptions are often partial and informal - in general, they are supplied as API informal documentation. In any case, the service specification is a black-box model of the system that claims to provide the service. The service provider system implementation and its white-box model, if any, are hidden and cannot be derived from the service black-box model, even the most formal one. The fundamental problem of the service economy is: How can the stakeholders trust that a system is able to provide the service that it claims to provide and that is described in the service specification/documentation when its implementation is hidden? The only sustainable response to this question is: by testing its actual visible behaviour against the service specification, i.e. by comparing the actual behaviour with the definition of the behaviour that is compliant with the specification.

Services are the building blocks of the digital economy: modern applications aggregate services through their APIs. Services become enablers of critical business transactions and, potentially, the weakest links in these transactions. The quality of the services that an organisation *provides* and *consumes* is now more important than ever. Hence, service testing is not only an activity of service providers, but also of service users and starts in the earliest phases of the life cycle of their applications. Approaches such as Test-Driven Development [Beck 2003] and Continuous Integration Testing [Huang et al. 2008] gain popularity.

Service testing is a technical activity conducted to supply stakeholders with information about the quality of the systems that provide (and consume) the services. This activity includes the stimulation and the observation of the system behaviour according to a specified procedure [NIST 02 3 2002]. In an historical perspective, there are three distinctive "waves" of testing practices that are linked to corresponding engineering main trends: (i) vertical application testing, (ii) component off-the-shelf testing, (iii) service testing (see section 2 - § 'Towards service testing').

The testing of vertical applications - pieces of software that are utilised by business people through a Graphical User Interfaces as a support of a business activity - is organised in two distinct phases: (i) the developer practices white-box testing during the development phase, and (ii) the business user practices a kind of black-box testing by manipulating the user interface in the beta-test phase.

At the end of the twentieth century, the practice of provisioning software in the form of *components off the shelf* proliferates. A component *off-the-shelf* is a piece of software that implements a specific business or technical function, is provisioned through the physical distribution of the executable code and is accessible only through APIs. Components off-the-shelf are utilised as building blocks of applications, but their delivery modality raises the trust problem and stimulates the black-box testing practice by the user/developer.

The service delivery mode allows businesses to get rid of the burden of installing and running the software on their premises, but the service user loses definitively the control of the service software life cycle. The trust problem is raised at an unprecedented scale and become really critical. Effective and efficient testing is the main path to trust building.

A generally accepted categorisation classifies the testing practice in three main activities: (i) functional conformance test, (ii) security/vulnerability test and (iii) quality of service test. The current focus of this research on test scheduling through probabilistic inference is on functional conformance test, i.e. test of the compliance of the Services Architecture Under Test (SAUT) with the services' and service compositions' functional specifications.

Functional conformance testing of services and service compositions (see section 2 - § 'Functional conformance test tasks') is an activity composed of a number of tasks that manipulate and put in place respectively the following *objects* and *processes*:

- *Test case* (object) - A test case is: (i) a collection of specifications of the *initial states* of the SAUT stateful components; (ii) a collection of specifications of *stimuli* (messages) to be sent to the SAUT components set to initial states.
- *Test run* (process) – A test run carries out the execution of a test case. It is the actual sequence of interactions in the SAUT that is triggered by the transmission of the initiating stimulus after the setting of the SAUT components' initial states. After the end - or the interruption - of the test run, good testing practices recommend the *reset* of the SAUT state [ETSI EG 202 810 2010].
- *Test outcome* (object) – A test outcome is a representation of the SAUT observable behaviours at the interfaces and of the SAUT components' final states that are produced by a test run.
- *Test oracle* (object) – A test oracle is the partial or complete specification of the compliant test outcome of a test case run.
- *Test sample* (object) – A test sample is the couple formed by a test case and the associated test oracle. It represents a specification of a compliant behaviour snapshot of the SAUT.
- *Test suite* (object) – A test suite is a collection of test samples on a SAUT that can be ordered statically (prioritised) or scheduled dynamically (see below).
- *Test session* (process) – A test session is the sequence of test runs corresponding to the partial or total running of a test suite on a SAUT.

Service functional conformance testing is checking the compliance of the actual behaviour of a system with its service specifications. A comprehensive representation of the structural, functional and behavioural specifications of a services architecture, in which the internal implementations of the components are not observable, should include [Schieferdecker 2012] the following definitions:

- The *service interfaces*.

- The *topology* of the services architecture under test (components, services, provided and required interfaces, service connections [De Rosa et al. 2014a]).
- The service *interaction protocols* between components (for instance through Harel state charts or other similar formalisms [Harel and Politi 1998] [De Rosa et al. 2014b]).
- The *pre/post conditions* and *invariants* of the service interactions (for instance through the design-by-contract approach [Meyer 1992] [De Rosa et al. 2014b]).
- The *data-flow requirements* on the service interactions, for single services and service compositions [De Rosa et al. 2014b].

The functional conformance testing activity is decomposable in the tasks listed below that are organised in two main cycles:

- Test generation cycle:
    - test case generation,
    - test oracle generation.
- Test run cycle:
    - test execution,
    - test arbitration,
    - test scheduling,
    - test reporting.

The first objective of the execution of a test case is to expose a SAUT faulty behaviour that reveals hidden defects (*searching for failures*). The second but not less important objective is to *help localising faulty functional and structural elements* (*troubleshooting*) of the SAUT that are the source of failures. Methods and tools of generation of test suites with characteristics such as *effectiveness* (they do the job), *size efficiency* (with a minimum number of test cases), *time efficiency* (in the minimum number of test runs) have been the most important challenge of the research on SOA testing from the beginning [Canfora and Di Penta 2009] [Bartolini et al. 2011] [Bozkurt et al. 2013].

Manual production of test cases is a "creative", labour and knowledge intensive process, needing both a deep knowledge of the structural, functional and behavioural specifications of the SAUT and of the test case generation strategies and methods. Automated production of test cases is necessarily *model-based*. The model-based generation of test cases utilises *techniques* such as *constraint propagation* on pre/post-conditions and control/data flow requirements, *strategies* such as *boundary value analysis*, *equivalent class partitioning*, *random*, *ad hoc*, and *combinatorial and statistical methods* such as *pairwise* and *orthogonal arrays*.

Test oracle generation is a "mechanical" process: given a test case, and the structural, functional and behavioural specifications of the SAUT, the test oracle can be calculated "mechanically" (to be precise, it is a true mechanical process - whether performed by a human or a machine - only if the SAUT specifications are formal).

Manual production of test oracles is a labour intensive and knowledge intensive process. The producer must possess a deep knowledge of the structural, functional and behavioural specifications of the SAUT. Automated production of test oracles is necessarily model-based. The test oracle generation software must be able to calculate or emulate the external behaviour of the SAUT components as described in the SAUT model.

Test execution is the accomplishment, for each test case, of a test run. Test execution is the result of a number of technical actions, such as: setting the SAUT components' initial states, sending the stimuli, receiving, observing and collecting the SAUT responses, getting the components' final states and resetting the component states. Other ancillary actions are the deployment of the SAUT and the configuration of the test system (if any).

If an automated test execution system is not available, the tester must accomplish manually all the actions listed above, and also the set up and configuration of the environment. In the domain of test execution automation a significant progress has been made in the last fifteen years with the availability of the TTCN-3 language and its execution environments[1]. This language (and the environment) has been utilized for building automated test execution environments for SOA testing [De Rosa et al. 2013].

Test arbitration is the production of a test verdict as results of the assessment of a test outcome. Generally speaking a test verdict is the result of match (*pass*) / mismatch (*fail*) between the test outcome and the test oracle. Test arbitration can be a tricky endeavour needing accurate analysis of the test execution context in order to avoid *false negatives*, i.e. *pass* verdicts that hide SAUT failures, and *false positives*, i.e. *fail* verdicts that shroud the correct behaviour of the SAUT.

"Eyeball" test arbitration is strongly dependent on the availability of test oracles. When test oracles are available it is a labour intensive process that mobilises general "syntactic matching" abilities, but no specific knowledge of the SAUT specifications. When test oracles are not available, it is both a labour intensive and a knowledge intensive process. The needed knowledge is the same that is mobilised for the production of test oracles. The TTCN-3 language and framework allows including oracles representations and programming sophisticated mechanisms of test arbitration.

Test scheduling is arranging the test cases of a test suite in a specific order of running with the objective of improving the *fault detection rate*, i.e. the early detection of faults through the early uncovering of failures and localisation of related faulty elements, in order to speed and foster the test/debug/fix cycle and the more general test/design/implement cycle.

Static scheduling is giving a static order to the test cases prior to running through *priorities* assigned to them (*prioritisation*). The order cannot be changed at test run time. Manual static scheduling is a knowledge intensive task, comparable to manual test case production.

Dynamic scheduling is choosing at run time the next test case to run. Dynamic scheduling requires the decision by the (human or artificial) scheduler of the test case to run at each schedule/execute/arbitrate cycle on the basis of some criteria. The test cases can be prioritized before the starting of the test session, but their order (priority) changes dynamically during the session. Automatic test scheduling is an enabler for Test-Driven Development and Continuous Integration Testing.

[1] http://www.ttcn-3.org/

Manual dynamic scheduling is a labour and knowledge intensive task that requires deep knowledge of the SAUT specifications and of the test context (the test suite) and a strategic reasoning capability. Automated test scheduling requires the automation of inference methods applied to failure seeking and troubleshooting. Dynamic scheduling applied to functional conformance testing is the main focus of this research.

While the test log is the detailed narrative of all the events of a test session, the *test report* is a concise account for debugging purposes, supplying aggregate information that highlights the relations between the test cases, the test verdicts and the SAUT functional and structural elements. The *test report* is the trade-union between the test team and the design/implementation team.

Manual test reporting is a knowledge intensive task. It requires the same capabilities needed for manual test scheduling and the ability of summarising information about complex and scattered issues. Automated test reporting is still a poorly investigated research topic and it will be investigated in the MIDAS Project

AN EXAMPLE OF SERVICES ARCHITECTURE: THE INTERBANK EXCHANGE NETWORK

In this paragraph we introduce an example that is utilised in all the other sections of this manuscript and is freely inspired by the InterBank Exchange Network (IEN) "business case" described in the Annex D of the OMG UTP 1.2 Specification UML Testing Profile V.1.2 [UTP 1 2 2012]. The IEN 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 Automated Teller Machine (ATM) in the United States." [UTP 1 2 2012].

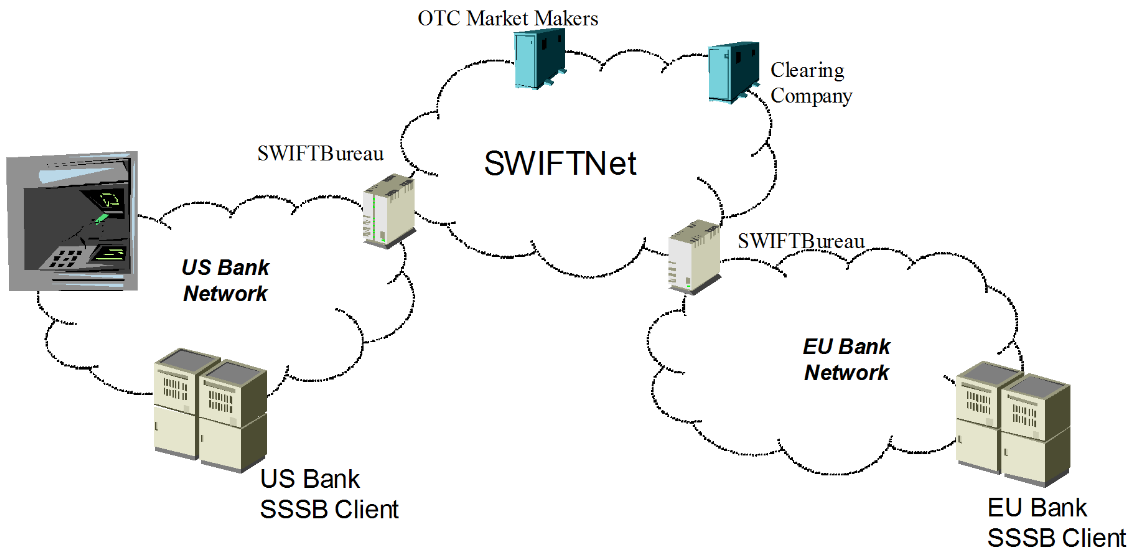


Figure 1. Overview of the Interbank Exchange Network (IEN).

The diagram in Figure 1, drawn from UML Testing Profile V1.2 [UTP 1 2 2012], gives an overview of the general architecture 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 between the logical networks of the US Bank and the EU Bank.

In the UTP specifications [UTP 1 2 2012] the example is presented as a collection of UML packages. In the MIDAS project (see § 'Background and context of the research' of this section) the example has been completely specified as a services architecture (WSDL, XSD) and implemented in modular service components with "tabular" implementation (for each service operation, a data base table of stereotyped request/responses entries) for documentation, training[2] and test of the platform of the test methods.

The services architecture of the IEN example that is utilized in the remainder of the manuscript is presented through a UML Component diagram in Figure 2.

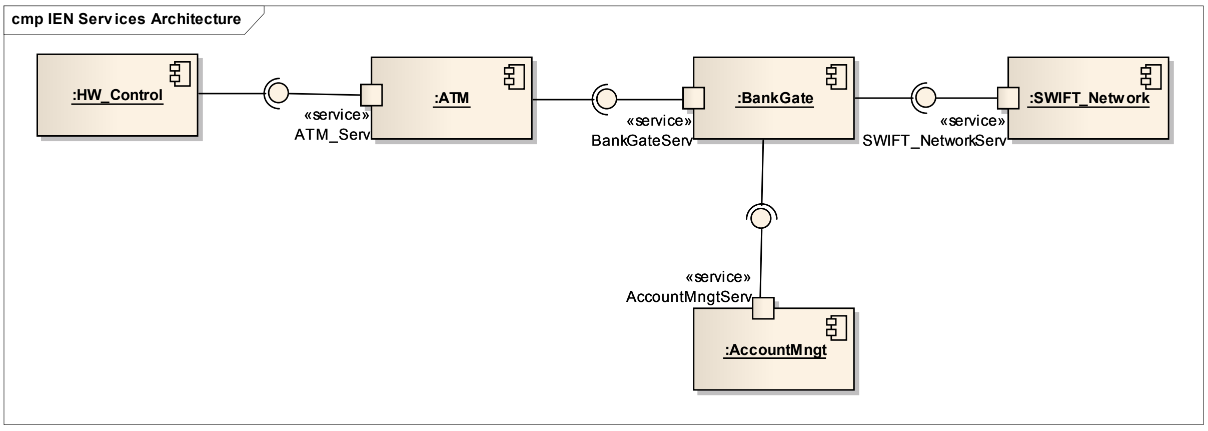


Figure 2. IEN Services Architecture.

The IEN services architecture includes five components:

- HW_Control - a human or artificial agent operating as an ATM (Automatic Teller Machine) user;
- ATM - an Automatic Teller Machine system;
- BankGate - the bank front office system;
- AccountMngt – the bank back office system, that manages the customers' accounts;
- SWIFT_Network - the Swift gateway that allows interbank communication.

Except for HW_Control, that doesn't expose any service and is used as a kind of "entry point" of the architecture, all the other components expose services, whose list is presented below:

- ATM_Interface.wsdl is exposed by ATM through its port ATM_Serv;
- BankGateInterface.wsdl is exposed by BankGate through its port BankGateServ;

[2] In the MIDAS project there are two real services architectures in the Health and Logistic domains that are utilised as targets for testing with the MIDAS facility. The data structures exchanged in these architectures are compliant to international standards and very complex, and their comprehension requires deep knowledge of the domain. Of course, this is true also for real examples in the banking domain, but the IEN, as a "naïve" banking system, can be easily understood by any common bank customer.

- AccountMngtInterface.wsdl is exposed by AccountMngt through its port AccountMngtServ;
- SWIFT_Interface.wsdl is exposed by SWIFT_Network through its port SWIFT_NetworkServ.

The services architecture depicted in Figure 2 allows running some families of service composition scenarios such as:

- "getBalance" – the user (HW_Control) inquires the balance of her/his account;
- "withdraw" – the user (HW_Control) requests to remove money from her/his account;
- "deposit" – the user (HW_Control) requests to put money in her/his account;
- "wire" – the user (HW_Control) requests to transfer money from her/his account to another account in another bank.

These scenarios can be represented through UML Sequence diagrams. The scenario GetBalance_OK is depicted in Figure 3, in which the user (HW_Control) asks ATM for its account balance amount through the getBalance(GetBalanceIn) request message. Note that SWIFT_Network is not actively involved in this scenario.

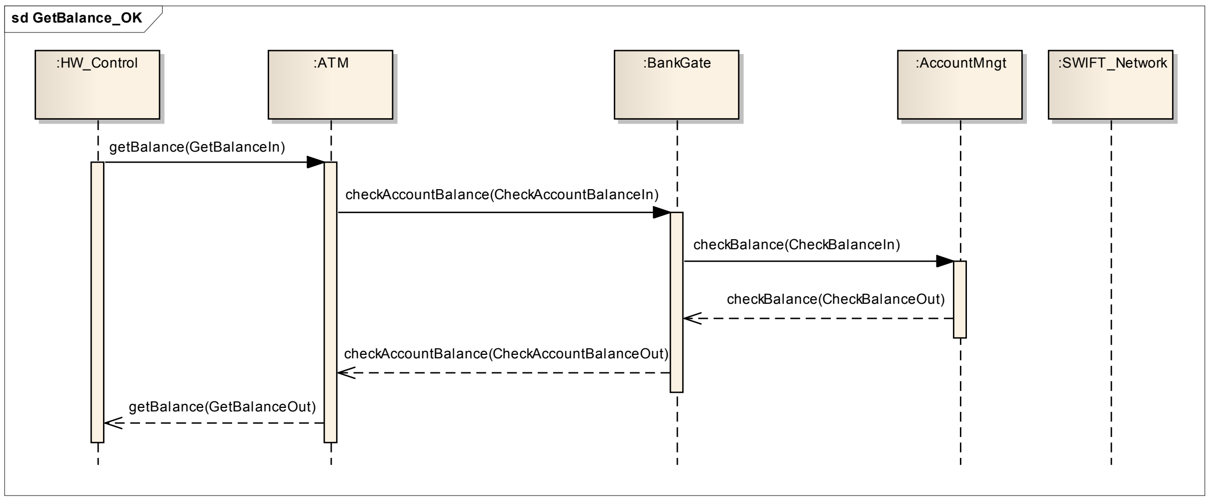

sd GetBalance_OK
:HW_Control
:ATM
:BankGate
:AccountMngt
:SWIFT_Network
getBalance(GetBalanceIn)
checkAccountBalance(CheckAccountBalanceIn)
checkBalance(CheckBalanceIn)
checkBalance(CheckBalanceOut)
checkAccountBalance(CheckAccountBalanceOut)
getBalance(GetBalanceOut)

Figure 3. GetBalance_OK scenario.

The scenario Withdraw_OK is presented in Figure 4. It is quite similar to the preceding one, with some differences: (i) pre/post-conditions are made explicit, (ii) the AccountMngt state changes. The pre-condition controlled by the ATM is that, in order to run this scenario, the requested withdraw amount (the value of a field of the input message) must be lesser than a withdraw threshold. The other pre-condition that is controlled by AccountMngt states that the requested debit amount must be lesser than the current account balance amount. Moreover, AccountMngt guarantees the post-condition that the account balance amount *immediately after* the debit operation execution is equal to the account balance amount *immediately before* the debit operation execution minus the debit amount (the value of the field of the debit input message).

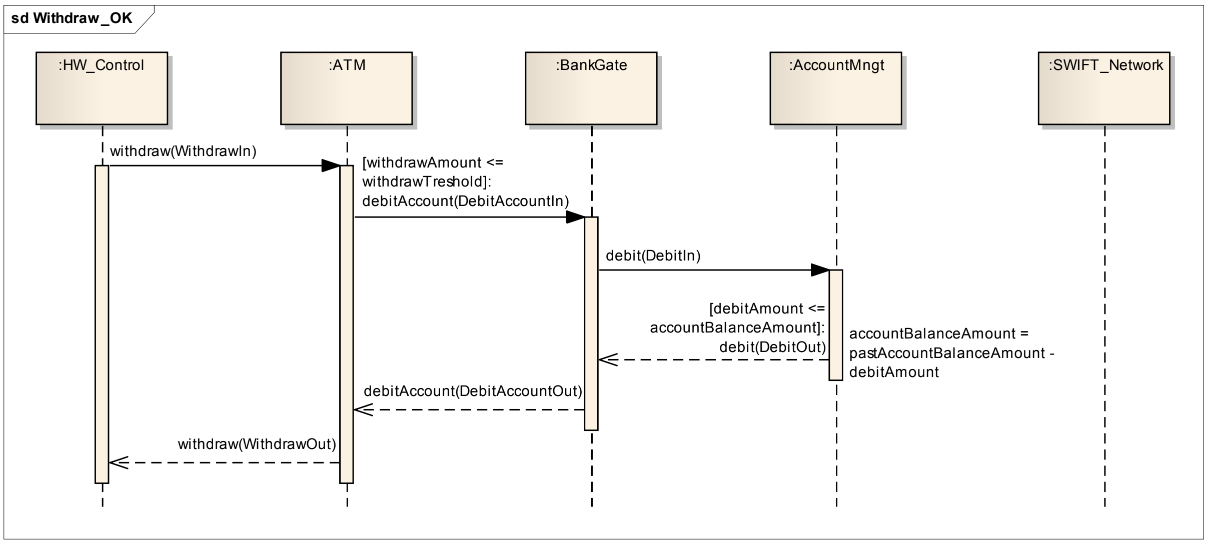

sd Withdraw_OK
:HW_Control
:ATM
:BankGate
:AccountMngt
:SWIFT_Network
withdraw(WithdrawIn)
[withdrawAmount <= withdrawTreshold]: debitAccount(DebitAccountIn)
debit(DebitIn)
[debitAmount <= accountBalanceAmount]: debit(DebitOut)
accountBalanceAmount = pastAccountBalanceAmount - debitAmount
debitAccount(DebitAccountOut)
withdraw(WithdrawOut)

Figure 4. Withdraw_OK scenario.

The scenario in which the "threshold" pre-condition is not respected by the amount field of the withdraw input message is presented in Figure 5. The immediate ATM response expresses the refusal of the withdraw request because the requested withdraw amount is greater than the withdraw threshold. Note that service engineering best practices allow designing service requesters that can legally violate the request pre-conditions, so that the burden of enforcing these pre-conditions falls to the service responder. Hence, this scenario represents a behaviour that is perfectly conformant to the functional specifications. Note also that BankGate, AccountMngt and SWIFT_Network components are not actively involved in the scenario.

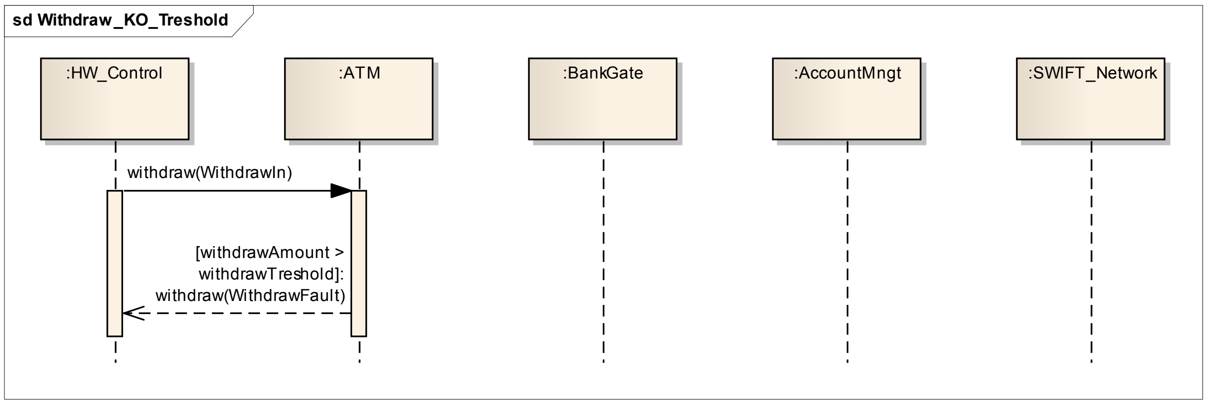


Figure 5. Withdraw_KO_Threshold scenario.

The scenario in which the "threshold" pre-condition is respected but the "balance" pre-condition is violated is presented in Figure 6. The requested debit amount is greater than the account balance amount. In this case AccountMngt refuses the debit and the account balance amount doesn't change: its value *immediately after* the operation execution equals the value *immediately before*.

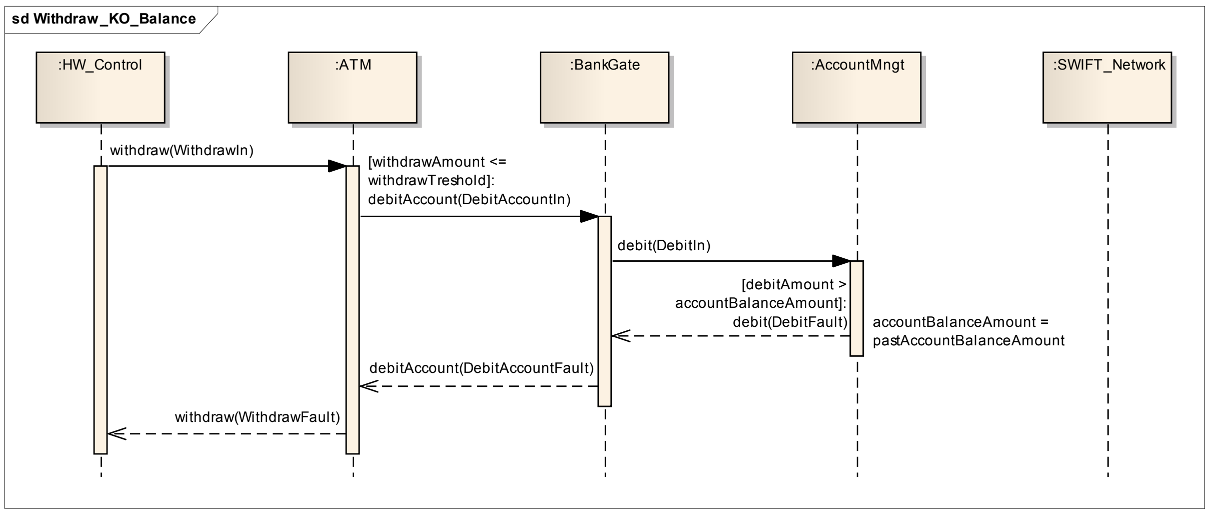
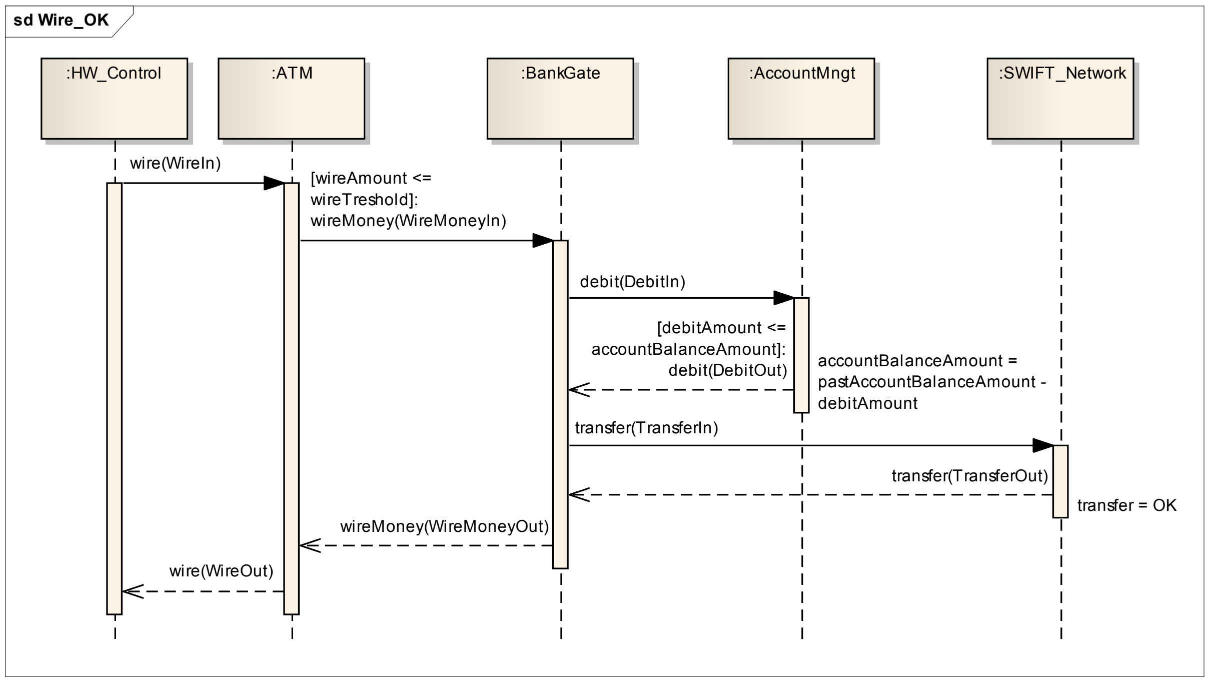

sd Withdraw_KO_Balance
:HW_Control
:ATM
:BankGate
:AccountMngt
:SWIFT_Network
withdraw(WithdrawIn)
[withdrawAmount <= withdrawTreshold]: debitAccount(DebitAccountIn)
debit(DebitIn)
[debitAmount > accountBalanceAmount]: debit(DebitFault)
accountBalanceAmount = pastAccountBalanceAmount
debitAccount(DebitAccountFault)
withdraw(WithdrawFault)

Figure 6. Withdraw_KO_Balance scenario.

The Wire_OK scenario depicted in Figure 7 is utilized as an example in other sections of this manuscript. The Wire_OK scenario is quite similar to the Withdraw_OK, except that in the latter a new post-condition is implemented by SWIFT_Network that bear on the success of the transfer, which implies that elements of fault-tolerance are introduced in the functional specifications. Note that the removal of the money from the customer account is done before the transfer[3].



[3] The bank behaviour modelled in this scenario is pretty realistic.

Figure 7. Wire_OK scenario.

The Wire_KO_Tranfer scenario depicted in Figure 8 represents the situation in which everything is OK until the removal of the money to be transferred from the account, but the transfer fails. By representing explicitly the possibility of technical failure at the functional level, the fault-tolerant behaviour is integrated in the functional specifications of: (i) SWIFT_Network, that is able to recognize and communicate the transfer failure, and (ii) BankGate, that performs the credit compensating operation[4]. In conclusion, the wire process has failed, but the final situation is safe.

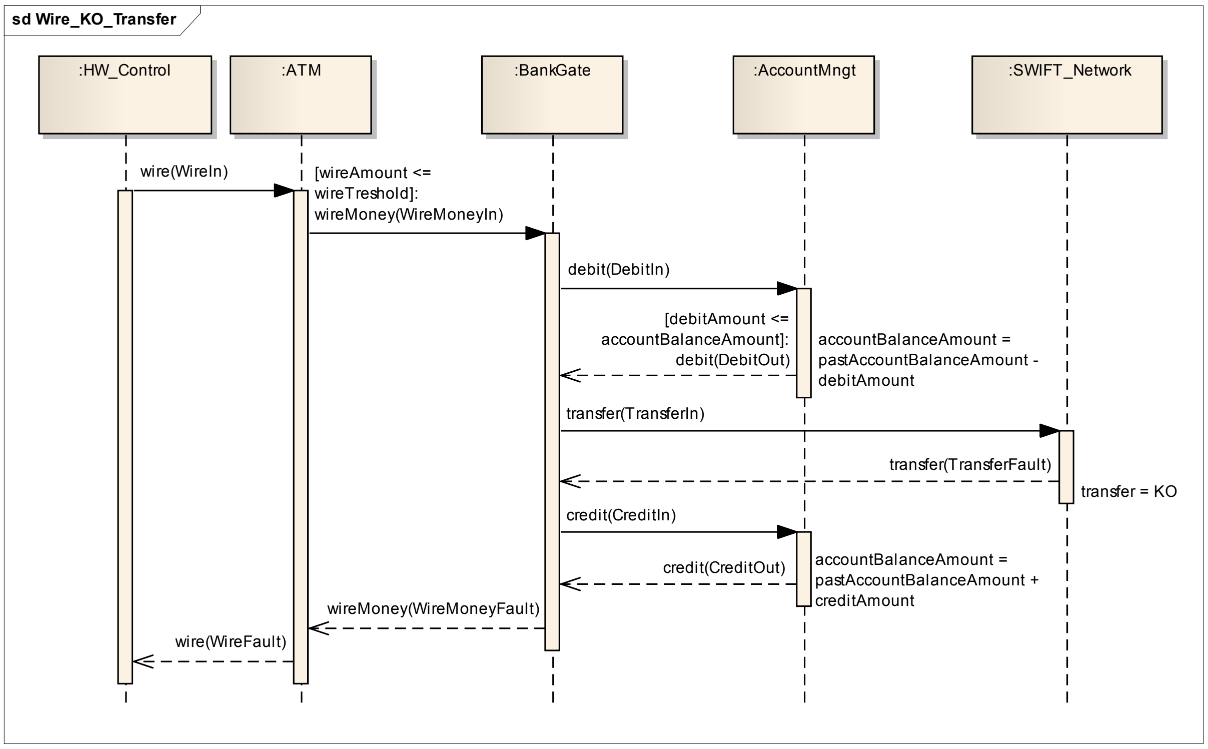

sd Wire_KO_Transfer
:HW_Control
:ATM
:BankGate
:AccountMngt
:SWIFT_Network
wire(WireIn)
[wireAmount <= wireTreshold]: wireMoney(WireMoneyIn)
debit(DebitIn)
[debitAmount <= accountBalanceAmount]: debit(DebitOut)
accountBalanceAmount = pastAccountBalanceAmount - debitAmount
transfer(TransferIn)
transfer(TransferFault)
transfer = KO
credit(CreditIn)
credit(CreditOut)
accountBalanceAmount = pastAccountBalanceAmount + creditAmount
wireMoney(WireMoneyFault)
wire(WireFault)

Figure 8. Wire_KO_Transfer scenario.

SOA functional conformance testing is stimulating and observing the behaviour of a services architecture in order to: (i) seek for failures, and (ii) localise the functional and structural faulty elements of the architecture that are responsible for the failures, at the lowest possible level of granularity. The general constraint is that the service components are black-boxes and their behaviours are observable only at the service interfaces.

A *test sample* is the representation of a scenario snapshot that is compliant with the service component architecture specifications. For instance, for the Wire_KO_Transfer scenario, the TS01 test sample is an ordered collection of instantiated messages (wire(WireIn$_{01}$), wireMoney(WireMoneyIn$_{01}$) ….), of instantiated state before variables' values (wireThreshold$_{01}$, accountBalanceAmount$_{01b}$) and of instantiated state after variables'

[4] In the reality of bank systems, these kinds of compensations are mostly performed asynchronously through batch procedures.

values (accountBalanceAmount$_{01a}$). If the services architecture is solicited with a stimulus that corresponds, for example to the wire(WireIn$_{01}$) sent to ATM, with wireThreshold set to wireThreshold$_{01}$ and accountBalanceAmount set to accountBalanceAmount$_{01b}$, it exhibits a behaviour that is conformant to the specifications if the actual exchanged messages matches the TS01 messages and the accountBalanceAmount variable value after the exchange equals accountBalanceAmount$_{01a}$.

Testing for functional conformance the services component architecture whose topology (components, services and service dependences) is represented in Figure 2 is producing and running a test suite made of a number of test samples for each scenario that allow:

- checking that the actual behaviour of the service component architecture does not violate the *pre-conditions* and the *sequence diagram* of the scenario, e.g. that, in a Wire-like scenario, when BankGate receives the wireMoney(WireMoneyIn$_{01}$) message from ATM – when wireThreshold = wireThreshold$_{01}$ - it issues a debit(DebitIn) message that matches debit(DebitIn$_{01}$) towards AccountMngt (and, for instance, neither a credit message towards AccountMngt, nor a transfer message towards SWIFT_Network that is not preceded by a successful debit request towards AccountMngt, nor any other interaction non-compliant with the sequence diagram);
- checking that the actual behaviour of the service component architecture does not violate the *data flow requirements*, e.g. checking that ATM, when receiving wire(wireIn$_{01}$), issues wireMoney(WireMoneyIn$_{01}$) – this is an implicit check that the amount field of the wireMoney(WireMoneyIn) message equals the value (semantically speaking – the syntax could be different) of the amount field of the just received wire(WireIn) message.

For stateful service components this level of checking is clearly not enough, and the test suite should also be able to check the *post-conditions* that bear on internal state variables of the service components which, in principle, are hidden. In any case, in order to test the stateful component behaviour the tester shall be able not only to **get** but also to **set** the state variables that are referenced in the pre/post-conditions (e.g. wireThreshold, accountBalanceAmount). For instance, checking the debit service operation in the Withdraw_OK, Wire_OK, Wire_KO_Transfer … scenarios needs the performance of the following actions:

1. set on AccountMngt a test case state view, with, for instance, a certain account balance state variable set to 2000€;
2. send to AccountMngt the test case debit message for this account, with, for instance, the debit amount field set to 1000€;
3. receipt the debit response message from AccountMngt and match it with the test oracle – in particular match the account balance amount field value of the message with the test oracle value (1000€);
4. get the AccountMngt state view and match it with the test oracle – in particular match the account balance state variable with the test oracle (1000€);
5. reset the AccountMngt state view.

Note that the action 3 and action 4 are distinct: the combinations of values of the balance amount field of the debit output message and of the state view account balance are a potential source of *false positives* and *false negatives*. Action 5 follows from good practices that require to leave the system under test in the conventional "initial" state in which it was before running the test. This good practice allows the logical independence of test cases and the execution of test runs in any order (which is essential for prioritising them).

In this research, and in the surrounding MIDAS project, the black-box stance towards service components is strict and the constraint that neither the tester nor the test system are able to put any probe, sensor, agent or whatever *inside* the service component implementation, which is out of reach, is enforced. The solution of the state visibility dilemma is to require, for each stateful component, the definition of a state view (the part of internal state whose visibility is meaningful for the functional specification of the service provision) and the related implementation of an ancillary state view management service (with set, reset and, optionally get operations) accessible by the test system as a testability requirement.

In summary, in order to test services architectures, the tester (and the test system) is confronted with two related but distinct hard challenges:

- consider the scenarios that are put in operation by the services architecture and produce for each of them a number of instantiations (test samples); the number of scenarios, even in a services architecture of limited complexity, can be explosively large; the number of test sample for each scenario can be practically unlimited; in conclusion, the quality (size, coverage, test effectiveness and test efficiency) of the test suite (collection of test samples) is a main research challenge;
- execute and arbitrate in the best way a given test suite, whose testing capability is not necessarily guaranteed a priori, and produce a meaningful report of the test session; faced to the quality challenge about the test suite, whatever the test generation methods, strategies and procedures have been employed, and in order to shorten the test sessions and increase their frequency, put in place test sample prioritisation strategies, better if run-time and dynamic, that reduce the fault detection rate.

INTELLIGENT DYNAMIC SCHEDULING OF SERVICE COMPOSITION TESTING

In summary, the only means for increasing the trust of the stakeholders in the dependability of the digital service ecosystems are appropriate testing procedures and processes (see section 2 – § 'The rationale for service testing automation'). But service and service composition testing is undoubtedly a difficult, hard and expensive activity [Canfora and Di Penta 2009] [Bartolini et al. 2011] [Bozkurt et al. 2013]. Furthermore, the *no-testing stance* can be even more costly for service businesses [NIST 02 3 2002]: if the service is the business, non-dependable service is non-dependable business.

First of all, service testing is intrinsically difficult because of information hiding about component internals. Moreover, the engineering methods, tools and technologies that are employed by the service developers and providers are hidden too, and cannot be trusted *a priori*. The establishment and management of inter-organisational service testing (collaborative testing) cycles, procedures and sessions on a multi-owner services architecture is a complex organisational task per se.

The cost of service testing has three components: (i) equipment expense, (ii) labour effort, (iii) time-to-market.

The equipment costs of service testing, including hardware, facilities, software licenses and maintenance, for the SAUT and the test system, are very high, especially for an activity that has traditionally been intermittent. The equipment cost has been a major concern of the research about testing. A solution of this problem is the adoption of cloud computing for testing [Maesano et al. 2013b], which is chosen in the MIDAS project. On the cloud, the marginal cost of computing resources tends to zero.

The labour effort is very high on critical tasks such as (i) production of test cases / oracles, (ii) configuration and set up of test environments, (iii) scheduling and monitoring the test runs, (iv) arbitration of test outcomes, (v) production of meaningful test reports, but also, generally speaking, planning and management of the overall testing cycle. Moreover, most of these tasks require: (i) deep knowledge of the target services architecture specifications and (ii) and high-level skills on testing approaches, methods, strategies and tools. Both competences are really scarce resources. Last but not least, manual testing requires sustained continuous attention and critical stance by the human tester and is error prone.

The last component of the testing cost is *time-to-market*. Service providers and users are confronted to the dilemma between: (i) long, painful and costly testing procedures that can provoke the missing of the market momentum and do not guarantee necessarily the service quality and (ii) precocious delivery of insufficiently tested services, with high business risks. The true solution of the cost and effectiveness problems of service testing is *extreme automation* [Maesano 2013], i.e. pushing the automation of all the testing tasks as far as possible.

Service functional conformance test automation is by definition model-based: what can be automated is the test of the compliance of the SAUT with its formal model. The idea behind extreme automation is that the only "manual" task for the developer is the production of the SAUT models that, once built reduces the marginal cost of human effort to zero.

The subject of this research is dynamic scheduling of service and service composition functional conformance test. Its objectives and constraints are (see section 2 - § 'Objectives and constraints for test scheduling automation'):

1) The test activity to be scheduled is grey-box, functional conformance testing of service compositions.
2) Automated dynamic scheduling has the objective of improving test time efficiency, i.e. the early detection of failures and localisation of faulty elements (*fault detection rate*), in order to allow more frequent automatic testing and shorter focused test session and to enable Test-Driven Development and Continuous Integration Testing.
3) The dynamic scheduler shall be implemented as-a-service, i.e. as a component of the testing facility that can be invoked for scheduling services.
4) The dynamic scheduling approach must be model-driven: the scheduler must be aware of the SAUT structural and functional model and of the structure of the test suite to be scheduled.
5) The scheduler shall be able to manage the uncertainty of the failure seeking and troubleshooting process by means of a probabilistic inference capability. Furthermore, it shall be able to take into account in its decision process evidences from the test run cycle and from other sources (e.g. the design/implementation team) and to manage the computational complexity of the inference.
6) Probabilistic reasoning about testing and troubleshooting shall be piloted by testing objectives, that are expressed through policies.

In summary, the objective of this work is a tool that, on the basis of efficient probabilistic inference, supplies dynamic scheduling services to test sessions of services architectures.

EXISTING APPROACHES

SOA TESTING

The research on SOA testing has flourished in the last fifteen years (see section 3), with a spectacular rise of the number of publications - from 21 in 2004 to 177 in 2010 [Bozkurt et al. 2013] - even if the SOA testing problem has not been explicitly taken into account by the service orientation research mainstream, where SOA testing is not taken into account as a specific activity/problem [Papazoglou et al. 2008].

An important characteristic of research about SOA testing is the *lack of real-world case studies* (between the papers mentioned above, only four of them relate experimentation with real services architectures [Bozkurt et al. 2013]).

From the beginning, the research has correctly recognised some fundamental issues that limit the testability of services architectures and confer to the SOA testing problem its specificity [Canfora and Di Penta 2009] [Bartolini et al. 2011] [Bozkurt et al. 2013]:

- limitations in observability of service code and structure due to information hiding of the implementations;
- lack of control of the services architecture component development life cycles.

The fundamental criticisms about the research thread on service unit and composition testing are:

- many works bear of white-box testing of the software implementing service providers and consumers, falling into the general trend about white-box program testing;
- the WSDL/XSD/SOAP paradigm has been privileged, and the spread of REST/XML and REST/JSON paradigms has been taken into account only very recently and in terms of white-box testing;
- service composition has been taken into account as *orchestration* (BPEL)[5] for a majority of works, sometimes as *choreography* [Bucchiarone et al. 2007], never as direct composition by program, which corresponds to the reality of the API economy. Direct composition between service components (without any superstructure) is utilised in the overwhelming majority of business cases. Furthermore, with the emergence of the API economy, generally based on the REST/JSON technical approach, the BPEL paradigm is on a side residual track.

Research on model-based testing (MBT) is based on the same approaches listed above, so its real applicability is limited. In particular, all the approaches of MBT of service composition emphasise the behavioural models (activity diagrams, state machines, BPEL scripts, etc.) without taking into account the necessity of a separated independent *structural* model of the services, components and services dependences. The SAUT structural model as independent from the functional and behavioural models is a missing topic in the current MBT

[5] Strictly speaking, BPEL scripting has no exclusive relationship with service composition. A BPEL script runs on a participant of a services architecture (the *orchestrator*) and can be replaced by a plain program written in a usual programming language running on a standard server. Testing BPEL scripts utilises techniques from software white-box testing.

research. The SAUT Construction model is an original result of this research. In the context of this research, it is applied to test execution, arbitration and scheduling, and is also applied to test generation in the MIDAS project (see section 4, § 'The SAUT Construction model').

The research topics that are directly relevant for the subject of this work are: (i) *Test run automation frameworks*; (ii) *Regression testing*.

The related work on test run automation frameworks can be classified in two categories: (i) *service unit test automation* equipped with *virtualisation* techniques, (ii) *service integration test automation*.

A service composition, realised explicitly by orchestration or choreography or implicitly by direct service exchange can be performed incrementally by *service component virtualisation*, i.e. by using stub, mock or surrogate services to test the behaviour of the service component nestled in the business process. Apart from the usual dominance of the research on orchestrated (BPEL) service composition, the automated generation of virtual service components in the test system is still poorly investigated, the majority of environments requiring the user to program "by hands" the virtual component.

The first challenge of integration testing framework is the localisation in the service composition architecture of the component whose behaviour is the source of the failure (troubleshooting). The majority of the approaches focus on modelling scenarios of service coordination and exchange (e.g. sequence diagrams, activity diagrams, other) but not explicitly the SAUT structural model, and they are unable to locate precisely the failure and the fault service component at an acceptable level of granularity.

One of the main innovations of the last fifteen years in the domain of test run automation has been the "invention" of TTCN-3, an international standardised complete programming language that has been designed with powerful traits that are specific for testing automation. The research on the utilisation of TTCN-3 for service testing has started early, but its spread remains very limited, because of the technical difficulty of the realisation ad utilisation of effective adapters for SOAP, WSDL, XSD and of the lack of adapters for the JSON format. The author of this manuscript has participated to the specification and implementation of a generic framework that implements the complete automation of the test execution/arbitration of web services and web service compositions that is integrated in the MIDAS platform and supports the test executor and arbiter whose activity is driven by the dynamic scheduler presented in this manuscript [De Rosa et al. 2013].

There has been an important work on SOA regression testing, i.e. testing unintended side effects of a *change* of the services architecture implementation or model. The work has been concentrated on: (i) *test case selection* (*selection*) techniques – that select test cases from a given test suite to test the modified parts of the system and (iii) *test case prioritisation* (*prioritisation*) techniques that schedule test cases for running in an order that attempts to meet some desirable properties. These techniques can be distinguished as *version-specific* (they apply to a specific version of the test/fix/debug cycle) and *general* (they select and/or prioritise in a manner that is independent from the specific version).

Unhopefully, the overwhelming majority of *selection* techniques are dedicated to service composition through *orchestration* (BPEL), so they are BPEL script white-box testing methods and require the identification the modified part of the script.

The main goal of *test case prioritisation* is that of increasing a test suite's *fault detection rate* (*time efficiency*), i.e. the *early* exposure of failures and localisation of faulty elements as the sources of the failures. The prioritisation techniques are based on the analysis of the coverage of *service functional and structural elements* and on the evaluation of the *fault-exposing potential* of the test cases.

In the author's best knowledge, all the coverage-based prioritisation techniques are *non-version-specific* (*general*) and a majority of them is *white-box* and targets service composition through BPEL orchestration. They target *structural* coverage of a BPEL script and the prioritisation of test cases with the highest coverage on the basis of some *coverage criterion*, by an algorithm that implements a *coverage strategy* and without any consideration for the modified part of the SAUT.

The *fault-exposing potential* of a test case would be measured as the probability of the test case of provoking a failure that reveals the faulty component responsible for the failure [Elbaum et al. 2002]. The calculation of these probabilities can be only the result of approximations, and is based either on statistical data from *group testing* - testing several competing implementations of the same service specification - or on the measure of sensitivity of test case to specification and implementation mutation. The weak point of the approaches based on the statistical evaluation in group testing is that it is dubious that the frequency of failure of the test case on a collection of independently designed and implemented service components could be an effective prior probability of failure on a newly implemented service component. The sensitivity of test cases to mutation seems a more promising approach.

This work introduces the dynamic scheduling of functional conformance test sessions as a new research topic in the testing domain and proposes a generalised approach to the dynamic prioritisation of test cases based on probabilistic inference, beyond the current application of static prioritisation to regression testing and re-testing.

PROBABILISTIC INFERENCE AND ITS APPLICATION ON TESTING AND TROUBLESHOOTING

The probability theory has shown to be the most promising framework for representing uncertainty within the decision mechanism, thanks to its ability of modelling with maximum accuracy and minimum number of parameters a complex reality. The hypothesis behind this research is that probabilistic inference can drive dynamic test scheduling that improves failure seeking and troubleshooting (see section 5).

PROBABILISTIC INFERENCE

The most popular models for probabilistic inference are the Markov Network and the Bayesian Network models. Their objective is to map conditional independence so that it is possible to factorize joint probability distributions in a framework that allows iterative and interactive calculation of marginal probability distributions through probabilistic inference.

Probabilistic Inference is method of calculating an updated state of the probabilistic model according to the observation or not of evidence realizations, the evidence over a variable being a measure of likeliness of an instantiation of the variable. It is possible to distinguish between *hard* and *soft* evidence. Hard evidence expresses total knowledge over the state of the variable: the likeliness of one instantiation is equal to 1 (maximised) and all other instantiations equal to 0. Soft evidence expresses a belief or a partial knowledge of the state of the variable and permits a distribution over all instantiations with at least two instances' likeliness different from 0.

When inserting evidence in a BN or a MN the result is the transformation of respectively the variable probability distributions and the factors.

There are several (families of) algorithms implementing probabilistic inference. The “classical” inference algorithms are those belonging to the *variable elimination* family and to the *junction tree inference* family.

The *variable elimination* (VE) inference is a simple but powerful and efficient algorithm that exploits the factorization from the joint probability distribution and try to eliminate random variables by considering only a sub-set of factors algorithm [Zhang and Poole 1994] [Zhang and Poole 1996] [Dechter 1998]. Between the methods variants and extensions there are: (i) *bucket elimination* [Dechter 1996] [Dechter 1998] [Darwiche 2010], (ii) *generalisation to junction trees* [Cozman 2000], (iii) *value elimination* [Bacchus et al. 2003]. *Variable elimination* is query-sensitive (the entire data structure must be re-initialised at each new query), whereas the *generalisation to junction tree* avoid re-runs.

The junction tree is a data structure represented by a cluster graph. Each cluster is initialized by knowing its local *potential* and its neighbours and sends one message, which is a *potential function*, to each of its neighbours. It is able to compute the marginal probability of its variables by combining its local potential with the messages it receives.

The Shafer-Shenoy inference algorithm transforms MN into junction trees and computes marginal probabilities using a message-passing scheme [Shenoy and Shafer 1986] [Shenoy and Shafer 1988]. Junction tree algorithms were among the first inference schemes for BNs. Currently, the most performant junction tree algorithm is *lazy propagation* [Madsen and Jensen 1999].

Junction trees algorithms and VE share the same complexity limitation due to the MNs tree-width. One approach to overcome the limitation induced by high tree-width for exact inference is to exploit local structures, i.e. alternate representations of factors exploiting local symmetries in the probability distribution.

A new family of inference algorithms compile MNs into *Arithmetic Circuits* (AC) via the transformation into a Conjunctive Normal Form (CNF) [Darwiche 2003] [Chavira and Darwiche 2005] [Chavira et al. 2006] [Chavira and Darwiche 2007].

A hypothesis of this research is that Bayesian inference by AC compilation is the best suited for dynamic scheduling applied to SOA testing, in particular for the advantages in terms of data structure size and inference time. The compilation time could be important, but its result can be saved (AC image) on disk and reused as many times as needed. Generally speaking, in services architectures the service functional and behavioural model doesn’t change slowly and, in the Continuous Integration Testing, it is possible to retest modified implementations towards the same SAUT and with the same Test Suite, i.e. to schedule the test session with the same AC image on a daily (nightly) basis with zero compilation costs and a rapid inference time. The AC image can also be used to suspend and resume a test session. This research has developed an alternative approach of the compilation that is presented in section 6 and whose experimental results are presented section 7.

In their position paper *at FSE/SDP workshop on Future of software engineering research* (2010), 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 [Namin and Sridharan 2010]. Building on prior seminal work, they consider the efficiency and effectiveness test case generation and the test prioritisation the most important application fields for Bayesian reasoning methods. The research on probabilistic inference in software testing and related domains such as software quality assessment is still in its infancy and is made of a few disparate tentative works on the different aspects of the discipline.

A seminal work about the use of probabilistic inference based on a BN framework to support input partitioning test methods [Rees et al. 2001] [Wooff et al. 2002] [Coolen et al. 2007] is aimed at understanding which kind of stimuli provokes software failures. The BN allows the tester to quickly discover what partition or combination of partitions can be associated with a failure. On re-testing, since the model identifies which parts of the system are unconnected to those where the test has failed, it can also indicate which tests can continue to be run before the fault needs to be located.

A novel approach to prioritizing test cases in order to enhance the fault detection rate within white-box testing is based on Bayesian Networks [Mirarab and Tahvildari 2007]. The idea is to incorporate source code changes, software fault-proneness, and test coverage data into a unified model. In this approach, dynamic scheduling (incorporating as evidence the feedback from the test system for each test run) is considered as one of the most important subject of their future research. In the author's best knowledge, there are not yet published results of this further research.

In diagnosing defective systems, the primary goal is to isolate the faults that best explain the symptoms in the most efficient way possible. Decision systems are useful in this context because they can model real world problems with high accuracy and can be designed in a transparent way, facilitating the coordination between experts and users.

Several methods and techniques (rule-based reasoning, case-based reasoning, neural networks, decision trees, Bayesian Networks, others …) can be used for troubleshooting complex systems. The Bayesian Network model presents several advantages: (i) well-founded explicit modelling of uncertainty in complex real-world domains where exact inference is intractable [Mirarab and Tahvildari 2007]; (ii) mathematically well-defined mechanism for representation and results that can be mathematically proven [Dechter 1996]; (iii) management of multiple hypotheses about the state of the target; (iv) capability of incremental integration of additional information about the state of the system; (v) knowledgeable mechanisms, in contrast with other black-box methods (neural networks, genetic algorithms); (vi) absence of combinatorial problems of the formulation (in contrast with decision trees); (vii) utilisation in industrial applications [Cooper et al. 1998] [Jensen et al. 1995].

Any complex system can be defined as a set of components, where any component can be a potential source of failures. In order to determine which component may be faulty, a set of actions is performed on each component. Actions can be defined as *passive* or *active*. Passive actions (Observing, Questioning, Testing …) involve information gathering and do not affect the system itself. Active actions (Repair…) influence the system by making changes. These actions are invoked based on the results of the passive actions. A *strategy* is a set of actions executed in a specific order.

Troubleshooting is an iterative and interactive process, which continually integrates and applies new information, or evidence, in order to determine the next action. A good strategy minimizes the time and cost (of testing) required to isolate and to fix the faulty components.

The structure of any system to be diagnosed can be represented as a directed acyclic graph (DAG), which shows causal relationships between symptoms, components and troubleshooting actions. Causes and effects can be connected using two main model categories: (i) the Naïve model (single fault assumption), (ii) the Causal Independence model (allowing multiple faults).

By using a BN, the relationships between the different elements of the analysis are probabilistic rather than deterministic, allowing a greater simplification of the system. Stochastic methods are considered appropriate for failure detection and diagnosis of complex systems in cases where there is no complete knowledge of the system, i.e. the detection and diagnostic process is undertaken in presence of uncertainty and the evidence data domain is too large to be completely analysed [Nielsen et al. 2000].

Troubleshooting of complex system with BN is henceforth a well-established discipline and practice and some significant realisations are presented in section 5. The hypothesis of this work is that, in the domain of grey-box testing of services architectures, the discipline of troubleshooting of complex system with BN and probabilistic inference can support the dynamic choice of test samples to expose failures and the localisation the faulty structural and functional elements that are the sources of the revealed failures.

This work contributes to the research on the application of Bayesian reasoning methods to the "reliable and efficient software testing and program analysis" [Namin and Sridharan 2010] that is considered an "ideal research paradigm" and a promising future research thread on software testing. In fact, the author thinks that the problem of intelligent dynamic scheduling of test sessions can be posed and solved only through probabilistic inference, because it is too complex for exact inference.

THE BAYESIAN TEST SCHEDULER AS A SERVICE

The Dynamic Scheduler is packaged as a Web service on the MIDAS platform that is able to provide scheduling services to different MIDAS test methods.

The MIDAS testing facility (see section 4, § 'The MIDAS testing facility'), targets black-box testing of single services and grey-box testing of services architectures. Its key features are:

1. Testing-as-a-service implemented on a public cloud infrastructure.
2. Programmable testing facility through service APIs.
3. Open platform equipped with an evolutionary registry/repository of test methods.
4. Extreme automation of SOA/API testing tasks.

The end users of MIDAS are service developers that utilise the MIDAS facility to automate the test of the services architectures. They are able to integrate the MIDAS facility in a unified Service Development Life Cycle characterised by: (i) Model-based design and test of services and services architectures; (ii) Test-Driven Development (iii) Incremental Integration and Continuous Integration Test.

Dynamic scheduling can be understood as a schedule/execute/arbitrate cycle, whose description is detailed in section 4, § 'The logical architecture of the dynamically scheduled test run cycle'). The *Scheduler* chooses the next *test case* to be executed - starting from a first one that it chooses on the basis of a *prioritisation policy* - and communicates it to the *Executor*. The *Executor* runs the *test case*, collects/logs the *test outcome* and communicates this outcome to the *Arbiter*. The *Arbiter* evaluates the *test outcome* with the test oracle and produces a *test verdict* that is communicated back to the *Scheduler*. The *Scheduler*: (i) either chooses another not yet executed *test sample* from the test suite, communicates its identifier to the *Executor* and the cycle continues, (ii) or decides to halt the cycle even if there are still *test samples* to run, (iii) or ends the cycle - there are no more *test samples* to run in the suite. This conceptual architecture of the schedule/execute/arbitrate cycle has been proposed by this research as a framework for the implementation of probabilistic methods and tools for test scheduling and is implemented in the MIDAS facility.

The implementation architecture of the MIDAS facility is generic and service oriented (see section 4, § 'The scheduled test run services architecture pattern'). All the components that implement test methods are realised as service providers in a flat (non-hierarchical) architecture that includes the basic components that realise the basic test tasks (test case generator, test oracle generator, test executor, test arbiter, test reporter, test scheduler) as well as the intermediate drivers (test generation workflows, test run managers) and the top driver.

With automated and intelligent dynamic scheduling available on a testing platform as a service [Maesano and De Rosa 2011] [Maesano et al. 2011] [Maesano et al. 2013] that already implements automated test execution and arbitration, the entire *test run cycle* can be automated and programmable. This automated test run cycle can be executed in background, allowing service developers to focus on their primary mission: design and implement appropriate services. This research is focused on the design and development of an automated intelligent scheduler that can unburden the service developer of the test run cycle implementation and management.

The Scheduler is a component that exposes a service interface (see section 4, § 'The Scheduler interface'). It is considered by the other components as a scheduling service. The concrete architecture of the automated test run cycle is composed of a Runner that receives from the Front End the test task requests and orchestrates the activity of the Scheduler service, the Executor/Arbiter service and the Reporter service. The execution and arbitration tasks are carried out by a unified service that, after an initialisation phase, is able to receive the indication of the test case(s) to run and to return the test verdicts. In summary, the Runner implements the logical architecture of the schedule/execute/arbitrate cycle by orchestrating the activities of the Scheduler and the Executor/Arbiter (see the sequence diagram in Figure 15).

The test scheduling approach of this work is model-based. In order to initialise and configure its embedded inference engine, the Scheduler utilise the structural model of the SAUT (SAUT Construction model) the model of the test scenarios (the Test Suite Definition model) and the Test Suite data set (see section 4, § 'The Scheduler configuration models and data sets').

The SAUT model (see section 4, § 'The SAUT Construction model') is a simple and powerful model that integrates the structural model of the SAUT (descriptive model) and the configuration model of the test system (prescriptive model). It is utilised (with the Test Suite Definition Model and the Test Suite data set) to configure and initialise both the Executor and the Scheduler and to bind the deployed SAUT with the test system (the Executor).

The SAUT model is built with a limited number of elements: *components*, *references*, *services*, *wires*. Components are nodes of a directed graph that expose services (provided interfaces), declare references (required interfaces) and are linked by wires that are edges from references to compatible services. Components are *actual* – actually deployed as components of the SAUT - or *virtual* – mock-up components that are put in place by the test system. Other elements of the SAUT Construction meta-model (*Atomic Participant*, *Compound Participant*) allow modelling component *structures*, i.e. declared references and exposed services and the pointers to the service specifications (e.g. WSDL/XSD documents) and, for *Compound Participants*, their recursive composition as a graph of actual components with declared references and exposed services that are linked by wires. A complete example of SAUT Construction model that applies to the services architecture depicted in Figure 2 p. 12 is developed in section 4, § 'The SAUT Construction model'.

The test scenario model contains the definition of scenarios, such as those depicted with sequence diagrams in the preceding § 'An example of services architecture: the Interbank Exchange Network', that are sampled in the associated Test Suite data set. Generally speaking, the scenarios *sampled* in the Test Suite are a subset of all the possible scenarios of service interactions in the associated SAUT model. The Test Suite data set contains a number of test samples for each scenario of the Test Suite Definition. Each couple test scenario modell / test suite data set has a *test coverage measure* of the SAUT.

An appropriate model of the problem domain is a crucial element for the success of the probabilistic inference application. This work proposes a simple but deep model of the problem domain, the grey-box functional conformance testing of distributed services architectures, that is able to take into account both the fault-exposing potential of the test cases and the test coverage of structural and functional decompositions of services architecture at the lowest possible level of granularity, given the black-box/grey-box stance of service testing.

The Scheduler decision module is an Arithmetic Circuit inference engine (see section 6). The construction of the Arithmetic Circuit is the job of the Scheduler initialiser (see section 4, § 'Initialisation request') and produces the Arithmetic Circuit structure through three logical steps:

1. It extracts a view of the SAUT model, of the scenario model and of the test suite data set (called a Test Scheduling Context model – TSC), that includes: (i) a structural/functional decomposition of the SAUT and (ii) a model of the Test Suite data set (see section 6, § 'Building the Test Scheduling Context model ').
2. It builds a virtual Bayesian Network (vBN) from the TSC model by associating random variables to TSC elements and stochastic dependences to TSC relationships (see section 6, § 'Building the virtual Bayesian Network by model transformation').
3. It compiles the vBN into an Arithmetic Circuit (AC) by using an original compilation algorithm (see section 6, § 'Building the Arithmetic Circuit by compilation').

The vBN includes variables corresponding to the structural/functional decomposition of the SAUT, such as components, ports (references/services), message types, and to the test suite: test samples (collections of message oracle), the messagte oracles. These variables ore organised by the statistical dependences. The model allows the finest granularity of the test coverage and the evaluation of the fault-exposing potential of each elementary oracle of a test sample.

This research propose an original algorithm for the technique of the inference by compilation [Darwiche 2001], which seems particularly suitable for mastering the computation complexity problem of test scheduling. The principle of inference by compilation is straightforward: from the *chain rule* [Darwiche 2003] follows that any Bayesian Network (BN) can be represented by a multi-linear function (MLF) with specific properties. An MLF is the sum of an exponential number of terms: there is a term for each possible instantiation of the network variables and the term is the product of the evidence indicators and the network parameters of the instantiation. The Arithmetic Circuit (AC) is a representation of such a function that facilitates its inference computing.

An interesting feature of the AC is that it allows avoiding re-computations of operations within sub-circuits. The MLF contains all the information about the variables and the dependences, so that, all the answers to probabilistic queries can be obtained by evaluating and differentiating the function. The structure of the Arithmetic Circuit with respect to the compiled vBN is detailed in section 6. The compilation method proposed in this research is a recursive dynamic algorithm detailed in section 6, § 'The first steps of the Scheduler initialization phase end with the construction of the vBN that is in fact a model of a BN (in the sense that the representation does not correspond with any optimized representation of "executable" BNs). The vBN model is stored in a light XML representation. Figure 41 sketches an example of vBN .

The classical Bayesian inference approach reaches its limits in size and computation speed very quickly with the increase of the number of: (i) **Participant**s, (ii) **SendingPort**s, (iii) **InteractionType**s, (iv) **Interaction**s and (v) **TestSample**s. The proposal of this research is to "compile" the classical representation of the Bayesian Network in a more compact structure (the Arithmetic Circuit), adapted to more efficient inference computation. A concise description of an original model-driven method for compiling a BN into an AC follows.

As it was explained in chapter 4 a BN uses is a graphical representation of a joint probability distribution that can be viewed as a MLF. The AC is a factorized version of the MLF that avoids redundancies like repeating sub-circuits. A MLF is the sum of an exponential number of terms where each term is an instantiation of all network variables multiplied by the probability parameters expressing their mutual dependences.

Returning to Figure 41, it is important to highlight that the **TestSample**, **InteractionType**, **SendingPort**, **Participant** and **System** BN nodes are OR nodes (they are instantiated to the value 1 if at least one of the parent nodes is instantiated to 1). This implies that the probability tables of the dependences contain values equal to 1 and 0. Consequently, some of the terms of the MLF vanish due to a value equal to 0.

After this initial simplification it is possible to observe that all remaining terms express all possible instantiation of the observed **Interaction**s, the input information, and the consequences on the state of the rest of the network variables.

$$f(V) = \{\lambda_{\bar{m}_0}\lambda_{\bar{m}_1} \dots \lambda_{\bar{m}_n}\}\{\lambda_{\bar{g}_i} \dots \lambda_{\bar{g}_v}\}\{\theta_{\bar{m}_0}\theta_{\bar{m}_1} \dots \theta_{\bar{m}_n}\} + \cdots + \{\lambda_{m_0}\lambda_{m_1} \dots \lambda_{m_n}\}\{\lambda_{g_i} \dots \lambda_{g_v}\}\{\theta_{m_0}\theta_{m_1} \dots \theta_{m_n}\}$$

Each term of the function can clearly be organized in three groups of variables:

$\{\lambda_{m_i}\ ;\ \lambda_{\bar{m}_i}\}$ - **Interaction** evidence indicators that represent the two possible instantiations of **Interaction** (expressing the verdicts *pass* or *fail*).

$\{\lambda_{g_i\ i}\ ;\ \lambda_{\bar{g}_i}\}$ - Evidence indicators for the other variables (**System**, **Participant**s, **SendingPort**s, **InteractionType**s and **TestSample**s). They also represent the two possible state of the related Bayesian Network variable.

θ_{m_i} – Network parameter consistent with the state of the interaction m_i and the topology of the vBN. Instead of the network parameters of the other BN variables, those network parameters are not removed during the simplification because their value is different from 0 and 1. For each interaction, the network parameter can belong to one of two categories:

- $\theta(m_i|\boldsymbol{U})$ – $\boldsymbol{U}$ being the state of all parents node, in the case of non-observed **Interaction**s the values of the **Interaction** instantiations influence the values of the preceding non-observed **Interaction**s.
- $\theta(m_i)$ - In the case of observed **Interaction**s the value is defined by the expert as an a priori opinion on the state probability of failure of the **Interaction**s.

The morphology of the MLF can be explained by transition properties of the BN. In fact, since the state of the **System** depends of the state of the **Participant**, the state of the **Participant** depends of the state of the **SendingPort** and so on (see Figure 41), the state of any element of the SAUT can be determined by the state of the cluster of **Interaction**s that are linked to it.

’. At the end of the initialisation phase, the Scheduler is equipped with an Arithmetic Circuit that allows it to behave as a Bayesian agent.

The inference engine on the AC is an internal module of the Scheduler. To each execute/arbitrate cycle (with test samples as inputs and test verdicts as outputs) corresponds a schedule inference cycle (with test verdicts as inputs and test samples as outputs). The Scheduler manages its internal inference engine by setting: (i) prior probabilities on its top variables at the initialisation and (ii) *assumptions/beliefs/observations* (*evidence realisations*) on the other variables potentially at any inference cycle.

At each inference cycle, the AC engine utilises a “double traversal” algorithm [Darwiche 2003] to calculate: (i) $P(\boldsymbol{e})$ and $P(X, \boldsymbol{e} - X)$ in a first traversal and (ii) $P(X|\boldsymbol{e})$ in the second traversal, X being any variable not instantiated in the evidence $\boldsymbol{e}$.

The Scheduler can put in place, on the basis of the AC inference, different “generic” scheduling policies based on different evaluations of the *fitness* probability distribution of the test samples that produced by the inference cycle, and different “generic” halting policies, based on the objectives of the test session related to the test/design/development cycle.

Three “generic” scheduling policies are put in place by the Scheduler for the choice of the next test sample to run: (i) *max-entropy policy* – the test sample with the maximum entropy [Khinchin 1957] of its *fitness* probability distribution (ii) *max-fitness policy* – the test sample with the max *fit* probability, which can be interpreted as a measure of the test sample fault-exposing potential, (iii) *min-fitness policy* – the test sample with the min *fit* probability.

Four generic halting policies are put in place by the Scheduler: (i) *n-fail-halt policy* – halt at the n-th test sample execution whose global verdict is *fail*, (ii) *n-misfit-halt policy* – halt at the n-th test sample execution whose global verdict is *pass*, (iii) *entropy-threshold-halt policy* – halt when the entropy of all the variables of a collection is less than an established threshold; (iv) *no-halt policy* – the test session stop only when there are no more non-executed test samples.

Some experimental results of the original algorithm presented in this manuscript, conventionally called **inf4sat** and some comparisons through simulation campaigns with classical probabilistic inference algorithms (e.g. Lazy-Propagation and Gibbs Algorithme) and the Darwiche algorithm of inference by compilation have been collected in terms of: (i) AC size, (ii) inference speed, (iii) compilation speed. The inf4sat algorithm is better that its "competitors" in memory size, robustness and inference speed, and pushes further the limits in terms of tractability of the network complexity. It is less performant than its competitors in terms of compilation speed. This trade-off is adapted to the testing domain where the service specifications are generally stable, but the service implementations are built through software engineering iterative processes such as Test Driven Development and Continuous Integration Testing, and is extremely well adapted for continuous re-testing (checking that the failures exposed in the last test session have been fixed) and regression testing (testing that the new implementation version doesn't produce non-compliant side effects). The compilation result (the *AC image*) is compiled once, and can be compiled asynchronously with respect to the test sessions, saved and reused in hundreds, thousands … of test sessions, where the probabilistic inference could be an effective tool for improving the test detection rate (the early exposure of failures and the localisation of the faulty components that are the sources of these failures), and the inference speed is a critical factor for shortening the test session duration and for augmenting the test frequency.

This Scheduler is integrated in the MIDAS platform and its utilisation within test sessions on the MIDAS Pilots is ongoing. The MIDAS Pilots are real and complex services architectures in the health and in the logistics domain that are used as real-world case studies for the MIDAS platform. The author is confident that the Scheduler implementation and usage (*policies*) in relationships with different testing objectives and approaches on two real world case studies will be confirmed and improved.

BACKGROUND AND CONTEXT OF THE RESEARCH

This research has been initiated within a joint project (BN4SAT – Bayesian Networks for Services Architecture Testing) of the Université Pierre et Marie Curie – Sorbonne Universités (UPMC), the Centre National de la Recherche Scientifique acting within the Laboratoire Informatique de Paris 6 (LIP6) and Simple Engineering France (SEF), a small company specialised on SOA engineering. The project was partially funded by the ANRT (Association Nationale de la Recherche et de la Technologie – Convention CIFRE n° 314/2009).

Sometime before the beginning of the BN4SAT project, Simple Engineering France had started utilising the TTCN-3 language [Willcock et al. 2011] for the automation of the test of complex services architectures. Today, the powerful features of the TTCN-3 language and of the associated tools[6] allow considering the automation of test execution and test arbitration tasks with TTCN-3 programming as a solved problem [De Rosa et al. 2013]. However, the specific characteristics of the services architectures make their testing hard, costly and difficult-to-manage, even when the full automation of the testing execution and arbitration is attained.

On the other hand, with the accelerated development of the digital service economy, SOA testing becomes more and more compulsory. It is the only means for increasing the stakeholders' trust in the services architectures that automate the processes of the digital economy. Hence, the decision was taken to go further

[6] http://www.testingtech.com/

to the automation of higher level tasks such as the dynamic prioritisation of test cases, i.e. dynamic test scheduling, to facilitate and to shorten the test/design/implement cycle in a context of Test-Driven Development and Continuous Integration Testing. In order to satisfy this requirement, the DECISION team of the LIP6 has proposed to investigate the application of probabilistic inference to the automation of the SOA test dynamic scheduling. The BN4SAT project has produced and a first prototype [Maesano and De Rosa 2011] [Maesano et al. 2011] and a positive assessment on the possibility of applying Bayesian inference to the test scheduling automation problem.

These first results have permitted to enlarge the perspective on SOA testing automation to all the testing tasks and to the full Model-Based Testing (MBT) approach. UPMC and SEF have promoted a European Consortium, including: (i) Academic and Research institutions such as Fraunhofer Fokus Institute (DE), University of Goettingen (DE), Consiglio Nazionale delle Ricerche (IT), ITAINNOVA - Instituto Technologico de Aragon (ES), (ii) Companies such as Dedalus S.p.A. (IT) and T6 (IT) and (iii) Non-profit organisations such as Sintesio (SI). This Consortium has issued a proposal for the EC FP7 Call 8 (Activity: ICT-8-1.2 - Cloud Computing. Internet of Services and Advanced Software Engineering) and has obtained a grant for a three year STREP Project MIDAS (Model and Inference Driven Automated testing of Services architectures - Project Number 318786).

The MIDAS project aims at developing a SOA testing facility delivered as Testing as a Service on cloud [Candea et al. 2010] [Floss and Tilley 2013] [Maesano 2013], enabling the model-based automation [Schieferdecker 2012] of all the SOA testing tasks (test case/oracle generation, test execution, test arbitration, test reporting, test scheduling and test planning). The focus is on functional conformance testing, security/vulnerability testing and usage-based testing.

A specific characteristic of the MIDAS project is the presence in the Consortium of two partners - ITAINNOVA and Dedalus - whose specific task is to put in place real complex services architectures (the pilots), respectively in the logistic (supply chain management) and health domains, and to utilise directly the MIDAS facility to test the pilots. These partners play the role of users of the MIDAS testing facility from the beginning, when it is still work in progress. The double advantage is that the MIDAS test methods are confronted to the complexity of the real services architectures – double complexity of the service composition network and of the data structures that are exchanged – and that the very early integration of the users in the project fosters the effectiveness and usability of the MIDAS testing facility. It must be said that one of the great problems of the SOA testing research is the *lack of real-world case studies* [Bozkurt et al. 2013] and that one of the main reasons that have pushed UPMC/LIP6 and SEF to build a Consortium and to candidate for a grant with the MIDAS project is the search for real-world case studies for the research that has started with the preceding BN4SAT project.

Hence this research has continued in the context of the MIDAS project, where UPMC is leading the Work Package 5 (Intelligent planning and scheduling of SOA test campaigns), with the financial support of the European Commission.

THE STRUCTURE OF THE MANUSCRIPT

Section 2 provides a description of the problem statement concerning SOA testing and SOA test scheduling. After a general introduction of the service approach of distributed applications as the main design paradigm of the digital economy, the importance and peculiarity of SOA testing, as black-box testing of single services and grey-box integration testing of service compositions, is highlighted. The section claims that effective and efficient testing is the only means for service developers and service users to improve the stakeholders' trust on the service digital ecosystem. In particular, functional conformance testing, i.e. checking the actual behaviour of single services and service compositions towards structural, functional and behavioural models, that are shared between service designers, providers, integrators and users is motivated as a basic step of trust building. This section introduces key concepts and defines main terms of SOA testing (test case …), and also key model concepts and terms such as service interface, service composition topology, service interaction protocols, service pre/post-conditions, data flow requirements. These definitions introduce the SOA testing task decomposition. The testing tasks, such as test generation, execution, arbitration, scheduling, reporting and planning are described and discussed as human as well as automated processes, in relations with the service life cycle, and in particular with the cycles that put in place Test-Driven Development and Continuous Integration Testing. The section highlights the difficulty and cost of SOA testing, the benefits of SOA test automation and, in particular, of the automated dynamic scheduling of test sessions and summarises the objectives and constraints that have driven this research.

Section 3 presents and discusses the related work about SOA testing. After a general presentation of the research on SOA testing and on the impact of the more recent paradigm of model-based testing on SOA testing, it focuses on the research threads that are directly related to this work on automated dynamic scheduling, i.e. (i) the test run automation thread and (ii) the regression testing thread. In the test run automation is shown that the research on TTCN-3 and some TTCN-3 frameworks pushes very far the automation of the test execution/arbitration on services and services architectures. There is no specific thread on dynamic test scheduling (the test run automation frameworks of the literature implement static scheduling) but the section presents relevant research about test case *selection* and *prioritisation* (static scheduling) of test cases in regression testing, always with the objective of improving the fault selection rate and the test coverage, in relation with the test regression main objective that is checking that maintenance and evolution actions on the services architecture under test do not produce unwanted side effects - but also with the other important objective that the implemented modifications are effective in correcting known faults. The research review shows that the efforts on regression testing share with this research some objectives (the early exposure of failures and the early detection of faults provoking the failures) and some concepts (fault index of service structural and functional elements, fault-exposing potential of test cases).

Section 4 presents the general architecture of the test run automation in the context of the MIDAS project, the place of the scheduler in this general architecture and its relationships with the other components, the scheduler service behaviour and interface and, in particular, the modelling elements that have been proposed by this research (the SAUT Construction model and the Test Suite data model) and are utilised to initialise the scheduler for a specific test session. These modelling approach is today adopted in the more general project (MIDAS) in which this research is integrated. After a short presentation of the MIDAS facility, and of the MIDAS internal services architecture, the logical architecture of the schedule/execute/arbitrate cycle and its realisation as services architecture on the MIDAS platform are presented. The scheduler service interface is presented and its operations are illustrated (with reference to the scheduler WSDL in the annex – section 0). The section ends with a detailed description (including a formal definition) of the SAUT Construction model and of the Test Suite data structure model and a some hints about the test system configuration, the test run and the structure and meaning of the test verdicts, that are the main input to the probabilistic inference cycle that drives the dynamic scheduling service.

Section 5 presents the related work about the probabilistic approach of the test dynamic scheduling. The section is composed of four parts. In the first part, the general approach related on the use of conditional independence and of probabilistic models (Bayesian networks, Markow networks) is presented and discussed. In the second part, the probabilistic inference is introduced, and its methods and algorithms are detailed and discussed, in particular *variable elimination* and its variants, *junction tree* algorithms and variants, the so-called *Shafer-Shenoy* and other approaches. In the third part, the motivations, methods and algorithms of the inference by compilation approach are presented and discussed. The last part of the section documents the use of probabilistic inference in testing and troubleshooting. The use of Bayesian models in different testing activities is still at its infancy but, on the basis of some interesting realisations an influential position paper published in 2010 [Namin and Sridharan 2010] considers Bayesian reasoning methods “an ideal research paradigm” for the future of the testing research. This inspiring work, in particular the applications on selection and prioritisation of test cases, is presented and discussed, with the fact that dynamic selection and prioritisation are considered important topics of future research. The section ends with the presentation and discussion of the troubleshooting approach through probabilistic inference that constitutes today a well-established research discipline.

Section 6 presents the central element of this research that is the probabilistic inference engine core implementation. The inference engine implements inference by compilation that is based on an Arithmetic Circuit representation (inf4sat). The section starts with a description of the model mapping and transformation process that, starting from the SAUT Construction model and the Test Suite model and data set end with an intermediate structure, the virtual Bayesian Network and shows how the Scheduler initialiser builds a *view* (the Test Scheduling Context model) of the aforementioned models that exhibits the structural and functional decomposition of the SAUT, the individual test samples of the test suite and their relationships. Then it associates to the elements of this model the appropriate random variables and to their relationships the stochastic dependences. In the second part the compilation step that transforms the virtual Bayesian Network into the Arithmetic Circuit is presented and the compilation methods as well as the target structure are detailed.

Section 7 presents the use of the inference engine as a driver of the test schedule/execute/arbitrate cycle by the scheduler and introduces and discusses the concept of scheduling and halting policies. In particular, the peculiarities of the inference mechanism and of its utilisation by the scheduler are presented, together with the three “benchmark” scheduling policies (the max-entropy policy, the max-fitness policy and the min-fitness policy) and the three main halting policies (n-fit-halt policy, n-misfit-halt policy and entropy-threshold-halt policy). Experimental results and comparison of the inf4sat algorithm of inference by compilation detailed in the section 6 and its main “competitors” is described in the second part of the section. The results of evaluation campaigns, in terms of Arithmetic Circuit size, inference speed and compilation speed.

2. SOA, SOA TESTING AND SOA TEST SCHEDULING

SERVICES ARCHITECTURE

In the *digital economy* tens, hundreds, thousands, … applications, systems, devices are connected and collaborate without human intermediation, putting in place the automation of business processes that support economic, social and administrative activities. The dependability and security of such digital ecosystem become more and more a critical issue [Brian Arthur 2011].

Service orientation is a *design and implementation style* [SoaML 1 0 1 2012] that allows organizations to put in place dynamic and automatic collaborations of distributed, autonomous, heterogeneous and loosely coupled systems, applications and devices in order to achieve flexible, dependable and secure business process automation.

Within the service oriented style, the collaboration among distributed systems, applications and devices is carried out automatically – without human intermediation - via the exchange of *service provisions* through remote *Application Programming Interfaces* (APIs). A service can be defined as an *activity that has an effect in the real and/or digital world* (the *service provision*) carried out by a system acting as a service *provider* for or in behalf of another system acting as a service *consumer* [SoaML 1 0 1 2012]. The service provision is coordinated by the interaction between the provider and the consumer through the service APIs. A *services architecture* is a network of participant systems, applications and devices that exchange service provisions in order to achieve business goals.

A service can be intended as described by a *service specification*. A service specification is a description of: (i) *interfaces* – required and provided by the provider and the consumer; (ii) *behaviours* - the protocols and contents of the interactions between the provider and consumer, i.e. the *behaviours* that are *observable* at the interfaces; (iii) *functions* – the service provisions. The service specification should also include security and quality of service aspects. Generally speaking, the interface description (the APIs) is, by definition, well-defined, *formal* and *machine readable*, while the functional and behavioural descriptions are often partial and informal (in general, they are supplied as API documentation).

Whether complete or partial, formal or not, a *service specification* is a *black-box model* of the system that claims to implement it, either as a service provider or as a service consumer. It does not include any information about the system implementation. Systems that are built following different *constructional* (*white-box*) *specifications* and that are implemented on different *technological platforms* can be able to fulfil the provider or consumer role of the same service specification.

Specifications can be represented by *models* of the specified object or process. Service engineering, as several other engineering activities, is always *model-based*, either implicitly or explicitly. Analysts and designers always build a *mental model* of the service that they have either to specify/design (*prescriptive model*) or to analyse/assess (*descriptive model*). When a model is represented explicitly, even informally, it is potentially communicable between humans. When a model is represented through a formal system of signs (the *meta-model*), it is also communicable to a machine [Bézivin 2005]. Formal models can be processed in different ways. They can be: (i) checked for some relevant properties, (ii) transformed until automatic generation of parts of the digital system, (iii) utilised to generate automatically test cases, test oracles, test system components. For example, a WSDL document [WSDL 1 1 2001] is a partial service specification - only the interfaces are concerned - that describes through a machine readable model the service APIs on a specific interoperability platform [SOAP 1 1 2000]. It can be used to generate code such as the consumer *proxy* and the provider *skeleton*[7] and to generate templates of compliant SOAP messages[8].

The service oriented approach of the cooperation between distributed systems is characterized by a sharp separation between *service specifications* (service models), *system specifications* (system models) and *system implementations*. A service model can be considered a *black-box model* of the system that claims to implement the service, whereas the system model is a *white-box model* of the system. The service, black-box model is made accessible to the stakeholders - the service providers and the service consumers - and works as a "contract" between them, whereas the system, white-box models are concealed and private to the system owners. Note that hiding the system white box model and the system implementation is not only a confidentiality issue, but also and, we can say, mostly, a way of mastering the complexity of the implementation of large systems [Parnas et al. 1983].

Furthermore, the system white-box model and implementation cannot be derived mechanically from the black-box models of the services that the system implements. This impossibility constitutes the true limitation of the model-driven approach of software and system engineering[9]: the analyst can build a *formal black-box (functional) model* of the service; the designer can build a *formal white-box (constructional) model* of the system that is supposed to implement the service; possibly, the software implementation can be derived mechanically from the constructional model[10]. The problem is that, in the most general case, the white-box model and its implementation cannot be *derived mechanically* from the black-box model, and the fact that a white-box model implements correctly a black-box model cannot be formally proved [Dietz 2010]. In a situation in which the implementation is hidden and cannot be mechanically derived from the service specification, how can we increase the stakeholders' trust in the conformance of the system behaviour to the service specification?

What can be done with a formal black-box model is to sample its behaviour, i.e. to generate (manually or automatically) an *extension* of it [Rapaport 2012]. A collection of *samples* of the *behaviour of the system at the interfaces* (service samples) that are compliant with the model can be used to check the actual system behaviour, by comparing them with snapshots of the actual behaviours at the interfaces. As the service user is unable to inspect the implementation and to prove the compliance of the implementation with the service specification, the *only way* at her/his disposal to increase its trust in the system conformance to the service specification is *testing the system implementation against the service specification*.

[7] http://axis.apache.org/

[8] http://www.soapui.org/

[9] http://www.omg.org/mda/

[10] In any case, AIT (Algorithmic Information Theory) confirms that a complete constructional model cannot be significantly shorter than its software implementation [Lewis 2001].

Services are the building blocks of the digital economy. Modern applications are *composites* aggregating not only their own internal components but also *private*, *partner* and *public services* through their APIs. Legacy systems are more and more equipped with service *wrappers* and *adapters* that allow them to interact directly with other applications and systems through APIs. In the Software Development Life Cycle (SDLC) a progressive shift from *software development* - *specify*, *design* and *implement components* – towards *service integration* - *select*, *evaluate* and *integrate services through APIs* - can be observed.

Services become enablers of critical business transactions and, potentially, the weakest links in these transactions. The quality of the services that an organisation *provides* and *consumes* is now more important than ever. The business impact of any application failure is the same regardless of whether the fault lies within the components developed by the organization or in the implementations of the services that its components use.

Hence, service testing is not only an activity of service providers, but also and mostly of service users. Furthermore, service providers and service users are *roles* of developers whose software consumes services in order to implement the services that it provides (*service composition*). Hence, a service developer has to test the services that her/his software uses, in addition to test the services that it provides, in the earliest phases of the SDLC.

Test-Driven Development style is a popular option for software development [Beck 2003], and is mandatory for service composition where, in principle, a service that is selected to be used should be tested (as a part of its evaluation) before its integration. As a consequence, testing is no more a singular event but a continuous activity from the beginning of the SDLC (Continuous Integration Testing – CIT [Huang et al. 2008]).

Technically speaking, services architectures are built on interoperability platforms between distributed systems that are mainly based on a limited number of technologies:

- SOAP [WSDL 1 1 2001] [SOAP 1 1 2000];
- REST/XML [Fielding and Taylor 2002] [XML 1 1 2006];
- REST/JSON [Fielding and Taylor 2002] [JSON 2013].

All these platforms ensure the syntactic interoperability of the participant systems. They can be mixed within services architecture. Service components using these interoperability platforms are loosely coupled and there are no technical limitations to the “mixed” composition of SOAP, HTTP/XML, HTTP/JSON services.

Historically, SOAP is the first service technology/standard and also the most complex one. More recently, the exponential growth of non-XML services, in particularly those based on HTTP and the JSON format of the interaction payloads, has been witnessed.

TOWARDS SERVICE TESTING

Service testing is a technical activity conducted to provide stakeholders with information about the quality of the system providing (and consuming) the service. This activity includes the stimulation and the observation of the behaviour of the system according to a specified procedure [NIST 02 3 2002].

In an historical perspective, there are three distinctive "waves"[11] of testing practices that are linked to corresponding engineering main trends:

1. vertical application testing,
2. component off-the-shelf testing,
3. service testing.

VERTICAL APPLICATION TESTING

Vertical application testing was the most common testing activity at the end of the twentieth century. Within this first wave, the object of development and test is a *vertical application*, i.e. piece of software built to be utilised directly by business people through a Graphical User Interface (GUI) as a support of a business activity. Application developers are in charge of requirement gathering[12], design, development and deployment of the application.

In principle, developers perform *white-box* testing[13] during the development phase: they check by analysis and observation of the running code that the software is free from technical errors (e.g. infinite loops). At a certain stage of the engineering cycle, the application is delivered to a group of selected end users that "test" its compliance with the business needs (beta test phase). In this phase, a sort of *black-box* testing[14] activity is performed manually by end users: they stimulate, observe and assess the behaviour of the application without any access to its implementation.

COMPONENT OFF THE SHELF TESTING

At the end of the twentieth century, enters the *component-based engineering* approach. The idea is that software can be provisioned in the form of *components off the shelf*. A *component* is a piece of software:

- that implements specific business and/or technical functionalities;
- that is provisioned through the physical distribution of its executable code to be installed on the user premises;

[11] The term 'wave' seems more appropriate than 'phase' for activities that continue today.

[12] Requirement gathering is often lead by business consultants, who operate as trade-unions between the end users and the technical developers.

[13] "**White-box testing** (also known as **clear box testing**, **glass box testing**, **transparent box testing**, and **structural testing**) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing)." http://en.wikipedia.org/wiki/White-box_testing

[14] "**Black-box testing** is a method of software testing that examines the functionality of an application without peering into its internal structures or workings." http://en.wikipedia.org/wiki/Black-box_testing

- whose functionalities are accessible only through APIs, the source code being not available.

The component-based approach introduces some important changes in the development practices and in the testing needs [Briand et al. 2006]. First of all, the component users are no more business people – they are developers that utilise components as building blocks of more complex applications and systems. Very soon, the problem is posed of the user trust in the functional and non-functional reliability of components that: (i) are black-box executable artefacts developed elsewhere and (ii) are to be used as building blocks of the user applications.

The rise of the component-based approach has stimulated the rise of *black-box* testing as a technical activity to be conducted by a diversified population that includes:

- component developers;
- components users that are implementers of composite applications/systems;
- independent testers (including certification authorities).

At its dawn, the component-based approach has raised all the challenges and problems that are encountered today with the service oriented style of provisioning of business and technical functionalities.

SERVICE TESTING

The service testing wave has followed closely the component-based one at the beginning of the twenty-first century. From the user point of view, a *service* is a business and/or technical functionality that is accessible at some network endpoint through a published API.

Compared to the component-based approach, the service oriented approach brings advantages for the user and for the implementer. For the user, the main advantage is that s/he has not the burden of gathering and installing on her/his premises the piece of software (and its updates) that implements the service as the business and technical functions are only accessible through APIs exposed at a network endpoint managed by the service provider. A counter effect is that the user loses definitively the control of the software life cycle. Within the component-based approach, if a new version of a component is available, the user can choose, under certain conditions, not to adopt it because the new version imposes adaptation costs that s/he doesn't want to afford. Within the service oriented approach, the provider can decide unilaterally to stop the provision of an ancient version of the service, and the user is obliged to "migrate" the applications that consume it.

For the provider, the trade-off between the component and service approaches is between the costs of packaging and distributing new versions of the component and the costs of the infrastructure that provides directly the service. Thanks to the increased availability of cloud computing that allows easy and cheap deployment of software systems, the latter costs had already decreased of at least one order of magnitude and continue to shrink[15]. Moreover, the generalization of the API paradigm allows the construction of large scale

[15] Amazon Web Services (https://aws.amazon.com), the first provider and the market leader of cloud services, has reduced its prices 43 times since the year of its inception (2006).

systems through *service composition*. The essential problem of trust in the functional and non-functional reliability of services as building blocks of the digital economy is raised today at an unprecedented scale.

The Table 1 shows the different perspectives of the service implementer and the service user on service testing. Service implementer and service user are both roles that the *service developer* plays in the SDLC.

Service implementer	***Service user***
Availability of the source/executable code that implements the service.	Unavailability of the source/executable code that implements the service.
Testing practice: white box testing of the code and black box testing of the service.	Testing practice: black box testing of single services and grey box testing[16] of interactions between services.
Context-independent view of the service provided by the implemented code.	Context-dependent view of the used services.
All configurations or aspects of the implemented code behaviour should be tested.	Only subsets of configurations and aspects of the service behaviour that are related to context-dependent use are tested.
Play the *testee* role - deployment of the service component for internal and external use - perform extended coverage unit test.	Perform context dependent unit test and incremental integration test on services that are deployed by their providers for testing purposes.

Table 1. Perspectives on service testing.

It must be noticed that testing does not give the proof of the absence of incorrect behaviour and, for this reason, testing is a potentially never-ending activity, because the absence of *revealed failures* does not ensure the absence of *concealed faults*. The obvious reason is that, *in general*, testing cannot demonstrate the absence of *faults* - this can only be achieved by *code correctness proofs*. In the scientific sense, black-box testing *does not verify* the service compliance and the software quality. As B. Meyer expresses vividly: "The only incontrovertible connection between testing and quality is negative, a falsification in the Popperian sense: a failed test gives us evidence of non-quality" [Meyer 2008]. Hence, *in general*, successful testing (tests that *fail*) *falsifies* service conformance and software quality. So, effective testing can be a means for improving the stakeholders' confidence in the compliance and quality of the service implementations, not a means for proving this compliance and quality.

On the other hand, Aichernig and Salas [Aichernig and Salas 2005] claim that, giving up the "*in general*" and under specific hypothesis, testing can serve as a *verification* tool: "Testing can show the absence of faults, if we have knowledge of what can go wrong". *Fault-based test case generation* is an approach where testers "anticipate" defects in order to generate test cases, i.e. test case generation is driven by the search of "anticipated" faults. The idea is to have enough test cases capable of detecting these anticipated faults. Test case generation takes place in two steps: (i) *business rule individuation* – individuating the business rule (pre/post condition, invariant, control and data flow requirements) that can be violated as a means of modelling the mistakes that may have been done during the development process; and (ii) *test cases*

[16] Grey-box testing of services architecture is based upon the knowledge of the structure of these architectures - their components and the relationships between them - and the observation of the interaction between these components, that are seen as black-boxes.

generation that will cover the modelled mistake, i.e. test cases that would certainly fail if the mistake were implemented.

Effective testing practices permit highlighting defects and vulnerabilities of the service implementation by stimulating the service under test in a manner that instigates these defects and vulnerabilities to provoke failures. Within service testing, the only testing means are mechanisms that allow *stimulating*, *reacting to* and *observing* the behaviours situated at the accessible interfaces of the components of the services architecture under test. Service *unit testing* is *black-box* testing of a *single service provider component/service/operation*. Service *integration testing* is *grey-box* testing of *service compositions*, i.e. distributed architectures of software components that cooperate through service provision/consumption, where the interactions between these components are observable. In the remainder of this manuscript, when the distinction is not relevant, the "canonical" term 'Services Architecture Under Test' (SAUT) is used for both a *single component under test* and a *multi-component architecture under test*.

The testing activities are classified in several ways [Canfora and Di Penta 2009] [Bartolini et al. 2011] [Bozkurt et al. 2013]. A generally accepted classification proposes three main categories:

- functional conformance testing,
- security/vulnerability testing,
- quality of service testing.

Functional conformance testing is testing the compliance of the SAUT behaviours with the services' functional specifications[17]. Security testing is, on one side, testing the compliance of the SAUT behaviours with the *security policies* (this activity is comparable with functional testing) and, on the other side, testing the vulnerability of the SAUT implementations to malicious attacks. Quality of service testing is testing the compliance of the SAUT behaviours with *Service Level Agreements* (SLAs) about performance, reliability, availability, continuity, integrity and other "non-functional" characteristics of the service[18].

This research focuses on functional conformance testing of services and service compositions, more specifically on ***model-based test scheduling methods and automated tools, applied on the grey box testing of multi-component services architectures***[19]. In the remainder of this manuscript, the term 'testing' without qualification shall be understood as 'functional conformance testing'. In the next section the tasks that are involved in the functional conformance testing of services and services architectures activity are detailed.

[17] Conventionally, the service "functional" specifications are not limited strictly to the service abstract functions, independently from any interaction protocol, but include the specification of the interfaces and of the interactional protocols. *Functional* testing is also used as a synonym of black-box testing, while *structural* testing is used as a synonym of white-box testing.

[18] For real time systems, the service functional specifications include also time constraints. Real-time services architectures, if any, are not in the scope of this research.

[19] In this research, black-box testing of a single service component is seen as a particular instantiation of the general SOA testing problem.

FUNCTIONAL CONFORMANCE TEST TASKS

This section hosts a description of the service functional conformance test tasks. This description and the associated taxonomy have been elaborated within the MIDAS project as a generally acceptable description of the testing practices [Maesano et al. 2013]. This description is preceded by the concise sketch of the objects and processes involved in the functional conformance testing activity, and of the model elements that allows model-based functional conformance testing.

TEST OBJECTS AND PROCESSES

A *test case* is the specification of: (i) a collection of states of the SAUT - for each stateful actual component of the SAUT (see below) - and (ii) a collection of *stimuli* (messages) to be sent to the SAUT. The states represent the initial states in which the SAUT components are supposed to be when one of them receives the initiating stimulus that triggers the execution of the test case. The initiating stimulus and the other subsequent stimuli (that are sent as reactions to messages received from the SAUT as responses to previous stimuli) are sent to the SAUT from the test system.

A *test run* is the performance of a test case. It is the actual partially ordered collection of interactions in the SAUT that is triggered by the transmission of the test case initiating stimulus after the configuration of the test case initial states. This collection of interactions constitutes a finite or infinite[20] exchange of messages between components and between the components and the test system. Each message issued by a component is a (SAUT) response and each message issued by the test system after the initiating stimulus is a *delayed* stimulus. After the end - or the interruption - of the test run, good testing practices recommend the reset of the SAUT [ETSI EG 202 810 2010], i.e. the setting of the SAUT to the state before the test run. In fact, each fair test run should be an auto-compensated transaction [Gray 1981] that leaves the SAUT in the state before the test run, and is realized through the invocation by the tester (that can be an automated test system) of the appropriate state management functions on each stateful component. These ancillary functions shall be implemented as services and are utilised by the tester in order to ensure the independence and repeatability of each test run and to support the arbitration of the test outcomes by getting state views.

A *test outcome* is a representation of the partially ordered collection of the SAUT responses and of the collections of the SAUT components' final states that are produced by a test run. Since test runs that are correctly implemented (see the preceding point) are independent from each other and can be performed in any order, each run of the same test case on the same SAUT always produces the same test outcome.

A *test oracle* is associated to one and only one test case. It is a specification of: (i) a partially ordered collection of *responses* and (ii) a collection of *final states*. The test oracle responses and states define the expected behaviours associated to the running of the test case. Oracles can be *passive* or *active*. A passive test oracle is able to check partially the behaviour of the component, but is not able to reproduce this behaviour. For instance, a partial specification (a *fragment*) of a SOAP message issued by the SAUT component as a response

[20] The expected exchange of messages constitutes a finite sequence. An actual unlimited sequence of messages is the expression of a failure (of the SAUT) or an error (of the Test System).

to a stimulus is a passive oracle. An active oracle can *reproduce* the behaviour of the SAUT component. For instance, a complete specification of a SOAP message instance issued by the SAUT component as a response to a stimulus is an active oracle. Active oracles allow the test system to emulate the component behaviour.

A *test sample* is the couple of a test case and the associated test oracle. It is a specification of: (i) the collection of the initial states the SAUT stateful components (the test case initial states), (ii) the partially ordered collection of the messages (stimuli and responses) that, starting from the initiating stimulus, are exchanged between the components (the test case stimuli and the test oracle responses), (iii) the collection of the final states of the SAUT stateful components (the test oracle specifications of final states). A test sample represents an instance (possibly specified partially with passive oracle) of the *extension* of the SAUT model, i.e. it models a snapshot (a family of snapshots for passive oracles) of the SAUT behaviour that is compliant with the SAUT specification.

A *test suite* is a collection of test samples on a SAUT that can be ordered statically (prioritised) or scheduled dynamically (see below).

A *test session* is the sequence of test runs corresponding to the performance on a SAUT of part or the totality of the test samples of a test suite.

SAUT MODEL REQUIREMENTS

In order to describe the testing tasks, and to compare the manual and automated executions of this tasks a sketch of the requirements for structural, functional and behavioural specifications of the SAUT, and of the model (meta-model) that constitutes a formal representation of these specifications must be provided. The automated and manual processes that implement the testing tasks are deeply dependent on the availability of formal (structural, functional and behavioural) models of the SAUT.

A comprehensive formal representation of the structural, functional and behavioural specifications of a services architecture for model-based black-box/grey-box testing should include:

- A formal definition of the *service interfaces*. This part of the model is available “by definition” of the service oriented approach (e.g. WSDL document). The interface model is part of the structural model of the SAUT.
- A formal definition of the *topology* of the services architecture under test. This topology includes the components their provided and required interfaces[21] and the actual service dependences between them, i.e. the connections between the required and provided interfaces that carry out the component interactions. The topology is part of the structural model of the SAUT [De Rosa et al. 2014a].
- A formal definition of the service *interaction protocols* between components (for instance through Harel state charts or other similar formalisms [Harel and Politi 1998] [De Rosa et al. 2014b], i.e. of the message types (defined in the interface model) that each service component can send and receive in

[21] The terms ‘provided interface’ and ‘required interface’ are intended to have the meaning used in UML [UML 2 4 1 2011].

each state of the “conversation” with the other components. The interaction protocols are part of the functional/behavioural model of the SAUT. Note that the modelled behaviour is the external behaviour observable at the components’ interfaces.

- A formal definition of *pre/post conditions* and on the *invariants* of the service operations (for instance through the design-by-contract approach [Meyer 1992] [De Rosa et al. 2014b]). These conditions are Boolean expressions on the states of the “conversation” between the components. They are part of the functional/behavioural model of the SAUT.
- A formal definition of the *data-flow* requirements of the interaction, i.e., for each component, the definition of the functional relationships between the content of the received messages and the content of the subsequent, protocol-dictated, emitted messages [De Rosa et al. 2014b]. The definition of the data-flow requirements is part of the functional/behavioural model of the SAUT. The interaction protocols and the data-flow requirements are the building blocks of the service composition specification.

The production of these SAUT models is a labour intensive and knowledge intensive process that shall be performed by someone with a deep knowledge of the SAUT specifications and of the modelling techniques. In the model-based Test-Driven SDLC, the early production of models drives both the implementation and testing concurrent processes.

TASK DECOMPOSITION

The service functional testing activity is decomposable in the tasks listed below organised in two main test cycles:

- Test generation cycle:
  - test case production,
  - test oracle production.
- Test run cycle:
  - test execution,
  - test arbitration,
  - test reporting,
  - test scheduling.

The test tasks listed above can be organised in *test plans*. Generally speaking, *test planning* is intended as a management activity that organises in time and space the testing tasks and their related resources. In this research a more restricted and operational definition of *test planning* is given in the paragraph below.

The service testing current practice is generally limited to service *unit testing* and is backed by popular tools (such as SoapUI) that:

- support the tester activity of construction of SOAP messages (stimuli and responses) by proposing SOAP templates that are generated by the tool from the interface specification (WSDL/XSD) and that the tester has to fill with appropriate texts and values;

- propose a built-in client that is able to send requests to the service under test and to collect the responses;
- propose a built-in server that is able to receive requests and send stereotyped responses defined by the tester (service *virtualization*).

Within the current practice, with the exception of the limited support described above, the tester carries all the testing tasks out “by hand”. Furthermore, the aforementioned category of tools supplies *no support for integration testing*, i.e. for the grey-box testing of multi-component services architectures implementing service compositions[22].

TEST CASE PRODUCTION

The number of test cases for functional conformance testing of a SAUT is potentially unlimited. The objective of the test case production task is the construction of a finite effective and efficient collection of test cases. The effectiveness of a collection of test cases is its ability to reveal failures and to localise faulty SAUT elements (troubleshooting). Its efficiency is its ability to reveal failures and to localise faulty SAUT elements with a minimum number of test cases (*size efficiency*) or with a minimum number of test runs (*time efficiency* or *fault detection rate*).

Methods and tools of generation of effective and efficient *test suites* have been the main focus of the research effort on testing from the beginning (see section 3). In the domain of vertical application testing, some of these strategies and methods are implemented as automated generation tools in sophisticated, off-the-shelf products that are commercially available. Automated test case generation for service testing has been an academic research target for fifteen years, but there are not yet commercially available tools for service composition test case generation (except for orchestrated service composition[23]).

The test case production processes are categorised and described in the Table 2 on the basis of the availability of structural, functional and behavioural SAUT models and of the degree of automation of the testing task.

#	***Process title***	***Process description***
TC1	Manual production *without* the support of the SAUT model.	Creative, labour intensive and knowledge intensive unstructured process. The producer must possess both a deep knowledge of the implicit structural, functional and behavioural specifications of the SAUT and of the test case generation strategies and methods.

[22] Generally speaking, the tools that support the run cycle for service composition test are dedicated to specific composition paradigms such as service orchestration (e.g. BPEL) and service choreography (e.g. WS-CDL).

[23] The editors of orchestration engines (BPEL) and development environments supply white-box testing tools for BPEL scripts.

TC2	Manual production *with* the support of the SAUT model.	This is a labour intensive task that shall be performed by someone that is able to apprehend the SAUT models and has knowledge of test case generation methods and strategies (from this point of view, it is also knowledge intensive). Conversely, the producer is not obliged to possess "first-hand" knowledge of the functional and behavioural specifications of the SAUT that are objectified in the SAUT models.
TC3	Automated production of test cases, necessarily based on the availability of the SAUT model.	The automated production of test cases is necessarily model-based. The model-based generation of test cases utilises *techniques* such as constraint *propagation on the* pre/post-conditions, control and data flow requirements and *strategies* and *methods*, such as *boundary value analysis*, *equivalent class partitioning*, *random*, *ad hoc*, and *combinatorial and statistical methods* such as *pairwise* and *orthogonal arrays*.

Table 2. Production of test cases.

TEST ORACLE PRODUCTION

Test oracles can be produced either at generation time (*static oracles*) or at run time (*dynamic oracles*). The production of a test oracle is a "mechanical" process: given a test case, and the structural, functional and behavioural specifications of the SAUT, the test oracle can be calculated "mechanically"[24].

Test oracles can be *passive* or *active*. A passive oracle can only be used to *check* the SAUT behaviour. An active oracle is also able to *reproduce* the SAUT behaviour. The active oracle can be produced only from a *complete* SAUT structural, functional and behavioural model[25].

The test oracle production processes are categorised and described in the Table 3. They are categorized on the basis of the availability of the SAUT model and the degree of automation.

#	***Process title***	***Process description***
TO1	Manual production *without* the support of the SAUT model.	Labour intensive and knowledge intensive unstructured process. The producer must possess a deep knowledge of the implicit structural, functional and behavioural specifications of the SAUT.
TO2	Manual production *with* the support of the SAUT model.	Given a test case and the SAUT model, the calculation of the test oracle is labour intensive and knowledge intensive and shall be performed by someone that is able to apprehend the SAUT model and to manually "generate" a snapshot of its functioning. Conversely, the producer is not obliged to hold "first-hand" knowledge of the structural, functional and behavioural specifications of the SAUT that are objectified in the SAUT models.

[24] To be precise, it is a true mechanical process (whether performed by a human or a machine) only if the SAUT specifications are formal. If the specifications are informal or implicit (in the head of the service designer) we cannot speak of a true mechanical process.

[25] The completeness of the SAUT model should be intended relatively to the test context and, in any case concerns only the external observable behaviour of the SAUT.

TO3	Automated production of test oracles, necessarily on the basis of the availability of the SAUT model.	The process is performed by test oracle generation software that is able to calculate or emulate the external behaviour of the SAUT components as described in the SAUT model. The automated production of test oracles is necessarily model-based.

Table 3. Production of test oracles.

TEST EXECUTION

The core of the *test execution* task is the accomplishment, for each test case, of a test run. In order to perform a test run, the tester, directly or with the help of a test system, shall be able to perform the following actions:

1. setting the SAUT components' states to the test sample initial states through the state-view management ancillary services,
2. issuing the test case stimuli towards the SAUT,
3. receiving the SAUT responses,
4. observing the SAUT responses,
5. possibly getting the SAUT components' final states through the state-view management ancillary services,
6. resetting the SAUT components' states through the state-management ancillary services.

In order to run the test cases, the test execution task comprehends other enabling and accompanying actions such as:

- the SAUT components deployment, that shall conform the SAUT construction model;
- the test system deployment – a crucial capability of the test system is the emulation of the SAUT virtual components that are responsible for sending the stimuli to the deployed SAUT actual components and for receiving the SAUT responses; another crucial capability of the test system is the deployment of *observers*, that are able to watch the interactions between actual components that are triggered by the test case stimuli; more advanced test system are able to automate other test tasks such as test arbitration, test scheduling, dynamic test case/oracle generation, test planning (see below);
- the pre-run configuration and the setup of the test system and of the SAUT components, including the binding of the test system with the SAUT components;
- the detailed logging of the test run and of all the actions listed above.

A collection of test cases (a test suite) can be executed either in *batch* mode or in *interactive* mode. Within *batch execution*, the execution of all the test cases of a test suite is effected in a predetermined sequence, and the tester (a human or a test system) regains control only when the sequential execution is either ended (all the test cases are executed) or is stopped by the verification of some halting condition. Within *interactive execution* the tester launches the run of test cases one by one (or by small groups) and regains the control at the end of each test run, the test outcome being available immediately after the execution.

The end user tools for service testing that are proposed today can perform only *unit test execution* (batch and interactive) and the logging of the test outcomes (with the exception of the TTCN-3-based tools, see next paragraph).

In the domain of test execution automation a significant progress has been made in the last fifteen years with the availability of the TTCN-3 language and its execution environments[26]. The language have been specified by a standard body (ETSI) and has the traits of a general purpose programming language with specialized constructs and built-in facilities for programming black-box test execution and arbitration. Some commercial off-the-shelf and open source implementations of the language and the environment are currently available[27]. The TTCN-3 language (and the accompanying environment) has been utilized for building automated test execution environments for embedded software and industrial applications (automotive, telecom). It has also been utilized for building automated test execution environments for SOA testing [De Rosa et al. 2013]. In the context of this research, a TTCN-3 engine is employed for service test execution and arbitration automation[28].

If an automated test execution system is not available, the tester must accomplish manually all the actions listed above. Developers/testers are often accustomed to implement dedicated test systems that are the results of composition of off-the-shelf products, open source components and home-developed software components. Generally speaking, these systems support clerical tasks for the interactive and batch execution of the test suites and the logging of the test outcomes.

TEST ARBITRATION

Test arbitration is the production of a *test verdict* as results of the assessment of a *test outcome*. The core action of the test arbitration is the comparison between the *test outcome* and the *test oracle* associated with the executed *test case*. Test verdicts are usually organized in four general categories [UTP 1 2 2012]:

1. *pass* - the test outcome *matches* the test oracle and the match is evaluated as a manifestation of the compliance of the SAUT actual behaviour with the SAUT modelled behaviour – the test *passes*;
2. *fail* – the test outcome *mismatches* the test oracle and the mismatch is evaluated as a manifestation of a SAUT *failure* (the SAUT actual behaviour is not compliant with the SAUT modelled behaviour) - the test *fails*;
3. *error* – whatever the test outcome, it is evaluated as a manifestation either of a defect of the test system or of an error of the SAUT configuration/initialization, including the binding with the test system;
4. *inconclusive* – the evaluation cannot conclude with one of the preceding verdicts;
5. *none* – there is no test outcome (verdict meta-value).

[26] http://www.ttcn-3.org/

[27] http://www.ttcn-3.org/index.php/tools, https://projects.eclipse.org/projects/tools.titan

[28] http://www.testingtech.com/products/ttworkbench.php

When testing complex services architectures, test arbitration can be a tricky endeavour needing accurate analysis of the test execution context in order to avoid *false negatives*, i.e. *pass* verdicts that hide SAUT failures, and *false positives*, i.e. *fail* verdicts that shroud the correct behaviour of the SAUT.

Test arbitration, such as test execution, can be performed either in "batch" or in "interactive" mode. With batch arbitration, the test cases of a test suite are executed as a lot in a predetermined sequence and, at the end, the test log is analysed for arbitration. With interactive arbitration, the test outcome of each test run is arbitrated immediately after the execution of the test case. The latter modality allows: (i) the *interruption* of the batch execution of the test cases when a certain condition is verified (for instance, at the first failure); (ii) *dynamic scheduling* of test cases, i.e. the dynamic choice of the next test case to run on the basis of the verdicts of the past test runs (with the prerequisite of interactive – manual or automated – execution).

A classification in ten categories of the test arbitration processes is detailed in Table 4. The categorisation is made on the basis of the batch/interactive modality, of the availability of test oracles, of the availability of the SAUT model and on the degree of automation.

#	***Process title***	***Process description***
A01	Batch manual "eyeball" arbitration *without* the support of static test oracles and *without* the support of the SAUT model.	This is an unstructured, labour intensive and knowledge intensive process. Part of this process could be the same as TO1 (Manual production of the test oracles *without* the support of the SAUT model - see Test oracle production). Furthermore, the human arbiter shall be able to evaluate test outcomes documented in the test log against test oracles that has been derived from implicit specifications.
A02	Interactive manual "eyeball" arbitration *without* the support of static test oracles and *without* the support of the SAUT model.	The process can be described as A01. Moreover, the oracle derivation and the outcome evaluation are effected on the fly for each test run. This interactive manual testing mode is useful if the tester is able to perform some strategic reasoning about the next test case to run.
A03	Batch manual "eyeball" arbitration *without* the support of static test oracles but *with* the support of the SAUT model.	Part of this process is structured as TO2 (Manual production of test oracles with the support of functional and behavioural models of the SAUT - see Test oracle production). The human arbiter shall be able to "calculate" test oracles from the test cases and the SAUT model and to evaluate the SAUT outcomes documented by the test log against the calculated oracles.
A04	Interactive manual "eyeball" arbitration *without* the support of static test oracles but *with* the support of the SAUT model.	The process can be described as A03 and the ability described in A02 applies too. Moreover, it can be said as for A02 that this interactive manual mode is useful if the tester is able to perform some strategic reasoning.
A05	Batch manual "eyeball" arbitration *with* the support of static test oracles.	This is a labour intensive process that mobilises general "syntactic matching" abilities, but no specific knowledge of the SAUT specifications[29]. The arbiter shall be able to match the test outcomes documented in the test log to test oracles and the test context.

[29] This kind of job is frequently "crowdsourced" (https://www.mturk.com/mturk/welcome).

A06	Interactive manual "eyeball" arbitration *with* the support of static test oracles	The process can be described as A05. Interactive arbitration is interesting if the arbiter is also able to perform strategic reasoning, as in A02 and A04.
A07	Batch automated arbitration	It combines in batch mode automated matching of the test outcomes (test log) to the test oracles and automated evaluation of the test run context. The prerequisite for batch automated arbitration are either the availability of static test oracles (manually or automatically calculated) or the capability of the arbiter to invoke the automated generation of oracles (dynamic oracles). Note that automated execution is not a prerequisite of batch automated arbitration (the test runs can be handled manually and the test outcomes can be collected and assembled manually).
A08	Interactive automated arbitration	It combines in interactive mode automated matching of each test outcome to the appropriate test oracle and automated evaluation of the test run context. The prerequisites for interactive automated arbitration are (i) interactive execution (possibly automated) and (ii) either the availability of static test oracles (manually or automatically calculated) or the capability of the arbiter to invoke the automated generation of test oracles (dynamic oracle). Interactive automated arbitration is useful if utilised by an intelligent scheduler (human or machine) that is able to perform strategic reasoning.
A09	Batch automated execution/arbitration	It combines the automated execution of the test cases (see preceding paragraph) with automated matching of each test outcome to the appropriate test oracle and automated evaluation of the test run context in batch mode. In this configuration, the batch processing of a test suite carries out, for each test case, both execution and arbitration. The batch executor/arbiter takes as input a test suite, i.e. an ordered collection of test samples, and produces a test log in which are documented, for each executed test case, both the test outcome and the test verdict. The batch executor/arbiter may be able to process a halting condition, possibly bearing on the test verdicts.
A10	Interactive automated execution/arbitration	It combines the automated execution of the test cases (see preceding paragraph) with automated matching of each test outcome to the appropriate test oracle and automated evaluation of the test run context in interactive mode. The interactive executor/arbiter takes as input a test sample, executes the test case, arbitrates the test outcome and returns as output the couple test outcome / test verdict. Interactive automated execution/arbitration is a prerequisite for automated dynamic test scheduling (see below).

Table 4. Test arbitration processes.

The TTCN-3 language and framework allows programming sophisticated mechanisms not only of test execution (see the preceding section), but also of test arbitration. The automated test execution environments for SOA testing [De Rosa et al. 2013] mentioned in the preceding paragraph perform also interactive automated arbitration and is the automated execution/arbitration environment that is dynamically scheduled in the context of this research.

TEST SCHEDULING

Test scheduling is giving to the test cases of a test suite a specific order of running. There is a distinction between (i) *static* and (ii) *dynamic* scheduling.

Static scheduling is *batch* scheduling, which is also known as *prioritisation*. The test cases of a test suite are ordered prior to execution through *priorities* assigned to them. The priorities can be assigned manually, by test case production methods or by the test system on the basis of different criteria. At run time, test cases are run in batch mode in the priority order that cannot be changed dynamically. When a test suite is prioritized and runs in batch mode, the batch execution system should be able to stop (or suspend) the test session when a halting condition is verified. Generally speaking, static scheduling can be carried out by batch execution/arbitration systems that are able to take account of the priorities and of the halting conditions - no specific scheduler agent is needed. Examples of static scheduling are the prioritisation techniques employed in regression testing [Rothermel et al. 2001].

Dynamic scheduling is the choice at run time of the next test case to run. Dynamic scheduling requires the action of a human or artificial agent that is able to decide at each test/execute/arbitrate cycle the test case to run on the basis of a decision that takes into account the context in which the cycle is situated and the history of the past cycles. In principle, the scheduler should decide the next test case to run on the basis of its "fitness" to different criteria (for instance the *fault-exposing potential* [Elbaum et al. 2002]). With dynamic scheduling the fitness of each not-yet-performed test case can change at each schedule/execute/arbitrate cycle on the basis of the evolving test session context and the past test verdicts. The result of the execution/arbitration of a test case can change the fitness distribution on the not-yet-executed test cases with respect to the original testing objectives, or even change the testing objectives and the related fitness distribution of the test cases.

Automated test scheduling requires the automation of methods for strategic reasoning and troubleshooting. An automated scheduler could be coupled as a question answering system (what is, given the last test verdicts, the next test case to run?) with manual test execution and arbitration. If the automated test execution/arbitration is available, the complete automation of the dynamically scheduled test run cycle is possible.

TEST REPORTING

Test reporting is the production of a meaningful account of the execution of a test session for debugging purposes[30]. The *test report* is the trade-union between the testing team and the debugging/fixing team. It should contain and organise all the information that can be gathered from the test session and that is useful to improve the debugging process.

While the test log is the detailed narrative of all the execution actions and of the test outcomes/verdicts, the test report is a concise story, supplying aggregate information that highlights the failure (and error) test verdicts, the relationships between failures and the relationships between failures and passes (tests that pass)[31] with the goal of enhancing the debugging process.

The manual production of a test report from the test log or the direct observation of the test session is a difficult task that requires skills such as:

[30] Test reporting systems collect data for administrative purposes that are very useful. This research considers the report as a debugging tool.

[31] When looking for a specific failure, the fact that a test passes could be meaningful information, for the scheduler but also for the debugger.

- the ability to summarize meaningfully (for the debugger) the content of a bulky journal without information loss,
- the ability to find meaningful correlations (for the debugger) between data that are sparse in the log.

The automatic production of a meaningful test report is not a trivial task, because it requires the automated comprehension and assessment of all the information included in the log, e.g. the explanations and the justifications of the test verdicts. The explanations and justifications of the choices of a dynamic test scheduler driven by strategic reasoning can be interesting information too.

The twin of the test report is the *fix report* from the debugging/fixing team. In order to insure the tractability of the test/debug/fix process, he test report should allow identifying: (i) the test session, (ii) the involved SAUT (the build numbers that must be supplied by the testers) and SAUT model, (iii) the test suite and (iv) the discovered failures. If the fix report that accompanies the deployed new build includes: (i) the new build number, (ii) the identification of one (or more) test report(s) and (iii) the identifiers of the failures published in these test reports that the new build is intended to have fixed, this information can be exploited to schedule enhanced regression testing.

TEST PLANNING AND MANAGEMENT

Test planning and management shall be intended in the context of this research as the arrangement and timetabling of the tasks described above. In particular, test planning and management organises the *test generation cycle*, the *test run cycle* and the relationship between them.

In principle, the test generation cycle and the test run cycle can be separate and managed independently. The test generation cycles produce collections of test samples (test suites) that are stocked for future use. These test suites are inputs of the test run cycle that is performed asynchronously.

Advanced test scheduling can supply inputs to test planning, by indicating, on the basis of the verdicts of past test runs, some specifications about the production of new test cases (see the section of future works). Test planning should supply directives and data to the test case production task, for example by focusing the test generation activity on the coverage of:

- specific service operations,
- specific services,
- specific components,
- specific regions of the service component architecture.

or, on the contrary, by conducting a breadth-first search for failures.

Automated test planning and management is the last step in test automation and requires a testing infrastructure in which all the test tasks described above are automated, programmable (callable through APIs) and interoperable. In this case, the test planner is an intelligent system that is able to drive all the test tasks on the basis of a general testing and troubleshooting activity. This automated planner

The Test-Driven Development [Beck 2003] and Continuous Integration Testing [Huang et al. 2008] approaches structure the service engineering process in three nested cycles, as sketched in Figure 9: the test generate/run cycle is driven by the service test/debug/fix cycle that is part of the more general service design/implement/test cycle. Roughly speaking, the test/debug/fix cycle targets the tuning of the service implementation without changes of the service specifications (structural, functional, behavioural), whereas in the design/implement/test cycle the service specifications (and the related implementations) are changed.

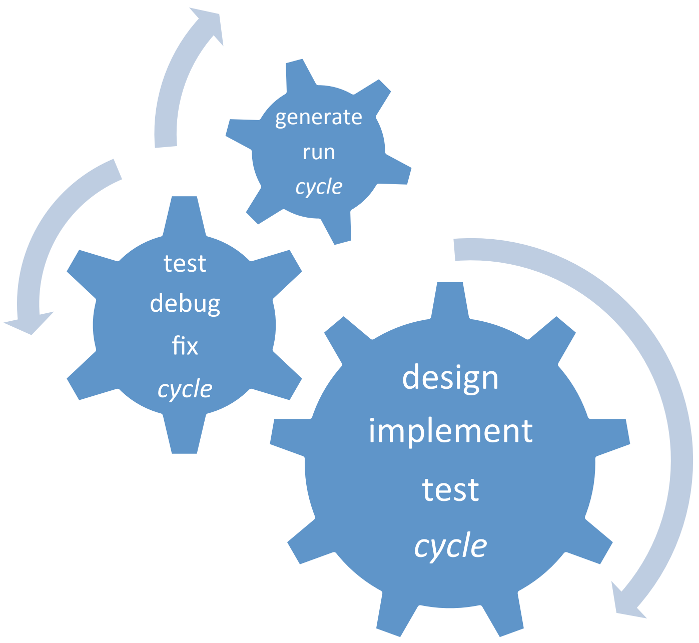


Figure 9. Service engineering cycles.

THE RATIONALE FOR SERVICE TESTING AUTOMATION

In summary, the only means for increasing the trust of the stakeholders in the dependability and security of the digital service ecosystems are appropriate testing procedures and processes. The problem is that service testing is undoubtedly a difficult, heavy and expensive activity. Paradoxically, some of the peculiarities that make service testing *mandatory* make also it *hard*. Many researchers [Canfora and Di Penta 2009] [Bartolini et al. 2011] [Bozkurt et al. 2013] highlight issues related to the complexity, difficulty and cost of service testing. A famous study [NIST 02 3 2002] has set the agenda, putting into evidence the costs of the lack of an adequate infrastructure for testing that makes the endeavour particularly problematic.

First of all, service testing is intrinsically difficult because of the specific characteristics of the service component architectures as test targets. The service oriented approach of information hiding about system internals, on one side allows managing the complexity of the digital economy, but, on the other side, increases the difficulty of the testing tasks. Moreover, information about the internal states of systems that are relevant for assessing the compliance of the implementations with the service specifications can be obtained only by indirect means (e.g. the implementation of ancillary state view management services for stateful components) that make the testability requirements heavier.

The lack of observability of implementations cannot be replaced by the trust in the engineering methods, tools and technologies that are employed by the service implementers, because, in the general case, also these methods and tools are not visible, and cannot be realistically assessed. A further difficulty relates to the lack of direct control of the service implementation lifecycles. The implementations' life-cycles in multi-owner service component architectures are out of each other control, even when very strict policies are agreed between independent stakeholders, such as rigorously controlled change management procedures concerning both service contract updates and service implementation new releases. In general, the services architecture stakeholders are independent organizations that have no hierarchical relationships among them. The establishment and the management of inter-organisational service testing (collaborative testing) cycles, procedures and sessions on a multi-owner services architecture is a complex organisational task per se.

All the difficulties depicted above grow with the increasing scale factor of the services architectures. The service approach facilitating the availability of powerful business functionalities in the Internet, it is easy to predict the viral development of very large scale services architectures, supporting structured and unstructured digital processes that involve huge numbers of participant systems, applications, devices and "objects" [Brian Arthur 2011]. Testing scenarios of end-to-end service exchanges will require the design and the implementation of dedicated strategies able to master the complexity of the task.

Testing does not prove the absence of faults. At best, it can either reveal that past failures that seem correlated to some defects or weaknesses that have been corrected do not happen anymore or improve the confidence in the absence of anticipated errors or susceptibilities. Because of the increasing complexity of the SAUT initialization, configuration and reset procedures and of the test system configuration, binding and initialization processes, the verdicts gathered from the arbitration of test outcomes can be flawed or uncertain (risk of false positives and false negatives, inconclusive assessments …).

The cost of service testing has three components: (i) equipment expense, (ii) labour effort, (iii) time-to-market.

The equipment costs of service testing, including hardware, facilities, software licenses and maintenance, for the SAUT and the test system, are very high. The adoption of Test-Driven SDLC brings more continuity to the testing activity in the early stages of the cycle, but the equipment needs are still discontinuous. This elasticity of demand for testing equipment raises also logistic and organizational problems. A possible solution of this problem is the adoption of cloud computing for testing [Maesano et al. 2013b].

The labour cost of manual testing is high, whatever the testing task involved and the labour skill required, for several reasons. First of all, the level of automation of the current testing practice is low. Commercially available tools[32] offers limited functionalities related to the mechanisation of few clerical tasks and are strongly limited to single-service testing. Even if some clerical tasks can be mechanised, service testing remains a human based activity for critical tasks such as: (i) production of test cases and test oracles, (ii) configuration and set up of test environments, (iii) scheduling of test runs, (iv) arbitration of test outcomes, (v) production of meaningful reports.

[32] http://www.soapui.org/, http://www.parasoft.com/, http://www.crosschecknet.com/products/soapsonar.php

Moreover, as already highlighted in the preceding section, these tasks are not only labour intensive, but also knowledge-intensive. At least two kinds of knowledge are mobilised by the testing tasks: (i) knowledge of the SAUT structural, behavioural and functional specifications and (ii) knowledge of approaches, methods and tools to be applied to the testing tasks. They are both scarce resources. Even when they are available, manual testing (generation, execution, arbitration, scheduling) is difficult and error prone (useless test cases, wrong oracles, false positives, false negatives …) and requires sustained continuous attention and critical observation capabilities.

Paradoxically, the availability of partially automated test execution tools that make easier clerical tasks such as the storage, management and execution of test suites incites the brute force execution of massive test sessions, short of any consideration of their efficacy in terms of failure seeking, troubleshooting and confidence improving. This approach finally increases the human effort needed for the eyeball assessment of large numbers of test outcomes, without measurable amelioration of the effectiveness of the overall process. Manual testing is not only difficult and expensive, but also its efficacy is questionable, above all for large scale services architectures. Last but not least, the efficacy of manual testing, all things being equal, is diminished by the lack of motivation for an activity that is considered boring and low rewarding. The difficulty of manual testing in front of the explosion of the digital service ecosystem cannot be overcome by the mere diffusion of methodological recommendations and training programmes. The true solution is test automation.

The last component of the testing cost is *time-to-market*. Time-to-market allows comparing testing costs with the risk of insufficient testing. Service providers and users are confronted to the dilemma between: (i) long and painful testing procedures that can provoke the missing of the market momentum and, as seen above, do not guarantee necessarily the service quality and (ii) precocious delivery of insufficiently tested services, with high business risks. The duration of the testing processes and procedures compared to the constraints of the time-to-market pushes to automate the testing tasks, but also to shorten the test cycles and to increase their frequency.

Testing automation, especially functional conformance testing, is forcefully model-based. All the testing automation modules take as inputs the structural, behavioural and functional models of the SAUT and the service implementations are tested *against* these models. Hence, the only “manual” task for the developer is the production of the SAUT structural, functional and behavioural models [De Rosa et al. 2014a] [De Rosa et al. 2014b]. This task is not labour intensive, but is neither trivial. It requires modelling skills, and the availability, during the modelling process, of a deep knowledge of the SAUT structural, functional and behavioural specifications. The adoption of the Test-Driven SDLC facilitates the approach: the early production of models drives concurrently the implementation of the services architecture and the test generation and run cycles that are implemented by automated processes running in background. Once the SAUT model is built, the marginal cost (in terms of human effort) of the fully automated testing tasks approaches zero. Furthermore, the testing automation modules embed the knowledge of test strategies, methods and practices.

Test automation should be considered not only the automation of the testing tasks, but also as the automation of the synchronisation between the test cycle and the service engineering cycles (Figure 9).

This research is focused on the automation of test scheduling. This automation is implemented on the basis of probabilistic inference mechanisms that take into account the SAUT structural model and the test suite in order to perform efficient failure seeking and localisation of faulty operations/services/components.

OBJECTIVES AND CONSTRAINTS FOR TEST SCHEDULING AUTOMATION

As a conclusion of this section, this paragraph summarises some very general objectives and constraints that have driven this research, whose main goal is the automation of dynamic scheduling of the test run cycle.

1. Grey-box functional conformance testing of service composition

The fundamental idea of composition testing (other same-meaning terms: 'grey-box testing of services architecture', 'service integration testing') is that what must be checked is not only the response to a service request from a single service component (unit testing), but (i) all the messages exchanged among the components of a services architecture during the entire end-to-end service transaction that is triggered by the first request (the stimulus) and (ii) all the states after of the stateful components.

The automated dynamic scheduler shall be able to cope with grey box testing of multi-component services architectures. This means that it shall be fully aware of the structural model of the service component architecture under test, of the test system configuration and of the test suite to be scheduled and it shall take into account, in order to schedule the test runs, the composite verdicts issued by the arbitration of the past test outcomes.

2. Automated dynamic test scheduling in TDD and CIT life cycle

The automated dynamic scheduler shall be a tool that improves the Test-Driven Development and the Continuous Integration Testing approaches of the SOA engineering life cycle. It shall allow complete automation of the test run cycle. Not only shall the test run cycle be automated, but also its invocation shall be effected automatically. The idea is that the development environment of the design/implementation team receives test reports and invokes automatically test sessions, possibly sending back change reports. In this context the main objective of the automated scheduler is to foster the early detection of faults (*fault detection rate*), through the improvement of early failure detection and faulty component localisation.

3. Test Scheduling as a service

The automated dynamic scheduler shall be a component of a more general architecture of testing automation. Test execution and arbitration automation are prerequisites for automated dynamic test scheduling. In order to automate the scheduling task the automated scheduler must cooperate with the logic of the process that combines automated interactive execution with automated interactive arbitration. In any case, the automated dynamic scheduler must be loosely coupled with the surrounding test system and testing environment.

The automated dynamic scheduling function shall be provided as scheduling-as-a-service. The scheduler shall be a scheduling service provider module that exposes an API that allows the initialization with the SAUT model and the test suite, and the request to supply the next test case to be run by furnishing the past test verdicts.

4. Model-based Dynamic test scheduling

In order to improve its failure exposure and fault localisation capabilities, the scheduler shall “acquire” and maintain a detailed knowledge of (i) the structural and functional decomposition of the SAUT and (ii) of the Test Suite. This information must supplied by the same SAUT model and the Test Suite data that are used by the other modules of the test run cycle. Moreover, it must be able to integrate in the model the information coming from the progress of the test session, such as the test verdicts, but also other information about the SAUT structural and functional elements. It must be able to suggest to the test system the most appropriate test cases to be executed.

5. Probabilistic inference for test scheduling

The scheduler shall be able to “reason” on an uncertain situation and to manage the uncertainty of the decision process. Moreover, the scheduler shall be able to take into account in its decision process new information coming from the test execution/arbitration, i.e. new observations (test verdicts) and new beliefs about the structural and functional elements of the SAUT.

The fundamental advantages of the usage of the probabilistic inference for decision are well-known, and seems particularly appropriate for test scheduling: (i) probability theory is able to model with maximum accuracy and minimum number of parameters a complex situation; (ii) the intermediate steps and the results of probabilistic inference can be mathematically proven; (iii) new assumptions, beliefs and observations can be integrated in the inference steps, (iv) the inference results are knowledgeable and can be explained in the context of a well-established mathematical theory. The use of probability models and probabilistic inference on these models bring a concise, sound and knowledgeable approach to test scheduling.

The limitations of the approach are fundamentally in the fact that many calculating procedures that support the probabilistic inference have high computational complexity, and managing this complexity is a research challenge. Automated test scheduling based on probabilistic inference is a new domain of investigation and the main focus of this research.

6. Scheduling and halting policies

The Scheduling service shall be able to implement different scheduling and halting policies on the results of its inference engine. These policies shall allow utilising the same probabilistic inference mechanism when pursuing different testing goals in different testing contexts.

3. RELATED WORK ABOUT SERVICE FUNCTIONAL CONFORMANCE TESTING

In the last fifteen years, the spread of the service orientation as a paradigm for the design and implementation of distributed architectures has changed the traditional understanding of software application design, delivery and utilisation [Vinoski 2002] [Vogels 2003] and has stimulated the emergence of a research thread about Service-Oriented Computing (SOC) and Service-centric Systems (ScS) [Papazoglou 2003] [Singh and Huhns 2006] [Bichier and Lin 2006].

The trend of the number of publications on Service-centric System Testing and Verification, drawn from [Bozkurt et al. 2013] is depicted in Figure 10. Bozkurt and colleagues declare that the total number of publications has risen from 21 in 2004 to 177 in 2010.

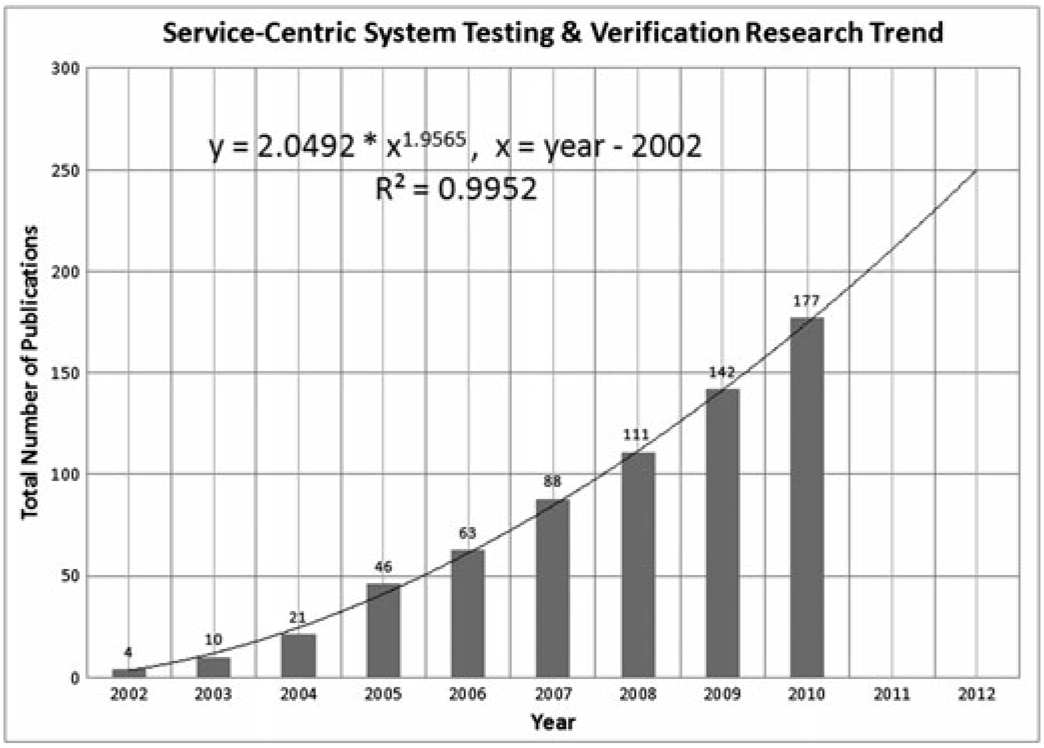


Figure 10. Total number of publications from 2002 to 2010, drawn from [Bozkurt et al. 2013].

In addition to research papers and communications, there are also a number of research surveys that give an overall sight of the various aspects of the SOA testing problem [Canfora and Di Penta 2009] [Bartolini et al. 2011] [Bozkurt et al. 2013], and in particular of issues related to the *complexity*, *difficulty* and *cost* of service testing.

Figure 11, always drawn from [Bozkurt et al. 2013] presents the type of *case studies* used by the researchers for experimentation and validation of their approaches and highlights one of the great problems in service testing research: the *lack of real-world case studies*. Let's speak Bozkurt and colleagues: "Unfortunately, 71% of the research publications provide no experimental results. The synthetic service portion of the graph mainly includes approaches for testing Business Process Execution Language (BPEL) compositions. The most common examples are the loan approval and the travel agency (travel booking) systems that are provided with BPEL engines. There are also experiments on existing Java code that are turned into services such as javaDNS, JNFS and Haboob. The real services used in experiments are generally existing small public services that can be found on the Internet. There are four experiments performed on existing projects such as government projects, Mars communication service and a human resources (HR) system".

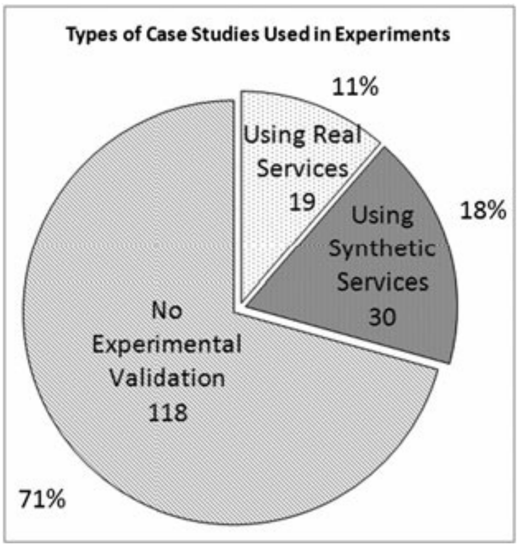


Figure 11. Distribution of case study types used in experimental validation drawn from [Bozkurt et al. 2013].

There are many plausible reasons for this lack of real-world case studies. The first is probably the fact that, in spite of the hype on SOA, only very recently the "API economy" phenomenon is going to make real the original dream of service orientation, i.e. the implementation of massively distributed architectures for business process automation through the loose coupling of independently designed systems, applications and devices on the basis of shared service specifications and mutually hidden implementations. The second reason is that SOA testing is complex and hard to put in place in real contexts, from the technical and organisational points of view, especially for multi-owner services architectures. The third reason is that, generally speaking, testing is a hot topic: businesses are reluctant to communicate about their testing practices and to expose their systems as targets of research approaches and tools for testing [NIST 02 3 2002]. In the MIDAS project, two real and complex services architectures under test, in the domains of Health and Logistics [Maesano et al. 2014], have been put in place by two business professional partners and the test campaign with the MIDAS platform, including the scheduler that is the product of this work, has started in October 2014.

The work that is reviewed in this manuscript is related to this research in different ways. First of all, the domain is *functional conformance testing of services architectures*. In this domain we analyse the research threads on *test run automation* and on *regression testing*.

In the author's best knowledge, there is no evidence of a specific research thread about *scheduling*, but *static scheduling* (*test case prioritisation*) has been largely investigated as a technique of regression testing. The authors of a systematic study on Web service regression testing research [Qiu et al. 2014] enumerate 48 prioritisation techniques.

Test generation has been the main research topic of SOA testing, but it is not related directly to the subject of this research. Test case generation and dynamic scheduling of test execution/arbitration have different but associated objectives: the former targets the effectiveness and space efficiency of the test suites, whereas the latter pursues the *fault detection rate* of a *given* test suite. The integration of test scheduling and test generation is a topic of future work (see the section of future works).

Model-based testing (MBT) is a research thread that emphasizes the use of formal models in testing. In the 1st generation of MBT studies, the main target is the automated generation of test cases from the SAUT *functional* and *behavioural* model. There is no specific consideration for a specific and independent SAUT *structural* model, that is utilised in this research for the configuration/generation of the test execution/arbitration environment and for the generation of the Bayesian Network whose inference drives the scheduling task. The structural model enables the model-based automation of test scheduling and allows both structural and functional localisation of faulty elements.

The research about the automation of the test run (execution and arbitration) is more directly related to dynamic scheduling automation. The automation of test execution and test arbitration are prerequisites for the automation of the test run cycle driven by an automated dynamic scheduler. The author of this manuscript has participated to a research program correlated to this work that had the objective of complete automation of test execution and arbitration on the basis of the TTCN-3 technology [De Rosa et al. 2013], and this test system has been deployed on the cloud platform of the MIDAS project [De Rosa et al. 2014d] .

Research about regression testing - the retesting of the SAUT after specification and/or implementation changes to check that these changes have not unintended side effects – has been focused in how to maximise the value of existing test suites for regression testing and minimise the costs of regression test execution, through techniques such as test case selection and test case prioritisation. Work about regression testing is related to this research in at least two ways: (i) test case prioritisation is a form of static scheduling for regression testing but the prioritisation criteria (*coverage*, *fault-exposing potential*) can be utilised for general testing and their relationships with dynamic scheduling have to be investigated and (ii) the role of dynamic scheduling in regression testing is a subject of investigation too.

TEST RUN AUTOMATION FRAMEWORKS

On the side of test run automation, an important step has been taken with the "invention" of TTCN-3 [Willcock et al. 2011]. TTCN-3 is an international standardised test language for defining executable test specifications for a wide range of computer and telecommunication systems. TTCN-3 is a complete programming language that has been designed with powerful traits that are specific for testing automation. It allows complete automation by programming of the test execution and arbitration tasks. The research on the utilisation of TTCN-3 for service testing has started early [Schieferdecker and Stepien 2003] [Xiong et al. 2005] [Werner et al. 2008] [Peyton et al. 2008]. Today De Rosa and colleagues [De Rosa et al. 2013] propose a generic TTCN-3 framework for testing services and service compositions that integrates the XSD mapping standard [ETSI ES 201 873-9 V4.5.1 (2013-04)]. This framework implements the execution/arbitration automation of the scheduled test run cycle in which the scheduler result of this research takes place.

The related work on test run automation frameworks can be arranged in two groups: (i) *service unit test run automation frameworks*, equipped with virtualisation techniques, (ii) *service integration test run automation frameworks*.

UNIT TEST AUTOMATION FRAMEWORKS (VIRTUALISATION TECHNIQUES)

A service composition, realised explicitly by orchestration or choreography or implicitly by chained invocation, may also be considered, as a whole, as a service unit. Unit testing of service compositions can be performed by simulation (real-world testing of service compositions with access to actual services is unrealistic), i.e. by using stub or mock services (*virtualisation*) to test the business process [Li et al. 2005].

Mayer and Lübke [Mayer and Lübke 2006] present a BPEL unit testing framework for repeatable, white-box BPEL unit testing. This framework uses a specialized BPEL-level testing language to describe interactions with a BPEL process to be carried out in a test case. The framework supports automated test execution and offers test management capabilities via well-defined interfaces.

Huang and colleagues [Huang et al. 2008] propose a simulation framework that addresses the service availability problem by using Continuous Integration and Testing (CIT) approach. The proposed framework automates the testing by using a *surrogate* generator that generates platform-specific code skeleton from service specifications and a *surrogate* engine that simulates the component behaviour according to skeleton code. Huang and colleagues claim that the proposed surrogates are more flexible than the common simulation methods such as stubs and mocks and that the simulation is platform independent.

Ilieva and colleagues [Ilieva et al. 2010] introduce a tool for end-to-end testing of BPEL processes called TASSA, that offers virtualisation, an injection feature, a data dependency analysis module and a test case generation tool.

INTEGRATION TEST AUTOMATION FRAMEWORKS

The first challenge of integration testing framework is the exposition of failures that facilitate the localisation, in the service composition architecture, of the components whose behaviour is the source of the failures (troubleshooting). Integration testing approaches are based on models of test scenarios (e.g. sequence diagrams, orchestration scripts, choreography models), but, to author's best knowledge, never on structural models of the services architecture that support the scenarios.

Peyton, Xiong and colleagues [Peyton et al. 2008] [Xiong et al. 2005] present an integration test framework for composite applications able to localise faults within the architecture. The test framework: (i) is based on defining test cases of expected behaviour for the applications and the web services used, and (ii) is implemented in TTCN-3 using a test agent architecture that supports coordinated grey-box testing of application behaviour and web service interaction. The test agents are intended to place themselves between a the composite application and the services that it invokes, and the authors consider implementing concurrent execution of requests from more that one user and mechanisms such as (i) caching of responses and sequencing and interleaving of requests and responses. Their mechanisms are limited to services whose state is initialised once and to non update operations. The result of the research is a TTCN-3 framework that allows programming by hand test cases and test architectures on the basis of a library of TTCN-3 elements.

Liu and colleagues [Liu et al. 2009] propose a Continuous Integration Testing (CIT) approach with which executable test cases carry information on their behaviour and configuration. Integration test cases are generated from sequence diagrams and run on a test execution engine to support this approach. The information of an executable test case is separated into two layers: the behaviour layer and the configuration layer. The behaviour layer represents the test logic of a test case and is platform independent, while the configuration layer contains the platform specific information. Liu and colleagues report the design of a test execution engine specially designed to execute the integration test cases. They utilize a global test case identifier to correlate the distributed test case execution traces that are captured by ITCAM - an IBM integrated management tool. A verification approach supporting Boolean expression and back-end service interaction verification is proposed to verify the test execution result.

Full integration functional testing can be generated only by a *control and data flow model* of the service composition [Mei et al. 2008] [Mei at al. 2009a] [Marconi et al. 2006] that specifies not only the control flow between interactions, but also the data flow requirements that are expressed as *transfer functions*, i.e. the formal correspondences (functional relationships) between the data involved in ingoing and outgoing interactions for each component. XPath is utilized for implementing transfer functions in BPEL applications. But XPath may extract wrong data from the XML messages received, resulting in erroneous results in the integrated process. In [Mei et al. 2008] Mei and colleagues utilise the mathematical definitions of XPath constructs as rewriting rules, and propose a data structure called XPath Rewriting Graph (XRG), which not only models how an XPath is conceptually rewritten but also tracks individual rewritings progressively. The mathematical variables in the applied rewriting rules are processed as if they were program variables, and are used to analyse how information may be rewritten in an XPath conceptually. The authors develop an algorithm to construct XRGs and a novel family of data flow testing criteria to test WS-BPEL applications. In [Mei at al. 2009a] the authors reuse the methods and techniques in order to test choreographies specified in WS-CDL. They utilise Labelled Transition Systems to specify message interactions for scenario-based specifications. By applying XRG patterns, they identify new data flow associations in choreography applications and develop related data flow testing criteria.

Bertolino and colleagues [Bertolino et al. 2011] present (role) CAST, a framework for on-line service testing. (role)CAST supports on-line testing of access policies of SOAP services whose roles have been defined by means of SAML assertion. The idea is to test a service composition within its real execution context. On-line testing consists of proactive service invocations designed by testers, and foresees the execution of such test invocations on a service while it is engaged in serving real requests (not be confused with run-time monitoring or passive testing. The authors “are aware though that such an approach poses big challenges, in terms of costs and potential impact, which may undermine its acceptance. Such challenges mainly account for possible testing side effects, in particular when stateful resources are considered. Nevertheless, in some contexts on-line testing can be regarded as a useful technique to increase trust among organizations interacting via deployed services.” (role)CAST is intended to test role –based access control and is made of four main components: (i) the Test Driver that configures and runs instances of Test Robots, (ii) the Test Robot that loads the test cases from a repository (iii) the Tester Backport, a component that extends the interface of a generic Identity Management Provider and and is used to collaborate with the test Robot, (iv) the Oracle, an abstract component that defines the minimal assumptions to be tested (partial oracle. (role)CAST provides the APIs that can be used to program the Tester Robot.

Besson and colleagues [Besson at al. 2011] [Besson et al. 2012] present Rehearsal, a framework for Test-Driven Development (TDD) on choreographies, that allows automating unit, integration, and scalability testing, and is available under the LGPL license[33]. Rehearsal uses SoapUI to automatically build a set of XML-Soap request envelopes to test service operations at runtime. Rehearsal provides features such as: (i) the WSClient, a dynamic generator of web service clients. With this feature, the developer can interact with a service without creating stub objects; (ii) message interceptor, a collection of mechanisms to intercept, store and then forward the messages exchanged between the services in the composition; (iii) WSMock, a feature for mocking services; with this feature, real services can be easily simulated by mock objects; (iv) Scalability Explorer that aims to support automated scalability tests by providing features for executing the choreography in different scenarios, collecting performance metrics, manipulating resources and reporting execution results.

REGRESSION TESTING

According to the systems and software engineering vocabulary[34], regression testing is selective retesting of a software system to verify that modifications do not cause unintended effects and that the software still complies with its specified requirements. Sometimes, re-testing, i.e. testing that the past failed test cases do not fail now, is considered as regression testing.

Yoo and Harman [Yoo and Harman 2012] investigate the utilisation of a number of different techniques to maximize the value of a regression test suite and to minimize the testing cost: (i) *test suite minimization* or *test case reduction* (*reduction*) techniques - that aim to eliminate redundant test cases in order to reduce the number of test runs, (ii) *test case selection* (*selection*) techniques – that select test cases from an existing test suite to test the modified part of the system and (iii) *test case prioritisation* (*prioritisation*) techniques that schedule test cases for running in an order that attempts to increase their effectiveness at meeting some desirable properties.

Reduction techniques try to eliminate redundant test cases in order to reduce the number of test runs. More precisely, *reduction* techniques aim at obtaining a minimal subset of a test suite that preserves a specified adequacy criterion (e.g.: coverage). Test suite minimization techniques are not really specific to regression testing and are often integrated into the process of test case generation as an optimization step and, in the service testing domain, has attracted less attention than others.

TEST CASE SELECTION

Regression testing based on large test sets accumulated over time, this is a costly process, especially if testing is manual. Often, changes made to a system are local, arising from fixing bugs or specific additions or changes to the functionality. Re-running the entire test set in such cases is wasteful. Instead, we would like to be able to identify the parts of the system that were affected by the changes and select only those test cases for rerun which test functionality that could have been affected.

[33] http://ccsl.ime.usp.br/baile/VandV

[34] ISO/IEC/IEEE 24765:2010 - http://www.iso.org/iso/catalogue_detail.htm?csnumber=50518

Selection techniques concern reusing test cases from an existing test suite to test the modified part of the system. The majority of test case selection techniques are modification aware [Yoo and Harman 2012], i.e. they require the identification of the modified parts of the SAUT.

Qiu and colleagues [Qiu et al. 2014] propose a universal framework to cover all possible change types in services architectures:

- Build change
    - Implementation change (ImC)
    - Binding change (BC)
- Model change
    - Interface change (IC)
    - Process changes (PC)

With *build change*, the implementations are changed, without any change of the structural, functional and behavioural model of the SAUT (corrective maintenance). The *implementation change* is a modification (new version) of a component implementation of a service provider with no change of the services' interfaces and of the SAUT structural, functional and behavioural models [Tarhini et al. 2006a]. *Binding change* is a replacement of a service component without any changes of the service interfaces and of the SAUT structural, functional and behavioural models – this is similar to the implementation change, with the difference that the service component is replaced with a component from another provider, instead of a new version of the component from the same provider [Li et al. 2012] [Di Penta et al. 2007].

With *model change*, the service models and processes change. *Interface change* is a change in the service interfaces (and the related implementations) [Li et al. 2012] – the interface change can be *backward compatible* (e.g. adding to the interface a new operation leaving the operational semantics of the other pre-existing operations unchanged) or not. *Process change* is a change in the service composition structure, control flow and data flow, without change of the service interfaces and implementations [Tarhini et al. 2006a] [Li et al. 2012].

A concrete change can mix several of the change types listed above.

All selection techniques reviewed in this section are dedicated to service composition. An important dimension of selection techniques is their *safeness*. With a *safe* technique, every test case from the original test suite that can expose faults in the modified program is still selected, i.e. all faults found with the full test suite are also found with the selected test cases [Rothermel and Harrold 1997].

Selection techniques are classified [Qiu et al. 2014] as: (i) Path analysis selection techniques; (ii) Graph walk selection techniques; (iii) Modification-based selection techniques; (iv) Dependency-based selection techniques.

Li and colleagues [Li et al. 2008] [Li et al. 2012] take the set of service paths in both old version *S* and new version *S'* as input, which are expressed as BFG (XBFG) paths generated from the BFG (XBFG) model, and compare the paths from *S* and *S'* to identify paths as new, modified, deleted, or unmodified. In particular, Liu and colleagues [Liu et al. 2007] address the consequences of concurrency in BPEL scripts for regression testing by proposing a test case selection technique based on *impact analysis* that identify the changes to the process under test and discover paths that are impacted by these changes.

Ruth and colleagues [Ruth et al. 2007] apply Rothermel's method [Rothermel and Harrold 1997] to general composite services. Their tool takes global control flow graphs of the original and of the modified systems, compares them, identifies the "dangerous" edges and selects the test case to be rerun. This approach assumes that the CFGs of participating services are provided by their developers via WS-Metadata Exchange[35] [Ruth and Tu 2007a]. In particular, Ruth and Tu [Ruth & Tu 2007b] propose an automated extension to the *selection* technique that tackles the concurrency issues. The multiple modified service problem is treated by using *call graphs* that make possible to determine the execution order of the modified services. A strategy called 'downstream services first' is applied in order to achieve fault localization. In this strategy, if a fault is found in a downstream service, none of the upstream services are tested until the fault is fixed. Ruth improves the approach by considering privacy issues on the edges (sensitive implementation details of participating services that service providers do not want to share) [Ruth 2008] [Ruth 2011]. In [Ruth and Rayford 2011], Ruth and Rayford expose an approach using only locally available information at each service and a publish/subscribe mechanism.

Wang and colleagues [Wang et al. 2008] propose an utilisation of the BPEL Flow Graph [Yuan et al. 2006] to put in place a BPEL regression testing framework that can generate and select test cases for regression testing using Rothermel and Harrold's selection technique [Rothermel and Harrold 1997].

Tarhini and colleagues [Tarhini et al. 2006a] identify modifications from generated Timed Labelled Transition Systems (TLTSs) [Tretmans 2008] and select test cases that would cover the modified part. This approach can also generate test cases for the new and existing services and operations.

Khan and Heckel [Khan and Heckel 2011] propose a model-based approach to the selection problem. In their approach service interfaces are described by visual contracts, i.e., pre and post conditions expressed as graph transformation rules [Heckel and Lohmann 2005]. *Dependency-based selection techniques* select test cases based on an analysis of the dependences and conflicts between visual contracts specifying the preconditions and effects of operations. The analysis of conflicts and dependences between these rules allows assessing the impact of a change of the signature, contract, or implementation of an operation on other operations, and thus deciding which of the test cases is required for re-execution. The approach is evaluated on a case study of a bug tracking service in several versions.

TEST CASE PRIORITISATION

Prioritisation techniques schedule test cases for running in an order that attempts to increase their effectiveness at meeting some desirable properties. The main goal of *test case prioritisation* is that of increasing a test suite *fault detection rate* (*time efficiency*), i.e. the early exposure of failures and localisation of source faulty elements. An improved fault detection rate can provide precocious and relevant feedback to the

[35] http://www.w3.org/TR/ws-metadata-exchange/

development team and can enable earlier debugging [Rothermel et al. 2001]. Test time efficiency is a must for Test-driven Development and Continuous Integration Testing.

Generally speaking, test cases are ordered by assigning a *priority* to each of them through some *primary rule*, and, possibly, test cases with the same priority are reordered through some *additional rule*.

Benchmark prioritisation techniques provide the lower and upper bound of the effectiveness of the other heuristics: (i) *randomly ordered technique*, (ii) *optimally ordered with maximum fault rate detection technique*. The latter technique is considered only as a theoretical upper-bound of effectiveness.

The prioritisation techniques are classified in two categories: (i) *Coverage-based prioritisation techniques* and (ii) *Fault-exposing potential prioritisation techniques*.

COVERAGE-BASED PRIORITISATION TECHNIQUES

Coverage-based prioritisation techniques are inspired from the traditional software testing goal of covering the whole program under test, and a quantitative measure of test coverage is considered an indirect measure of system quality. In service testing, coverage-based techniques rely on the coverage of *service-related elements*. Hence test cases are ordered on the basis of some *coverage criterion*, by an algorithm that implements a *coverage strategy*.

In the reviewed works, Service Activities (SAs) (with possibly a weight – Weighted SA [Chen et al. 2010]) and Service Transitions (STs) are the most commonly used coverage criteria. Other criteria are: invoked Services (Ss) [Askarunisa et al. 2010b], WSDL tag (any XML element defined in in WSDL/XSD) [Mei et al. 2009], WSDL occurrences (as a secondary rule to differentiate cases with the same WSDL tag coverage) [Mei et al. 2011]. For orchestrated architectures, other elements such as workflow branches (WBs), XPath Rewriting Graph branches (XRGBs), WSDL elements (WEs) and combinations of these preceding elements are used as coverage criteria, possibly for additional rules [Mei et al. 2009] [Mei et al. 2009] [Mei et al. 2013a] [Mei et al. 2013b]. The coverage strategy algorithms are in general variants of the *Greedy search algorithm*.

These prioritisation techniques can be compared by three dimensions. The first dimension is whether the coverage criterion is *functional* or *structural* (or both), i.e. if the service related elements that are covered are part of the structural architecture (white-box) or of the functional architecture (black-box) of the SAUT.

Another (complementary) dimension is the coverage *level of granularity* of the test case, i.e. the smallest element type that is taken into account to calculate the coverage. Generally speaking, more the coverage is fine-grained, more the time required for prioritising the test suite is longer.

Las but not least, coverage-based prioritisation technique can be classified as *general* or *specific* [Rothermel et al. 2001]. The key difference between them is the usage of a specific version's modification information on prioritising test cases. In the former case (general) the test case priority order will be useful over a series of subsequent modified versions of the SAUT. In the latter case, the order will be useful on a specific version of the SAUT. All the coverage-based techniques mentioned in this paragraph are general.

Chen and colleagues [Chen et al. 2010] propose a model-based test case prioritisation approach based on impact analysis of BPEL processes. The authors introduced a model called BPEL flow graph (BPFG) into which BPEL processes are translated for change impact analysis. Test cases are prioritized according to the proposed weighted dependence propagation model. Li and colleagues [Li et al. 2010] extend the BFG model into another graph called eXtensible BFG that they claim is better suited to regression testing.

Mei and colleagues [Mei et al. 2011] provide a strategy for black-box service-oriented testing. They formulate new test case prioritisation strategies using tags embedded in XML messages to reorder regression test cases and reveal how the test cases use the interface specifications of web services. Furthermore, they propose [Mei et al. 2012] *Preemptive Regression Testing* (PRT), for the regression testing of (dynamic) web services. A *dynamic* web service is a web service that can dynamically change its own processing logic or bind to and use new external services during the course of an execution (*late change*). Whenever a late-change on the service under regression test is detected, PRT pre-empts the currently executed regression test suite, searches for additional test cases as fixes, runs these fixes, and then resumes the execution of the regression test suite from the pre-emption point.

Athira and Samuel [Athira and Samuel 2010] depict a model-based test case prioritisation approach for service compositions. The approach discovers the most important activity paths using a UML activity diagram of the service composition under test. Athira and Samuel technique identifies the differences between original model and modified model. The information activity paths for each test case are plotted and the most promising paths are identified. The test cases which cover these paths are considered as the most beneficial test cases.

FAULT-EXPOSING POTENTIAL (FEP) BASED PRIORITISATION TECHNIQUES

The *fault-exposing-potential* (FEP) of a test case measures its ability of stimulating the execution of a faulty component in a manner that exposes the failure that reveals the fault [Elbaum et al. 2002]. These measures can be considered only as probabilities, and the calculation of these a priori probabilities can be only the result of approximations.

Tsai and colleagues propose the application of their group-testing approach [Tsai et al. 2004] to regression testing [Tsai et al. 2005] by using the test case *potency*. Group-testing is testing of several implementations of the same service interface, and the potency of a test case is its capability of revealing faults in a number of implementations. In [Tsai et al. 2007] [Tsai et al. 2008] [Tsai et al. 2009] they improve the computation of *potency* to combine *potency* and a *coverage relationship model* (*CRM*) between test cases as an additional test case ranking criterion. In group testing, test-case selection and ranking are the key factors in enhancing testing efficiency. Statistical data can be utilised [Whittaker and Thomason 1994] to rank test cases using two criteria:

- *Potency* – it is the probability that a test case can detect a fault (for example, if a test case fails 30 services out of 100, its potency is 0.3); and
- *Coverage relationship model* (CRM) – it is the amount of additional coverage one test case can bring given that we've applied other test cases.

Instead of evaluating how test cases are derived, CRM evaluates the test case coverage by learning the previous results. Because CRM is based totally on test results, two test cases derived from two different testing techniques and addressing two different code segments might have identical coverage in the CRM. Tsai and colleagues define multiple rules that guarantee the selection of the most potent test cases. The approach can be applied to regression testing when a new version of a service with the same specifications is created..

Other approaches [Bai and Kenett 2009] prioritise test cases on the basis of the *risk* of the targeted service feature, i.e. the products of the probability to fail of the feature and a measure of the impact of the consequence of the failure. The risky features should be tested earlier and with more tests.

Change sensitivity [Nguyen et al. 2011] measures the importance of test cases based on the assumption that sensitive test cases that potentially stimulate more service changes have a higher ability to reach the faults caused by changes. The method is based on the idea that the most important test cases are those that have the highest sensitivity to changes injected into the service responses (mutations). Nguyen and colleagues adopted several mutation operators to mutate services and calculate the change sensitivity based on the proportion of mutants that are "killed" (if mutated response of service differs from the original response).

Askarunisa and colleagues [Askarunisa et al. 2011] propose an automated testing framework for composite web services where the domain knowledge of the web services is described using Protégé tool[36] and the behaviour of the entire business operation flow for the composite web service is described with OWL-S. Prioritisation of test cases is performed based on various coverage criteria for composite web services. They simulate the capability of detecting faults by seeding faults. Test cases are prioritized by their *fault rate*, that is, detected seeded faults with regard to the execution time. In addition, *fault severity* was also proposed based on the combination of *fault rate* and *fault impact* (importance of a fault).

DISCUSSION

From the beginning, the research has correctly recognised the true fundamental issues that limit the testability of services architectures and confer to the SOA testing problem its specificity [Canfora and Di Penta 2009] [Bartolini et al. 2011] [Bozkurt et al. 2013]:

- limitations in observability of service code and structure due to information hiding of the implementations, the only exception being the observability of the orchestration scripts in orchestrated service compositions (BPEM) that encourage the use od white-testing techniques. This issue limits the testing of services architecture to black-box testing of single components and grey-box testing of multi-component architectures;
- lack of control of the services architecture component development life cycles, due to independent ownership of these components and to the independent infrastructure on which these components run.

The two issues above are not so different from those that characterise component off-the-shelf testing [Rehman et al. 2007].

From the beginning, the most important research thread in SOA functional conformance testing has been *test case generation*. In contrast, the automated generation of test oracles, that is a key point of functional conformance test automation, is still the focus of a minority of investigations and approaches. The difficulty of the test oracle generation task is related to the fact that, by definition, it can start only from a formal specification/model of the structure, functions and behaviours of the services architecture components.

[36] http://protege.stanford.edu/

The important emerging trend in the SOA testing research domain is model-based testing (MBT). Firstly, MBT has redesigned the test generation panorama, integrating previous approaches (specification-based, contract-based, category partition) in a more coherent and uniform vision. Whereas test automation replaces manual test execution with automated test scripts, MBT replaces manual test designs with automated test designs and test generation. The first-generation MBT showed various shortcomings such as failing to distinguish between white-box and black-box models. The white box model can be used for code generation and white-box test case generation, but only the black-box model of a single service component and grey-box models of service component architectures can be used for black-box and grey-box test case generation.

"Second-generation" MBT uses specific black-box and grey-box models (called also scenario, usage or environmental models) and a three-phase-process for functional conformance testing: (i) design black-box and grey-box models that must be precise and rich enough to allow automated derivation of functional conformance test cases and oracles from them; (ii) determine test generation criteria, either on the basis of structural model of the services architecture under test (coverage), or on the basis of expected or known faults (fault index), or driven by risk (the product between fault index and fault gravity) and ; (iii) generate the tests [Schieferdecker 2012]. But all the approaches reviewed lacks surprisingly of an effective structural (meta-)model of the SAUT as a distributed service component architecture that is necessary for fault localisation.

About service composition testing, a very important effort is concentrated on testing service *orchestrations* (BPEL). The main reasons are that the service composition is driven by a program (e.g. a BPEL script) whose source text is available (allowing white-box testing techniques), that is run by a special agent (the orchestrator) whose internal behaviour (the execution of the script) is accessible too. Hence, the same white-box test methods that have been applied to program testing are applicable to orchestration testing. This research thread should not be overestimated for two reasons: (i) the utilisation of the orchestration paradigm has not spread over the development of the web service technology and is confined in a niche market; (ii) the development of the API economy and the emergent popularity for REST-based and JSON-based service approaches has made the aforementioned niche even smaller. The competing *choreography* decentralised paradigm has been the target of interesting research and experimentation [Wieczoreck et al. 2008] [Wieckzorek et al. 2009], but is also a small market niche, even less significant than orchestration. *Basic service composition*, i.e. direct service exchange between service components without orchestration and choreography, is the prevalent service composition approach in the API economy, but has drawn the attention of a limited number of researchers.

So, the test case generation for basic service composition is now an important research thread. In the MIDAS project, service component behaviour is modelled through Protocol State Machines represented in the emerging standard SCXML [De Rosa et al. 2014b] based on Harel's state charts [Harel and Politi 1998]. This approach is independent from the service exchange technology platform and can represent service compositions in which all the interoperability platforms (SOAP, REST/XML, REST/JSON) are utilised. The *control flow requirements* are represented by the states/transitions of the PSM, the *pre/post-conditions* are represented by Boolean expressions on elements and values of the messages and the component state view resources, and *data flow requirements* are represented by *transfer functions* expressing the elements and values of an outgoing message as functions of the elements and values of the ingoing messages and state view resources. Boolean expressions and transfer functions are expressed in ecmascript.

The fundamental weakness of all (in the author's best knowledge) the MBT research works applied to SOA testing is (i) the lack of a clear distinction between the structural, functional and behavioural models of the

SAUT, and (ii) the weakness of the structural modelling elements that are essential for fault localisation, especially in gey-box testing. All the approaches utilise as models sequence diagrams, activity diagrams, orchestration (BPEL) scripts, choreography models, states machines and variants that mix the structural, behavioural and functional information about the SAUT. In particular, there is no autonomous structural model of the SAUT that expresses the topology of the components, their required and provided service interfaces, how the components are "wired" through these interfaces, independently from the scenarios of service exchange, orchestration, composition. The lack of this specific model prevents the effective localisation of faulty elements. As explained in section 4 (§ 'The SAUT Construction model'), the SAUT structural meta-model allows enabling the test system, through the Scheduler probabilistic inference, with troubleshooting (location of faulty elements) capability.

Speaking about test run automation, an important step has been taken with the "invention" of TTCN-3 [Willcock et al. 2011]. TTCN-3 is an international standardised test language for defining executable test specifications for a wide range of computer and telecommunication systems. TTCN-3 is a general programming language that has been designed with powerful traits that are specific for test automation. It allows complete automation by programming of the test execution and arbitration tasks. The research on the utilisation of TTCN-3 for service testing has started early [Schieferdecker and Stepien 2003] [Xiong et al. 2005] [Werner et al. 2008] [Peyton et al. 2008].

Hence, the only environment that pushes as far as possible the automation of the execution and arbitration tasks in grey-box testing is that based on the powerful TTCN-3 technology [Peyton et al. 2008] [Xiong et al. 2005]. TTCN-3 allows implementing complete test systems for grey-box testing of complex services architectures very easily. These systems are easily provided with "agents" playing *emulators* (as requesters and responders) of service components (for service virtualisation) and *interceptors* for observing the interaction between service components. Also the problems related to the automated management of the state of stateful components, i.e. setting, getting and resetting the visible state variables (state view) could be easily solved by TTCN-3 programming. As already noticed, also this environment lacks singularly of tools for building and using the structural model of the SAUT. This absence prevents the full automated generation of the test system.

De Rosa and colleagues [De Rosa et al. 2013], including the author of this manuscript, have implemented a generic and complete model-based TTCN-3 framework for testing services and service compositions. This framework is able to integrate not only a formal representation of the test scenarios (message sequences) and of the test suite, but also a structural model of the SAUT. Starting from these formal representations, the framework implements the execution/arbitration automation of the scheduled test run cycle, including (i) the configuration/generation of the test system, containing the test components (proxies and emulators of the service components, interceptors on the channels between SAUT components) (ii) the integration of the test suite, (iii) the management of the SAUT component state variables, (iv) the binding of the test system with the SAUT components. This framework allows the scheduler to drive the automated dynamic scheduling of the execution and arbitration tasks. The same elements (SAUT structural model, test suite) allow configuring, generating and initialising the TTCN-3 executor/arbiter and the Bayesian agent that realises the dynamic scheduler.

Research about test scheduling is still at its infancy. The few papers that treat the issue are in the section on regression testing. Apart from test *reduction*, which should be considered as an optimisation step of the test case generation process, the works on regression testing propose either test case *prioritisation*, or test case *selection*. The majority of regression testing techniques that are proposed in the literature are adapted to white-box testing, which is out of the scope of this research.

Selection techniques for *implementation change* (change of the component implementation, without any change of the structural, functional and behavioural model of the SAUT) require the identification of the modified parts of the SAUT. With grey-box testing of service component architecture, we have seen that the structural description of the SAUT stops at the atomic component level – the atomic component being the component that cannot be described as a service sub-component architecture - the further sublevels (service interface, service interaction type, message) are functional. Hence, the smallest modified part of the SAUT is the atomic component, and regression testing (looking for unexpected side effects) should maximise the functional coverage of the component, where re-testing should focus on the functional elements (service interfaces, interaction types) that has failed in the "last" test session and whose fixing should have been achieved with the new implementation version. An interesting situation could be when the test system can obtain from the SAUT deployment system not only the identity of the component whose new build is deployed, but also the trace of the failures revealed in the past test sessions that are in principle fixed by the new build. What is the possible usage of such information is a matter of future investigation.

Regression testing on *binding change* is a kind of *acceptance test* of the new component implementation. With black-box or grey-box testing, the *selection* criteria should be the maximum functional coverage of the atomic component or the maximum structural/functional coverage of the composite component (whose service sub-component architecture is observable).

In case of *interface / process* change, there are two different situations: (i) the interface / process change is *backward compatible* (new operations are added to the service definition without modifying the existing operations, new processes are designed on the basis of the existing services) – the existing test suite can be reused as such, for instance to test the impact of the change; (ii) the interface/process change is not backward compatible and some test cases are no more applicable – it depends on the breadth of change, and, in the author's best knowledge, there are no automated treatments available for the problem when it is considered in its generality.

All the *coverage-based* prioritisation techniques reviewed in the preceding section and are *non-version-specific* (*general*) and a majority of them are *white-box*, allowing *structural* coverage. All these techniques allow prioritising test cases with the highest (structural or functional) coverage without any consideration for the modified part of the SAUT.

The *fault-exposing potential based* prioritisation techniques utilise the ability of the test case in detecting faults that is measured as a probability of stimulating a faulty component in a manner that reveals the fault. The techniques reviewed are all general (non-version-specific): estimation of the test case potential do not consider the specific modifications present in the modified version of a program; however, they attempt to factor in the potential effects of modifications in general.

The weakness of the approaches based on the statistical evaluation of the fault-exposing potential in group testing is that it is dubious that the frequency of failure of a test case on a collection of independently designed and implemented service components is a measure of the prior probability of failure on the next service component. Other approaches based on risk and change sensitivity seem more realistic but are always white-box approaches.

This work introduces the dynamic scheduling of functional conformance test sessions as a new research topic in the testing domain and proposes a generalised approach to the dynamic prioritisation of test cases, beyond the current application of static prioritisation to regression testing and re-testing, based on probabilistic inference.

4. THE SCHEDULER ARCHITECTURE

The Bayesian Scheduler is packaged as a Web service on the MIDAS platform and is able to provide its services to different MIDAS test methods. This section situates the Scheduler in the more general architecture of the test cycle and documents its interface and behaviour.

THE MIDAS TESTING FACILITY

The MIDAS testing facility, which is the main result of the MIDAS project, targets black-box testing of single services and grey-box testing of services architectures[37]. Its key features are:

1. Testing-as-a-service implemented on a public cloud infrastructure.
2. Programmable testing facility through service APIs.
3. Open platform equipped with an evolutionary registry/repository of test methods.
4. Extreme automation of SOA/API testing tasks.

The end users of MIDAS are service developers that utilise the MIDAS facility to automate the test of the services architectures that they implement and utilise.

TESTING-AS-A-SERVICE IMPLEMENTED ON A PUBLIC CLOUD INFRASTRUCTURE

The MIDAS testing facility is implemented on a public cloud infrastructure[38] and is delivered as a service. The user has nothing to install on premises: s/he interacts with the MIDAS test functionalities through Application Programming Interface (APIs) on the Internet (Web services). Furthermore, the MIDAS portal provides utilitarian (documentation, learning, ancillary tools …), administrative (accounting, billing …) and community services through a Graphical User Interface (GUI).

The MIDAS facility exhibits all the features of an advanced software-as-a-service:

- self-provisioning - the user selects the testing services that s/he wants to utilise;
- scalability and elasticity of computing and storage resource allocation;
- control and security of access;
- integrity and privacy of data (models and test data);
- pay per use and transparent accounting and billing;
- documentation and e-learning;
- user community tools.

[37] Other terms generally utilised with the same meaning: ‘service integration testing’, ‘service composition testing’. The service unit test can be seen as a “collapsed” case of service integration test.

[38] The MIDAS underlying cloud infrastructure is Amazon Web Services (http://aws.amazon.com/).

Once the end user has obtained her/his credentials, s/he has access to the MIDAS functionalities. The accounting and billing policies are based on the metering of the utilisation of the underlying public cloud computing resources, third party resources and MIDAS “logical” resources (testing services, ancillary services).

One important concern of testing and troubleshooting research is the testing cost. The cloud implementation of the testing facility lowers substantially (at least one order of magnitude) the global cost of physical computing resources, guarantees the scalability and elasticity of these resources and reduce to zero their marginal cost. Of the three cost components - equipment expense, labour effort, time-to-market (section 2, § ‘The rationale for service testing automation’), the latter two become the most important cost issues. Paradoxically, the easy availability of cheap computing resources on cloud makes the computing complexity of several automated testing tasks (for instance, test generation and test prioritisation) more critical.

PROGRAMMABLE TESTING FACILITY THROUGH SERVICE APIS

MIDAS is not a site, it is a service. The MIDAS test methods and functionalities are fully accessible through public Application Programming Interfaces (APIs) that are documented on the MIDAS portal. Thus, the MIDAS facility is *programmable*: the service developer carries out client software that invokes the MIDAS testing services. The MIDAS APIs allows the service developer to integrate the MIDAS testing services with her/his Service Development Life Cycle (SDLC) environment.

More specifically, the interaction through API allows the developers of:

- Modelling tools,
- Integrated Development Environments (IDEs),
- Application Lifecycle Management (ALM) tools,
- Computer-Aided Software Testing (CAST) tools,
- SOA governance tools,
- API management tools,

to build testing processes that integrate the MIDAS services within their environments. As a proof of concept, a MIDAS-profiled Eclipse Modelling Framework[39] that is able to access the MIDAS facility is downloadable from the MIDAS portal.

The MIDAS APIs allow invoking testing services are *asynchronous* and *non-blocking*. They enable not only automatic invocation but also asynchronous and background execution.

OPEN EVOLUTIONARY PLATFORM

The MIDAS facility is built on an *open platform*. The researchers and experts on testing and the developers of testing tools usually produce *test components* that automate *testing tasks* (test case/oracle generation, test execution, test arbitration, test reporting, test scheduling, test planning) following specific approaches. In the MIDAS terminology, these contributors are called *test method developers*.

[39] http://www.eclipse.org/modeling/emf/

The test method developers are able to upload, register and install their test components on the MIDAS platform. The functionalities implemented by the test components, or by the combination of these components (*test composites*), that are installed on the MIDAS platform are published on the MIDAS facility and are made accessible to end users as *test methods*. The relationship between *test methods* and the *test components/composites* is similar to that occurring between *services* and the *systems* that implement them. Thus, test components are loosely coupled with the test methods that they implement. A test component that provides a specific service (for instance, test scheduling) can be reused within the implementation architecture of several test methods.

Moreover, the MIDAS API that allows MIDAS users to invoke test methods is *generic*. It offers a restricted number of *generic* operations that are the same for all the test methods, but are able to convey input and output documents whose schemas are *specific* to the test method. The generic API is implemented by the MIDAS gateway, which is a front-end that selects the test method on the basis of its identifier, allocates the resources needed for its execution, deploys the components that implement the method, routes the request with its specific content to the front component and manages the exchanges with the client.

The test method developers implement test components, and embed them on virtual machines on their premises. Afterwards, they upload the virtual machine images on the platform, register them on the test component repository and register the related test methods on the test method registry. This process is executed in a MIDAS sandbox that allows safely checking, verifying and testing in isolation the method and its components. Once a certification process is achieved successfully, these test methods are published on the registry and made available to the MIDAS users.

The objective of the MIDAS open platform is to “capture” new advances in the research and practice of SOA/API testing and to make them available for the MIDAS users as new enhanced test methods that foster the automation of effective test cycles.

EXTREME AUTOMATION OF SOA/API TESTING TASKS

The proof of concept of the full automation of all the testing tasks is the main objective of the MIDAS project. The MIDAS features described in the preceding paragraphs are instrumental for this objective. The MIDAS partners “play” the role of test method developers and develop test methods in the domains of: (i) functional testing; (ii) security and vulnerability testing; (iii) usage-based testing. These test methods aim at putting in place automation tools for:

- the basic testing tasks (test case production, test oracle production, test execution, test arbitration and test reporting),
- the test generation cycle as a whole,
- the test run cycle as a whole,
- the test cycle as a whole, and its interaction with the *debugging/fixing* cycle.

The components that bring the automation the testing tasks are presented in a tree diagram sketched in Figure 12. The basic test tasks are automated by specific components (test case generator, test oracle generator, test executor, test arbiter, test reporter). The test generation cycle is driven by a generation workflow. The test run cycle is driven by a test Scheduler. A test planner drives the overall test cycle.

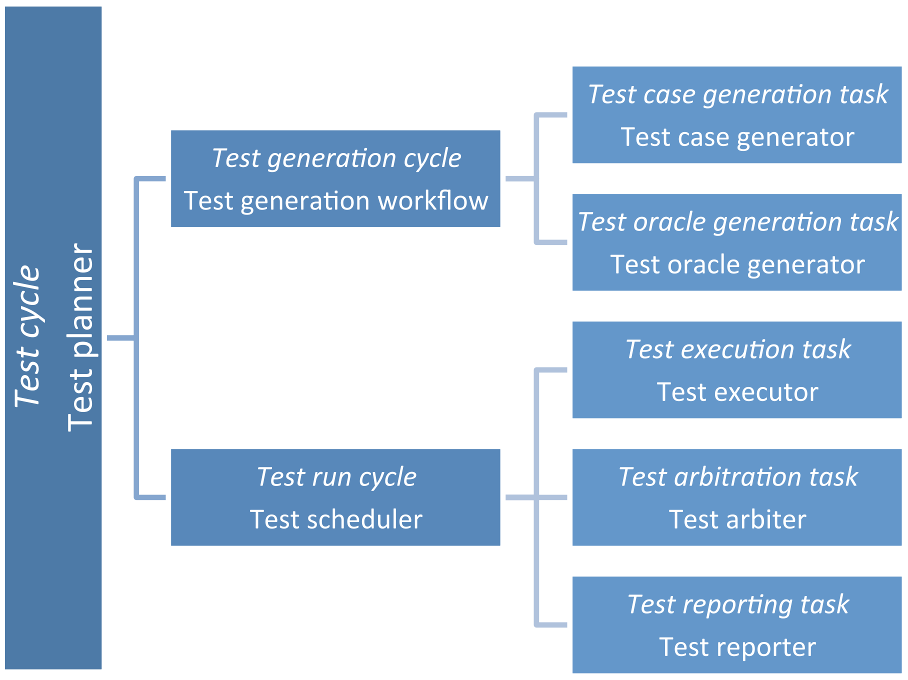


Figure 12. Test cycles and tasks and test automation components.

This automation approach is model-based. Advanced test automation is driven by the availability of SAUT structural, functional and behavioural models. The degree of test automation is related directly to the model richness.

The objective of test extreme automation available on a programmable TAAS facility is a unified Service Development Life Cycle characterised by:

- Model-based service design and test – services and services architectures models drive both the development cycle and the test cycle;
- Test-Driven Development – tests can be produced from models automatically, concurrently with the development of the implementation code, and can be executed automatically as soon as the code is released; tests can repeated automatically against each new implementation build (regression testing automation);
- Incremental Integration and Continuous Integration Test – partial service integrations can be put in place and tested as soon as integrated; tests can be automatically enriched and reiterated at each incremental integration step of new service components.

We have seen that the cloud implementation of the MIDAS facility lowers the testing equipment expenses of at least one order of magnitude. Extreme automation and programmability of the test system contribute to lower the other two cost component of test: the labour effort and the time to market.

THE LOGICAL ARCHITECTURE OF THE DYNAMICALLY SCHEDULED TEST RUN CYCLE

The dynamic Scheduler chooses the next (*n*+1) test case to run on the basis of the past (1..*n*) test verdicts. The implementation of automated dynamic scheduling requires the establishment of a test run cycle architecture that involves modules implementing the execution and the arbitration tasks.

The conceptual schema of the automated schedule/execute/arbitrate cycle proposed by this research is depicted in Figure 13, where a diagram sketches a dynamic scheduler (*Scheduler*), an interactive executor (*Executor*) and an interactive arbiter (*Arbiter*) and the logic of data exchange between them.

The dynamic scheduling cycle can be described as follows. The *Scheduler* chooses the next *test case* to be executed - starting from a first one that is choses on the basis some initialisation procedure - and communicates it to the *Executor*. The *Executor* runs the *test case*, collects/logs the *test outcome* and communicates this outcome to the *Arbiter*. The *Arbiter* evaluates the *test outcome* with the test oracle and produces a *test verdict* that is communicated back to the *Scheduler*. The *Scheduler*: (i) either chooses another not yet executed *test case* from the test suite, communicates it to the *Executor* and the cycle continues, (ii) or decides to halt definitely or suspend temporarily the cycle even if there are still *test case*s to run – possibly, the cycle can be resumed, (iii) or ends the cycle - there are no more *test case*s to run.

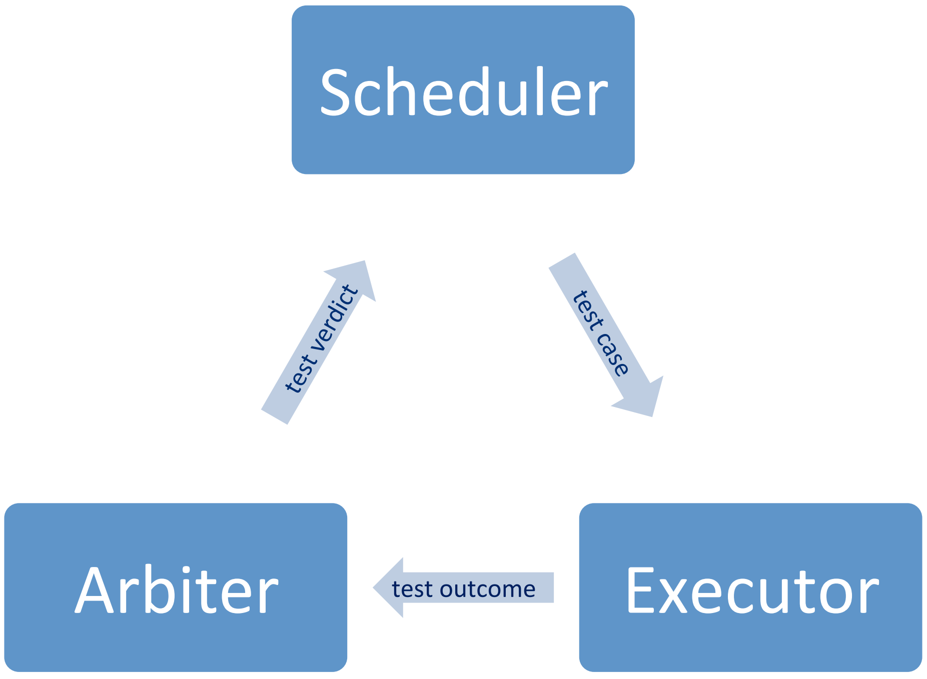


Figure 13. Conceptual schema of the automated schedule/execute/arbitrate cycle.

This scheduling architecture is logical in the sense that its primary goal is to clarify the logical relationships between abstract entities (*Scheduler*, *Executor* and *Arbiter*) and the abstract objects that are exchanged among them in a way that is independent from any concrete implementation of these entities and of their interactions. This conceptual architecture of the schedule/execute/arbitrate cycle has been proposed by this research as a framework for the implementation of probabilistic inference for test scheduling. Both the conceptual and the concrete architectures are adopted by the MIDAS project [Maesano et al. 2013b].

With automated and intelligent dynamic scheduling available on a testing platform as a service [Maesano and De Rosa 2011] [Maesano et al. 2011] [Maesano et al. 2013] that already implements automated test execution and arbitration, the entire *test run cycle* is automated and programmable. This automated test run cycle can be executed in background, allowing service developers to focus on their primary mission, i.e. design and implementation of appropriate services.

THE SCHEDULED TEST RUN SERVICES ARCHITECTURE PATTERN

The internal architecture of the MIDAS platform is generic and service oriented. All the test components that implement test methods are realised as service providers in a flat (non-hierarchical) architecture that includes the test components that realise the basic test tasks (test case generators, test oracle generators, test executors, test arbiters, test reporters) as well as the intermediate drivers (test generation workflows, test schedulers) and the top drivers (test planners).

A specific test method interface instantiates the generic interface with the method-specific input/output document schemas. A specific test method implementation realises the test method specific interface and is built as a *service composite*. In this architecture the test components providing test task specific services (test execution services, test arbitration services, test scheduling services …) are *orchestrated* by a main component that: (i) implements the test method interface and interacts with the test method consumers and (ii) drives the execution of the specialised test components.

The generic test run method pattern proposed by the MIDAS project has the prototypical internal architecture sketched in Figure 14[40]. The operating environment is organised in three regions: (i) the User environment in which operates the user client software, (ii) the MIDAS Gateway in which operates the MIDAS Front-end and (iii) the MIDAS internal execution environment for test methods, in which the services architecture of test components that implement the run test method is deployed and executed.

[40] The sketched architecture is generic and many technical details are dropped from the real one that is available on the MIDAS facility in order to highlight the relevant features.

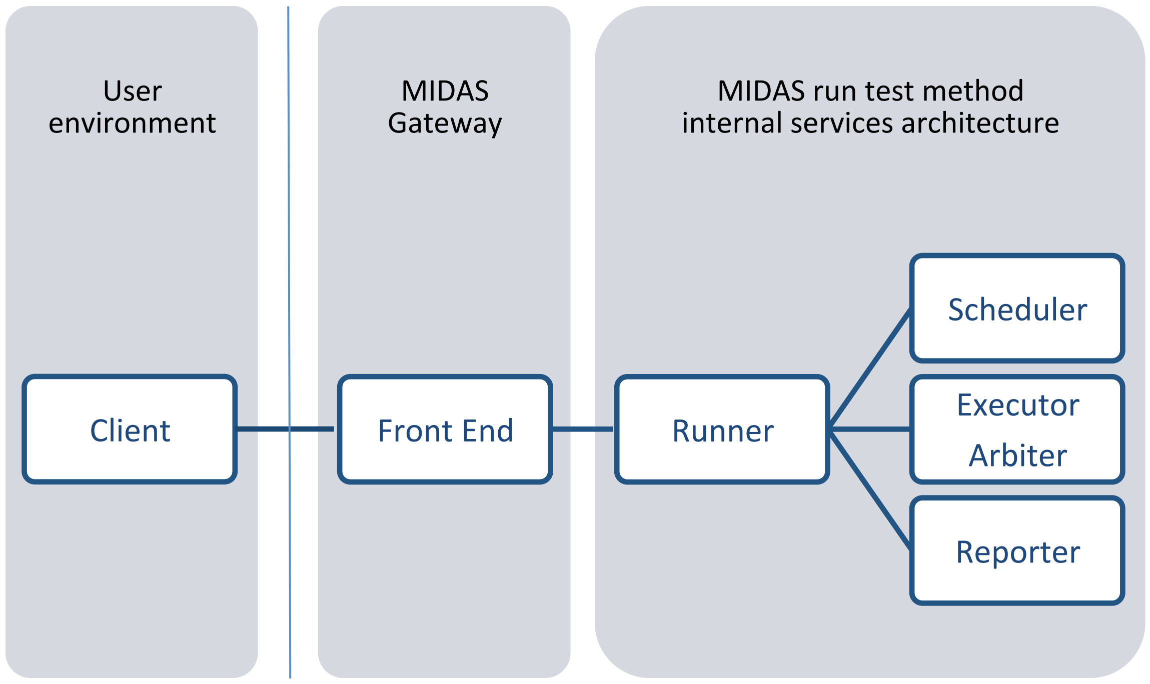


Figure 14. MIDAS scheduled run test method architecture pattern.

The generic test run method is implemented by a service component architecture pattern with four components (Runner, Scheduler, Executor/ Arbiter and Reporter). The Runner exposes to the Front End the generic service API (core Run API) specialised to this run test method and the other internal components expose to the Runner their specialised service APIs (Sched, Exec/Arb and Rep). Note that the task specific test components do not interact directly: they exchange information indirectly via the Runner. Furthermore, thanks to the MIDAS services architecture, each specific component is not reserved to the exclusive use of a specific method but can be reused in other test method implementations.

The sequence diagram of the scheduled test run cycle realised by the services architecture sketched in Figure 14 is illustrated in Figure 15. The user client software (*Client*) invokes the run test method on the MIDAS front end (*Front End*) through the generic *invokeTestTask* operation with, as parameters, the method universal identifier and an input document, which is compliant with the Input schema associated to the method. The *Input* document contains information such as SAUT Construction model locator, the Test Suite Definition model locator, the Test Suite data set locator and other parameters. The *Front End* controls the credentials of the client, looks for the run test methods on the *test method registry* using the *test method identifier*, gets the description of component architecture that implements the method, locates in the test component repository the test components that implement the method. If these actions are successful, the *Front End* allocates the cloud resources for the test composite architecture deployment and installs the architecture. When everything is configured and ready to go, the *Front End* invokes the method deployed entry point (the *Runner*), passing to it the Input document through the core *Run* interface.

The *Runner* invokes the initialisation operations on the *Executor/Arbiter* and the *Scheduler* and supplies them the appropriate information taken from the Input document, in particular the locators of the SAUT Construction model, the Test Suite Definition model and the Test Suite data set. This initialisation phase is complex: (i) the *Scheduler* utilises this phase to build in two sequential steps the virtual Bayesian Network and the Arithmetic Circuit that handles the probabilistic inference. The *Executor / Arbiter* component configures the Test Execution System and perform the binding between the Test Execution System and the SAUT. When the *Scheduler* and the *Executor / Arbiter* are initialised, the *schedule/execute/arbitrate* cycle can start. The *Runner* journalises in the test log all the exchanges with the test components.

This pattern is able to implement the schedule/execute/arbitrate cycle that operates as described in the preceding section: the Runner interrogates the *Scheduler* about the next test case to execute, then invokes the *Executor / Arbiter* with the returned test case. The *Executor / Arbiter* drives the execution/arbitration of the test case and returns the corresponding test verdict and the cycle is repeated until the *Scheduler* returns either a halting directive or the end of the test suite, together with information about the test session and the state of the SAUT. Afterwards, the Runner invokes the Reporter, passing to it the returned information from the Scheduler and the test log. The Reporter builds a meaningful test report that is intended for the debugging/fixing team.

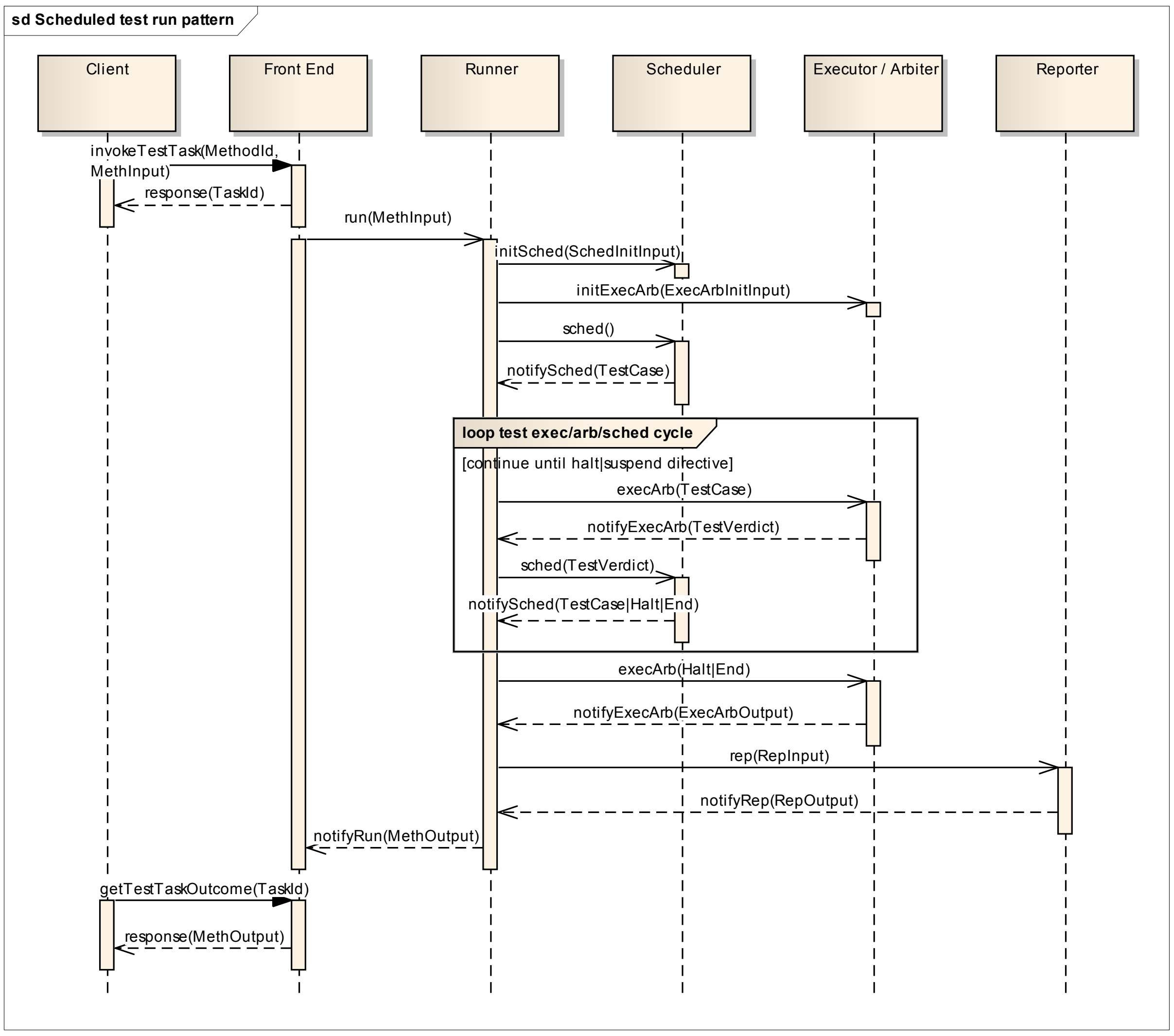

sd Scheduled test run pattern
Client
Front End
Runner
Scheduler
Executor / Arbiter
Reporter
invokeTestTask(MethodId, MethInput)
response(TaskId)
run(MethInput)
initSched(SchedInitInput)
initExecArb(ExecArbInitInput)
sched()
notifySched(TestCase)
loop test exec/arb/sched cycle
[continue until halt|suspend directive]
execArb(TestCase)
notifyExecArb(TestVerdict)
sched(TestVerdict)
notifySched(TestCase|Halt|End)
execArb(Halt|End)
notifyExecArb(ExecArbOutput)
rep(RepInput)
notifyRep(RepOutput)
notifyRun(MethOutput)
getTestTaskOutcome(TaskId)
response(MethOutput)

Figure 15. Sequence diagram of the scheduled test run pattern.

THE SCHEDULER INTERFACE

As already stated, the implementation architecture of a MIDAS test method is service oriented: the internal test components that implement a test method cooperate by service exchange through APIs.

The Scheduler automates the test scheduling task for a family of functional conformance test methods. It specialises the generic scheduling core interface, whose WSDL/XSD document is in the section 0.

In fact, the Scheduler service interface (Sched) is bidirectional: both the provider (the Scheduler) and the consumer (the Runner) must expose an interface. The Scheduler service operations that are coordinated through the Sched service interface are listed in the Table 5.

Service operation	***Provider interface***	***Consumer interface***
Initialisation	initTestSched	notifyInitTestSchedOutcome
Test sample request	requestTestSched	notifyTestSchedOutcome
Abort	abortTestSched	notifyAbortTestSchedOutcome

Table 5. Scheduler service operations.

INITIALISATION REQUEST

The Scheduler launches the Initialisation phases when receiving the initTestSched operation request. The request provides the TestSessionId and the SAUT/TSD/TS files' URIs.

The initialisation is effected in two steps:

- the creation of a virtual Bayesian Network (vBN) from the SAUT Construction model and the Test Suite Definition model and the Test Suite data set;
- the generation of an Arithmetic Circuit (AC) from the vBN.

This process is described in detail in the following section 6. Once the initialisation is accomplished successfully or failed (an exception is raised) the Scheduler returns notifyInitTestSchedOutcome.

If the AC can be built successfully, it contains the probability distributions and dependences that are utilised in the Scheduler decision process (see section 6).

The Scheduler saves the AC data structure on disk in a lightweight XML format called the **AC Image**. The Scheduler initialisation can be supplied with the AC Image URI, instead of the SAUT/TSD/TS files' URIs. A typical utilisation of the AC Image is for re-testing and regression testing.

TEST SAMPLE REQUEST

The invocation of the requestTestSched from the Runner on the Scheduler starts a new schedule/execute/arbitrate cycle. The Runner takes as input the TestSessionID and a set of test verdicts collected from the Arbiter during the previous schedule/execute/arbitrate cycle.

In order to identify the next test sample to execute, the Scheduler inserts the collection of local verdicts in the corresponding AC Interaction nodes as observations. More specifically, the verdicts at scheduling cycle n ($\mathrm{V_n}$) are added to the collection of past evidences ($\mathrm{e_{n-1}}$) to form the actual knowledge of the system behaviour: $\mathrm{e_n} = \mathrm{e_{n-1}} \cup \mathrm{V_n}$. The insertions of $\mathrm{V_n}$ in the AC consist in changing the evidence nodes into permanent constants nodes valued to 0 or 1 (pass or fail).

The Scheduler returns a notifyTestSchedOutcome message that provides one (or more) test sample identifier(s) selected on the basis of the policies described in the next section.

ABORT REQUEST

The requestTestSchedAbort erases the AC that is built by the initialization procedure and modified by the inference. The completion of the task is confirmed by a notifyTestSchedAbortOutcome message.

THE SCHEDULER STATE MACHINE

With respect to its internal inference engine, the Scheduler can be seen as an automaton with two states: Non-Initialized and initialized (see Figure 16). The Scheduler changes its state according to the operation request that it receives. Note that in the Initialised state the Scheduler can be re-initialised.

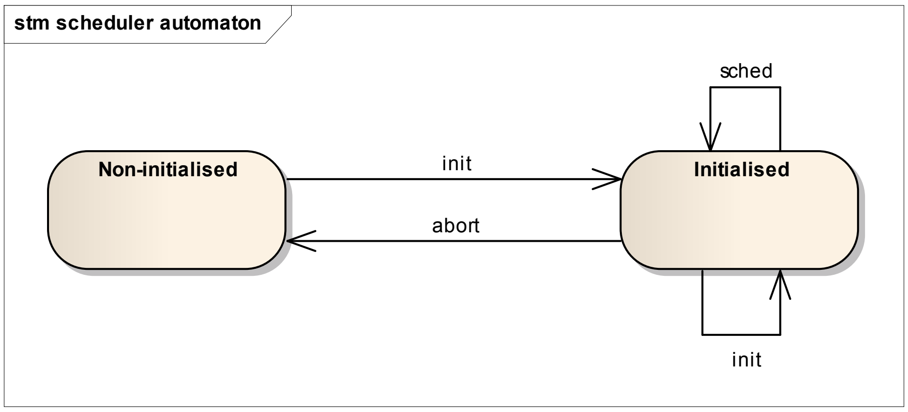


Figure 16. Scheduler inference engine protocol state machine.

THE SCHEDULER AS A STAND-ALONE SYSTEM

The Scheduler is implemented as an independent service that is loosely coupled with its interlocutors: it is an "oracle" system based on a probabilistic inference engine. Hence, it can be used stand-alone as a support of a user that either interacts directly with the test system (the executor/arbiter), or utilises the Scheduler as a support for simulation and what-if reasoning.

Note that the Scheduler *suggests* the next test case to run on the basis of the past test verdict, but does not expect the interlocutor's acceptance of its suggestion. In the next request for test samples, the user can supply to the Scheduler test verdicts related to the execution/arbitration (actual or virtual) of test samples that are different from those suggested in the last cycle, and the Scheduler is able to perform perfectly the probabilistic inference step on those “unexpected” evidences. This is true also for the automated scheduled test run cycle described above: a “smart” Runner can decide to give to the Executor instructions that are different from the suggestions of the Scheduler and, in the next step, to supply to the Scheduler test verdicts related to the execution of other test samples. This mechanism can be interesting for the development of the scheduling policies (see section 7) and for other usages that will be investigates in the future (section 0).

The stand-alone utilisation mode described above is utilised to “test”: the scheduler mechanism, (ii) the scheduler functionality and its policies (with respect to a specific SAUT/TSD model and TS data). The investigation of its utilisation as a simulation and what-if analysis tool will be investigated in the future (section 9).

THE SCHEDULER CONFIGURATION MODELS AND DATA SETS

The test scheduling approach of this work is *model-based*. In the initialisation phase, in order to build its embedded inference engine, the Scheduler uploads the SAUT Construction model, the Test Suite Definition model and the Test Suite data set.

THE SAUT CONSTRUCTION MODEL

The SAUT Construction model is a structural model of the services architecture under test and its surrounding environment. It describes the SAUT as a *construction* made of *components* that are connected by *wires* that represent active service dependences (connections between *required* and *provided interfaces*). It can be represented by a directed graph (components = nodes, wires = edges). The diagram in Figure 17 presents a graph of an example of SAUT construction.

A SAUT construction is made of: (i) the *SAUT composition,* i.e. the set of components (called *actual* components) that are part of the actually deployed services architecture - in the diagram they are represented by the black nodes inside the *SAUT* composition boundary; (ii) the *SAUT environment*, i.e. the set of components (called *virtual* components) that *are not part* of the deployed services architecture but that are linked through *at least one* wire with *at least one actual* component - in the diagram they are represented by the grey nodes situated between the SAUT composition boundary and the SAUT environment boundary; (iii) the *SAUT structure*, i.e. the collection of all the wires between the components of the union of the composition set and the environment set – that are represented by the directed links between nodes. Note that between two *components* there could be more than one *wire*. In Figure 17 the white nodes, outside the SAUT environment boundary, represent components that *are not part of* the SAUT construction.

In a SAUT construction there are as many *wires* as many active service dependences, each *wire* linking a *required service interface (reference)* to a *provided service interface (service)*. The wires of the SAUT structure are classified as (i) *ActualToActual* (between two actual components) wires – for instance between **comp01** and **comp03**, (ii) *VirtualToActual* wire (from a virtual component to an actual component) – for instance between **comp08** and **comp01**, (iii) *ActualToVirtual* wires (from an actual component to a virtual component) – for instance between **comp04** and **comp07**.

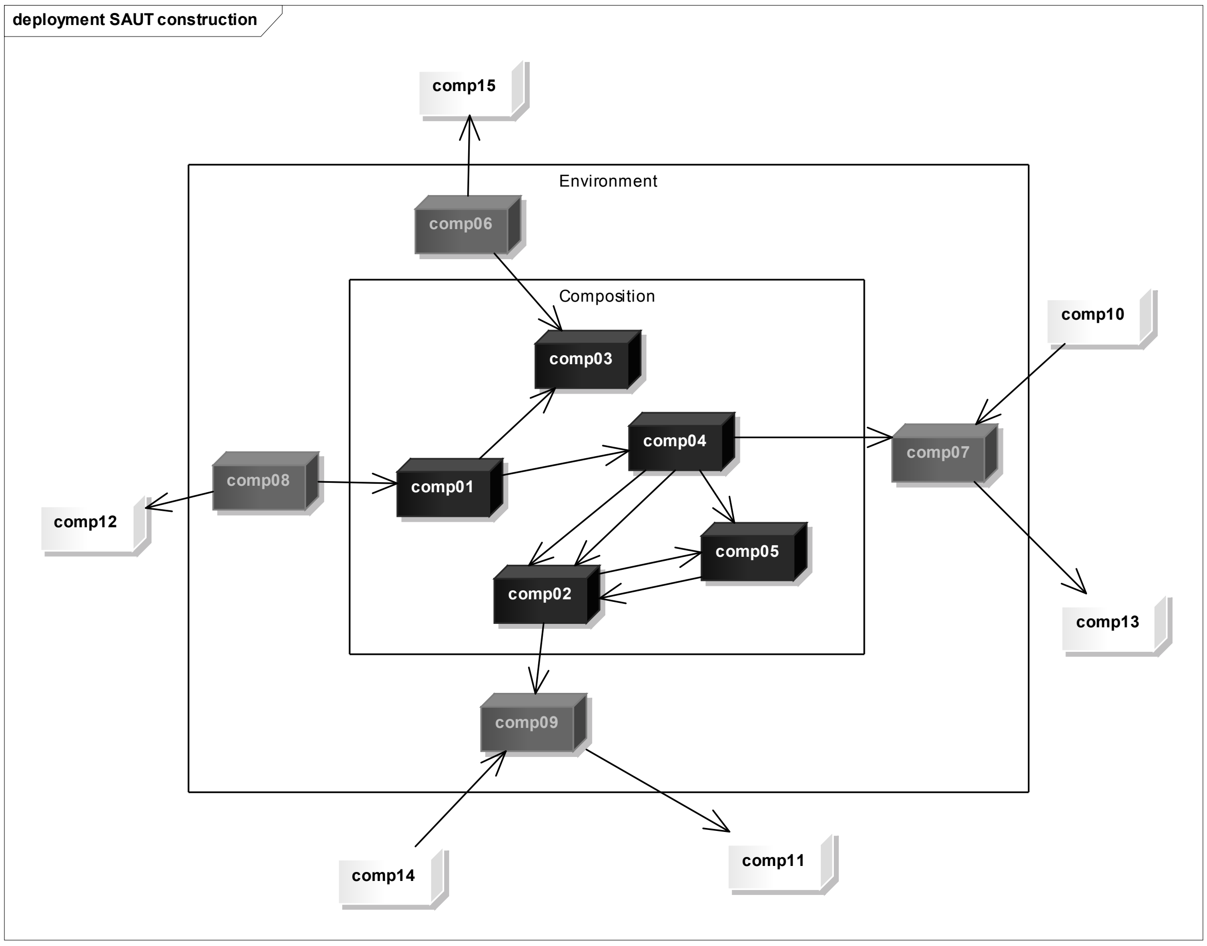


Figure 17. Generic example of SAUT construction.

FORMAL DEFINITION OF SAUT CONSTRUCTION

A formal definition of *SAUT* and *SAUT construction* is presented below. It is inspired by the Dietz's 'enterprise ontology' [Dietz 2010]:

let K be a class of *components*
let $x \in K$, let P_x the class of *references* declared by x
and let Σ_x the class of *services* exposed by x
let $P_K = \{ \cup_x^K \; P_x \}$ be the class of references of the components of K
let $\Sigma_K = \{ \cup_x^K \; \Sigma_x \}$ be the class of services of the components of K
let $\curvearrowright$ be a binary, non-reflexive and non-transitive relationship between component references and component services:
for $x,y \in K$, $r \in P_x$, $s \in \Sigma_y$, $x(r) \curvearrowright y(s)$
means 'there is a *wire* from the r reference of x (source) to the s service of y (target)'
let $\rhd$ be a binary relationship between components
for $x,y \in K$, $x \rhd y$ if and only if $\exists r,s\!: r \in P_x \;\wedge s \in \Sigma_x \;\wedge x(r) \curvearrowright \; y(s)$
$x \rhd y$ means 'there is at least one wire from x to y'

$\rhd$ is obviously non-reflexive and non-transitive
let X be a class of SAUTs
let $\prec$ be a binary relationship between a component and a SAUT
for $x \in K$ and $\sigma \in X$, $x \prec \sigma$ means 'x is part of σ'
let $\sigma \in X$
the *construction* of σ is a triple $< C(\sigma), E(\sigma), S(\sigma) >$ where
the *composition* C of σ is defined as
$C(\sigma) = \{x \in K \mid x \prec \sigma\}$
the *environment* E of σ is defined as
$E(\sigma) = \{y \in K \mid y \notin C(\sigma) \wedge \exists x{:}(x \in C(\sigma) \wedge (x \rhd y) \vee (y \rhd x))\}$
the *structure* S of σ is defined as
let $S_1(\sigma) = \{<x(r),y(s)> \in P_K \otimes \Sigma_K \mid x \in C(\sigma) \wedge (y \in C(\sigma) \vee y \in E(\sigma)) \wedge r \in P_x \wedge s \in \Sigma_y \wedge x(r) \curvearrowright y(s)\}$
let $S_2(\sigma) = \{<y(r),x(s)> \in P_K \otimes \Sigma_K \mid x \in C(\sigma) \wedge y \in E(\sigma) \wedge r \in P_y \wedge s \in \Sigma_x \wedge y(r) \curvearrowright x(s)\}$
$S(\sigma) = S_1(\sigma) \cup S_2(\sigma)$

The statements below allow a precise definition of *region*:

let $\sqsubset$ be a binary, non-reflexive and non-transitive relationship between SAUTs:
let $\sigma_1, \sigma_2 \in X$ be two SAUTs whose constructions are
$<C(\sigma_1),E(\sigma_1),S(\sigma_1)>$ and $<C(\sigma_2),E(\sigma_2),S(\sigma_2)>$ then
$\sigma_2 \sqsubset \sigma_1$, that means '$\sigma_2$ is a *region* of σ_1', if and only if
$C(\sigma_2) \subseteq C(\sigma_1)$ and
$E(\sigma_2) \subseteq (C(\sigma_1) \setminus C(\sigma_2)) \cup E(\sigma_1)$ and
$S(\sigma_2) \subseteq S(\sigma_1)$.

The formal notion of region allows establishing the definitions of *compound* component and of two operators: *aggregation* (∇) and *disaggregation* (Δ). The definition of compound component and of aggregation operator ∇ is presented below:

let $\sigma_1,\sigma_2 \in X$ and $\sigma_2 \sqsubset \sigma_1$
let $x_\alpha \in K$ such that $x_\alpha \notin C(\sigma_1)$, $x_\alpha \notin E(\sigma_1)$
let P_α the class of references declared by x_α
let Σ_α the class of services exposed by x_α
$x_\alpha \simeq \sigma_2$ ('x_α is a *compound* of σ_2') and
$\sigma = \nabla(\sigma_2,x_\alpha)\sigma_1$ ('σ is an *aggregation* of $\sigma_1$by replacing the region σ_2 of $\sigma_1$with the compound x_α')
if and only if
$C(\sigma) = (C(\sigma_1) \setminus C(\sigma_2)) \cup \{x_\alpha\}$
$E(\sigma) = E(\sigma_1)$
$S(\sigma) = (S(\sigma_1) \setminus S(\sigma_2)) \cup$
$\{<x_\alpha(r_1),y(s)> \in P_\alpha \otimes \Sigma_K \mid (y \in C(\sigma) \vee y \in E(\sigma)) \wedge s \in \Sigma_y \wedge ((x_\alpha(r_1) \curvearrowright y(s)) \Leftrightarrow$
$(\exists x\, \exists r_2\, (x \in C(\sigma_2) \wedge r_2 \in P_x \wedge <x(r_2),y(s)> \in S(\sigma_1)))\} \cup$
$\{<y(r),x_\alpha(s_1)> \in P_K \otimes \Sigma_\alpha \mid y \in C(\sigma) \vee y \in E(\sigma) \wedge r \in P_y \wedge ((y(r) \curvearrowright x_\alpha(s_1)) \Leftrightarrow$
$(\exists x\, \exists s_2\, (x \in C(\sigma_2) \wedge r_2 \in P_x \wedge <y(r),x(s_2)> \in S(\sigma_1)))\}$

The definition of the *disaggregation* operator Δ is straightforward:

let $\sigma, \sigma_1, \sigma_2 \in X$, $x_\alpha \in K$ such that

$x_\alpha \prec \sigma, \sigma_2 \sqsubset \sigma_1, x_\alpha \simeq \sigma_2$ and $\sigma = \nabla(\sigma_2, x_\alpha)\sigma_1$
then $\sigma_1 = \Delta(x_\alpha, \sigma_2)\sigma$ (‘$\sigma_1$is a *disaggregation* of σ by replacing a compound $x_\alpha$with its region σ_2 in σ’)
$\sigma_1 = \Delta(x_\alpha, \sigma_2)\nabla(\sigma_2, x_\alpha)\sigma_1$

Roughly speaking, a compound component of a SAUT is a non-atomic component whose internal architecture is a region of the SAUT. The application of the Δ (*disaggregation*) operator to a SAUT region with a compound that have the structure of the region replace the region with the compound that have the same region boundary structure.

THE SCA4SAUT NOTATION

In the MIDAS project, the SAUT Construction model is represented through a SCA Assembly Model XML document [SCA_AM_V1_2 2011]. Service Component Architecture (SCA) is a set of specifications that describe how to build applications and systems as services architectures. SCA extends and complements prior approaches to implementing services, and it is built on open standards. Thus, it is a standard specification for building and deploying services architectures whose component systems are implemented in different technologies. An important characteristic of the SCA V1.0 Assembly Model is that it is machine readable and allows the effective configuration of a service component architecture by a SCA *run time frameworks* such as Apache Tuscany[41].

The MIDAS Service Component Architecture for Services Architecture Under Test (SCA4SAUT) notation is a restriction of the SCA V1.0 Assembly Model language [De Rosa et al. 2014a]. Hence, the SCA4SAUT model is a standard SCA V1.0 Assembly Model. The main elements of the SCA4SAUT model are:

- *SAUTs,*
- *Atomic Participants*,
- *Compound Participants*.

These elements are modelled with the SCA ‘composite’ root XML element. The UML representation of the SCA4SAUT meta-model is presented in three partial views (Figure 18, Figure 19 and Figure 20)

A *SAUT* construction is modelled through a SCA composite that has a specific composition and environment arrangement:

- The SAUT composition is made of at least one (for unit test) or more (for integration test) actual components.
- The SAUT environment is made of at least one virtual component. Each virtual component must either declare one reference that must be wired with an actual component service or expose one service that must be wired with an actual component reference. In a SAUT construction, at least one service exposed by one actual component MUST be wired to a reference declared by a virtual component.

The *structure* of each SAUT component, i.e. the references that it declares and the services that it exposes, is defined by designating a *Participant* composite as a *structure specification*.

[41] http://tuscany.apache.org/

Participants can be *Atomic* or *Compound*: (i) the *Atomic Participant* composite is the basic building block of the SCA4SAUT model - it represents a combination of references and services and their bindings with their *service interface definition* in the Service model, i.e. the portType/port defined in the WSDL document; (ii) the *Compound Participant* composite represents a combination of references and services but also an *aggregation*, i.e. a collection of sub-components linked by wires that "implements" the references and services that are respectively declared and exposed by the *Compound*; the structures of these sub-components is recursively specified by *Atomic* and *Compound Participants* composites; the references and services of each sub-component can be either wired with the compatible ones of other sub-components or *promoted* as interfaces of the *Compound Participant* as a whole.

A reference and a service can be wired if and only is they are bound to the same service interface definition. Note that the wire source (reference) and target (service) must belong to different components – the wire relationship is non-reflexive, hence *loops* are not allowed in the component/wire directed graph. The *ActualToActual* wires are also called *actual* wires, and the *VirtualToActual* and *ActualToVirtual* wires are also called *virtual* wires. The SAUT a*ctual* components (composition) are used to represent entities that are aggregations of physically deployed, localized reference and service endpoints. These endpoints are the observation/stimulation points of the service testing activity, i.e. the locations where the behaviour of the deployed actual components can be stimulated and observed. Conversely, the SAUT v*irtual* components (environment) are used to model virtualized systems that carry out the stimulation and the observation of the *actual* components' behaviours at their interfaces. They shall be implemented by the test system.

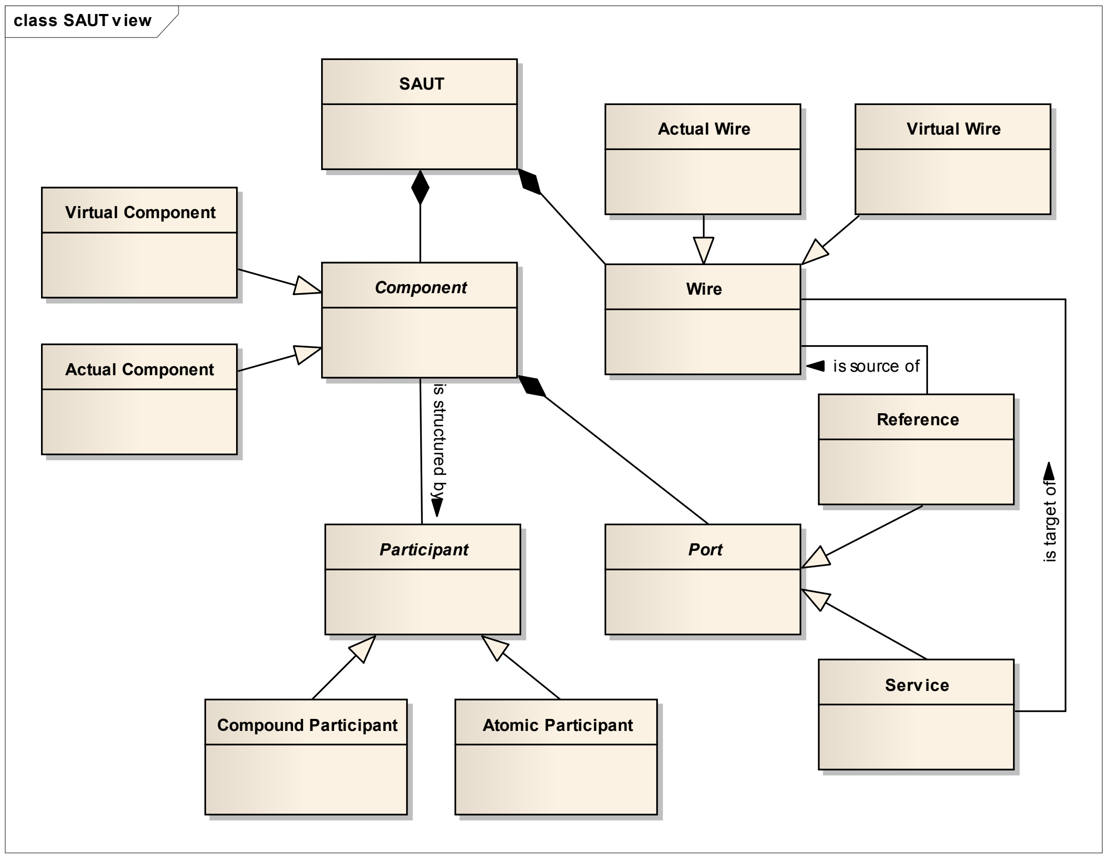


Figure 18. SCA4SAUT meta-model – SAUT view.

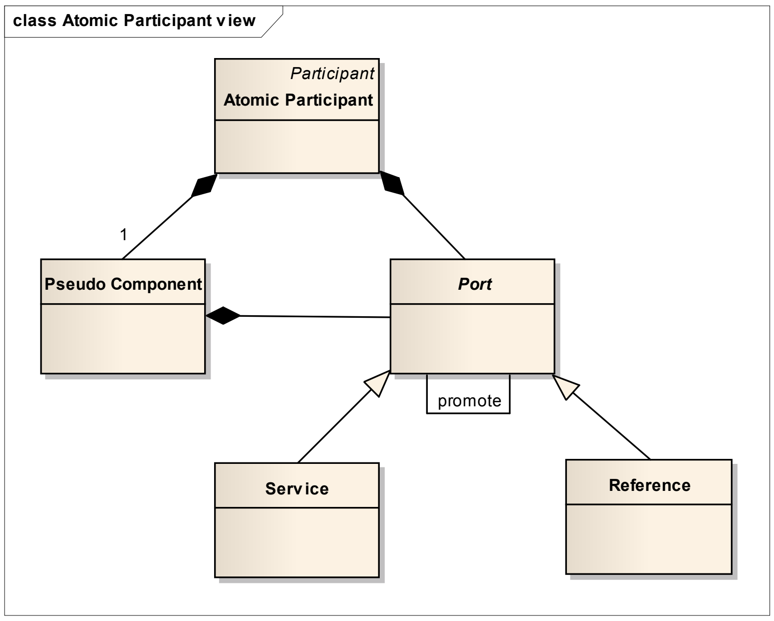


Figure 19. SCA4SAUT meta-model – Atomic Participant view.

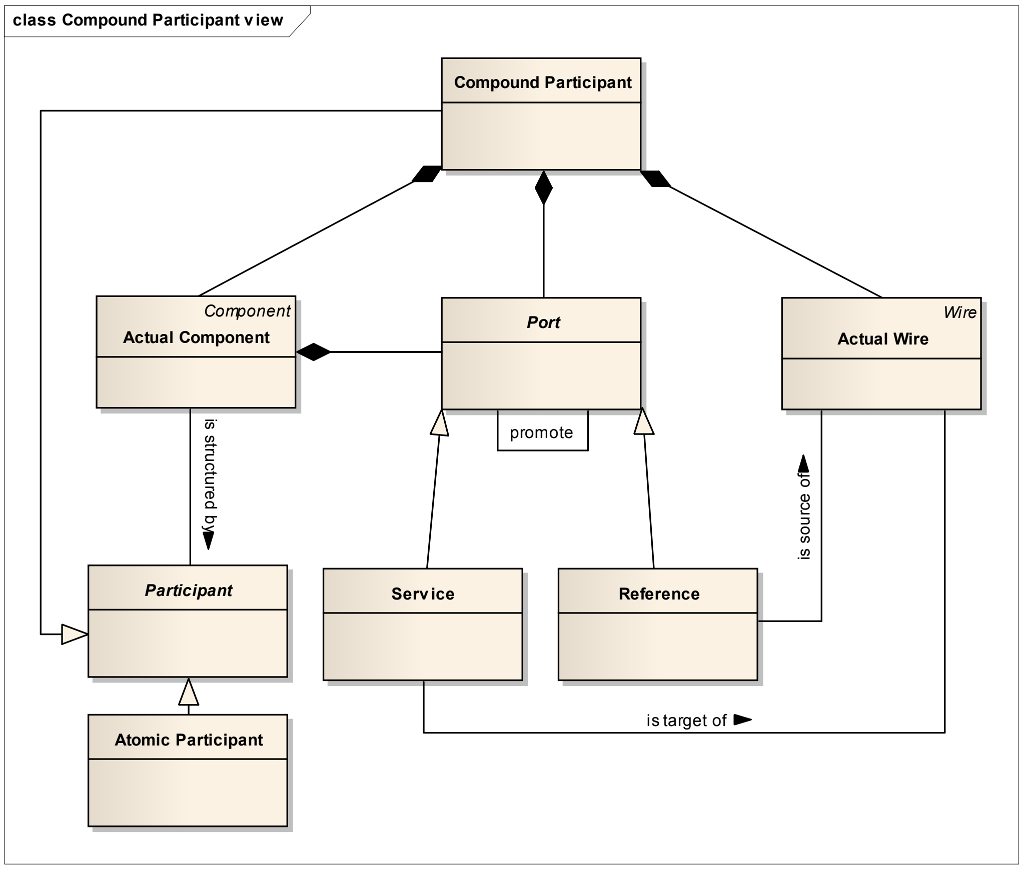


Figure 20. SCA4SAUT meta-model – Compound Participant view.

AN EXAMPLE OF SAUT CONSTRUCTION MODEL

In this paragraph an example of SAUT Construction model is illustrated that applies to the IEN example services architecture described in section 1 (§ 'An example of services architecture: the Interbank Exchange Network') and depicted Figure 2. The UML Component of the SAUT Construction model example is presented in Figure 21.

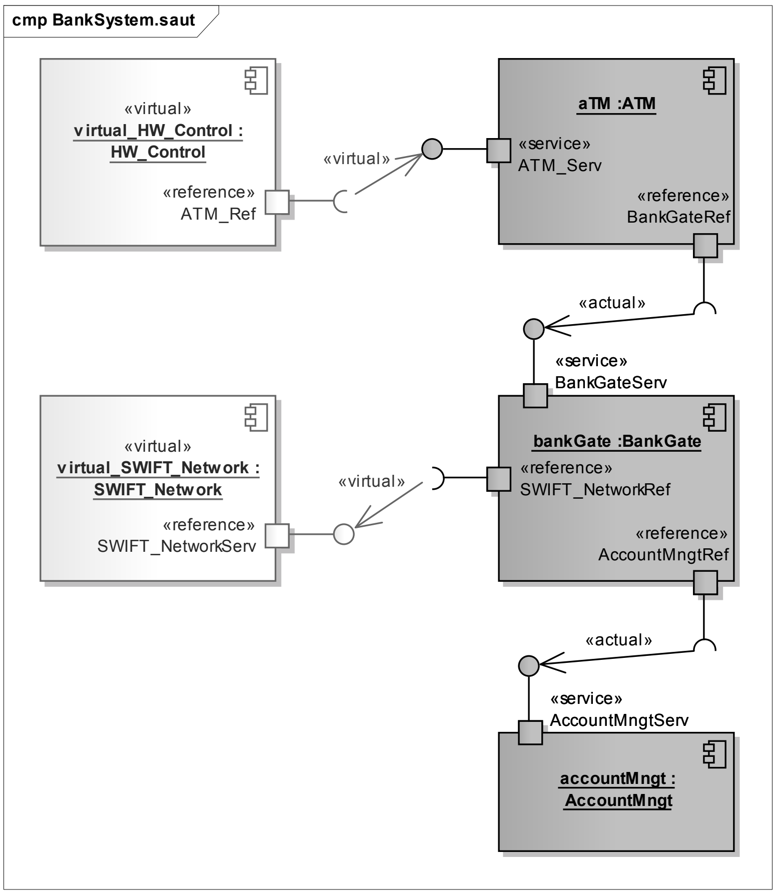

cmp BankSystem.saut
«virtual»
virtual_HW_Control : HW_Control
«reference» ATM_Ref
«virtual»
aTM :ATM
«service» ATM_Serv
«reference» BankGateRef
«actual»
«service» BankGateServ
«virtual»
virtual_SWIFT_Network : SWIFT_Network
«reference» SWIFT_NetworkServ
«virtual»
bankGate :BankGate
«reference» SWIFT_NetworkRef
«reference» AccountMngtRef
«actual»
«service» AccountMngtServ
accountMngt : AccountMngt

Figure 21. BankSystem.saut - an example of a SAUT Construction model.

The **BankSystem** SAUT Construction model includes three *actual* components (SAUT Composition):

- **aTM**: it represents an Automated Teller Machine; it is equipped with a service (**ATM_Serv**) and a reference (**BankGateRef**);
- **bankGate**: it represents a component of the bank system that acts as a front-office between the bank and its environment; it is equipped with a service (**BankGateServ**) and two references (**SWIFT_NetworkRef** and **AccountMngtRef**);
- **accountMngt**: it represents a back-office component that manages the bank accounts; it is equipped with a service (**AccountMngtServ**).

The **BankSystem** SAUT Construction model includes two *virtual* components (SAUT Environment):

- **virtual_HW_Control**: it represent a virtual agent (a bank customer) that operates the ATM; it is equipped with a reference (**ATM_Ref**);
- **virtual_SWIFT_Network**: it represents a virtual Swift network that allows transactions with other banks; it is equipped with a service (**SWIFT_NetworkServ**).

The virtual components shall be implemented by the test system as *emulators* that are utilised to stimulate the actual components to which they are wired and to collect and check their responses with the oracles.

The SAUT comprehends two *actual* wires:

- source: **aTM/BankGateRef**
  target: **bankGate/BankGateServ**;
- source: **bankGate/AccountMngtRef**
  target: **accountMngt/AccountMngtServ**.

For each *actual* wire the test system shall implement an *interceptor* that is able to receipt and to forward the messages forth and back, to arbitrate them and, if the oracles are *active*, to emulate the issuers of these messages (see § 'The logical structure of the global test verdict').

A *virtual* wire (*virtualToActual*) links a virtual component/reference to an actual component/service:

- source**: virtual_HW_Control/ATM_Ref**
  target: **aTM/ATM_Serv**.

Another *virtual* wire (*actualToVirtual*) links an actual component/reference to a virtual component/service:

- source: **bankGate/SWIFT_NetworkRef**
  target: **virtual_SWIFT_Network/SWIFT_NetworkServ**.

The SAUT Construction model sketched with the UML Component diagram in Figure 17 is described as a SCA composite in the snippet below.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<sca:composite xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:am="urn:BankNet:AccountMngt"
        xmlns:atm="urn:BankNet:ATM"
        xmlns:bg="urn:BankNet:BankGate"
        xmlns:hwc="urn:BankNet:HW_Control"
        xmlns:swift="urn:BankNet:SWIFT_Network"
        name="BankSystem.saut"
        targetNamespace="urn:BankNet:BankSystem.saut">
        <sca:property name="saut"/>
        <sca:component name="virtual_HW_Control">
         <sca:property name="virtual"/>
         <sca:property name="init"/>
         <sca:implementation.composite name="hwc:HW_Control.composite"/>
         <sca:reference name="ATM_Ref"/>
        </sca:component>
        <sca:component name="aTM">
         <sca:implementation.composite name="atm:ATM.composite"/>
         <sca:service name="ATM_Serv"/>
         <sca:reference name="BankGateReference"/>
         <sca:property name="stateViews">
          <sca:value name="ATM_StateView01"/>
         </sca:property>
        </sca:component>
        <sca:component name="bankGate">
         <sca:implementation.composite name="bg:BankGate.composite"/>
         <sca:service name="BankGateServ"/>
         <sca:reference name="SWIFT_NetworkReference"/>
```

```
          <sca:reference name="AccountMngtRef"/>
         </sca:component>
         <sca:component name="accountMngt">
          <sca:implementation.composite name="am:AccountMngt.composite"/>
          <sca:service name="AccountMngtServ"/>
          <sca:property name="stateViews">
           <sca:value name="AccountMngtStateView01"/>
          </sca:property>
         </sca:component>
         <sca:component name="virtual_SWIFT_Network">
          <sca:property name="virtual"/>
          <sca:implementation.composite name="swift:SWIFT_Network.composite"/>
          <sca:service name="SWIFT_NetworkServ"/>
         </sca:component>
         <sca:wire source="virtual_HW_Control/ATM_Ref"
          target="aTM/ATM_Serv"/>
         <sca:wire source="aTM/BankGateRef"
          target="bankGate/BankGateServ"/>
         <sca:wire source="bankGate/AccountMngtRef"
          target="accountMngt/AccountMngtServ"/>
         <sca:wire source="bankGate/SWIFT_NetworkRef"
          target="virtual_SWIFT_Network/SWIFT_NetworkServ"/>
</sca:composite>
```

Snippet 1. BankSystem.saut.composite.

The virtual component `virtual_HW_Control` is an initiator (`init` property). Its structure is specified by `HW_Control.composite`.

The actual component `aTM` is stateful. Its structure is specified by `ATM.composite`. Its state view (state management operations and resources) is specified by the view `ATM_StateView01` defined in the `ATM.composite`. The state view definition is not detailed herein because it is out of the scope of the Scheduler (it is utilised by the Executor to manage the state view of the stateful component – see § 'The logical structure of the global test verdict').

The structure of the actual stateless component `bankGate` is specified by `BankGate.composite`.

The actual component `accountMngt` is stateful. Its structure is specified by `AccountMngt.composite`. Its state view (state management operations and resources) is specified by the view `AccountMngtStateView01` defined in the `AccountMngt.composite`.

The virtual component `virtual_SWIFT_Network` is a *responder*. Its structure is specified by `SWIFT_Network.composite`.

The wires connect the component references and services in the way sketched in Figure 21.

The snippet below shows an example of definition of an *Atomic Participant.* It the *Atomic Participant* `BankGate` contained in the XML document BankGate.composite, that specifies the structure of the actual component `bankGate` shown in the Snippet 2.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<sca:composite xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdli="http://www.w3.org/ns/wsdl-instance"
  name="BankGate"
  targetNamespace="urn:BankNet:BankGate">
  <sca:component name="BankGate.component">
    <sca:service name="BankGateServices">
      <sca:interface.wsdl
        interface="urn:ien:bank:services:wsdl#wsdl.porttype(BankGatePortType)"/>
      <sca:binding.ws
        wsdlElement="urn:ien:bank:services:wsdl#wsdl.port(BankGateSOAP11HTTPPort)"
        wsdli:wsdlLocation="urn:ien:bank:services:wsdl BankGateInterface.wsdl"/>
    </sca:service>
    <sca:reference name="AccountMngt4BankGatePortTypeService">
      <sca:interface.wsdl
        interface="urn:ien:account:services:wsdl#wsdl.porttype(AccountMngt4BankGatePortType)"/>
      <sca:binding.ws
        wsdlElement="urn:ien:account:services:wsdl#wsdl.port(AccountBankSOAP11HTTPPort)"
        wsdli:wsdlLocation="urn:ien:account:services:wsdl AccountMngtInterface.wsdl"/>
    </sca:reference>
    <sca:reference name="SWIFT_Network4BankGateService">
      <sca:interface.wsdl
        interface="urn:ien:swift:services:wsdl#wsdl.porttype(SWIFT_Network4BankGatePortType)"/>
      <sca:binding.ws
        wsdlElement="urn:ien:swift:services:wsdl#wsdl.port(SwiftBankSOAP11HTTPPort)"
        wsdli:wsdlLocation="urn:ien:swift:services:wsdl SWIFTInterface.wsdl"/>
    </sca:reference>
  </sca:component>
  <sca:service name="BankGateServ" promote="BankGate.component/BankGateServices"/>
  <sca:reference name="AccountMngtRef"
    promote="BankGate.component/AccountMngt4BankGatePortTypeService"
    multiplicity="0..1"/>
  <sca:reference name="SWIFT_NetworkRef"
    promote="BankGate.component/SWIFT_Network4BankGateService"
    multiplicity="0..1"/>
</sca:composite> </sca:composite>
```

Snippet 2. BankGate.composite.

The bindings of the references AccountMngt4BankGatePortTypeService and SWIFT_Network4BankGateService and of the service BankGateServices to the appropriate WSDL portType/port(s) are specified in the (unique – it is an *Atomic Participant*!) component BankGate.component. The specified services and references are promoted to the level of the composite root element BankGate respectively through the references AccountMngtRef and SWIFT_NetworkRef and the service BankGateServ.

In the preceding Snippet 1 the references and services respectively declared and exposed by the component `bankGate`, whose structure is specified by the *Atomic Participant* BankGate.composite (Snippet 2), refer *by name* to the references and services promoted therein.

THE TEST SUITE REPRESENTATION

The Test Suite Definition / Test Suite (TSD/TS) specification [De Rosa et al. 2014c] defines a couple of XML-based descriptive notations (XSD) for the representation of respectively the Test Suite Definition model and of the Test Suite data set. The Test Suite data set is made of data structures that contain SOAP payloads (test case stimuli and active oracles), SOAP payload fragments[42] (passive oracles) and XML resources (state-before and state-after views).

THE TEST SUITE DEFINITION MODEL

A TSD document is a definition of a family of test suites. Several TSD documents can be associated to the same SAUT Construction model, and each TSD document is associated to one and only one SAUT Construction model.

A TSD XML document root element **testSuiteDefinition** includes three types of elements:

[42] The passive oracles are represented as SOAP payload fragments compliant with the W3C specification Web Services Fragment (WS-Fragment) - W3C Recommendation 13 December 2011 - http://www.w3.org/TR/2011/REC-ws-fragment-20111213/

- **interactionClass**(es),
- **componentStateView**(s),
- **testSampleClass**(es).

For each wire / operation that is involved in one of the **testSampleClass**es (see below) that are instantiated in the Test Suite, there shall be defined the corresponding **interactionClass**(es). There are three interaction class categories:

- {*operation*.input} – defined for each involved operation (request/reply and one-way) – it is issued by the reference;
- {*operation*.output} – defined for request/reply operations – it is issued by the service;
- {*operation*.fault} – optionally defined for request/reply and one-way operations – it is issued by the service.

An **interactionClass** is universally identified by name and designates: (i) the **addresser**, i.e. the component/reference or component/service that issues the message; (ii) the **addressee**, i.e. the component/service or component/reference that receipts the message; (iii) the **messageClass**, i.e. the class of the transmitted message - of one of the three categories listed above.

For instance, the interaction class that corresponds to the wire(WireOut) message in Figure 23 is documented in the TSD as in the snippet below.

```
<interactionClass name="wireOutputClass"
    addresser="aTM/ATM_Serv"
    addressee="virtual_HW_Control/ATM_Ref"
    messageClass="wire.output"/>
```

Snippet 3. The wireOutputClass *interactionClass*.

The **componentStateView**s define the state views that are involved in the test samples of the Test Suite. **componentStateView**s are classified in *Component Stateless Views*, *Component Statebefore Views* and *Component Stateafter Views*, depending on the availability of the restState, setState, getState operations (specified in the SAUT model). Table 6 summarises the characteristics of the three view categories.

	resetState	setState	getState
empty view	NO	NO	NO
stateless view	YES	NO	NO
statebefore view	YES	YES	NO
stateafter view	YES	YES	YES

Table 6. componentStateView categories and the implemented state view management operations.

Components without views (empty) do not need any state management. Components with Stateless view implements only a resetState operation to be performed by the test system at the end of the test run.

Components with Statebefore view are stateful and shall implement also a setState operation to be performed by the test system at the beginning of the test run that allow them to be in a state consistent with the run of the test sample. Components with Stateafter view implement also a getState operation that allows the test system to retrieve their current state. This information item is not more detailed because it is not utilised by the Scheduler (only by the Executor/Arbiter).

The UML representation of the TSD meta-model elements and of their associations with the SAUT Construction meta-model elements (Figure 18) is sketched in Figure 22.

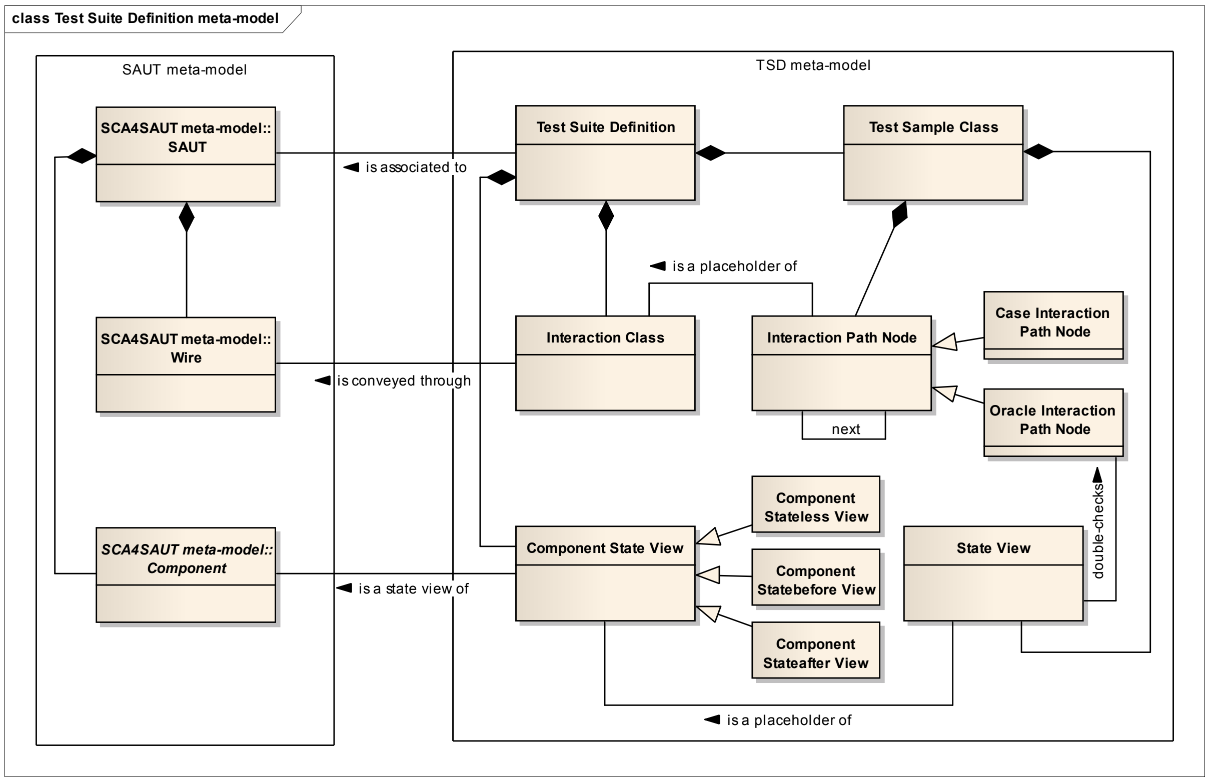


Figure 22. Test Suite Definition meta-model and its associations with the elements of the SAUT Construction meta-model.

The **testSampleClass** allows classifying the test samples. The sequence diagram of an example of **testSampleClass** (**Wire_OK_SampleClass**) is depicted in Figure 23. The **Wire_OK_SampleClass** defines a class of test samples for the **Wire_OK** scenario depicted in Figure 7, that can be executed on the BankSystem.saut, whose *SAUT Construction model* definition is in the Snippet 1 and whose UML component diagram is in Figure 21 (this *SAUT Construction model* is defined on the IEN Services Architecture whose UML component diagram is depicted in Figure 2 p. 12).

In the **Wire_OK_SampleClass** Sequence diagram in Figure 23 the interactions issued by the *actual* components are represented with thick arrows, whereas the stimuli issued by the *virtual* components are represented with thin arrows.

The **testSampleClass**es are either calculated automatically and checked by the MIDAS functional conformance test case/oracle generation - from the SAUT construction model [De Rosa et al. 2014a] and the Service component protocol state machine model [De Rosa et al. 2014b] - or they are directly defined by the tester.

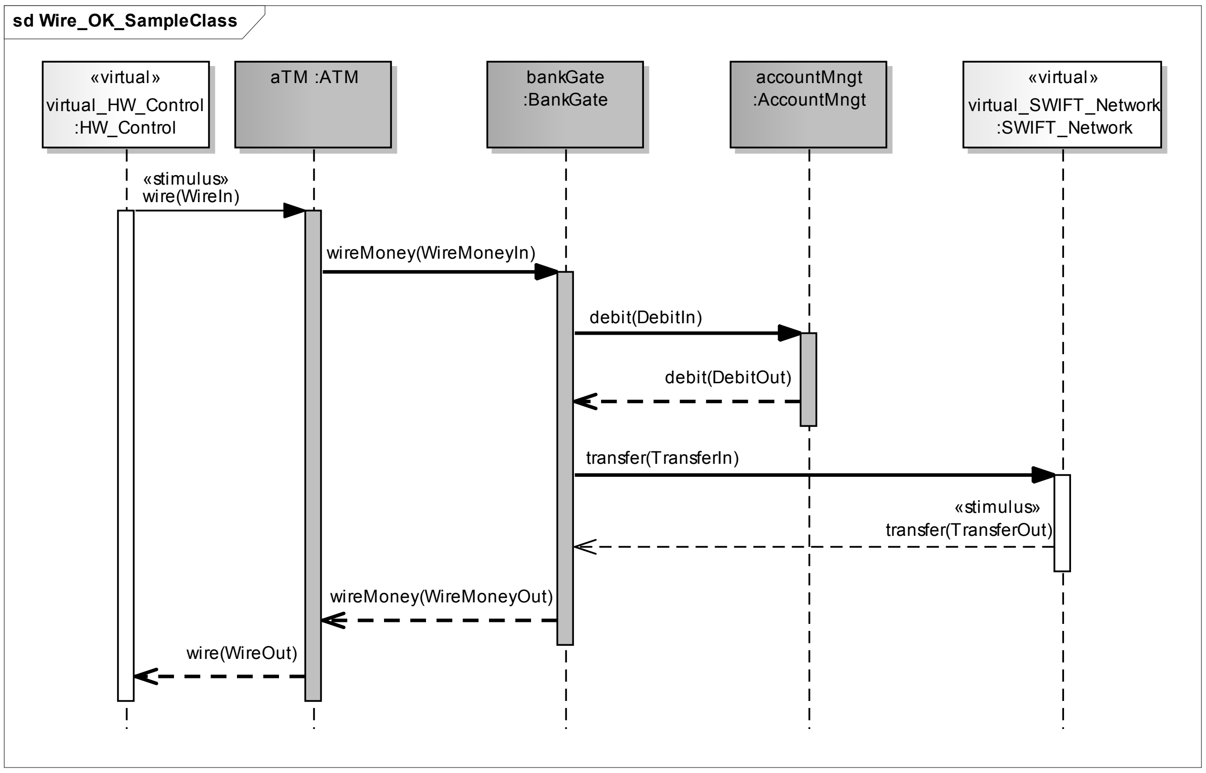


Figure 23. The Sequence diagram of the Wire_OK_SampleClass *testSampleClass*.

The snippet below gives a sketch of the *testSampleClass* **Wire_OK_SampleClass**.

```
<testSampleClass name="Wire_OK_SampleClass">
  <interactionPathNode name="wire_input"
    class="wireInputClass" use=”case”
    first="true">
    <next id="wireMoney_input"/>
  </interactionPathNode>
  <interactionPathNode name="wireMoney_input"
    class="wireMoneyInputClass" use=”oracle”>
    <next id="debit_input"/>
  </interactionPathNode>
  <interactionPathNode name="debit_input"
    …
  </interactionPathNode>
    …
  <stateView id="ATM_StateView01"/>
  <stateView id="AccountMngtStateView01"/>
</testSampleClass>
```

Snippet 4. The Wire_OK_SampleClass *testSampleClass*.

A **testSampleClass** is universally identified by name and includes: (i) a collection of **interactionPathNode**s and (ii) a collection of **stateView**s.

The **interaction Path Nodes** are classified (by means of the use attribute) in *Case Interaction Path Nodes*, the stimulus specifications, and *Oracle Interaction Path Nodes*, the response specifications.

The **interactionPathNode**s are organised in a lattice (represented by the **next** child element that includes the identifier of a next **interactionPathNode**). Each **interactionPathNode** is universally identified by name and is a placeholder of an **interactionClass**.

Each **stateView** is a placeholder of a **componentStateView** that is identified by name through the **id** attribute. The collection of **stateView**s designates the resources schemas that represent the state view of the *stateful* components that are relevant for the **testSampleClass**.

THE TEST SUITE DATA SET

A Test Suite (TS) data set is a structured collection of test samples in their native format, e.g. as XML, SOAP, SOAP fragment elements. The structure and content of a Test Suite data set is described in the associated Test Suite Definition model.

The TS data set model and its associations with the TSD meta-model elements are presented in Figure 24.

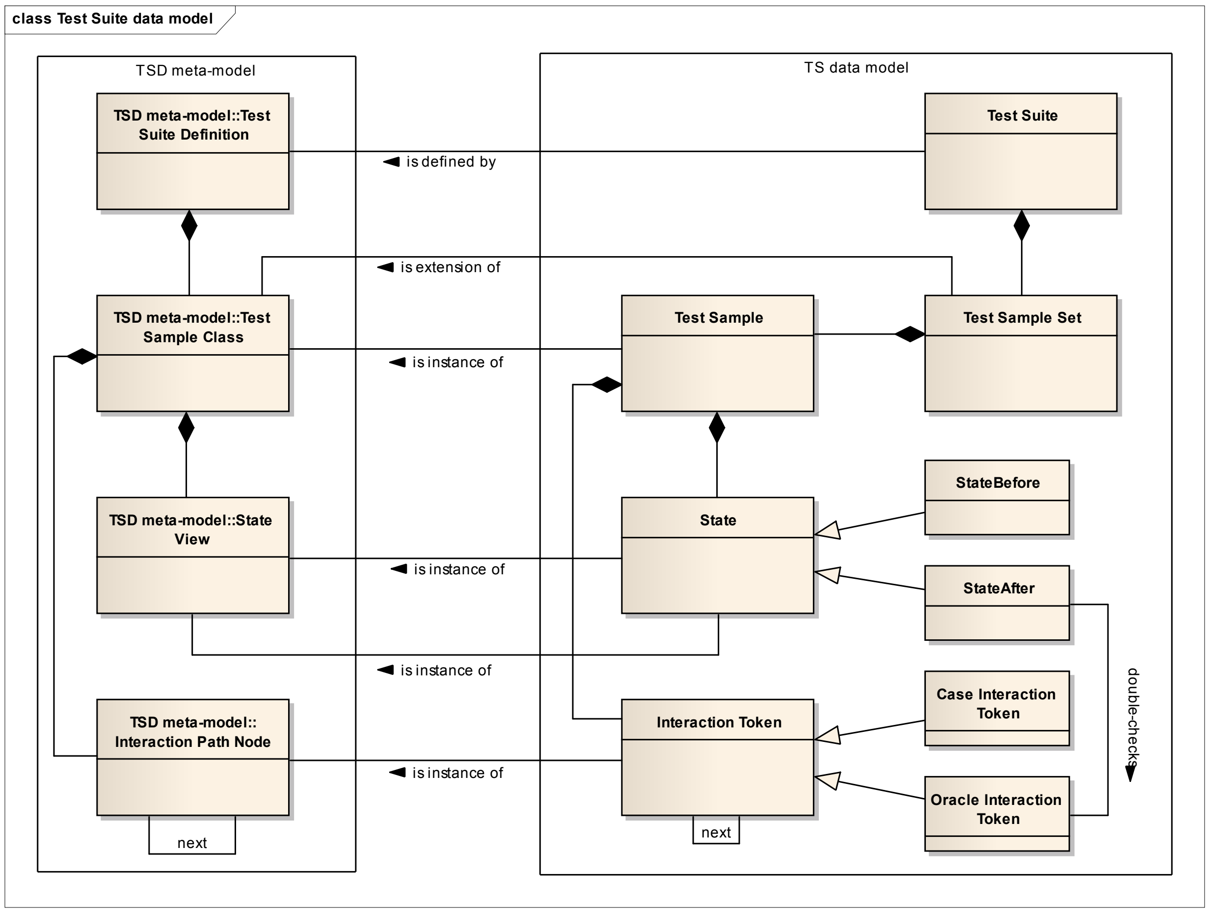


Figure 24. The Test Suite data model elements and their associations with the elements of the TSD meta-model.

A **testSuite**, the root document of a TS XML file, includes a collection of **testSampleSet**s*,* each **testSampleSet** being a collection of **testSample**s. A **testSuite** is associated with one and only one defining TSD XML document.

Each **testSampleSet** is associated with one and only one **testSampleClass** of the defining TSD and all the included **testSample**s instantiates this **testSampleClass**.

Each **testSample** element represents a collection of test case/oracle message payloads organised in the interaction path and a collection of resources representing the states of the involved stateful components.

A **testSample** element includes: (i) a collection of **interactionToken**s (at least a singleton), (ii) a collection of **stateBefore**s (possibly empty) and (iii) a collection of **stateAfter**s (possibly empty).

In each **testSample** there is an **interactionToken** for each defining TSD **testSampleClass/interactionPathNode**. The **interactionToken** instantiates the **TSD/interactionPathNode** and contains a **payload** that instantiates the appropriate TSD **interactionClass/messageClass**. The **payload** contains the instantiated message or message fragment. The **testSample** includes also **stateBefore** and **stateAfter** child elements, and each **stateAfter** element refers to the **interactionToken** that it *double-checks*. Their content is not detailed because it is not utilised by the Scheduler (only by the Executor/Arbiter), that uses only their identifiers.

For instance, a **testSample** that instantiates the *testSampleClass* **Wire_OK_SampleClass** whose sequence diagram is in Figure 23 includes:

- two *Case* **interactionToken**s, instances of: **wire(WireIn)**, **transfer(TransferOut)**;
- six *Oracle* **interactionToken**s, instances of: **wireMoney(WireMoneyIn)**, **debit(DebitIn)**, **debit(DebitOut)**, **transfer(TransferIn)**, **wireMoney(WireMoneyOut)**, **wire(WireOut)**;
- two *Before* **state**s, instances of: **aTM_StateBefore**, **accountMngtStateBefore**,
- one *After* **state**, instance of **accountMngtStateAfter** that *double-checks* **debit(DebitOut)**, implementing the post-condition enforcement.

THE TEST SYSTEM, THE TEST RUN AND THE TEST VERDICT

THE TEST EXECUTION/ARBITRATION SYSTEM CONFIGURATION

The SAUT Construction model is utilised not only by the Scheduler to configure the internal probabilistic inference engine, but also by the Executor/Arbiter to configure the Test Execution/Arbitration System – called also the ‘test system’ in the remainder of the manuscript. A glance of the test system configuration is necessary to understand the logical structure of the test verdict, which is the input of the Scheduler in the schedule/execute/arbitrate cycle.

The SAUT Construction model sketched in Figure 21 allows the configuration of the test system illustrated in Figure 25. Test system configuration for BankSystem.saut.. The SAUT components are in grey color, whereas the test system components are white.

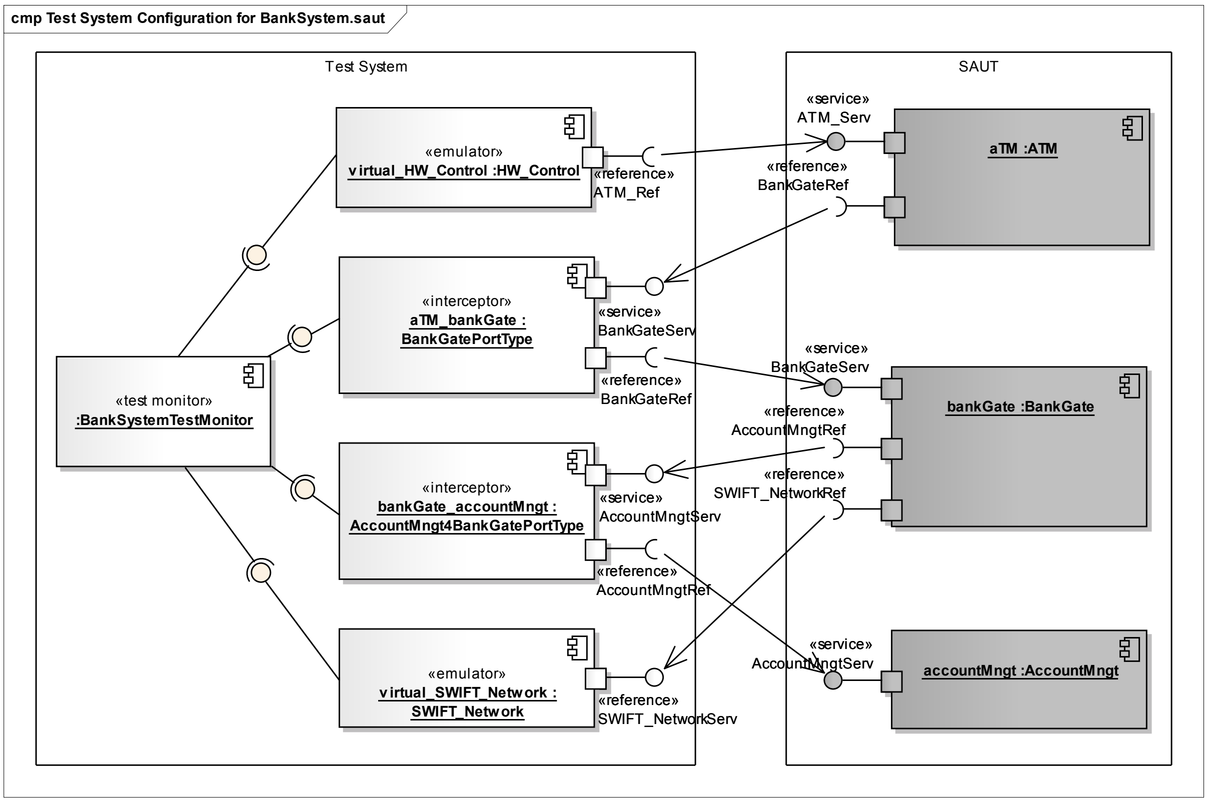

cmp Test System Configuration for BankSystem.saut
Test System
SAUT
«emulator» virtual_HW_Control :HW_Control
«reference» ATM_Ref
«interceptor» aTM_bankGate : BankGatePortType
«service» BankGateServ
«reference» BankGateRef
«test monitor» :BankSystemTestMonitor
«interceptor» bankGate_accountMngt : AccountMngt4BankGatePortType
«service» AccountMngtServ
«reference» AccountMngtRef
«emulator» virtual_SWIFT_Network : SWIFT_Network
«reference» SWIFT_NetworkServ
«service» ATM_Serv
«reference» BankGateRef
aTM :ATM
«service» BankGateServ
«reference» AccountMngtRef
«reference» SWIFT_NetworkRef
bankGate :BankGate
«service» AccountMngtServ
accountMngt :AccountMngt

Figure 25. Test system configuration for BankSystem.saut.

The Executor/Arbiter initialisation process creates four *test components* and a *test monitor* that coordinates them. It creates two *emulator* test components, one for each *virtual* component:

- **virtual_HW_Control** *requester emulator* test component, representing the eponym SAUT virtual component – when initialised by the *test monitor*, it issues a *stimulus* request to **aTM/ATM_Serv** service, waits for the response (test outcome), receipts the response, arbitrates the test outcome and returns the verdict (*pass* or *fail*) to the test monitor (the global test run stops) – the generic behaviour of a *requester emulator* is depicted in the UML Activity diagram in Figure 26;
- **virtual_SWIFT_Network** *responder emulator* test component, representing the eponym SAUT virtual component - when initialised by the test monitor, it waits for a request message (test outcome) from **bankGate/SWIFT_NetworkRef**, receipts the message and arbitrates it; if the verdict equals *pass* then the emulator issues a *stimulus* response to **bankGate/SWIFT_NetworkRef** and returns the verdict to the test monitor; otherwise (the local verdict equals *fail*), it returns the verdict to the test monitor (the global test run stops) – the generic behaviour of a *responder emulator* is depicted in the UML Activity diagram in Figure 26;

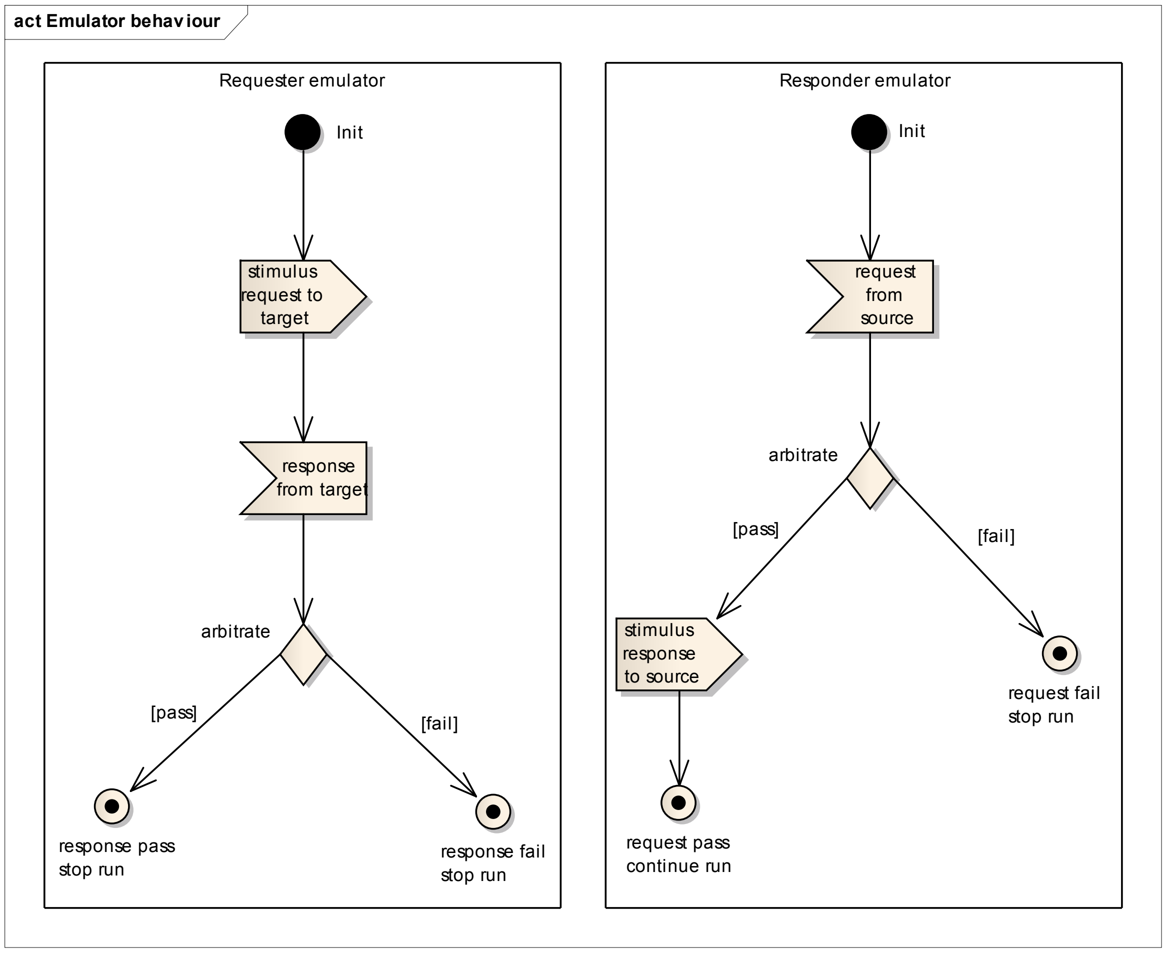


Figure 26. Emulators generic behaviour.

The Executor/Arbiter initialisation process creates two *interceptor* test components, one for each *actual* wire:

- **aTM_bankGate** *interceptor* test component – when initialised by the *test monitor*, it is able to intercept the interactions forth and back between the **aTM/BankGateRef** reference and the **bankGate/BankGateServ** service related to the operations of the **BankGatePortType** portType – the generic behaviour of an interceptor is depicted in the UML Activity diagram in Figure 27;
- **bankGate_accountMngt** *interceptor* test component – it is able to intercept the interactions between the **bankGate/AccountMngtRef** reference and the **accountMngt/AccountMngtServ** service related to the operations of the **AccountMngt4BankGatePortType** portType.

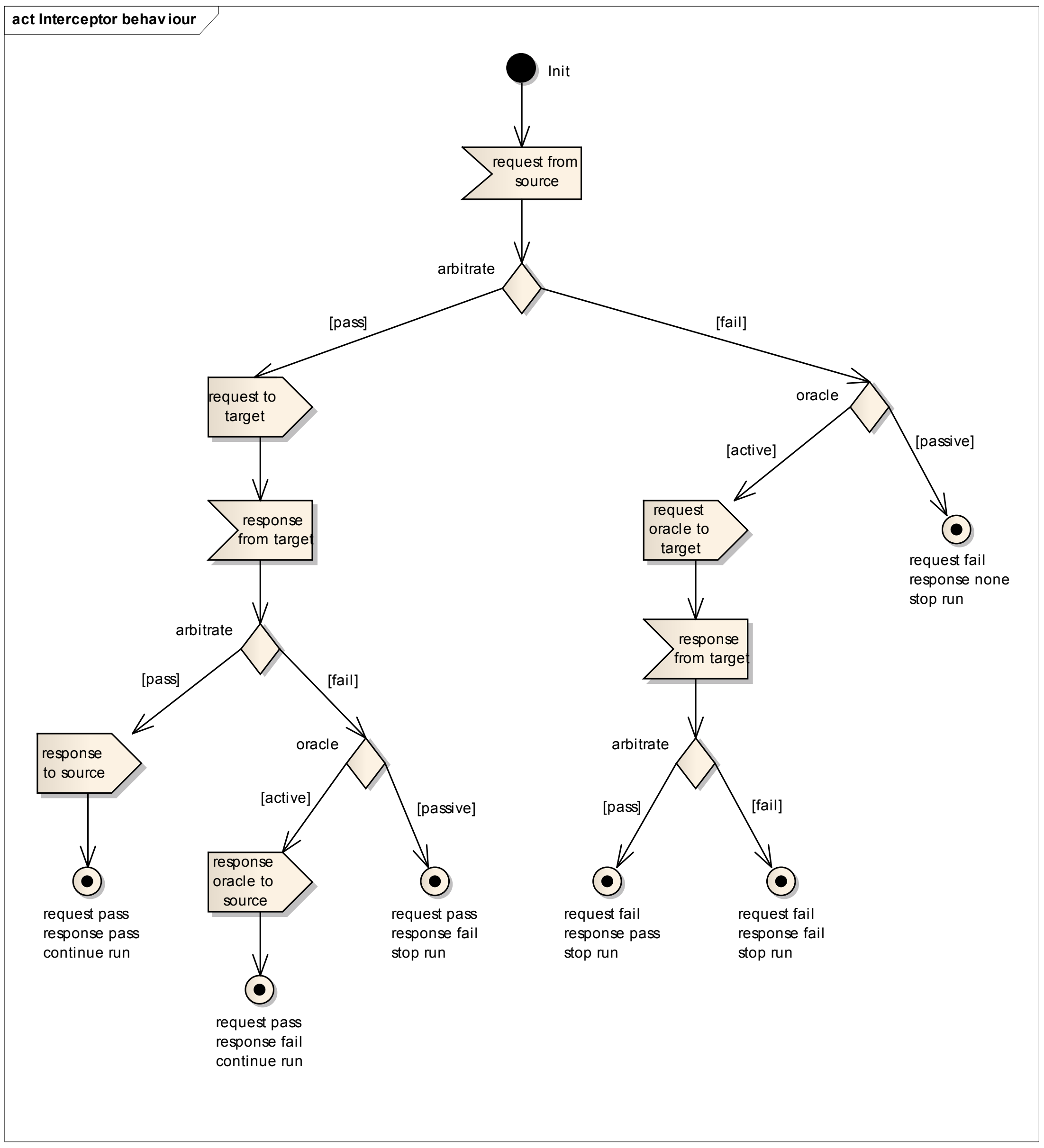


Figure 27. Interceptor generic behaviour.

The **aTM_bankGate** *interceptor* test component behaviour is detailed in the Table 7. The **bankGate_accountMngt** *interceptor* behaviour is similar.

When initialised by the *test monitor*: 1. wait for a request message from **aTM/BankGateRef** 2. receipt the request message 3. arbitrate it	[request verdict = *pass*] 1. send the request message to **bankGate/BankGate Serv** 2. wait for the response message 3. receipt the response message 4. arbitrate it	[response verdict = *pass*] 1. send the response message to the **aTM/BankGateRef** return the two verdicts - the test run continues	
		[response verdict = *fail*]	[active oracle] 1. sends the response oracle message to **aTM/BankGateRef** return the two verdicts - the test run continues
			[passive oracle] returns the two verdicts - the test run stops
	[request verdict = *fail*]	[active oracle] • send the request oracle message to **bankGate/BankGate Serv** • wait for the response message • receipt the response message • arbitrate it	[response verdict = *pass*] return the two verdicts - the test run stops
			[response verdict = *fail*] return the two verdicts - the test run stops
		[passive oracle] return the request verdict (response verdict = *none*) - he test run stops	

Table 7. Behaviour of the aTM_bankGate *interceptor.*

In order to simplify the description, the treatement of the *timeouts* is not detailed. The *test monitor* collects the *local test verdicts* from the test components and the Executor/Arbiter sends the *global test verdict*, i.e. the collection of *local test verdicts*, via the Runner, to the Scheduler. The Scheduler utilizes the *pass/fail* local verdicts as *observations* to instantiate the corresponding variables - the *none* local verdicts (the test has not been performed) do not instantiate the corresponding variables.

THE TEST RUN - AN EXAMPLE

The sequence diagram in Figure 28 depicts the test run of a test sample that instantiates the *testSampleClass* **Wire_OK_SampleClass** (Figure 23) in which there are no local failures, i.e. all the SAUT responses match the test oracles and the retrieved state matches the state after.

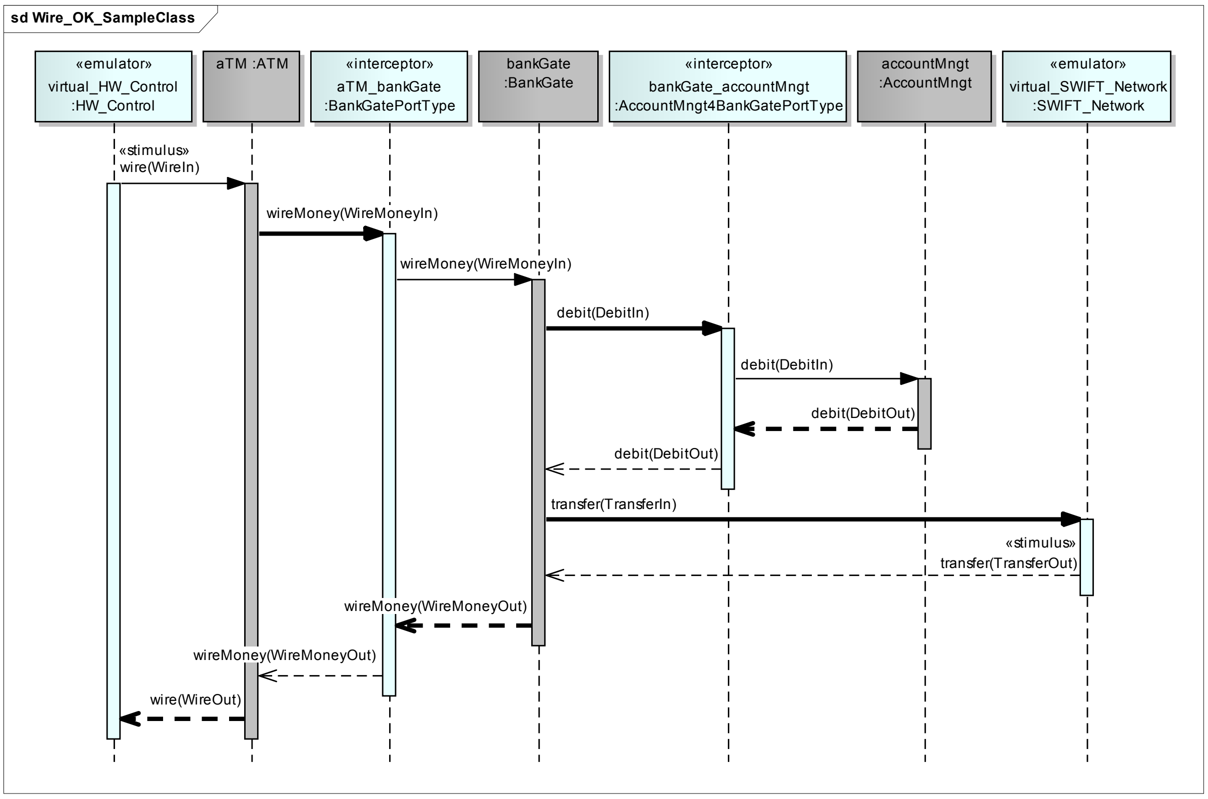


Figure 28. Sequence diagram of the execution of a test sample that instantiates the Wire_OK _SampleClass with no local failures.

The interlocutors of the sequence diagram are the SAUT components and Test System components sketched in Figure 25. The interactions issued by the SAUT components (*outcomes*) are represented with thick arrows, whereas the interactions issued by the Test System components (*stimuli* and *forwards*) are represented with thin arrows.

The test run of a **Wire_OK_SampleClass** instance (test sample) that conforms to the Sequence diagram in Figure 28, is described below:

1. the test monitor begins the test run and sets **aTM** with **ATM_StateBefore** *State Before* and **AccountMngt** with **AccountMngtStateBefore** *State Before* through their respective setState operations;
2. the **virtual_HW_Control** *emulator* issues the **wire(WireIn)** *stimulus* towards the **aTM** SAUT component;
3. **aTM** receipts the stimulus and issues the **wireMoney(WireMoneyIn)** message towards the **bankGate** SAUT component that is intercepted by the **aTM_bankGate** *interceptor*;

4. the **aTM_bankGate** *interceptor* arbitrates the intercepted message, sets the local verdict to *pass*, communicates it to the *test monitor* and forwards the message to **bankGate**;
5. **bankGate** receipts the message and issues the **debit(DebitIn)** message towards the **accountMngt** SAUT component that is intercepted by the **bankGate_accountMngt** *interceptor*;
6. the **bankGate_accountMngt** *interceptor* arbitrates the message, sets the local verdict to *pass*, communicates it to the *test monitor* and forwards the intercepted message to **accountMngt**;
7. **accountMngt** receipts the message and issues the **debit(DebitOut)** message towards **bankGate** that is intercepted by the **bankGate_accountMngt** *interceptor*;
8. the **bankGate_accountMngt** *interceptor* arbitrates the message (*pass*), gets the **accountMngt** state with the appropriate **accountMngt** getState, arbitrates the retrieved state with **accountMngtStateAfter** *State After* (*pass*), sets the local verdict to *pass*, communicates it to the *test monitor* and forwards the intercepted message to **bankGate**;
9. **bankGate** receipts the message and issues the **transfer(TransferIn)** message towards the **virtual_SWIFT_Network** *emulator*;
10. the **virtual_SWIFT_Network** *emulator* receipts the message, arbitrates it, sets the local verdict to *pass*, communicates it to the *test monitor* and issues a **transfer(TransferOut)** stimulus towards **bankGate**;
11. **bankGate** receipts the stimulus and issues the **wireMoney(WireMoneyOut)** message towards **aTM** that is intercepted by the **aTM_bankGate** *interceptor*;
12. the **aTM_bankGate** *interceptor* arbitrates the message, sets the local verdict to *pass*, communicates it to the *test monitor* and forwards the intercepted message to **aTM**;
13. **aTM** receipts the message and issues a **wire(WireOut)** message towards the **virtual_HW_Control** *emulator*;
14. the **virtual_HW_Control** *emulator* receipts the message, arbitrates it, sets the local verdict to *pass* and communicates it to the *test monitor*;
15. the test monitor resets the **aTM** and **accountMngt** states through their respective resetState operations, communicates the global verdict to the main Executor/Arbiter and ends the test run.

THE LOGICAL STRUCTURE OF THE GLOBAL TEST VERDICT

In the schedule/execute/arbitrate cycle, the Executor/Arbiter job is to perform test runs (see the example of the preceding paragraph) and to produce test verdicts. The test verdicts have a simple structure and an intuitive meaning, but also some subtle properties that are highlighted in this paragraph.

The *global verdict* following a *Test Sample* execution is a collection of *local verdicts*, one for each *Oracle Interaction Token*.

The *local verdict* value may be:

- *pass* – there is a test outcome (collected message) and it matches the oracle;
- *fail* – there is a test outcome (collected message) and it mismatches the oracle;

- *none* – there is no test outcome (no collected message - the test run has been stopped before)[43].

The *Oracle Interaction Token* specifies:

- *single-check oracle*: it is the content of the *Oracle Interaction Token* - the test outcome is arbitrated by matching the message with the *Oracle Interaction Token* content; if the match succeeds, then the *local verdict* is set to *pass*, otherwise it is set to *fail*;
- *double-check oracle*: it is the couple formed by the *Oracle Interaction Token* content and the content of the *State After* that is linked to it (by the *double-checks* association) - the test outcome is arbitrated (i) by matching the message with the *Oracle Interaction Token* content and (ii) by retrieving the state of the component that issued the message through the appropriate getState operation and by matching the retrieved state with the *State After* content; if both matches succeed the *local verdict* is set to *pass*, otherwise it is set to *fail*.

The *double-check oracle* allows to check not only the message, but also the service operation post-condition. Note that there are four possible results of the arbitration with a double-check oracle:

- both the collected message and the retrieved state match the respective oracles – the service operation was correctly executed and the message reported correctly its execution; the message local verdict is set to *pass*;
- the message matches its oracle but the state mismatches its oracle – the message reports the incorrect execution of the operation as it were correct; the message local verdict is set to *fail*;
- the message mismatches its oracle but the state matches its oracle – the message reports incorrectly the correct execution of the operation; the message local verdict is set to *fail*;
- both the message and the state mismatch the respective oracles – the service operation is implemented incorrectly (the message report can be "consistent" or not with the service operation implementation); the message local verdict is set to *fail*.

The relationships of the single-check and double-check local verdicts with the message and state arbitrations are summarised in Table 8.

message matching	**state matching**	**single-check verdict**	**double-check verdict**
match	*match*	*pass*	*pass*
match	***mismatch***	***pass*** **(false negative)**	***fail***
mismatch	*match*	*fail*	*fail*
mismatch	*mismatch*	*fail*	*fail*

Table 8. Single-check and double-check verdicts.

[43] The Scheduler does not yet treat the *inconclusive* and *error* values of the test verdict. Intuitively, these values should be processed as the *none* value, because they do not add information from the strict point of view of testing and troubleshooting.

If, for a *transformative* service operation, i.e. an operation that updates the component state (for instance, the operation invoked through the **debit(DebitOut)** message), the test generation process (whether manual or automated) does not produce an **accountMngtStateAfter** *State After* specification that is able to *double-check* the **debit(DebitOut)** *Interaction Token* specification, the Executor/Arbiter is obliged to adopt the *single-check oracle* arbitration process that does not prevent *false negatives*. The single-check oracle for state-change operation can be characterised as *partial oracle* that augments the uncertainty of the arbitration process.

As previously introduced, the *message* oracles can be classified as *active* oracles and *passive* oracle. A *passive* oracle can only be used to *check* the SAUT behaviour, while an active oracle is also able to *reproduce* the SAUT behaviour. An active *Oracle Interaction Token* content is a message structure (e.g. a SOAP message) that, after TTCN-3 compilation, can be utilised by the Executor as a TTCN-3 template for total matching of the incoming SOAP message from the SAUT, but also as a TTCN-3 record type for filling TTCN-3 record that are encoded in SOAP message and sent to the SAUT. The *message* active oracle is utilised by an *interceptor* to *emulate* the correct behaviour of the component that issues a mismatch message.

The usage of oracles for emulation must be managed carefully. For instance, if we suppose, in the example of the preceding paragraph at the step 4, that the **aTM_bankGate** *interceptor* arbitrates the received **wireMoney(WireMoneyIn)** message to *fail* (local verdict), and it sends the oracle message in place of the intercepted message (the oracle is active), and the test run can continue until the reception by the **aTM_bankGate** *interceptor* of the **wireMoney(WireMoneyOut)** message from **bankGate**, it arbitrates the intercepted message, sets the local verdict, communicates it to the test monitor but the test run cannot continue, because responding to **aTM** that sent a **wireMoney(WireMoneyIn)** message non conformant to the test sample with a **wireMoney(WireMoneyOut)** message conformant to the test sample is non sense.

A passive *Oracle Interaction Token* content is an incomplete specification that is represented by a message fragment structure (e.g. a SOAP message fragment[44]) that, after TTCN-3 compilation, can be utilised by the Executor as a TTCN-3 template for partial matching of the incoming SOAP message from the SAUT (after TTCN-3 encoding), **but not** as a TTCN-3 record type to generate TTCN-3 records that, after being decoded to SOAP messages, are sent to the SAUT.

Oracles, including also state views, may be specified incompletely for a number of reasons (e.g. insufficient knowledge of the tester). Incomplete (passive) oracles and single-check oracles of state-change components maintain the uncertainty of testing and, primarly, of troubleshooting. In this situation, the failure of a message may be loosely coupled with the fault index of the component/port (reference or service) that issues it (*fault propagation*). For example, if in the preceding example we suppose that the oracles are passive and in step 12 the **aTM_bankGate** *interceptor* arbitrates the **wireMoney (WireMoneyOut)** message to *fail*, it is not certain that this is not a *false positive*. In fact, in the preceding step 7, **accountMngt** issued a **debit (DebitOut)** message towards **bankGate** that was arbitrated to *pass* and forwarded by the **bankGate_accountMngt** *interceptor*. But some parts of the message, unchecked by the oracle, contained information that provoked the issuance by **bankGate** of the *failed* **wireMoney(WireMoneyOut)** message. In fact, the behaviour of **bankGate** was correct with respect to the receipted **debit(DebitOut)** message. So, a *false negative* **debit(DebitOut)** provoked *a false positive* **wireMoney(WireMoneyOut)**. The test system tends to localise the fault in **bankGate/BankGateServ**

[44] http://www.w3.org/TR/ws-fragment/

that performs correctly, while it is actually localised in **accountMngt/AccountMngtServ** and the test report, if it does not take into account the uncertainty of the fault localisation, deceives the **accountMngt** and **bankGate** debugging/fixing teams, which, in addition, may be independent.

These kinds of problems highlight the difficulty of service component architecture testing, and the fundamental uncertainty of the task. It is clear that the results of the execution/arbitration tasks must be treated with the support of a probabilistic reasoner. This work does not yet take into account this aspect of the passive oracles, but the author think that the proposed probabilistic model is the basis for taking into account this aspect and other aspects in future research.

5. RELATED WORK ABOUT THE PROBABILISTIC APPROACH

The artificial intelligence research is interested in representing the uncertainty within the decision mechanisms. At this effect, the use of the probability theory is considered to be one of the most promising frameworks [Pearl 1988]. Probability theory, through its various Bayesianist approaches, allows modelling conditional independences and stochastic relation between events and inferring on the state of a subgroup of those events according to observations.

In this work, the focus is on the Bayesian Networks (BN). But, to gain a good comprehension of the field and its applications, it is important to review the related work on Markov Networks (MN), a counterpart of the BNs, because they play an important role in probabilistic inference.

Taking into account uncertainty through probability and assumptions, beliefs and observations permit the decision-making despite incomplete knowledge: confronted to a complex reality that is impossible to fully represent and for which the exact logical inference is untreatable, the stochastic approach enables the decision process. Furthermore, the probabilistic framework provides a knowledgeable representation of the relations and of the influences between events.

Moreover, this representation is understandable even for people outside of the field of expertise. One can resume the success of BN in the industrial world to three facts: (i) simplicity and concision of the representation, (ii) understandability, with low requirements in probability theory knowledge and (iii) knowledgeability of the representation and reasoning mechanisms. This is why there are many examples of the use of probabilistic line of action in many business and technical activities (see § 'Probabilistic approach to testing and troubleshooting').

The hypothesis of this research is that probabilistic reasoning can enhance dynamic scheduling of SOA testing (grey-box testing of distributed services architectures) by improving the *fault detection rate*, i.e. the precocious expositions of failures and the localisation of faulty components. The intent of this section is to provide the reader with a state of the art on the probability theory models (BNs and MNs), their implementation, their exploitation for probability inference and the use of those models in the domain of testing (seeking for failures) and troubleshooting (fault localisation).

The following notation is used in this section:

- X is a random variable,
- $\boldsymbol{X} = \{X_1, \dots, X_n\}$ is a set of random variables,
- x is an instantiation (a value) of X,
- $\boldsymbol{x}$ is a set of instantiations for the set of variables $\boldsymbol{X}$,
- The notation $\pi(X)$ is used to represent the *parent* nodes of X in a directed acyclic graph.
- Let X be a variable and $\boldsymbol{U}$ be the set of *parents* of X, $\theta(x|\boldsymbol{u})$ is the probability of the state of the value x given the instantiation $\boldsymbol{u}$ of the parent nodes $\boldsymbol{U}$.
- Let $\boldsymbol{e}$ be a set of instantiations of one or more variables, $\boldsymbol{e} - X$ is the subset of $\boldsymbol{e}$ without any instantiation of X.

PROBABILISTIC MODELS

One of the uses of probability theory is to model with maximum accuracy and minimum number of parameters a complex reality. The intent of the models presented below is to map conditional independence so that it is possible to factorize joint probability distributions. The result is the creation of an easy to use framework that permits computation of marginal probability distributions and probabilistic inference.

INDEPENDENCE NOTION

Beginning with an example allows facilitating the comprehension of the concept of independence among events. Let's first consider an experiment where a fair coin C_1 is flipped twice in sequence X, Y. The coin used is fair and has two sides *head* and *tail* $V(X) = V(Y) = \{H, T\}$. When flipped the coin can return with equal probability any of its two sides: $X \rightarrow V(X)$. After the first coin flip, it is evident that it is impossible to know what would be the result of the second flip before it happens on the basis of the result of the first one. The singular events X and Y are independent from one another and are mathematically denoted $X \coprod Y$. Now, let's consider the possibility of the use of a gaffed coin C_2 with only one output H. Let's redefine the sequential flipping of X and Y as such: depending on the result of X the second flip Y can use C_1 or C_2 according to the rule 'If $X = H$ then C_1 else C_2'. It is evident that in this case the events X and Y are not independent. In fact you can now infer that if the first flip returns T then the second will obligatory return H and also if the second flip returned a T then the first had to return H.

Marginal independence is an intuitive notion corresponding to the fact that not all events are interconnected. There is a second kind of independence that has proven to be more useful in the field of uncertain reasoning when representing related events: **the conditional independence**.

To have a better understanding of the conditional independence it is useful to present another example. Considering two machines represented by two random variables M_1, M_2 ($Val(M_1) = Val(M_2) = \{OK, NOK\}$) connected to the same source of energy represented by the random variable P ($Val(P) = \{on, off\}$). It is possible to observe that M_1 is NOK, which can be the result of two cases: i) the P is off or ii) M_1 is NOK because of some random failure. It is possible to check if the fault relies in P being off by looking at M_2. If M_2 is NOK there is a strong probability that P is off. It is possible to say in this case that M_1 and M_2 are correlated. However if P is observed (wheter its state is on or off) then looking at M_2 states is irrelevant for troubleshooting M_1. This means that if P is observed then M_1 and M_2 becomes independent. Henceforth M_1 and M_2 are conditionally independent knowing P.

Conditional independence is very useful to model correlation between random variables. This key concept is at the core of the BN and allows merging the graphical representation part with the probabilistic part of the model.

A more formal definition of the conditional independence states that two sets of random variables $\boldsymbol{X}$ and $\boldsymbol{Y}$ are conditionally independent given a third set of random variables $\boldsymbol{Z}$, if and only if the conditional probability distribution shows their independence given all value $\boldsymbol{z} \in Val(\boldsymbol{Z})$: $P \vDash (\boldsymbol{X} \coprod \boldsymbol{Y} \mid \boldsymbol{Z} = \boldsymbol{z})$. In the case that $\boldsymbol{Z}$ is empty it is possible to speak of marginal dependence between $\boldsymbol{X}$ and $\boldsymbol{Y}$.

A result of the conditional independence is that $P(X, Y|Z) = P(X|Z) * P(Y|Z)$. Conditional independences satisfy the graphoid axioms of: i) *Symmetry*, ii) *Decomposition*, iii) *Weak Union* and iv) *Intersection* (only in the case P >0) [Pearl 1988]. The definition of conditional independences allows factorising probability distributions and the factorisation permits the model analysis and the development of algorithms for probabilistic inference.

According to the used probabilistic model there are different graphical representations of conditional independences. The main representations are: (i) the *undirected graph* and (ii) the *directed graph* representations. Figure 29 sketches these representations applied to a classical example of probabilistic network well known as Asia model (whose BN graph representation is fully developed Figure 33).

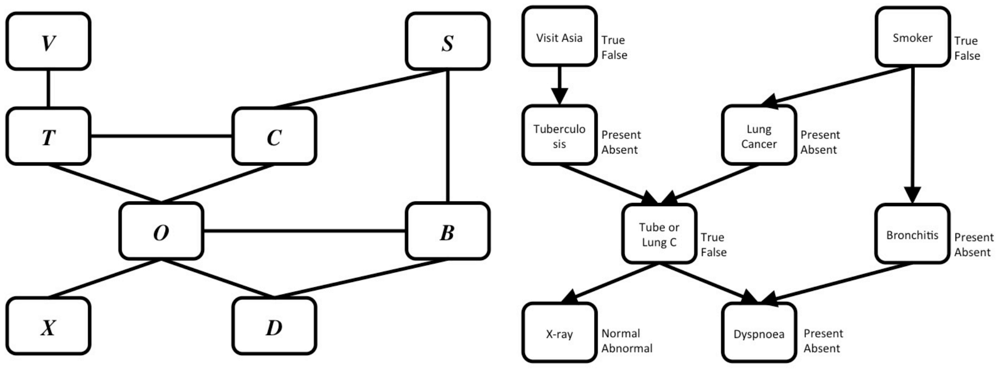


Figure 29. Undirected (left graph) and directed (right graph) representations of the ASIA model.

The relation between the graphical representation and the theory of the probabilistic model can be understood with the concepts of *active paths* and *separations* for undirected graphs and *active trail* and *d-separations* for directed graphs.

Let $G(\boldsymbol{V}, E)$ be an undirected graph, let $X_1 - \cdots - X_n$ be a path in G and $\boldsymbol{Z} \subset \boldsymbol{V}$. The path $X_1 - \cdots - X_n$ is active given Z if and only if $\forall X \in \{X_1, \dots, X_n\}, X \notin \boldsymbol{Z}$. In Figure 29 the undirected representation shows that the path $V - T - C - S$ is active given $\{B, O\}$ contrary to the path $V - T - O - X$, which is not active given $\{B, O\}$.

Considering three disjoint sets of nodes $\boldsymbol{X}, \boldsymbol{Y}$ and $\boldsymbol{Z}$ in G, a *separation* by $\boldsymbol{Z}$ of $\boldsymbol{X}$ and $\boldsymbol{Y}$ is when there aren't any active paths in G, linking $\boldsymbol{X}$ to $\boldsymbol{Y}$ given $\boldsymbol{Z}$. The notation for the separation of $\boldsymbol{X}, \boldsymbol{Y}$ by $\boldsymbol{Z}$ is $sep(\boldsymbol{X}\ ;\ \boldsymbol{Y}|\boldsymbol{Z})$. In the left graph depicted in Figure 29 $sep(\{T, X\}; \{S, B, D\}|\{O, C\})$ is a possible separation.

Equivalent concepts exist for the directed representation. Let's consider the directed graphs $\vec{G}(\boldsymbol{V}, E)$.

A trail $X_1 \leftrightarrow \cdots \leftrightarrow X_n$ in $\vec{G}$ is considered active given a set $\boldsymbol{Z}\ \subset \boldsymbol{V}$, if and only if :

1. Whenever there is a *v-structure* $X_i\ \rightarrow X_{i+1} \leftarrow X_{i+2}$ (such as $T \rightarrow\ O \leftarrow C$ in the example Figure 29) along the trail $X_1 \leftrightarrow \cdots \leftrightarrow X_n$, only X_{i+1} or one of his descendants must $\in \boldsymbol{Z}$.
2. Otherwise, no node $X \in Z$ must be present on the trail. [Pearl 1988]

In the right representation of Figure 29 $\mathrm{V} \rightarrow \mathrm{T} \rightarrow \mathrm{O} \leftarrow \mathrm{C} \leftarrow \mathrm{S}$ is an active trail given $\{\mathrm{O}, \mathrm{X}, \mathrm{D}\}$ and $\mathrm{C} \leftarrow \mathrm{S} \rightarrow \mathrm{B}$ is not an active trail given $\{\mathrm{S}\}$.

Considering three disjoint sets of nodes $\boldsymbol{X}, \boldsymbol{Y}$ and $\boldsymbol{Z}$ in $\vec{G}$, a d-separation by $\boldsymbol{Z}$ of $\boldsymbol{X}$ and $\boldsymbol{Y}$ in $\vec{G}$, denoted $dsep_{\vec{G}}(\boldsymbol{X};\ \boldsymbol{Y}|\boldsymbol{Z})$, occurs if and only if there aren't any *active trail* given $\boldsymbol{Z}$ linking any nodes $X \in \boldsymbol{X}$ and $Y \in \boldsymbol{Y}$. The directed separations associated with $\vec{G}$ are defined as such: $\mathcal{I}(\vec{G}) = \{dsep_{\vec{G}}(\boldsymbol{X};\ \boldsymbol{Y}|\boldsymbol{Z})\}$.

Looking to the right graph in Figure 29 it is possible to distinguish $dsep(\{D\};\{S\}|\{B,O\})$ and $dsep(\{V\};\{S\}|\{T,C\})$ but O does not d-separate V from S.

The separation between variables described for the two types of graphs permits to map the conditional independences between probability distributions. In the following sections it is shown how the BN and the MN use respectively the directed graph and the undirected graph representations of conditional independence among the probability distributions. Before highlighting the differences between the two representations it is important to precise the difference between the various types of independences mapping:

$$G\ is\ I-map\ of\ P \Leftrightarrow [sep(X,Y|Z) \Rightarrow X \perp Y|Z]$$

$$G\ is\ D-map\ of\ P \Leftrightarrow [X \perp Y|Z \Rightarrow sep(X,Y|Z)]$$

$$G\ is\ P-map\ of\ P \Leftrightarrow [X \perp Y|Z \Leftrightarrow sep(X,Y|Z)]$$

MARKOV NETWORK (MN)

A Markov Network is a probability distribution that uses an undirected graph as I-map. The probability distributions are encoded as *factors*. They are determined by the dependences represented in the undirected graph. Calculation over them is required for probabilistic inference.

Considering a set of variables $\boldsymbol{X}$, a factor is considered to be a function $\phi: Val(\boldsymbol{X}) \to \mathbb{R}$. A factor is non negative. When used in MNs factors do not map random variables values to probability. In most cases, they assign some nonnegative value that represents the amount of belief that an outcome may realize itself.

To be able to calculate inference, operators over factors are needed.

Multiplication operator

Considering three disjoint sets of variables $\boldsymbol{X}$,$\boldsymbol{Y}$ and $\boldsymbol{Z}$, the multiplication of two factors $\phi_1(\boldsymbol{X}=\boldsymbol{x},\boldsymbol{Y}=\boldsymbol{y}) * \phi_2(\boldsymbol{Y}=\boldsymbol{y},\boldsymbol{Z}=\boldsymbol{z})$ return a third factor $\phi_3(\boldsymbol{X}=\boldsymbol{x},\boldsymbol{Y}=\boldsymbol{y},\boldsymbol{Z}=\boldsymbol{z}): Val(\boldsymbol{X},\boldsymbol{Y},\boldsymbol{Z}) \to \mathbb{R}$ such that:

$$\forall\ \boldsymbol{x},\boldsymbol{y},\boldsymbol{z}\ \ \phi_3(\boldsymbol{X}=\boldsymbol{x},\boldsymbol{Y}=\boldsymbol{y},\boldsymbol{Z}=\boldsymbol{z}) = \phi_1(\boldsymbol{X}=\boldsymbol{x},\boldsymbol{Y}=\boldsymbol{y}) * \phi_2(\boldsymbol{Y}=\boldsymbol{y},\boldsymbol{Z}=\boldsymbol{z})$$

Marginalization operator

Considering a set of variable $\boldsymbol{X} = \{X_1,\dots,X_n\}$ and its factor $\phi(\boldsymbol{X})$ the marginalization over the variable $X_i \in \boldsymbol{X}$ $\phi_{-i}(\boldsymbol{X}\backslash X_i) = \sum_{\boldsymbol{Val}(X_i)} \phi(\boldsymbol{X})$, $\phi_{-i}: Val(\boldsymbol{X}\backslash X_i) \to \mathbb{R}$ such that:

$$\phi_{-i}(X_1,\dots,X_{i-1},X_{i+1},\dots,X_n) = \sum_{x \in Val(X_i)} \phi(X_1,\dots,X_i = x,\dots,X_n)$$

The two operators are required for the probabilistic inference. Even if factors are not always assigned probabilities values, Markov Networks are used to calculate inference and to do so they are coupled with specific probability distributions called Gibbs distributions.

Gibbs Distribution

Considering a set of variables $\boldsymbol{X}$, $P_{\boldsymbol{\Phi}}$ is the probability distribution over $\boldsymbol{X}$. $P_{\boldsymbol{\Phi}}$ Is a Gibbs distribution parameterised by a set of factor $\boldsymbol{\Phi} = \{\phi_1(\boldsymbol{C_1}), \dots, \phi_k(\boldsymbol{C_k})\}$ such that $\bigcup_{\boldsymbol{j=1}}^{\boldsymbol{k}} \boldsymbol{C_j}$ if and only if:

$$P_{\boldsymbol{\Phi}}(\boldsymbol{X}) = \frac{1}{Z}\tilde{P}_{\boldsymbol{\Phi}}(\boldsymbol{X})$$

where

$$\tilde{P}_{\boldsymbol{\Phi}}(\boldsymbol{X}) = \prod_{j=1}^{k} \phi_j(\boldsymbol{C_j})$$

is a un-normalized measure and

$$Z = \sum_{Val(\boldsymbol{X})} \tilde{P}_{\boldsymbol{\Phi}}(\boldsymbol{X})$$

Z is called the partition function that is a normalizing constant.

Computing the partition function is a very demanding task for the Markov inference algorithm. This research will explain how the compilation of the MLF permits a very fast computation of the function [Darwiche 2003]. A Gibbs distribution $P_{\boldsymbol{\Phi}}(\boldsymbol{X})$ factorize over a Markov Network if in the graph every $\boldsymbol{C_i}$ belonging to $\boldsymbol{\Phi} = \{\phi_1(\boldsymbol{C_1}), \dots, \phi_k(\boldsymbol{C_k})\}$ is a sub-graph of the original Markov Network graph. Each $\boldsymbol{C_i}$ is called a *clique* and each factor, that parameterizes the Markov Network, is called a *clique potential*.

A Markov Network can be defined as a pair noted $\mathcal{H} = (P_{\boldsymbol{\Phi}}, G)$, where $P_{\boldsymbol{\Phi}}$ is a Gibbs distribution over the graph G, where G is an undirected graph mapping and an I-map of $P_{\boldsymbol{\Phi}}$. The previous section explains how the *separations* determine a structure equivalent to independences in a probability distribution. Hence, if a probability distribution factorizes over an undirected graph then the graph is an I-map of the probability distribution.

It is theorized that for any probability distribution P and Markov Network $\mathcal{H}$, both over set of variables $\boldsymbol{X}$, if P is a Gibbs distribution that factorizes over $\mathcal{H}$, then $\mathcal{H}$ maps independences for P (*Soundness theory*) [Koller and Friedman 2009]. This theory demonstrates not only the fact that a probability distribution factorizes over an undirected graph gives us information about independences existing in it, but it also provides the guarantee that some dependences represented by edges can exist.

It is also possible to ascertain that if two variables X and Y are not separated by a set of variables $\boldsymbol{Z}$, then X and Y are dependent over Z in some probability distribution P that factorizes over $\mathcal{H}$. Mapping the independences between variables leads to map their dependences (*completeness theory*). The *Hammersley-Clifford* theorem states that if P is a positive distribution[45] then the opposite of the soundness theory is also true. Hence if $\mathcal{H}$ maps independences for P then P is a Gibbs distribution that factorizes over $\mathcal{H}$ [Koller and Friedman 2009].

[45] Positive distribution is strikly positive.

All the previous listed theorems establish the bases that permit to use the Markov Network as valid factorization of the joint probability distributions. Let's introduce a last notion: *Markov Blanket.*

For a given graph $\mathcal{H}$ and a random variable X in $\mathcal{H}$, we define the *Markov Blanket* of X over $\mathcal{H}$, denoted $MB_{\mathcal{H}}(X)$, as the collection of all the neighbours of X in $\mathcal{H}$. A Markov Blanket of X contains all the nodes that separates X form the rest of the graphs node. Thus, a Markov Blanket is the only knowledge required to predict the node's possible outcome [Pearl 1988]. Markov Networks are useful for optimizing inference calculation (see § 'Inference Algorithms').

BAYESIAN NETWORK (BN)

A BN is a probability distribution that uses a directed graph as I-map, where an MN relies on an undirected graph. This difference may not seem important, but it differentiates the applications of MNs and BNs. A first noticeable difference is the use of conditional probability distributions for the BN instead of a combination of factors and partition functions for the MNs.

A Bayesian Network is a representation of a probability distribution over a set of variables $\boldsymbol{V}$. The model uses a graphic representation to map the conditional independences between variables using d-separations [Pearl 1988]. Let:

- $\boldsymbol{V} = \{X_1, X_2, \dots, X_n\}$ be a set of random variables,
- $G = (\boldsymbol{V}, E)$ be a Directed Acyclic Graph (DAG) where $\boldsymbol{V}$ are the vertexes and E are the edges. G encodes a comprehensive representation of the complex relationships between the variables $\boldsymbol{V}$
- $P(X_i|\pi(X_i))$ be the probability parameter for each node $\mathrm{X_i}$ of $\boldsymbol{V}$ that allows quantifying the stochastic dependence relationship between the states of the variables. For each variable X_i all the probability parameters of $P(X_i|\pi(X_i))$ are arranged in a Conditional Probability Table (CPT). Each CPT contains as much parameters as there are instantiations of variables Xi and $\pi(X_i)$ combinations. In the case where the node does not have any parent, $P(X_i|\pi(X_i))$ becomes $P(X_i)$, i.e. a prior probability distribution.

The conditional independences mapped in the DAG allow the factorisation into the joint law according to the local Markov property that is also defined as the *chain rule* for the BN [Pearl 1988]:

$$P_J(X_1, X_2, \dots, X_n) = \prod_{i=1}^{n} P(X_i|\pi(X_i))$$

BNs differ from MNs in two ways: i) they rely on DAGs where MNs use undirected graphs; ii) the probability distributions associated with a BN factorize differently than those associated with a MN. These differences have a noticeable impact on the use of these frameworks. The first one is the ease for outsiders to model knowledge using BNs. Eliciting conditional probability distributions (CPD) as Conditional Probability Tables is more intuitive than eliciting factors, simply because CPTs are filled with probabilities, which are a more universally known mathematical object. This is the first reason why in this research the choice was made to utilize the BN framework over the MN. However, it will be show that both frameworks return similar results.

Active trail in BNs are more complex than their undirected counterpart. This is mainly due to the treatement of the v-structure (see Independence). To understand their role it is useful to intepretate active trails as flows of information. Depending on the graph topology the flow can be blocked or activated given evidences. There are three structures that can be found in a BN:

- **Chains** $X \rightarrow Z \rightarrow Y$ or $X \leftarrow Z \leftarrow Y$: When evidence on Z is given the trail is blocked between X and Y.
- **Common parents** $X \leftarrow Z \rightarrow Y$: When evidence on Z is given the trail is also blocked between X and Y.
- **V-structure** $X \rightarrow Z \leftarrow Y$: When evidence on Z is given the trail is not blocked between X and Y.

In the first two cases, the middle node blocks the flow of information given the evidence Z. An easy way to visualize the effect of observed evidences on the flow of information is to remove outgoing arcs from evidenced nodes. The third case behaves in a different manner. In fact the flow is blocked in $X \rightarrow Z \leftarrow Y$ if Z is not observed. The probability of X doesn't change the belief of Y when Z is not observed, but if Z is observed, then X affects the belief of Y.

The fact that d-separation correctly represents conditional independence assumptions for probability distributions that factorize over a BN is less intuitive than separation is for MNs. To justify the assumption the following theorems are necessary. First the **d-separation soundness** explains that for a directed graph $\vec{G}$, which nodes are a set of random variables $\boldsymbol{X}$ and P, which is a joint distribution over $\boldsymbol{X}$. If P factorizes according to $\vec{G}$, then $\vec{G}$ maps some of the conditional independencies for P. This theorem is very useful from a modelling perspective. Indeed the modelling of a valid BN passes through the specification of direct dependencies between the variables then filling the resulting CPT. The result is a factorized probability distribution for which the DAG maps the conditional independences. Consequently, defining a BN is a simple intuitive task.

The second theorem that justifies the relation between the d-separation and the conditional representation assumptions is the **d-separation completeness**. This theorem shows that for a directed graph $\vec{G}$, which nodes are a set of random variables $\boldsymbol{X}$, if X and Y are not d-separated given Z in $\vec{G}$, then the X and Y are dependent given Z in some distribution P that factorizes over $\vec{G}$.

Both theorems' proofs can be found in [Koller and Friedman 2009]. They state that d-separation is a valid graphical representation of conditional independences. With this understanding of both theorems it is possible to understand the following results.

The **d-separation soundness** theorem implies its reverse: for a directed graph $\vec{G}$, which nodes are a set of random variables $\boldsymbol{X}$ and P, which is a joint distribution over the same space, if $\vec{G}$ maps the conditional independences for P, then P factorises according to $\vec{G}$. For almost all distribution P that factorise over $\vec{G}$ that is, for all distributions, except for a set of measure zero in the space of conditional probability distribution parametrizations, we have that $\mathcal{I}(P) = \mathcal{I}(\vec{G})$. The meaning of tha last statement is is that almost all probability distributions that factorize over a BN graph are perfectly represented in terms of independence assumptions.

It is also possible in the BN framework to define the Markov Blanket for a random variable X. For a directed graph $\vec{G}$ the Markov Blanket for $X \in \boldsymbol{V}$, denoted $MB_{\vec{G}}(X)$ is defined to be X's parents, children and children's parents. In opposition with the Markov blanket in undirected graphs, a markov blanket of some node X in a direted graph includes nodes that are not direct neighbours of X. It is due to the particularity of v-structure present in directed graphs and the fact tha those structurs induces dependences between parents of same nodes. Hence the Markov Blanket for X includes all nodes that share a child node with X.

As the MN model use the factors to calculate marginal the BN model benefit from similar operations:

The use of the prior joint law allow to calculate the *marginal probability* of any subset $\boldsymbol{Y}$ of $\boldsymbol{V}$:

$$P_J(\boldsymbol{Y}) = \sum_{V \setminus Y} P_J(\boldsymbol{V})$$

When certain variables are observed to have specific values it is possible to calculate the marginal probability which is the goal of inference algorithms.

$$P_J(\boldsymbol{Y}|W = w_0) = \frac{P_J(\boldsymbol{Y}, W = w_0)}{P_J(W = w_0)}$$

FROM BAYESIAN NETWORK TO MARKOV NETWORKS

The two model detailed in the previous sections are closely related, but they cannot necessarily represent independences that the other can. To illustrate that assertion the following examples will provide a case where the dependences are correctly translated form a BN to a MN, and another case where dependences are obligatory lost in the translation.

For the first example let's consider the BN : $A \rightarrow B \rightarrow C$. It represents the factorization of the probability distribution: $P(A, B, C) = P(A) * P(B|A) * P(C|B)$. The equivalent dependence representation into a MN could be $M: A - B - C$. The joint probability distribution of the MN would be $P(A, B, C) = \frac{1}{Z}\phi_1(A, B) * \phi_2(B, C)$. In this case the possible value for the MN's factors could be $\phi_1(A, B) = P(A) * P(B|A)$ and $\phi_2(B, C) = P(C|B)$ with $Z = 1$. The example shows perfectly that both model encodes the same dependences: $dsep(A; C|B)$ and $sep(A; C|B)$. This example illustrates a situation in which both a BN and MN encode the same set of independences. However, in most situations BNs and MNs with similar graphical structures encode different dependences.

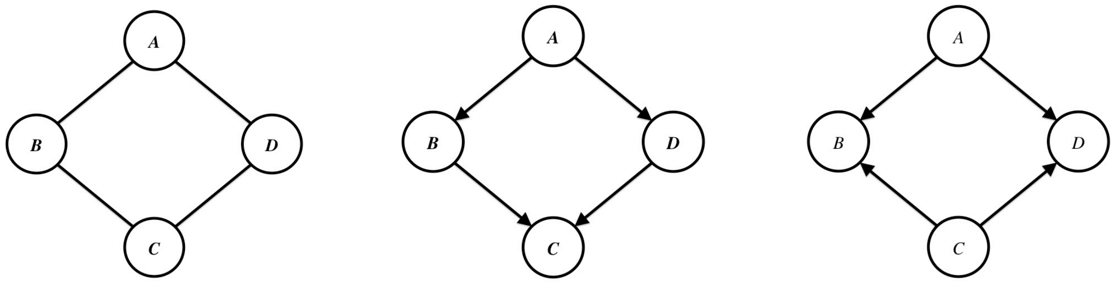


Figure 30. BNs misrepresentations (Figure 30.b and Figure 30.c) of dependences expressed by MN (Figure 30.a).

The following example proves the previous statement. Let's consider the following dependences concerning the variables $V = \{A, B, C, D\}$: $A \coprod C|\{D, B\}$ and $D \coprod B|\{A, C\}$ as represented in a MN graph by the first illustration (Figure 30.a). The second illustration (Figure 30.b) is a first attempt of representing the same independences with a BN graph. The graph represents the $dsep(A; C|\{B, D\})$ but does not succeed in representing the d-separation of B and D given $\{A, C\}$.The third illustration (Figure 30.c) is not able to represent the d-separation of A and C given $\{B, D\}$. The joint probability distribution induced by the graph Figure 30.b returns $P(A, B, C, D) = P(A) * P(B|A) * P(D|A) * P(C|B, D)$. The one induced by the graph Figure 30.c returns $P(A, B, C, D) = P(A) * P(B|A, C) * P(D|A, C) * P(C)$. By considering those probability distributions as factors, it is possible to see that they induce links non-visible in the MN ($B - C$ for Figure 30.b and $A - C$ for Figure 30.c). Conversely, one can notice that it is impossible for a MN to represent the independences expressed by the BN graph in Figure 30.b.

In most cases, we can transform a model into the other by losing independence assumptions - this usually implies that we add links or arcs. As it will be explained later, transforming a BN into a MN is simple: each pair of nodes that belong to the same CPT are linked by an edge. CPTs can then directly be used as factors and the partition function is equal to 1. The process is important since several inference algorithms reason over MNs. However, transforming an MN into a BN is more difficult and will not be discussed in this manuscript [Koller and Friedman 2009]. CPTs are normalized factors, thus they can be used to infer a MN from BN. In the BN illustrated in Figure 30.b the joint probability distribution factorizes according to the following CPTs: $\{P(A), P(B|A), P(D|A), P(C|B, D)\}$. We know that factors in MN are also called clique potentials because all random variables in a same factor form a clique in the MN graph. This is not the case for Figure 30.b since variables B,D and C do not form a clique in the BN graph. Given how BNs are defined, such missing links only occurs among parents of the same node. Adding such edges is called *moralization.* The moral graph of a DAG $\vec{G}$ over X is the undirected graph over X that contains an undirect edge between X and Y if:

1. $X \rightarrow Y$ or $Y \rightarrow X$ exists in $\vec{G}$
2. there is a node Z such that $X \rightarrow Z \leftarrow Y$ exists in $\vec{G}$

Of course some BNs are already moral (example Figure 30.c). Adding edges to moralize a BN remove independences. This entails a certain property for the moralized graphs. The moral graph of a DAG offers the minimal mapping of the independences represented by the BN graph. This is obvious, since the moralization create links where there are no edges before [Koller and Friedman 2009].

The moralization of a DAG is the first step in two major inference algorithms: (i) variable elimination and (ii) junction tree (see the respective sections in the same chapter).

MODELING USING BAYESIAN NETWORKS

When using BN to model specific domains, experts usually require specific functionalities that are presented in this paragraph. In BNs local structures are used to represent conditional probability distributions. It is important to differentiate the data structure used for the probabilistic computations from the one used for modelling purpose. Different data structures induce different costs for constructing a BN. CPTs are a straightforward representation of a discrete conditional probability distribution and are considered as the classic data structure used to encode probabilities in BNs. However the difficulty to fill CPTs grows exponentially with the number of random variables. Regarding ergonomics, there has been a considerable amount of research carried out by the different software companies selling BN oriented products. However, in our case we are more interested in alternative data structures that require fewer parameters, which help reducing the memory consumption of large CPTs and the number of computations. A glimpse will be taken to probabilistic and deterministic functions that are a must-have feature when dealing with expert knowledge. These functions span from standard probability distributions to logical operators.

Contextual Specific Independences. Multidimensional tables prevent from exploiting any structure in the conditional probability distribution, i.e. exploiting the fact that conditional probability distributions are constant for different instantiations of random variables. Such independence is called context specific independence and can be represented by using trees or rules [Boutilier and al. 1996][Koller and Friedman 2009]. In many cases, context specific independence can be exploited to reduce CPTs memory consumption. An example showing representation of a local structure expressing Z and Y are independent ginve $X = \bar{x}$ and dependent otherwise in a classic CPT (left) and in a tree (right) is shown in Figure 31.

		Z	
X	Y	z	$\bar{z}$
x	y	0.9	0.1
x	$\bar{y}$	0	1
$\bar{x}$	y	0.99	0.01
$\bar{x}$	$\bar{y}$	0.99	0.01

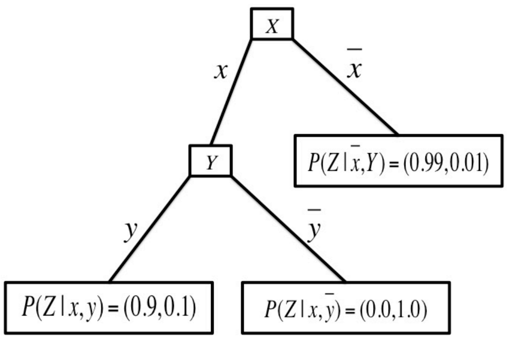

X
x
$\bar{x}$
Y
$P(Z \mid \bar{x}, Y) = (0.99, 0.01)$
y
$\bar{y}$
$P(Z \mid x, y) = (0.9, 0.1)$
$P(Z \mid x, \bar{y}) = (0.0, 1.0)$

Figure 31. CPT and tree exposing local structure within the conditional probability distribution.

Deterministic Functions.

When confronted to real world applications, we frequently encounter deterministic relationship, i.e. cases where the state of a random variable is known with no uncertainty if its parents are known. Such variables are called deterministic variables and are easily identifiable by the fact that probability values are equal either to one or to zero. A deterministic function is a triple (X_f, f, ϕ_f) where X_f is a discrete random variable in a BN, f a function such that $f\colon \pi(X_f) \to Val(X_f)$ and ϕ_f a CPT that maps $\pi(X_f)$ to zeros or ones. Deterministic functions can be considered observed if all their parents are observed. Classic deterministic functions are **logical or**, **logical and** and **logical xor**. There are also many specific functions, such as K-gates that are true only if at least K parents are true:

$$P(X|\pi(X)) = \begin{cases} 1 \; if \; |\{Y = True, Y \in \pi(X)\}| \geq K \\ 0 \; otherwise \end{cases}$$

Parametric Distribution.

Another classic conditional probability distribution variation is to use parametric distributions i.e. conditional distributions dependent over a parameter usually constant or context dependent. Formally, such parameters are considered as observed random variables and thus they do not require a prior distribution, which can be problematic if the parameter is not discrete. Usually the parameter is directly part of the CPT specification: $P(X) = (1 - \lambda, \lambda)$.

Noisy-or, Noisy-and and generalized linear models.

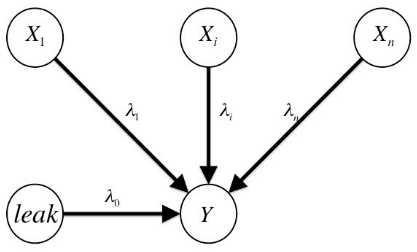


Figure 32. Noisy-Or gate in the case where causes can be independent

In many situations, a consequence can have multiple independent causes. Such situations are represented using a Noisy-or function. Figure 32 illustrates a *Noisy-or* with $n + 1$ parameters where Y and X_i are binary discrete random variables. Each λ_i is the probability $P\left(Y = true \middle| X_i = x_i, X_j = \bar{x}_j\right), j = 1, \dots, i-1, i+1, \dots, n$, except for leak probability, λ_0, representing an unknown cause. Since causes are supposed to be independent each other, the probability distribution $P(Y|X_1, \dots, X_n)$ can be inferred using the following equation (each x_i is supposed to be equal to 0 or 1):

$$P(Y = true|x_1, \dots, x_n) = 1 - (1 - \lambda_0) \prod_{i=1}^{n} (1 - \lambda_i)^{x_i}$$

There have been considerable works on *Noisy-or* and *Noisy-and* [Cozman 2004] [Xiang 2010]. Noisy-or functions are part of a more general class of probabilistic functions called generalized linear models [Koller and Friedman 2009].

Probabilistic Functions and Discrimination.

In contrast with deterministic functions, probabilistic functions are functions that return probability values different from zeros and ones. They are usually classic discrete probability distributions: the Poisson distribution, the Bernoulli distribution, the binomial distribution, the geometric distribution, and the negative binomial distribution are the most used ones. We can also proceed with the discretization of continuous distributions, since exponential or normal distributions are often useful to model some continuous random variable behaviour.

The graphical representation of ASIA, which is a classic example of a BN, is depicted in Figure 33. ASIA supports the diagnosis of respiratory diseases. Note that in the following descriptions the term 'node' is used for *variable* and the term 'link' for *dependence*.

In this example it is possible to separate the following nodes into 3 different categories:

- *"Medical Cause":* Visit Asia, Smoker
- *"Diagnostic":* Tuberculosis, Lung Cancer, Tuberculosis or Lung Cancer and Bronchitis
- *Indicator":* X-ray, Dyspnoea

Note that each node can be instantiated into two possible states as it is shown in the Figure 33. For each node there is a distribution of probability for each state according to the state of the parent nodes, for instance $P(Tuberculosis \mid Visit\ Asia)$

The ASIA example allows intuitively guessing the relationships between *"Medical Causes"* and "*Diagnostics"* and between "*Diagnostics"* and "*Indicators"* by looking at the links. For instance, visiting Asia increases the chance of contracting tuberculosis and tuberculosis or lung cancer can be diagnosed by looking at X-rays and patient signs of dyspnoea. By transitivity, the graph represents the relationships between *Causes* and *Indicators*: visiting Asia increases the chance of having signs of dyspnoea. It also highlights conditional independences: for example, *Causes* have no mutual relations in this model (smoking has no influence with visiting Asia and vice versa). With such BN it is possible to infer the distribution of any marginal probability according to any observations (example $P(Visit\ Asia \mid Dyspnoea = Present)$ or $P(Dyspnoea \mid Visit\ Asia = True)$).

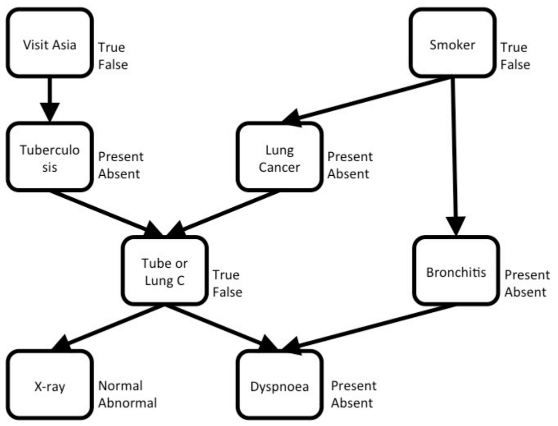


Figure 33. ASIA BN.

BN VS. MN

BNs and MNs exploit independences present in probability distributions to factorise them and offer a tractable representation for uncertain reasoning. BNs can be transformed in MNs that are their undirected counterpart playing an important role in probabilistic inference.

In spite of their theoretical similarity, MN and BN differ greatly in their applications. The most notable difference lays in the fact that BN utilise conditional probability distributions where MN combine factors with a partition function. The probability distribution associated with a BN factorises differently than the one associated with a MN. These subtle differences have huge impact in the use of these frameworks. The most notable one is the ease for experts to model knowledge using BNs: eliciting conditional probability distributions as CPTs is an easier task that eliciting factors, because probabilities are mathematical objects commonly known. This remains true even if the active trails in BN are more complex that their undirect counterparts.

INFERENCE ALGORITHMS

Probabilistic inference is a family of methods for calculating an updated state of the network (MN or BN) according to the observation or not of evidence realizations denoted $\boldsymbol{e}$. The inference computation permit to return the probability $P(X|\boldsymbol{E} = \boldsymbol{e})$. In the remainder of this manuscript, the plain term 'inference' refers to probabilistic inference.

An evidence over a variable X is a measure of likelihood $e(X) = p(E = e|X)$. It is possible to distinguish between *hard* and *soft* evidence. Hard evidence expresses total knowledge over the state of the variable: the likelihood of one instantiation is equal to 1 (maximised) and those of all the other instantiations are equal to 0. Soft evidence expresses the belief or partial knowledge of the state of the variable. Soft evidence is represented by a distribution of likelihood over the variable with at least two instances' likelihood different from 0.

A result of probabilistic inference when inserting evidence in a BN and in a MN is the transformation of respectively the variable probability distributions and the factors. The effect is also visible in the graphic representation for hard evidence. For the BN, the nodes representing the observed variables are modified by removing the outgoing edges - the new probability distributions if X is observed with an evidence $e(X)$ are $P'\big(X\big|\Pi(X)\big) = P\big(X\big|\Pi(X)\big) * e(X)$. Each child node Y of the observed node X ($Y \in child\ (X)$) retains a new distribution $P'(Y|\Pi(Y)\backslash X) = P\big(Y\big|\Pi(Y)\big) * e(X)$. In the MN graphical representation the node and the links to the other variable are simply removed and all concerned factors are projected over sub-factors $\phi' = \sum_X \phi * e$.

There are several methods for calculating inference such as *most probable explanation (MPE)*, *maximum a posteriori (MAP)* and *maximizes the expected utility (MEU).* These methods are implemented by different algorithms detailed in [Pearl 1988] [Jordan 1999] [Koller and Friedman 2009] [Darwiche 2009].

Probabilistic inference calculation is an NP-hard problem [Cooper 1987] [Cooper 1990] [Dagum and Luby 1993]. There are two categories of inference methods: i) exact and ii) approximated. This research utilises exact inference.

Many practical and non-trivial probabilistic inference applications are confronted with the issue of the computational complexity. The most important parameter for inference complexity is the MN tree-width, whose computation is also NP-hard. In fact, every method for inference shows an NP-hard complexity in at least one of its tasks.

This section will focus on the description of the "classical" inference methods: in particular the *variable elimination* method family and the *junction tree inference* method family - the method of the latter family detailed hereby is the so-called Shafer-Shenoy [Shafer and Shenoy 1990].

VARIABLE ELIMINATION

The *variable elimination* (VE) inference algorithm [Zhang and Poole 1994] [Zhang and Poole 1996] [Dechter 1998] is a simple but powerful and efficient algorithm. Variables can be eliminated from the joint probability distribution that calculates the target probability $P(\boldsymbol{Y}|\boldsymbol{E}=\boldsymbol{e})$:

$$P(\boldsymbol{Y}|\boldsymbol{E}=\boldsymbol{e}) = \frac{\sum_X P(\boldsymbol{Y},\boldsymbol{X},\boldsymbol{E}=\boldsymbol{e})}{P(\boldsymbol{E}=\boldsymbol{e})}$$

The direct computation of the joint probability distribution is demanding in memory and time. VE exploits the factorization from the joint probability distribution and tries to eliminate variables by considering only sub-sets of factors.

In the ASIA example (Figure 33) it is possible to factorize the joint probability distribution as follow.

$$P(V,T,L,B,O,X,D,S) = P(V) * P(S) * P(T|V) * P(L|S) * P(B|S) * P(O|T,L) * P(X|O) * P(D|O,B)$$

Removing the variable S would consist of calculating

$$P(V,T,L,B,O,X,D) = \sum_S P(V) * P(S) * P(T|V) * P(L|S) * P(B|S) * P(O|T,L) * P(X|O) * P(D|O,B)$$

Since the variable S is localized to the factor $\phi(S,L,B) = P(S) * P(L|S) * P(B|S)$, the sum over S can be calculated as such:

$$P(V,T,L,B,O,X,D) = P(V) * P(T|V) * P(O|T,L) * P(X|O) * P(D|O,B) * \sum_S \phi(S,L,B)$$

It is possible to eliminate other variables until the remaining subset of variables that is relevant to the target probability distribution. Note that the order of elimination is essential to the VE inference performance, because computing optimal order of variable elimination is a NP-hard problem [Kjærulff 1990] [Bodlaender 1993].

VARIABLE ELIMINATION VARIANTS

There exist several extensions and variations of VE. One of the most notable is the *bucket elimination framework* that extends the application of VE to other inference methods (MPE and MAP) and to other algorithmic problems (constraint satisfaction among others) [Dechter 1996] [Dechter 1998] [Darwiche 2010]. A generalization of VE to junction trees has been proposed in [Cozman 2000], where VE is changed to save intermediate computation results in a structure similar to a junction tree. Finally, an optimization of VE has been proposed that focuses on eliminating values instead of variables: at every iteration individual factor entries, instead of variable entries, are eliminated [Bacchus et al. 2003].

The VE algorithm is *query sensitive*, i.e. the queried variable must be specified in advance; hence, the entire data structure must be re-initialised at each new query. The junction tree algorithms generalize VE to avoid re-runs - it compiles the probabilities into a data structure (junction tree) that supports the execution of a large class of queries.

JUNCTION TREES INFERENCE ALGORITHMS

The junction tree is a data structure represented by a cluster graph that follows the following rules:

1. *Single connection*: there is only one path between each pair of clusters.
2. *Coverage*: for each clique there is a cluster covering it.
3. *Running intersection*: for each pair of clusters B and C that contain X, each cluster on the unique path between B and C also contains X.

The first step in building a junction tree is to triangulate the MN's graph. The § 'From Bayesian Network to Markov Networks' already explains how a BN can be encoded into an MN (at the cost of probable independences loss). The following MN must be triangulated.

In order to define the triangulation, the notion of chordal graph is introduced.

Let $G = (V, E)$ be an undirected graph. G is chordal if all cycles of length > 3 have a chord, i.e. for all cycles $l = X_1 - \cdots - X_n$ $(n > 3)$ two nodes X_i and X_j exists in l such that the edges $X_i - X_j$ exists in E but not in l. Transforming a graph G into a chordal graph is called triangulating G.

A naïve graph triangulation algorithm will choose randomly nodes, remove it form the graph and connect between them all its neighbours with new connections called fill-ins. The graphical representation of this action corresponds to the VE node elimination. The number of fill-ins added by the elimination of a node is a good estimator of the elimination quality. Ideally this value needs to be low but finding minimal triangulation is NP-hard. This is why triangulation relies mostly on approximate algorithms based on different heuristics. The complexity of this type of algorithm is independent of the complexity of the inference algorithm and the fact of finding an optimal solution for triangulation does not guaranty the feasibility of the inference. The contributions on triangulation are numerous [Rose et al. 1976] [Robertson and Seymour 1986] [Kjærulff 1990] [Bodlaender 1993] [Koller and Friedman 2009].

Cliques in a triangulated graph represent any intermediate factor created by VE and can be represented using a *clique graph*. Considering an undirected graph G, a clique graph U of G is an undirected graph where nodes represent cliques C_i of nodes of G. $C_i \in U$ if there is not any C_j such that $C_i \subseteq C_j$. An edge $C_i - C_j$ exists in U if $C_i \cap C_j \neq \emptyset$. Conversely if $C_i \cap C_j \neq \emptyset$, then a path linking C_i to C_j exists in U.

Hence building a clique tree U from a triangulated undirected graph G consist basically in two step:

1. Remove node n from G;
2. Add a clique node C in U, representing n and its direct neighbours in G, making sure that another clique doesn't already exists including the new clique C and link the clique C to cliques with nodes in common.

What remains to do is to extract a clique tree from the clique graph. Not all clique trees are junction trees since junction trees must respect the running intersection property (see above). Hence not all clique trees are suitable for inference. The running intersection property ensures the correctness of junction tree based algorithms. A possible solution for junction tree extraction is to use a maximum spanning tree algorithm over a weighted clique graph. A weighted clique graph is a clique graph for which each edge $C_i - C_j$ has a weight $|C_i - C_j|$ [Shibata 1988] [Jensen 1988].

In order to present the junction tree algorithm, the notion of *tree-width* needs to be introduced. The tree-width of a tree is the size of the largest clique minus one. The tree-width of a graph is the minimum width among all possible junction tree of a graph.

Different junction trees are obtained with different elimination orders. Finding the junction tree with the smallest clusters is an NP-hard problem.

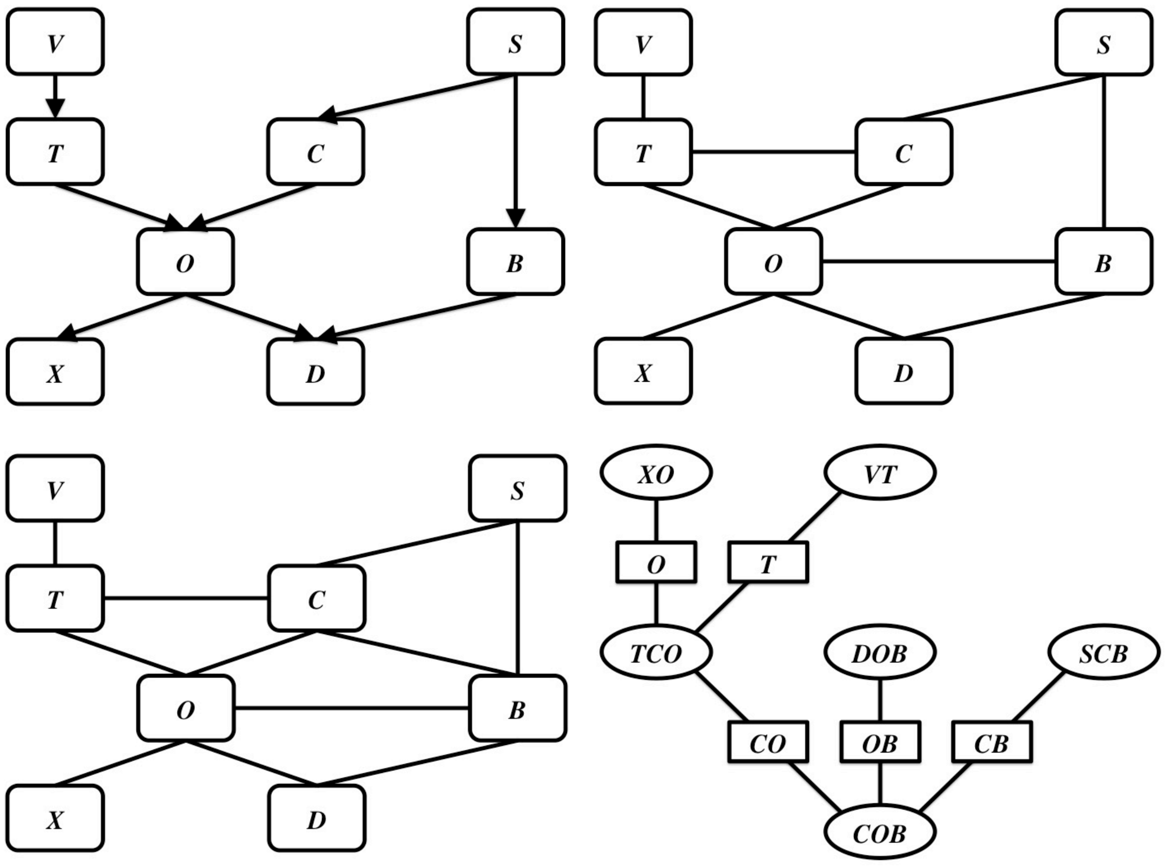


Figure 34. ASIA example transformation from BN to Junction Tree: a. BN, b. MN, c. triangulated chordal graph and d. Junction Tree.

Junction tree algorithms take as input a decomposable probability and its junction tree. They have the same distributed structure. Each cluster knows its local potential and its neighbours and broadcasts one message, which is a potential function, to its neighbours. Each cluster is able to compute the marginal probability of its variables by combining its potential with the messages it receives. These algorithms obey the *message passing protocol*: a cluster A is allowed to send a message to a neighbour B only after it has received messages from all neighbours except B.

A procedure compliant with the protocol is:

1. Choose one cluster R to be the root.
2. Execute *collect*(R) – *collect*(C): for each child B of C recursively call *collect*(B) and pass a message from B to C

3. Execute *distribute*(R) – *distribute*(C): for each child B of C, pass a message to B and then recursively call *distribute*(B).

SHAFER-SHENOY ALGORITHM

The Shafer-Shenoy inference algorithm transforms MN into junction trees and computes marginal probabilities using a message-passing scheme [Shenoy and Shafer 1986] [Shenoy and Shafer 1988].

The message sent from a cluster B to a neighbour C is:

$$\mu_{BC}(\boldsymbol{u}) \triangleq \sum_{\boldsymbol{v} \in \mathbb{X}_{B \setminus C}} \varphi_B(\boldsymbol{u} \cup \boldsymbol{v}) \prod_{(A,B) \in E, A \neq C} \mu_{AB}(\boldsymbol{u_A} \cup \boldsymbol{v_A})$$

Cluster B computes the product of its local potential and the message from all clusters except C, marginalises all variables that are not in C, and then sends the result to C. All messages are well defined because of the *single connection* property of the junction tree.

The cluster belief at C is defined as:

$$\beta_C(\boldsymbol{u}) \triangleq \psi_C(\boldsymbol{u}) \prod_{(B,C) \in E} \mu_{BC}(\boldsymbol{u}_B)$$

OTHER JUNCTION TREE BASED ALGORITHMS

Historically, junction tree algorithms where among the first inference implementations. Among them, there are different axiomatizations and message passing schemes that are variants of Shafer-Shenoy [Lauritzen and Spiegelhalter 1990] [Shafer and Shenoy 1990] [Lepar and Shenoy 1998]. Currently, the most performant junction tree algorithm is *lazy propagation* [Madsen and Jensen 1999]. Lazy propagation exploits d-separation to prevent unnecessary computation. There have been efforts to produce variants of junction tree algorithms without resort to graphical concepts [Draper 1995][Darwiche 1998], but these efforts have not produced a variable elimination-like scheme for inference.

OTHER APPROACHES

Probabilistic inference algorithms are not limited to the two approaches presented in this section. Other approaches exploiting distinct features of Bayesian Networks and Markov Networks have been proposed.

Belief propagation, also called the *polytree algorithm*, is a polynomial inference algorithm for polytree Bayesian Networks [Pearl 1982] [Kim and Pearl 1983] [Pearl 1988]. It can be understood as a message-passing algorithm that reasons on the Bayesian Network's graph. Belief propagation cannot be applied to non-polytree BNs for exact inference, but, if applied several times to non polytree BNs, it functions as an efficient approximate algorithm, and is called *generalized belief propagation* [Yedidia et al. 2001]. The reason is that cycles induce a repetition of messages sent by nodes in the same cycle. A solution called *global conditioning* or *loop cutset conditioning* cuts the cycles by conditioning over DAG's cutset [Pearl 1986] [Pearl 1988] [Suermondt and Cooper 1990].

Finding the minimal cutset in a graph is NP-hard [Garey and Johnson 1979] [Hao and Orlin 1994] [Becker et al. 2000]. However, it is possible to avoid conditioning the network over every cutset variables and to limit the conditioning to smaller and sometimes distinct portions of the DAG. Such technique is called *local conditioning* and is by order of magnitude faster than *global conditioning* [Díez 1996] [Fay and Jaffray 2000].

Recursive conditioning is another inference algorithm using the message passing scheme that relies on recursive splitting of the graph [Darwiche 2001] [Allen and Darwiche 2003] [Grant 2010] [Grant and Horsch 2005]. In the current state of the art, global and local conditionings are not considered efficient inference algorithms. Conversely, recursive conditioning offers interesting time-space trade-offs that can be crucial when computing inference on limited resources. This advantage is emphasized by the fact that conditioning can be applied with the popular VE inference algorithm.

BAYESIAN NETWORK COMPILATION

Junction trees algorithms and VE share the same complexity limitation due to the MNs tree-width. One approach to overcome the limitation induced by high tree-width for exact inference is to exploit local structures. Local structures are alternate representations of factors exploiting local symmetries in the probability distribution. However, junction trees and VE algorithms can exploit specific operators for these local structures, but do not exploit them from a graphical point of view.

This shortcoming has led to the development of a new family of inference algorithms that compile MNs into *Arithmetic Circuits* (AC) via the transformation into a Conjunctive Normal Form (CNF) [Darwiche 2003] [Chavira and Darwiche 2005] [Chavira et al. 2006] [Chavira and Darwiche 2007].

A hypothesis of this work is that Bayesian inference by AC compilation is the technique that gives a solution to the specific computation complexity challenges raised by the use of probabilistic inference for dynamic scheduling applied to SOA testing, in particular the presence of large tree-width.

COMPOSITION OF AN ARITHMETIC CIRCUIT

The principle of inference by compilation is straightforward: the *chain rule* [Darwiche 2003] states that the probability of the set of instantiations $\boldsymbol{x}$ of all the network variables $\boldsymbol{X}$ is the product of all the network parameters consistent with $\boldsymbol{x}$.

$$P(x_1, \dots, x_n) = \prod_{i=1}^{n} P(x_i|\boldsymbol{u}_i)$$

where $\boldsymbol{u_i}$ is the (possibly empty) set of values for the parents of X_i in $\boldsymbol{X}$.

It follows that any Bayesian Network (BN) can be represented by a multi-linear function (MLF) with specific properties (see below for more details).

The MLF is composed of two types of variables:

- Evidence indicator λ_x: it is a Boolean variable - there is one evidence indicator for each instantiation of the network variable;

- Network parameter $\theta_{x|\boldsymbol{u}}$: it is a variable that refers to the probability distribution of the state of the value x given the state of the parent nodes $\boldsymbol{u}$ - there is one network parameter for each different value of a conditional probability table.

The MLF is the sum of an exponential number of terms. There is a term for each possible instantiation of the network variables. The term is the product of the evidence indicators and the network parameters of the instantiation. The Arithmetic Circuit (AC) is a representation of such a function that facilitates its computing.

An AC over a set of variables $\boldsymbol{X}$ is a rooted directed acyclic graph whose leaf nodes are labelled with numeric constants or variables and intermediate nodes are labelled with arithmetic operations. The root of the circuit is the output of the function. The number of edges that it contains measures the size of an AC.

An interesting feature of the AC is that it allows avoiding re-computations of operations within sub-circuits.

In our case, the AC is used to represent the multi-linear function expressed by the BN, so the only used arithmetic operations are additions and multiplications.

The MLF contains all the information about the variables and the dependences, so that, all the answers to probabilistic queries can be obtained by evaluating and differentiating the function.

A BN is a graphical representation of a joint probability distribution that can be viewed as a MLF. The AC is a factorized version of the MLF that avoids redundancies like repeating sub-circuits.

A MLF is the sum of an exponential number of terms where each term is an instantiation of all network variables multiplied by the probability parameters expressing their mutual dependences.

The following Figure 35 sketches a very simple example of BN.

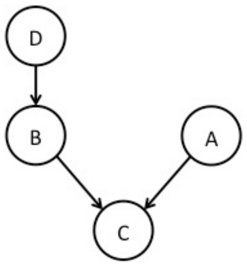


Figure 35: Example of a Bayesian Network.

The MLF corresponding to this BN is:

$f(\mathrm{B}, \Theta) =$

$$\lambda_a\lambda_b\lambda_c\lambda_d\theta_{(c|ab)}\theta_a\theta_{(b|d)}\theta_d + \lambda_a\lambda_b\lambda_{\bar{c}}\lambda_d\theta_{(\bar{c}|ab)}\theta_a\theta_{(b|d)}\theta_d + \lambda_a\lambda_{\bar{b}}\lambda_c\lambda_d\theta_{(c|a\bar{b})}\theta_a\theta_{(\bar{b}|d)}\theta_d +$$
$$\lambda_a\lambda_{\bar{b}}\lambda_{\bar{c}}\lambda_d\theta_{(\bar{c}|a\bar{b})}\theta_a\theta_{(\bar{b}|d)}\theta_d + \lambda_{\bar{a}}\lambda_b\lambda_c\lambda_d\theta_{(c|\bar{a}b)}\theta_{\bar{a}}\theta_{(b|d)}\theta_d + \lambda_{\bar{a}}\lambda_b\lambda_{\bar{c}}\lambda_d\theta_{(\bar{c}|\bar{a}b)}\theta_{\bar{a}}\theta_{(b|d)}\theta_d +$$
$$\lambda_{\bar{a}}\lambda_{\bar{b}}\lambda_c\lambda_d\theta_{(c|\bar{a}\bar{b})}\theta_{\bar{a}}\theta_{(\bar{b}|d)}\theta_d + \lambda_{\bar{a}}\lambda_{\bar{b}}\lambda_{\bar{c}}\lambda_d\theta_{(\bar{c}|\bar{a}\bar{b})}\theta_{\bar{a}}\theta_{(\bar{b}|d)}\theta_d + \lambda_a\lambda_b\lambda_c\lambda_{\bar{d}}\theta_{(c|ab)}\theta_a\theta_{(b|\bar{d})}\theta_{\bar{d}} +$$

$\lambda_a\lambda_b\lambda_{\bar c}\lambda_{\bar d}\theta_{(\bar c|ab)}\theta_a\theta_{(b|\bar d)}\theta_{\bar d} + \lambda_a\lambda_{\bar b}\lambda_c\lambda_{\bar d}\theta_{(c|a\bar b)}\theta_a\theta_{(\bar b|\bar d)}\theta_{\bar d} + \lambda_a\lambda_{\bar b}\lambda_{\bar c}\lambda_{\bar d}\theta_{(\bar c|a\bar b)}\theta_a\theta_{(\bar b|\bar d)}\theta_{\bar d} +$
$\lambda_{\bar a}\lambda_b\lambda_c\lambda_{\bar d}\theta_{(c|\bar a b)}\theta_{\bar a}\theta_{(b|\bar d)}\theta_{\bar d} + \lambda_{\bar a}\lambda_b\lambda_{\bar c}\lambda_{\bar d}\theta_{(\bar c|\bar a b)}\theta_{\bar a}\theta_{(b|\bar d)}\theta_{\bar d} + \lambda_{\bar a}\lambda_{\bar b}\lambda_c\lambda_{\bar d}\theta_{(c|\bar a\bar b)}\theta_{\bar a}\theta_{(\bar b|\bar d)}\theta_{\bar d} +$
$\lambda_{\bar a}\lambda_{\bar b}\lambda_{\bar c}\lambda_{\bar d}\theta_{(\bar c|\bar a\bar b)}\theta_{\bar a}\theta_{(\bar b|\bar d)}\theta_{\bar d}$

The corresponding AC is presented in Figure 36. Note that it is possible that node I and node II are roots of sub-circuits that should have been duplicated, but are built only once avoiding calculation redundancy.

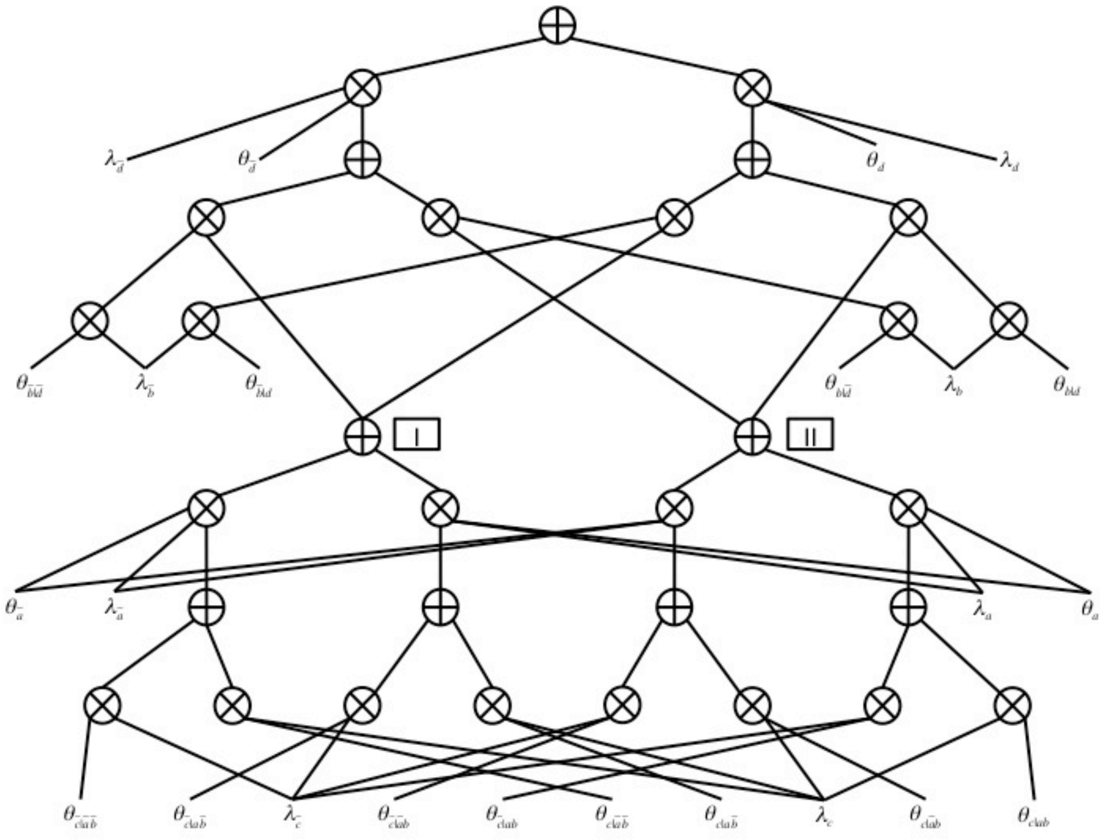


Figure 36: Example of an Arithmetic Circuit.

OPERATIONS OVER AC FOR INFERENCE

The probability of the observations of evidence $\boldsymbol{e}$ (collection of variable instantiations) can be calculated with the MLF by replacing all evidence indicators consistent with the observations with the value 1 and all evidence indicators contradicting the observations with the value 0.

$$f(\boldsymbol{e}) = Pr\,(\boldsymbol{e})$$

The partial derivative of a MLF in respect to evidence indicator λ_x gives the probability $P\,r(x, \boldsymbol{e} - X)$ of a new observed evidence in which x is the only difference with a previous observation. The partial derivative is calculated replacing all evidence indicators λ_x with the value 1 and all terms not containing λ_x with 0. The result is equivalent to conditioning the function to the event $X = x$, i.e. $\frac{\partial f}{\partial \lambda_x} = f\,(x)$. Thus, when differentiating the polynomial with respect to the evidence indicator of x and evaluating the result at evidence $\boldsymbol{e}$ the result is equal to:

$$\frac{\partial f}{\partial \lambda_x}(\boldsymbol{e}) = Pr\,(x, \boldsymbol{e} - X)$$

The computation of the probability of an event x given an observation $\boldsymbol{e}$, when $x \notin \boldsymbol{e}$, is straightforward – it can be obtained from the Bayes theorem

$$Pr(x|\boldsymbol{e}) = \frac{1}{f(\boldsymbol{e})}\frac{\partial f}{\partial \lambda_x}(\boldsymbol{e})$$

An AC can be evaluated and differentiated in time and space that are linear to its size. The evaluation traverses the circuit upward computing the value of a node after computing its children. After the evaluation it is possible with one downward traversal to calculate all the first differentials of all variables given evidence $\boldsymbol{e}$.

In [Darwiche 2003], the author gives a description of an algorithm for evaluation and differentiation that is implemented in this inference-based test Scheduler.

COMPILATION ALGORITHMS

The premise of the BN compilation is that every BN can be expressed into a MLF, whose evaluation and differentiation solves the exact inference over its variables. An exponential sized MLF can be compiled into a non-exponential sized AC. For the authors of [Shaafer and Shenoy 1986] and [Jensen et al. 1990] the compilation of the MLF consists in the factorisation of the MLF that can be reduced to the factorisation of the encoded CNF. Park and Darwiche [Park and Darwiche 2003] show that building a jointree for a Bayesian Network consists basically in transforming a MLF into an AC.

Darwiche [Darwiche 2002] has been the first to propose a method that compiles efficiently a MLF into an AC. This method avoids to be confronted with the exponential size of the MLF because uses first an encoder that transforms a BN into a CNF. The CNF permits to process the MLF in the Boolean domain. The CNF is factored by all its variables one at a time by using Boolean algebra until a result that is similar to a decision diagram. This diagram has the specificity of being a smooth deterministic Decomposable Negational Normal Form (d-DNNF). The particularity of this diagram is that it can be easily translated into an AC.

Different solutions for factoring the CNF have been proposed. Darwiche [Darwiche 2002] presents a first approach to encode a MLF into a CNF. This approach introduces the concept of Context Specific Independences that are different from structural independences and can be exploited for the compilation.

In Chavira and Darwiche [Chavira and Darwiche 2005] propose the use of the DPLL algorithm [Davis and al. 1962] [Darwiche 2004] to factorize the encoded CNF. The algorithm chooses one by one the variables, e.g. x, and factorises the CNF by x and $\overline{x}$, adds the two factors and recursively processes them. It also keeps track of the factorisation into a cache to avoid redundant future calculation. At the beginning of this investigation the author has used the CNF encoding approach to directly encode the SAUT structural model into a d-DNNF.

Other methods exist that do not use CNF encoding. Compiling Bayesian Networks using Variable Elimination Chavira and Darwiche [Chavira and Darwiche 2007] propose a method using Variable Elimination as a basis for compilation with Algebraic Decision Diagrams (ADD). The method exploits local structures and transforms a large body of research (CPT) into a more structured representation of factors (ADD) and uses a recursive algorithm that allows compiling the BN into an AC by-passing the CNF encoding. Other method exists that do not use CNF encoding such as Recursive conditioning [Darwiche 2000].

PROBABILISTIC APPROACH TO TESTING AND TROUBLESHOOTING

TESTING AND PROBABILISTIC INFERENCE

In their position paper *at FSE/SDP workshop on 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 [Namin and Sridharan 2010]. They provide a brief overview of some popular Bayesian reasoning methods (Bayesian classification, Markov decision processes, BN, stochastic sampling etc.), along with a justification of why they are applicable to different software testing tasks.

Namin and Sridharan illustrate the well-known advantages of the usage of probabilistic inference - intractable exact inference in complex real-world domains, mathematically well-defined mechanism for representation, explicit modelling of uncertainty, management of multiple hypothesis about the state of the target, capability of integrating additional information about the state of the system. Furthermore, the Markov assumption (a key principle in Bayesian reasoning) - given the system state S_{t-1} at time $t - 1$, the current action and the current observation, the state S_t at time t can be estimated conditionally independent of all prior states, actions and observations - is well suited for testing: the presence/absence of a fault does not change during the test session and each test run does provide an independent test verdict. The Markov assumption is the basis of the two step iterative Bayesian inference process where a *prediction* step updates the belief of all possible hypotheses of system state based on the prior belief and the actions taken since then and a further *correction* step “corrects” the updated belief based on the correspondence between the expected and actual observations. Such an adaptive procedure is well-suited for exposing failures and determining the location of faults that are source of the failures.

Namin and Sridharan consider test case generation a major software testing challenge that can be formulated using Bayesian reasoning methods. Efficiency and effectiveness are the main concerns in this domain. Prior research has already resulted in strategies of test case generation (see below [Rees et al. 2001] [Wooff et al. 2002] [Gras et al. 2006]) and in approaches of test cases prioritization for regression testing (see below [Mirarab and Tahvildari 2007]) that are based on BN. Other domains of applicability are (i) mutation testing, (ii) representing and measuring the reliability of program component (defect density, time to failure) and (iii) software economics and metrics. Furthermore, Namin and Sridharan discuss some practical challenges to the widespread use of Bayesian methods - such as sensitivity to prior probabilities and the steep learning curve for users - along with possible solutions to these challenges.

Rees, Wooff and colleagues [Rees et al. 2001] [Wooff et al. 2002] 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 provokes software failures. Indeed, starting from a partitioning of the input domain, the BN allows the tester to quickly discover what partition or combination of partitions can be associated with a failure. In the BN, probabilities of the cause of failure being in some nodes would increase substantially, leading the tester to those areas most likely to contain the fault. Further work is to be done on re-testing in such cases. Since the model identifies which parts of the system are unconnected to those where the test has failed, it can also indicate which tests can continue to be run before the fault needs to be located. The probabilistic model can be used either to generate and prioritise the test suite in a purely automatic fashion or to function as a decision support system for evaluating and choosing how to sequence a suite suggested by the tester and to check for additional tests which may have been overlooked [Coolen et al. 2007].

Gras and colleagues [Gras et al. 2006] present the Motorola Labs' Bayesian test assistant (BTA), an advanced decision support tool to optimize all verification and validation activities, in development and system testing. Motorola Labs built a library of causal models to predict, from key process, people and product factors, the quality of artefacts at each step of the software development. BTA links the predictions from development models by mapping dependences between components or subsystems to predict the level of risk in each system feature. As a result BTA generates a test strategy that optimizes the writing of test cases. During system test, BTA scores test cases to select an optimum set for each test step, leading to a faster discovery of defects.

An important aspect of regression testing is to prioritize the test cases on the basis of specific criteria. Mirarab and Tahvildari [Mirarab and Tahvildari 2007] present a novel approach to prioritizing test cases in order to enhance the rate of fault detection. Their approach is based on probability theory and utilizes Bayesian Networks (BN) to incorporate source code changes, software fault-proneness, and test coverage data into a unified model. The performance of this technique is evaluated by using APFD (Average Percentage Faults Detected) measure on eight consecutive versions of a large-size Java application augmented with hand-seeded faults. The results show that when there are reasonable numbers of faults in the source code, this proposed novel technique is capable of achieving better values of APFD in the comparison with other techniques. The obtained results indicate a significant increase in the rate of fault detection when a reasonable number of faults are available. In the pursuit of future research, the authors consider the possibility of interactive prioritization incorporating feedback by simply making evidence nodes in BN after each test run. In the author's best knowledge, there are not yet published results of this further research.

The selection of software tests is a very important activity to ensure that the software reliability requirements are met. Generally tests are run to achieve maximum coverage of the software code and very little attention is given to the achieved reliability of the software. Using an existing methodology, Periaswamy and McDaid [Periaswamy and McDaid 2006] describe how to use Bayesian Networks to select unit tests based on their contribution to the reliability of the module under consideration. In particular, the work examines how the approach can enhance test-first development by assessing the quality of test suites resulting from this development methodology and providing insight into additional tests that can significantly challenge the achieved reliability. In this way, the method can produce an optimal selection of inputs and the order in which the tests are executed to maximize the software reliability. To illustrate this approach, a belief network is constructed for a software system that incorporates the expert opinion, expressed through probabilities of the relative quality of the elements of the software, and the potential effectiveness of the software tests.

Ziv and Richardson [Ziv and Richardson 1997] present an approach that allows developers' beliefs regarding software components to be modelled and updated directly. This approach is part of an overall strategy that calls for explicit modelling of software uncertainties using BN. The authors present several kinds of software uncertainty, how they may be modelled and how BN may be used to confirm, evaluate or predict software uncertainties. The case study concerns an existing software system under development at Beckman Instruments. The BN models, once built, may be used by developers and managers in future software understanding, evolution and maintenance activities. The Beckman study gives a feedback about the factors that may affect confidence.

Fenton and colleagues [Fenton et al. 2002] argue that quality control of software has not reached the same levels of sophistication as it has with traditional manufacturing because insufficient thought is being given to the methods of reasoning under uncertainty that are appropriate to this domain. They describe a large-scale Bayesian network built to overcome the difficulties that have so far been met in software quality control. This approach exploits a number of recent advances in tool support for constructing large networks. The authors describe how the network has been validated and illustrate the range of reasoning styles that can be modelled with this tool.

In spite of the Namin and Sridharan's plea for Bayesian reasoning methods [Namin and Sridharan 2010], the research on probabilistic inference in software testing and related domains such as software quality assessment is still in its infancy and is made of disparate tentative works on different subjects of the discipline. In particular, the usage of probabilistic inference as a support of test scheduling is put in place, in the author's best knowledge, only by Mirarab and Tahvildari [Mirarab and Tahvildari 2007]. Their approach is static (prioritization for regression testing) but they consider dynamic scheduling (incorporating as evidence the feedback from the test system for each test run) as one of the most important subject of their future research.

TROUBLESHOOTING AND PROBABILISTIC INFERENCE

In diagnosing defective systems, the primary goal is to isolate the faults that best explain the symptoms in the most efficient way. For complex systems, determining which components are causing troubles is not always straightforward and often prone to inaccuracies. Decision systems are useful in this context because they can model real world problems with high accuracy and can be designed in a transparent way, facilitating the coordination between experts and users. Troubleshooting system failures can be approached with several methods and techniques such as rule-based reasoning, case-based reasoning, neural networks, decision trees, probabilistic models and others.

The Bayesian Network approach to troubleshooting bestows several advantages. First of all, the BN inference results are probabilistic, allowing managing the uncertainty in the decision process and the intractability of exact inference. Moreover, they can be mathematically proven [Dechter 1996]. In addition, its results are knowledgeables, in contrast to other approaches, such as Neural Networks, which act as black boxes. Furthermore, the Bayesian Network approach does not present the combinatorial problems of other methods such as Decision Trees, and is more suitable for large systems. Last but not least, Bayesian Networks have already been proven to work in real world applications, such as a system developed by NASA for pilot-aircraft interaction [Cooper et al. 1998] and a network printer troubleshooting system developed by Hewlet Packard [Skaanning et al. 2000] [Jensen et al. 2001].

Generally speaking, a troubleshooting system is able to choose actions of two categories:

- *Passive* actions (information gathering, observation, non-intrusive testing…);
- *Active* actions (repair actions…).

Complex troubleshooting problems are often modelled using various simplifying assumptions and approximations in order to make them computationally feasible. Common basic assumptions include:

- *System Failure* - The system is assumed to be defective at the start of the troubleshooting process.
- *Single Fault* - There is exactly one component of the system that is faulty.
- *Component Faulty State* - Each component has exactly one faulty state.
- *Observable Components* - All components are observable with deterministic states.
- *Repair Action Success* - Repair actions on faulty components are always successful.
- *Unique Repair Action* - All repair actions uniquely correspond to specific faulty components.

These assumptions can be loosened or modified. Because Bayesian Networks are probabilistic, further assumptions can be made in order to reduce the number of parts under consideration to those with the highest probability of failure.

Any complex system can be defined as a set Cp of n components $\{\mathrm{Cp}_1, \mathrm{Cp}_2, \dots, \mathrm{Cp}_\mathrm{n}\}$, where any component Cp_i can be a potential source of the problem. In order to determine which Cp_i may be faulty, passive and active actions are performed. Passive actions, such as *Observing*, *Questioning* and *Testing*, gather information from the system without affecting the system itself and are required in order to understand the state of components. Active actions (*Repair*, *Solution*) are invoked on the basis of the results of the passive actions and impact the system by making changes. A *strategy* is a set of actions executed in a specific order. Generally speaking, actions have costs. Given a cost structure of the actions, the goal of a troubleshooting system is to find an optimal strategy for testing, i.e. a strategy having an Expected Minimal Cost (ECR).

Troubleshooting is *iterative* and *interactive*. It is done in a stepwise process, which continually integrates and applies new information, or evidence, in order to determine the next troubleshooting action. In the first step, the determination is done without information. Generally speaking, strategies begin with passive actions to determine the state of a component, and then enact any active action required if the component is deemed faulty. The information produced by each action is fed back into the system, a new determination is made and the strategy (next actions to be performed) is modified accordingly. In this way, the strategy is continually being updated and executed. A good strategy minimizes the time required to isolate and fix the faulty component or components (if the *Single Fault* assumption is abandoned).

The structure of any system to be diagnosed can be represented as a directed acyclic graph (DAG), which shows causal relationships between symptoms, components and troubleshooting actions. Causes and effects can be connected using two main models:

1. The Naive model that requires the Single Fault assumption, where effects are independent from each other. This model is used in simple situations but is not realistic for complex problems. Different causes are represented as states of the parent variable cause. A cause may have several effects and one effect to only one cause, leading to a non-realistic representation of real world problems.
2. The Causal Independence model allows for multiple faults within a single system. This model requires that the troubleshooting actions have effective relations to the faults, in contrast with the Naive model. Faults can be dependent or independent, and the actions themselves can be dependent (linked to more than one fault) or independent as well [Heckerman and Breese 1996] [Breese and Heckerman 1996]. The causal independence model is closer to reality of complex systems.

The goal of the troubleshooting process is to isolate and repair any faults within the system as efficiently as possible. In order to determine the most efficient set and order of actions, the cost and/or efficiency of each action must be evaluated (belief in the fitness of the action). A good troubleshooting algorithm should take into consideration a number of criteria including the probability of failure and the cost of repair.

By using a BN, these relationships are probabilistic rather than deterministic, allowing a greater simplification of the system. Stochastic methods are considered appropriate for failure detection and diagnosis of complex systems in cases where there is no complete knowledge of the system, i.e. the detection and diagnostic process is undertaken in presence of uncertainty and the evidence data domain is too large to be completely analysed [Nielsen et al. 2000].

Decision-theoretic troubleshooting was firstly extended to BNs by Heckerman and colleagues under the *Single Fault* assumption [Heckerman et al. 1994] [Heckerman et al. 1995]. They consider observations of a component after and before a repair as base observations. Costs of repair and observation are independent from previous repairs and observations. They propose three troubleshooting algorithm: (i) the first algorithm stops at the detection of the first fault and does not update the knowledge provided by of the test cycles; (ii) the second algorithm also follows the *Single Fault* assumption but updates the knowledge at each test cycle; (iii) the third algorithm outplays the *Single Fault* assumption.

Skaanning, Jensen and colleagues [Skaanning et al. 2000] [Jensen et al. 2001b] model the problem of troubleshooting printing systems under the *Single Fault* assumption. They present the SACSO system for troubleshooting of printing systems that represents the relationships between three types of variables: i) fault ii) action and iii) query for fault identifications. They propose a heuristic method based on a two step look-ahead analysis to obtain a quasi-optimal strategy. The method is based on the greedy algorithm proposed by Heckerman and colleagues [Heckerman et al. 1994] [Heckerman et al. 1995].

An alternative BN approach to diagnose complex system proposes the use of fault trees [Bobbio et al. 2001]. Fault trees, a well-known diagnostic technique, allow locating the faulty components by means of inquiries. Bobbio and colleagues propose the use of fault trees for system troubleshooting. Fault trees allow locating the faulty components by means of inquiries. The construction of the fault tree for a complex system proceeds in a top-down fashion, from events to their causes, following the system breakdown, until elements revealing faults of basic components are reached. The authors use fault trees to define the minimal cut set - minimal set of components that need to be all defective to cause the system failure. The fault tree is transformed into a Bayesian Network, which is able, starting from a failure evidence, to locate the component or set of components with the highest fault probability.

Wang and colleagues [Wang et al. 2006] seize an innovative approach to service level management for network enterprise systems by using integrated monitoring, diagnostics, and adaptation services in a service-oriented architecture. The autonomous diagnosis for troubleshooting of web service interruptions is based on BN models. In this paper, Wang and colleagues present a method for building the diagnostic models. In particular, they focus on two types of Bayesian network models of different structural complexity (two-layer vs. three – layer). The results show that the two-layer model outperforms the three-layer model. This challenges the common belief that adding unnecessary nodes in a Bayesian network and growing its structural complexity does not deteriorate performance. Hence such practice of building more complex models than necessary should be approached cautiously within the context of the applied domain.

In the domain of service composition testing, i.e. grey-box testing of services architecture, troubleshooting helps locating the faulty element that is the source of the failure. Troubleshooting of complex system with BN is henceforth a well-established discipline and practice and some remarkable realisations are presented in this section. BNs and probabilistic inference seems to be good candidates to manage the relationship between passed/failed test runs (passive actions in the troubleshooting language, they do not change the state of the system under test) and faulty/faultless SAUT elements. These elements are organized in a hierarchy and are structural – in the higher levels of the hierarchy, atomic or compound components of the service architecture – and functional – in the lowest two levels, component's required/provided interfaces and interface's operation interactions. The latter elements (interfaces and interactions) are functional because, by definition, we do not know where, inside the component, is located the software that implements them. A key enabler of troubleshooting is the SAUT construction model (see section 4, § 'The SAUT Construction model') that allows the association of random variables with the SAUT elements. On the other hand, identified passive actions (e.g. test cases), with known links with the SAUT elements, allow iterative and interactive evaluation of their *fitness* (fault exposing potential), i.e. their capability of provoking a failure that reveals a fault.

This work contributes to the research on the application of Bayesian reasoning methods to the "reliable and efficient software testing and program analysis" [Namin and Sridharan 2010] that is considered an "ideal research paradigm" and a promising future research thread on software testing. In fact, the author thinks that the problem of intelligent dynamic scheduling of test sessions can be posed and solved only through probabilistic inference, because it is too complex for exact inference.

6. THE SCHEDULER IMPLEMENTATION: THE PROBABILISTIC INFERENCE ENGINE

An Arithmetic Circuit (AC) that is obtained by transformation of a virtual Bayesian Network (vBN) supports the probabilistic inference engine that drives the scheduler. The vBN is built on the basis of the specific topology of the services architecture under test (SAUT Construction model) and of the test scenarios (Test Suite Definition model) and data (Test Suite data set). The reasons of this transformation method and of the final representation are essentially mastering the size and time complexity of the utilisation of the Bayesian network approach and are discussed theoretically in this section, with some experimental results presented in the next section 7.

The Scheduler initialization phase implements a M2M (model-to-model) transformation phase from the SAUT Construction model, the Test Suite Definition model and the Test Suite data set to a *virtual* Bayesian Network (vBN) model (passing through the constitution of a Test Scheduling Context model) and a compilation phase from the virtual Bayesian Network to the Arithmetic Circuit.

BUILDING THE TEST SCHEDULING CONTEXT MODEL AS A VIEW

As discussed in the previous section, the information contained in the SAUT Construction model, in the Test Suite Definition model and in the Test Suite data set is utilised to configure the Test Execution/Arbitration system as well as for the construction of the vBN. In fact, only part of this information is utilised for building the vBN, that is called the *Test Scheduling Context* (TSC) model. The UML representation of the TSC meta-model is presented in Figure 37 as a *view* of the SCA4SAUT/TSD/TS meta-model.

The TSC meta-model is a composite view of (i) the SAUT construction (SCA4SAUT) meta-model – involving the **SAUT**, **Actual Component**, ***Port*** (**Service**, **Reference**) elements, (ii) the TSD meta-model – involving the **Interaction Class** element, and (iii) the TS data model – involving the **Oracle Interaction Token**, **Test Sample** elements.

The Test Scheduling Context (TSC) meta-model is conceptually splitted in two parts. The first part allows the description of the structural and functional decomposition of the SAUT. Starting from the root **SAUT** element, the first level of the tree decomposition represents the SAUT composition, i.e. the SAUT **Actual Component**s. This first level represents a *structural* decomposition of the SAUT and the lowest possible level of granularity of the *structural coverage* of the SAUT. At this level, the potential structural coverage is complete. The second level includes all the **Reference**s and the **Service**s respectively declared and exposed by each **Actual Component**. This second level represents a *functional* decomposition step, whose potential coverage extent is complete at this granularity level. The decomposition is functional since it does not allow locating directly the component software parts of the **Actual Component** that implement a ***Port***, due to information hiding of the component implementations. The third level of decomposition is supplied by the TSD model and is always *functional* – it represents the **Interaction Class**es that are issued by ***Port***. Note that there are three categories of **Interaction Class**es: (i) {*operation*.input} issued by a **Reference**, (ii) {*operation*.output} issued by a **Service** and (iii) {operation.fault} issued by a **Service**. This information is supplied by the TSD model. For this level of granularity, the Test Scheduling Context model has the coverage extent of the corresponding TSD Model.

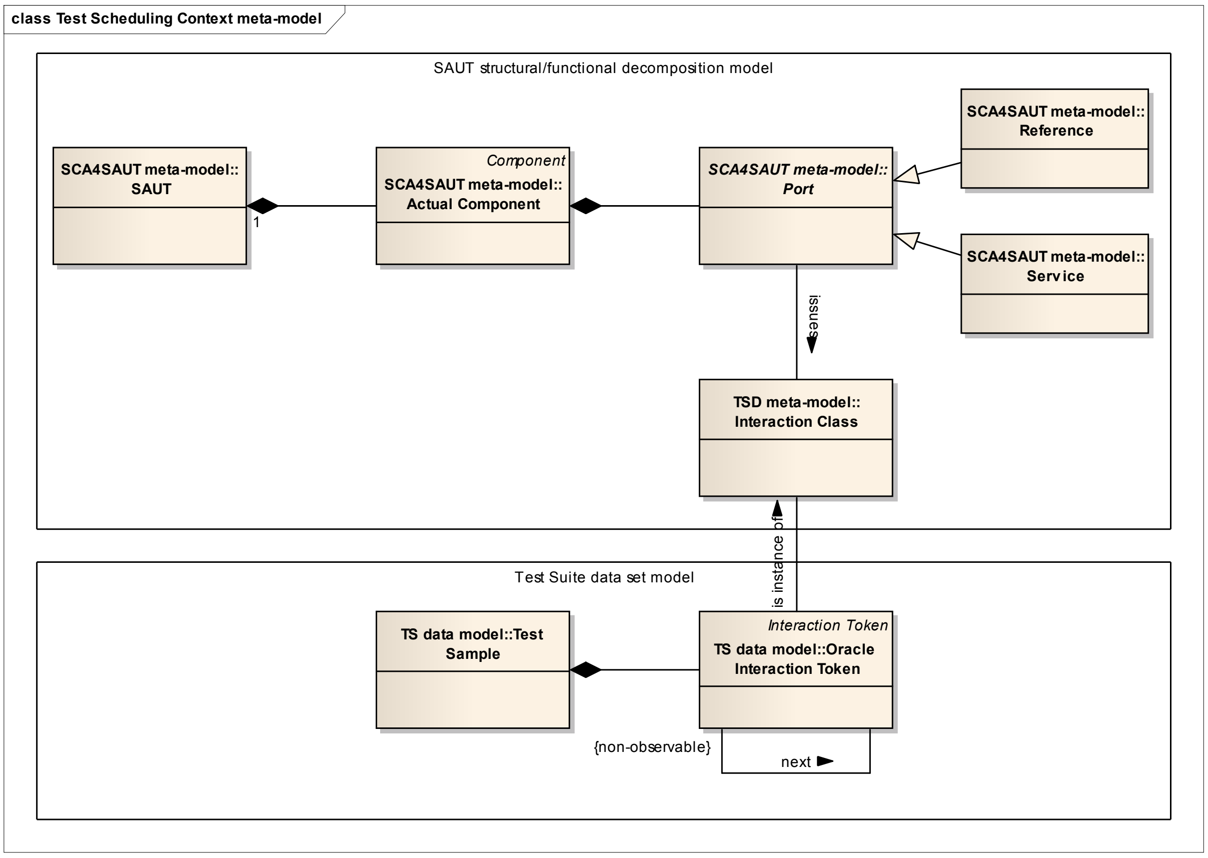

class Test Scheduling Context meta-model
SAUT structural/functional decomposition model
SCA4SAUT meta-model::SAUT
1
Component
SCA4SAUT meta-model::Actual Component
SCA4SAUT meta-model::Port
SCA4SAUT meta-model::Reference
SCA4SAUT meta-model::Service
issues
TSD meta-model::Interaction Class
is instance of
Test Suite data set model
TS data model::Test Sample
Interaction Token
TS data model::Oracle Interaction Token
{non-observable}
next

Figure 37. Test Scheduling Context meta-model.

The second part of the TSC meta-model (Test Suite data set model) allows the representation of the Test Suite. The model integrates a reduced view of the elements of the Test Suite data set – the **Test Sample**s and their **Oracle Interaction Token**s – that are shared with the Test System and that enable the data exchange in the schedule/execute/arbitrate cycle. The important information for the **Test Sample** is if it has been executed or not, and for each of its **Oracle Interaction Token**s is if it is *observable,* if it have been executed and the *local verdict*. The **next** link between **Oracle Interaction Token**s is the instantiation of the **next** association between the **Interaction Path Nodes** that are instantiated by these **Oracle Interaction Token**s (passing through the **Case Interaction Tokens**). The link is established in the TSC model only if the source **Oracle Interaction Token** (according to the **next** association) is non-observable (see below). Note that a local verdict is not necessarily available (value = *none*) even if the **Oracle Interaction Token** is observable, and its **Test Sample** has been executed (but not completely).

An example of TSC model is sketched in Figure 38. The example *SAUT* (**BankSystem**) is that detailed in the Snippet 1 and depicted in Figure 21. The *Interaction Class*es that are modelled are those involved in the **Wire_OK_SampleClass** depicted in Figure 23. The unique *Test Sample* (**Wire_OK_01**) is an instantiation of the **Wire_OK_SampleClass**. The *next* association between *Oracle Interaction Token*s is not modelled in this Test Scheduling Context, because, according to the Test System configuration, all the SAUT responses related to the *Test Sample* Oracle *Interaction Token*s are *observable* (see below).

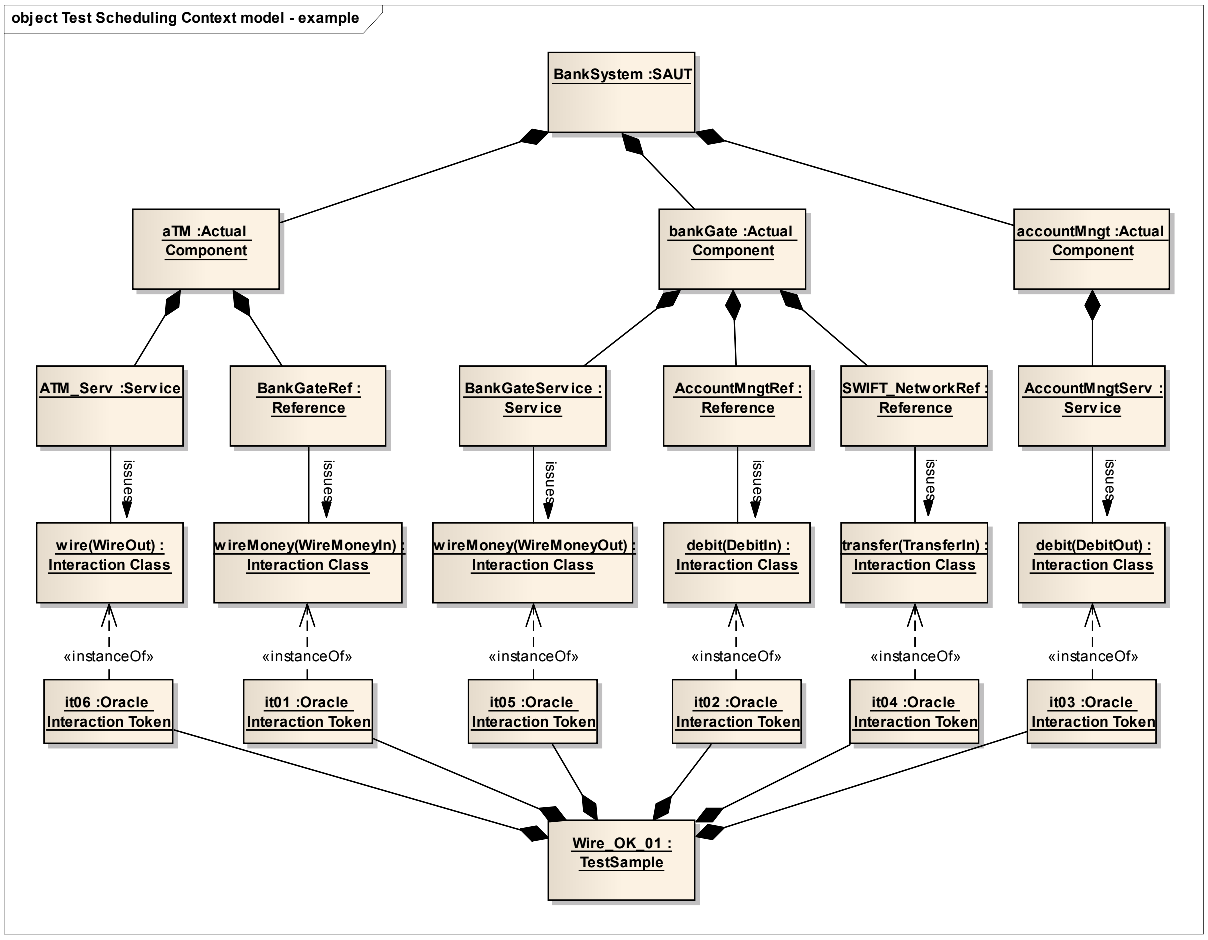

object Test Scheduling Context model - example
BankSystem :SAUT
aTM :Actual Component
bankGate :Actual Component
accountMngt :Actual Component
ATM_Serv :Service
BankGateRef : Reference
BankGateService : Service
AccountMngtRef : Reference
SWIFT_NetworkRef : Reference
AccountMngtServ : Service
issues
wire(WireOut) : Interaction Class
wireMoney(WireMoneyIn) : Interaction Class
wireMoney(WireMoneyOut) : Interaction Class
debit(DebitIn) : Interaction Class
transfer(TransferIn) : Interaction Class
debit(DebitOut) : Interaction Class
«instanceOf»
it06 :Oracle Interaction Token
it01 :Oracle Interaction Token
it05 :Oracle Interaction Token
it02 :Oracle Interaction Token
it04 :Oracle Interaction Token
it03 :Oracle Interaction Token
Wire_OK_01 : TestSample

Figure 38. Test Scheduling Context model – BankSystem example.

BUILDING THE VIRTUAL BAYESIAN NETWORK BY MODEL TRANSFORMATION

The vBN is an acyclic directed graph whose nodes are typed random variables and whose links represents stochastic dependence between the variables. The random variable types and their relationships of stochastic dependence are associated to the elements and relationships of the TSC meta-model. The Figure 39 depicts the vBN meta-model and the Table 9 details the correspondence between the TSC model and the vBN model.

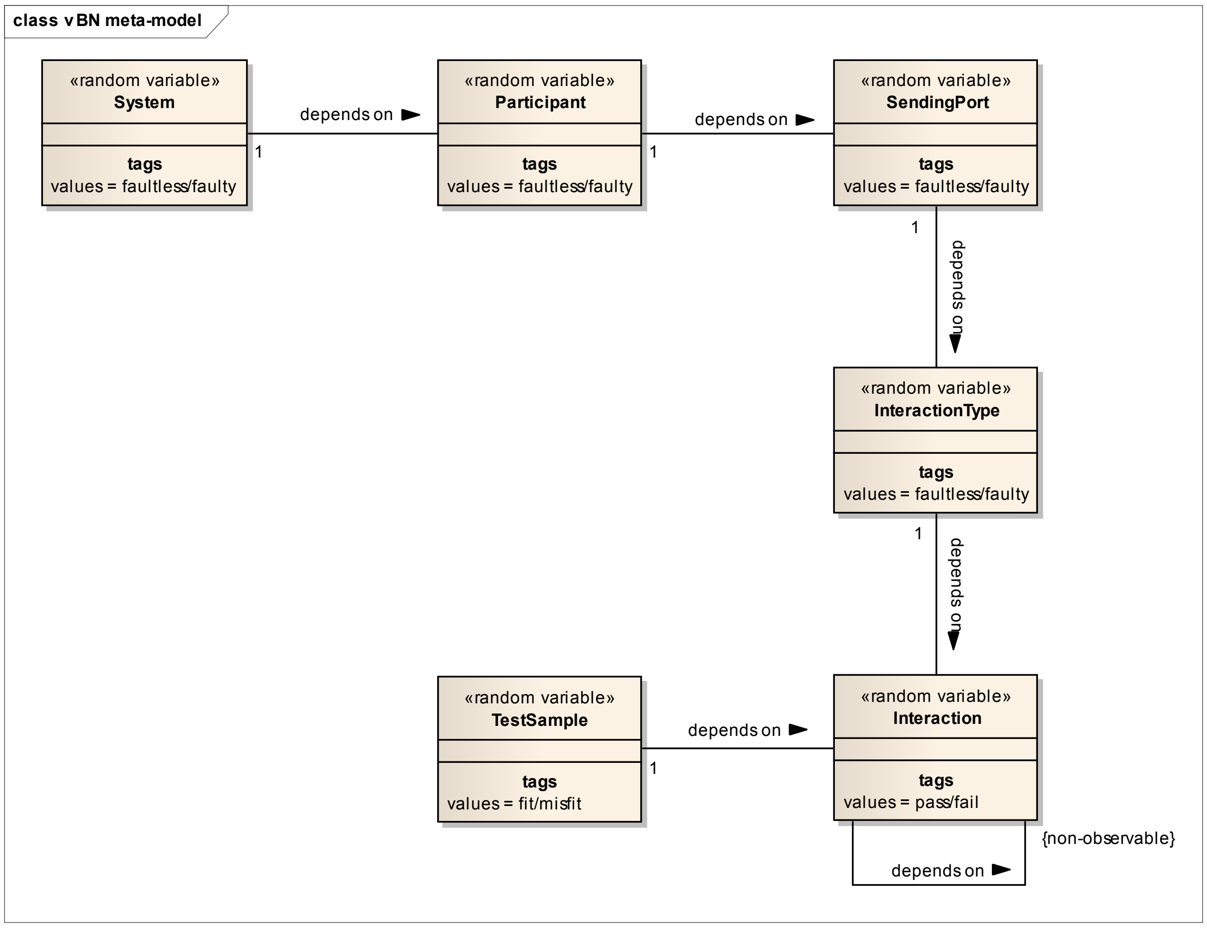

class vBN meta-model
«random variable» System
tags
values = faultless/faulty
depends on
1
«random variable» Participant
tags
values = faultless/faulty
depends on
1
«random variable» SendingPort
tags
values = faultless/faulty
1
depends on
«random variable» InteractionType
tags
values = faultless/faulty
1
depends on
«random variable» TestSample
tags
values = fit/misfit
depends on
1
«random variable» Interaction
tags
values = pass/fail
{non-observable}
depends on

Figure 39. vBN meta-model.

Note that the “canonical” BN *parent* relationship is the inverse of the *depends on* relationship. The *model mapping* from the TSC model to the vBN model (variables and relationships between variables) is detailed in the Table 9.

TSC elements and relationships	***vBN variables and dependences***
SAUT root element, *has* a universal identifier (**SAUT id**), *aggregates* (composition) **Actual Component**s.	**System** DAG bottom node, *denotes* the **SAUT** *fault index*, Boolean variable (*faultless* = 0 / *faulty* = 1), is identified by the corresponding **SAUT id**, *depends on* all the **Participant**s, is instantiated (*assumption*) to *faulty* in the initialization phase.
Actual Component SAUT structural element, *has* a universal identifier (**component id**), *is part of* a **SAUT**, *owns* ***Port****s*, *(declares* **References**, *exposes* **Services**).	**Participant** *denotes* the **Actual Component** *fault index*, Boolean variable (*faultless* = 0 / *faulty* = 1), *is identified by* the corresponding **component id**, *is parent of* **System**, *depends on* all the **SendingPort**s, corresponding to the ***Port****s* (**Reference**s, **Service**s) of its corresponding **Actual Component**, can be instantiated with a *belief* at any inference cycle.
Port **(Reference, Service)** SAUT functional element, *has* a universal identifier (**port id**), *is* a **Reference** of the **Actual Component** that *is in* a **Wire source** OR *is* a **Service** of the **Actual Component** that *is in* a **Wire target**, *issues* **Interaction Class**es.	**SendingPort** *denotes* the ***Port*** *fault index*, Boolean variable (*faultless* = 0 / *faulty* = 1), *is identified by* the corresponding **port id,** *is parent of* the **Participant** corresponding to the **Actual Component** in which its corresponding ***Port*** is specified, *depends on* all the **InteractionType**s, corresponding to the **Interaction Class**es that are issued by the corresponding **Port**, can be instantiated with a *belief* at any inference cycle.

Interaction Class SAUT functional element, *has* an universal identifier (**interaction class id**), *is issued by* a ***Port***, is of one of the three categories: 1. {*operation*.input} *issued by* a **Reference**, 2. {*operation*.output} *issued by* a **Service**, 3. {*operation*.fault} *issued by* a **Service**.	**InteractionType** *denotes* the interaction class *fault index*, Boolean variable (*faultless* = 0 / *faulty* = 1), *is identified by* the corresponding **interaction class id**, *is parent of* the **SendingPort** corresponding to the ***Port*** that *issues* the corresponding **Interaction Class**, *depends on* all the **Interaction**s corresponding to the **Oracle Interaction Token**s that *are instances of* the corresponding **Interaction Class**, can be instantiated with a *belief* at any inference cycle.
Oracle Interaction Token elementary oracle/outcome/local verdict *has* an universal identifier (**interaction token id**), *is an instance of* an **Interaction Class**, *belongs to* one and only one **Test Sample**, *represents* 1. an *oracle*, 2. the related outcome in a test session, 3. the *local verdict* associated to the outcome, *may be non-observable*, *may have* **previous** non observable **Oracle Interaction Token** of the same **Test Sample**, when the owner **Test Sample** is executed if it is executed (there is an outcome) its *local verdict* is arbitrated to *pass* or *fail*, otherwise (when the execution of the owner **Test Sample** stops before its elementary outcome) it is not arbitrated (local verdict = *none*).	**Interaction** DAG top node, *denotes* the *local verdict* associated with the response of the SAUT for the corresponding **Oracle Interaction Token**, Boolean variable (*pass* = 0 / *fail* = 1), *is identified by* the corresponding **interaction token id**, *is parent of* the **InteractionType** corresponding to the **Interaction Class** of the corresponding **Oracle Interaction Token**, *is parent of* the **TestSample** corresponding to the **Test Sample** that owns the corresponding **Oracle Interaction Token**, if non-observable, *is parent of* the **Interaction** corresponding to **next Oracle Interaction Token**, is initialised with a *prior probability* once when the owner **Test Sample** is executed, if the *local verdict* is *pass/fail*, it is instantiated with the corresponding *observation,* otherwise (the verdict is *none*, or possibly *inconclusive* or *error*) it is not instantiated.

Test Sample compound oracle/outcome/global verdict, *has* an universal identifier (**test sample id**), *is* an instance of a **Test Sample Class**, *owns* **Oracle Interaction Token**s, when it is executed, if all the owned **Oracle Interaction Token**s have an outcome and all the local verdicts are set to *pass*, the global verdict is set to *pass*, otherwise, if there at least one **Oracle Interaction Token** local verdict set to *fail*, the **Test Sample** global verdict is set to *fail*	**TestSample** DAG bottom variable, denotes the test sample *fitness*, Boolean variable (*misfit* = 0 / *fit* = 1), *is identified by* the corresponding **test sample id** *depends on* all the **Interaction**s corresponding to the **Oracle Interaction Token**s that are elements of the corresponding **Test Sample**, if the corresponding **Test Sample** has not been executed, it can be instantiated with a *belief*, when the corresponding **Test Sample** has been executed: if the *global verdict* is *fail*, the *fit* dependent probability is calculated to **1** (the **Test Sample** certainly *fits*), otherwise (the *global verdict* is *pass*) the *fit* dependent probability is calculated to **0** (the **Test Sample** certainly *misfits*).

Table 9: Correspondence between the TSC elements and relationships and the vBN random variables and dependences.

The vBN constructor of the Scheduler creates one and only one **System** random variable whose name is the **SAUT id** of the corresponding **SAUT**. The **System** variable is Boolean (*faultless* = 0 / *faulty* = 1) and represent the probability of being *faulty* of the **SAUT**. The unique **System** variable establishes the links between the different **Participant** variables of the vBN that are used by the vBN inference logic. These links are realised by the stochastic dependence on the vBN **Participant** variables that, when created (see below), become *parents* of the **System** variable.

For each TSC **Actual Component**, the vBN constructor creates a **Participant** stochastic Boolean variable (*faultless* = 0 / *faulty* = 1) whose name is the **component id** of the corresponding **Actual Component**. The **Participant** variable represents the probability of a SAUT **Actual Component** of being *faulty*. The vBN constructor establishes a *parent* link from the created **Participant** variable and the **System** variable, establishing a stochastic dependence of the latter on the former.

For each **Actual Component** / ***Port*** (**Reference** / **Service**) the vBN constructor creates (i) a **SendingPort** stochastic Boolean variable (*faultless* = 0 / *faulty* = 1) whose name is the **port id** of the corresponding ***Port*** and (ii) a *parent* link from the created **SendingPort** to the **Participant** variable that corresponds to the **Actual Component** that owns the ***Port***, establishing a stochastic dependence of the latter on the former. A **SendingPort** variable represents the probability of being *faulty* of the aforementioned ***Port***, i.e. that a fault is localised in the **Actual Component** implementation of the ***Port***.

For each **Actual Component** / ***Port*** / **Interaction Class**, the vBN constructor creates: (i) an **InteractionType** stochastic Boolean variable (*faultless* = 0 / *faulty* = 1) whose name is the **interaction class id** of the corresponding **Interaction Class** and (ii) a *parent* link from the created variable to the **SendingPort** variable that corresponds to the ***Port*** issuing the **Interaction Class**. The *parent* link from the **Participant** to the **SendingPort** variable establishes a stochastic dependence of the latter on the former. The **InteractionType** variable is used to localise the failure on the issuance of a specific *message type*.

For each **Test Sample** of the TS data set, the vBN constructor creates a **TestSample** stochastic Boolean variable (*misfit* = 0 / *fit* = 1) whose name is the **test sample id** of the corresponding **Test Sample**. The **TestSample** variable represents the probability distribution of the **Test Sample** to "fit" the search for failures and the localisation of faulty elements, on the basis of specified criteria (see below).

For each **Oracle Interaction Token** of each **Test Sample**, the vBN constructor creates an **Interaction** stochastic Boolean variable (*pass* = 0 / *fail* = 1) whose name is the **interaction token id** of the corresponding **Oracle Interaction Token**. The vBN constructor creates: (i) a *parent* link from the created **Interaction** to the **InteractionType** that corresponds to the **Interaction Class** of the **Oracle Interaction Token**, establishing a stochastic dependence of the latter on the former; (ii) a *parent* link from the created **Interaction** to the **TestSample** that corresponds to the **Test Sample** that owns the corresponding **Oracle Interaction Token**; (iii) if the created **Interaction** is *non-observable*, a *parent* link from it to the **Interaction** that corresponds to the **next Oracle Interaction Token**, if any.

The vBN model corresponding to the TSC model depicted in Figure 38 is presented in Figure 40. The links between variables are the canonical BN *parent* link.

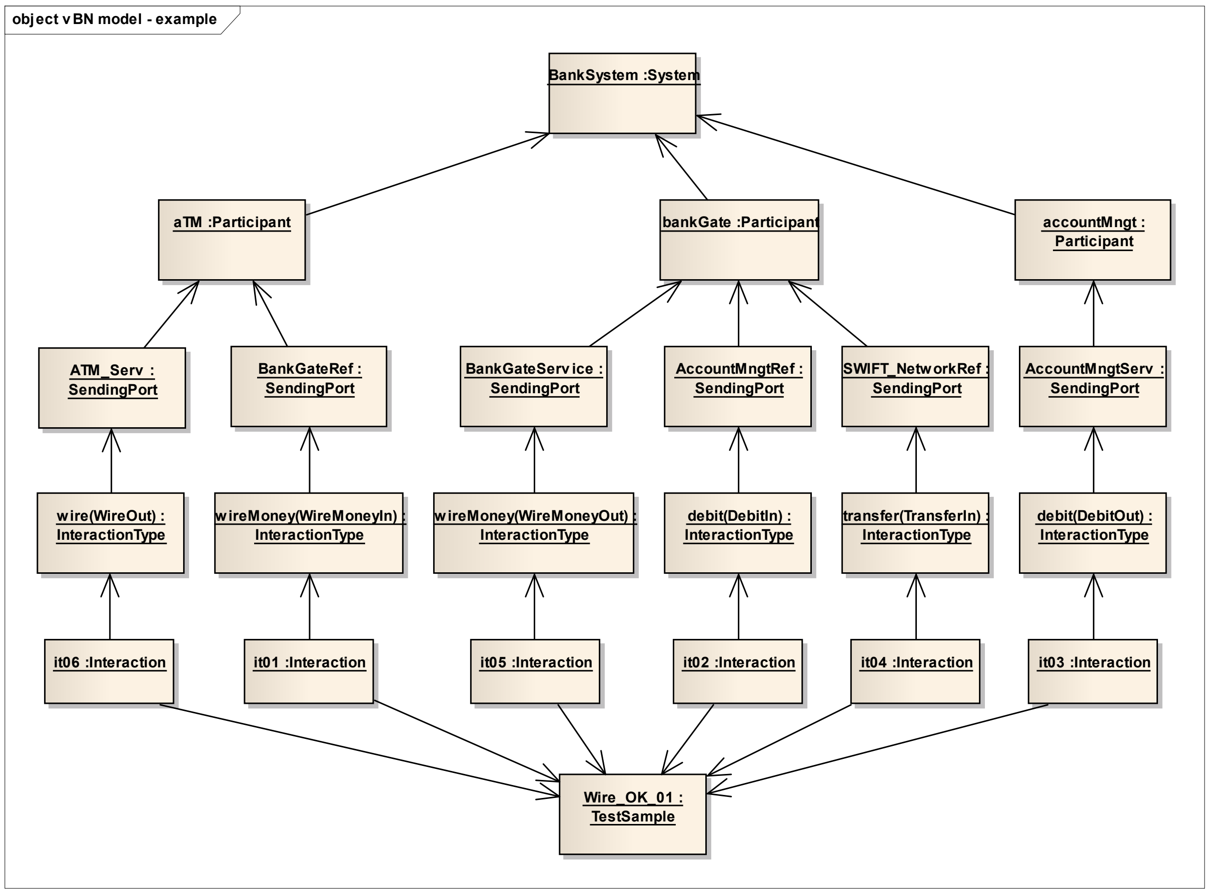

object vBN model - example
BankSystem :System
aTM :Participant
bankGate :Participant
accountMngt : Participant
ATM_Serv : SendingPort
BankGateRef : SendingPort
BankGateService : SendingPort
AccountMngtRef : SendingPort
SWIFT_NetworkRef : SendingPort
AccountMngtServ : SendingPort
wire(WireOut) : InteractionType
wireMoney(WireMoneyIn) : InteractionType
wireMoney(WireMoneyOut) : InteractionType
debit(DebitIn) : InteractionType
transfer(TransferIn) : InteractionType
debit(DebitOut) : InteractionType
it06 :Interaction
it01 :Interaction
it05 :Interaction
it02 :Interaction
it04 :Interaction
it03 :Interaction
Wire_OK_01 : TestSample

Figure 40. vBN model – BankSystem example.

In the test run cycle, the Scheduler supplies a **test sample id** to the Executor/Arbiter (through the Runner). The Executor/Arbiter executes/arbitrates the corresponding **Test Sample** and returns a *global verdict data structure* whose identifier is the **test sample id** (see section 4, § 'The test system, the test run and the test verdict').

The vBN considers each **Oracle Interaction Token** *pass/fail verdict* of the executed **Test Sample** as an *observation* (*evidence* realisation) that instantiates the **interaction token id Interaction** variable. The **Interaction** variables corresponding to the non-arbitrated **Oracle Interaction Tokens** (verdict = *none*) – because either of the halting of the **Test Sample** execution/arbitration cycle or of the lack of observers in the Test System – are not instantiated. Filling the vBN with the evidence realisations triggers the vBN inference step that updates the probability distribution (*fitness*) of the **test sample id TestSample** variable (in fact to **1** if fail, to **0** if pass) and of all the **TestSample** variables corresponding to the still not executed **Test Sample**. The Scheduler exploits this information in order to choose, on the basis of some criteria that can be indicated through scheduling *policies* and *directives,* the next **test sample id** to be supplied to the Executor/Arbiter (see the next section 7).

The vBN constructor creates a *parent* link between **Interaction**s that are *parents* of the same **TestSample**, on the basis of the **next** relationship between the corresponding **Oracle Interaction Token**s, which is derived from the **next** relationships between the **Interaction Path Node**s in the **Test Sample Class** that classifies the corresponding **Test Sample**. A *parent* link is established for each *non-observable* **Interaction** to the Interaction corresponding to the **next Oracle Interaction Token** owned by the same **Test Sample**. Those *non-observable* **Interaction**s correspond to **Oracle Interaction Token**s that are not monitored by the Test System. Hence, the probability distributions of each **Interaction** variable is: (i) either an a priori probability distribution if the **Interaction** has no previous (non-observable) **Interaction**s, or (ii) a conditional probability table that represents the stochastic dependence of this **Interaction** on the previous (non-observable) ones. This kind of probabilistic inference is intended to manage the concrete and difficult situation of *fault propagation*: the outcome corresponding to an **Oracle Interaction Token** doesn't match the oracle, and the conclusion could be the *fail* verdict and that the chain **InteractionType** / **SendingPort** / **Participant** is *faulty*, but in fact the preceding message addressed to that **Actual Component** and corresponding to a previous non-observable **Oracle Interaction Token** *failed,* whereas the "reaction" of the **Participant** to this message was correct. Another "generic" example of vBN is presented in Figure 41. Note the probabilistic dependence that is derived by the next link: I_5 is observable, but I_2, I_4, I_1 are not, and the evidence of I_5 allows inferring the probability distribution of the preceding **Interaction**s.

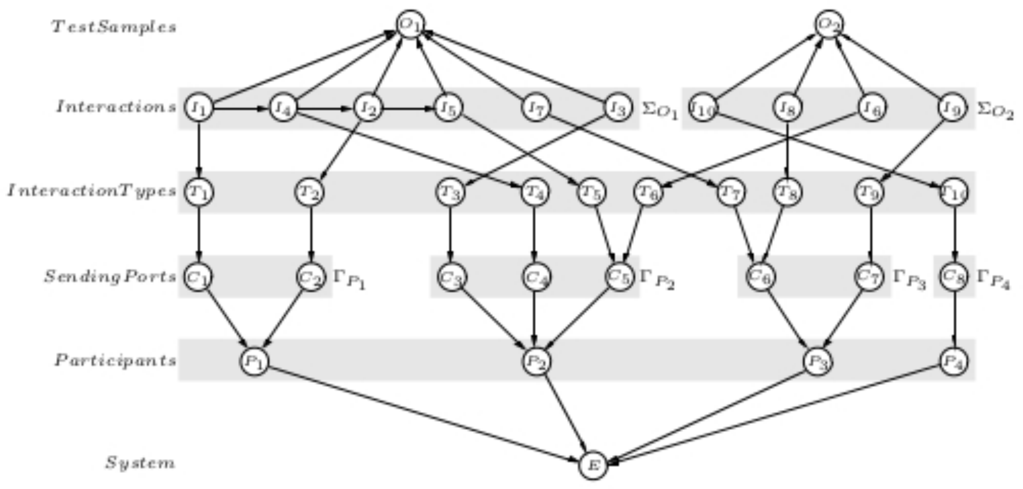


Figure 41: Graphical representation of an example of virtual Bayesian Network.

The **System** variable is instantiated to *faulty* (**1**) in the initialization phase in order to represent the "classical" troubleshooting hypothesis that the SAUT is *faulty*.

An appropriate model of the problem domain is a crucial element for the success of the probabilistic inference application. This work proposes a simple but deep model of the problem domain, the grey-box functional conformance testing of distributed services architectures, that is able to take into account both the fault-exposing potential of the test cases and the test coverage of structural and functional decompositions of services architecture at the lowest possible level of granularity, given the black-box/grey-box stance of service testing.

BUILDING THE ARITHMETIC CIRCUIT BY COMPILATION

The first steps of the Scheduler initialization phase end with the construction of the vBN that is in fact a model of a BN (in the sense that the representation does not correspond with any optimized representation of "executable" BNs). The vBN model is stored in a light XML representation. Figure 41 sketches an example of vBN .

The classical Bayesian inference approach reaches its limits in size and computation speed very quickly with the increase of the number of: (i) **Participant**s, (ii) **SendingPort**s, (iii) **InteractionType**s, (iv) **Interaction**s and (v) **TestSample**s. The proposal of this research is to "compile" the classical representation of the Bayesian Network in a more compact structure (the Arithmetic Circuit), adapted to more efficient inference computation. A concise description of an original model-driven method for compiling a BN into an AC follows.

As it was explained in chapter 4 a BN uses is a graphical representation of a joint probability distribution that can be viewed as a MLF. The AC is a factorized version of the MLF that avoids redundancies like repeating sub-circuits. A MLF is the sum of an exponential number of terms where each term is an instantiation of all network variables multiplied by the probability parameters expressing their mutual dependences.

Returning to Figure 41, it is important to highlight that the **TestSample**, **InteractionType**, **SendingPort**, **Participant** and **System** BN nodes are OR nodes (they are instantiated to the value 1 if at least one of the parent nodes is instantiated to 1)[46]. This implies that the probability tables of the dependences contain values equal to 1 and 0. Consequently, some of the terms of the MLF vanish due to a value equal to 0.

After this initial simplification it is possible to observe that all remaining terms express all possible instantiation of the observed **Interaction**s, the input information, and the consequences on the state of the rest of the network variables.

$$f(V) = \{\lambda_{\bar{m}_0}\lambda_{\bar{m}_1} \dots \lambda_{\bar{m}_n}\}\{\lambda_{\bar{g}_i} \dots \lambda_{\bar{g}_v}\}\{\theta_{\bar{m}_0}\theta_{\bar{m}_1} \dots \theta_{\bar{m}_n}\} + \dots + \{\lambda_{m_0}\lambda_{m_1} \dots \lambda_{m_n}\}\{\lambda_{g_i} \dots \lambda_{g_v}\}\{\theta_{m_0}\theta_{m_1} \dots \theta_{m_n}\}$$

Each term of the function can clearly be organized in three groups of variables:

$\{\lambda_{m_i}\ ;\ \lambda_{\bar{m}_i}\}$ - **Interaction** evidence indicators that represent the two possible instantiations of **Interaction** (expressing the verdicts *pass* or *fail*).

$\{\lambda_{g_i\ i}\ ;\ \lambda_{\bar{g}_i}\}$ - Evidence indicators for the other variables (**System**, **Participant**s, **SendingPort**s, **InteractionType**s and **TestSample**s). They also represent the two possible state of the related Bayesian Network variable.

θ_{m_i} – Network parameter consistent with the state of the interaction m_i and the topology of the vBN. Instead of the network parameters of the other BN variables, those network parameters are not removed during the simplification because their value is different from 0 and 1. For each interaction, the network parameter can belong to one of two categories:

- $\theta(m_i|\boldsymbol{U}) - \boldsymbol{U}$ being the state of all parents node, in the case of non-observed **Interaction**s the values of the **Interaction** instantiations influence the values of the preceding non-observed **Interaction**s.
- $\theta(m_i)$ - In the case of observed **Interaction**s the value is defined by the expert as an a priori opinion on the state probability of failure of the **Interaction**s.

The morphology of the MLF can be explained by transition properties of the BN. In fact, since the state of the **System** depends of the state of the **Participant**, the state of the **Participant** depends of the state of the **SendingPort** and so on (see Figure 41), the state of any element of the SAUT can be determined by the state of the cluster of **Interaction**s that are linked to it.

For the following explanations, the use of the Binary Decision Diagram (BDD) will allow a much easier understanding of the topology of the AC.

[46] In section on future works the limitation of such approach is explained and a possible improvement using Noisy-OR nodes is proposed [Pearl 1988].

Taking in consideration everything explained previously, it is possible to represent the instantiation of an **Interaction** variable of the MLF by the following Figure 42.

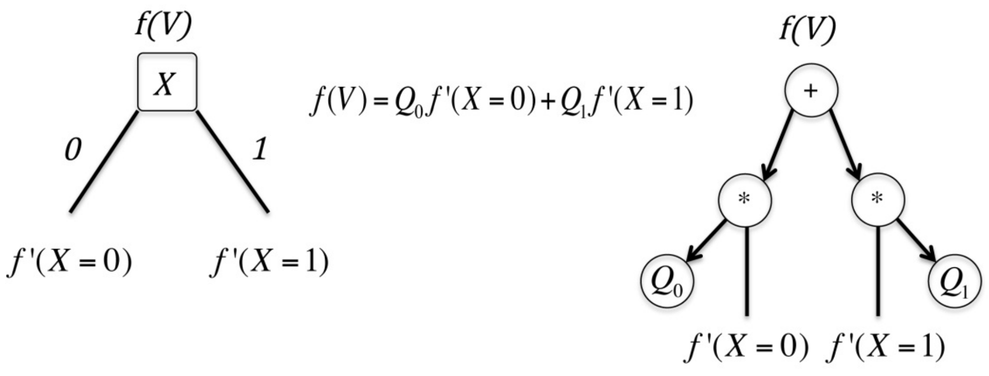


Figure 42: CA representation of a BDD (instantiation of X).

Here the instantiation of the network variable X to the value 1 implies that the evidence indicator $\bar{x}$ equals 0. When solving the MLF with the value $\bar{x}$ = 0 some terms of the function vanish, returning a shorter MLF. The same happens with the instantiation of X to the value 0, except that it is the evidence indicator x that equals 0.

Looking deeper, by resolving the MLF with the instantiation of a BN variable, a set of evidence indicators, common to all the terms of the remaining function, can be factorized. Here they are symbolized by Q.

$$f(X = 1) = Q_{x=1} f'(X = 1)$$

Q_i is a set of evidence indicators of the MLF, which nodes can be inserted in the Arithmetic Circuit with a multiplying node like in Figure 42.

f' is a simplified function, in which all evidence indicators that are common to all terms are moved in Q_i by the factorization.

Later it will be explained that the AC location where to insert the Q evidence indicator nodes depends on different factors and will influence the size of the AC. For clarity, we will consider for now that the Q nodes are inserted at the root of the sub-circuit as shown in Figure 42.

Using BDD representation (each path from the root to any leaf of the BDD is a term of the MLF) it is achievable to draw the coarse representation of the AC regardless of the **Interaction** order and without worrying about redundancies.

One of the purposes of the AC is to avoid the redundancies of calculation represented by sub-circuits. When two simplified MLFs are equal, only one sub-circuit needs to be built. For instance, the two paths of the circuit presented in Figure 43 are merged together.

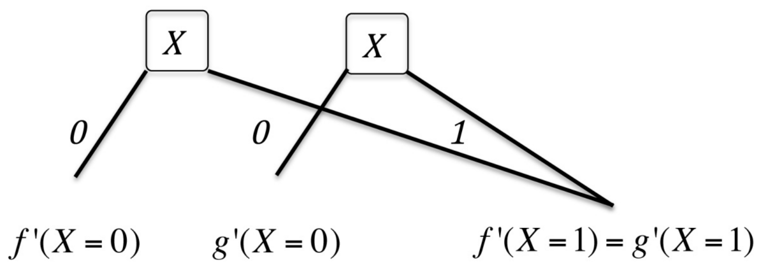


Figure 43: Merging two paths in a sub-circuit.

One of the objectives of this research is to minimize the size of the AC by finding an order of **Interaction**s that maximizes the redundancies. There are two easy conclusions that can be drawn at this point:

1. For two simplified functions to be equal and cause a merging of two circuit paths they need to contain the same terms, which implies that the same Bayesian Network variables must have been eliminated (regardless of their instantiation value) in the paths before the merging. Since all MLF variables depend on the instantiation of a cluster of **Interaction**s, it is logical to deduce that whatever path is followed on the AC the same cluster of **Interaction**s must be instantiated so that paths can merge. A global order of **Interaction**s to instantiate must be common to each circuit path.
2. When choosing the global order it is not interesting to take sequentially **Interaction**s belonging to different clusters since it only delays the elimination of the BN variables and consequently the merging of paths. When choosing an **Interaction** to instantiate you actually choose a cluster, instantiate all **Interaction**s and restart with another cluster.

Splitting a cluster of **Interaction**s means that in the middle of instantiating the **Interaction**s of the cluster other **Interaction**s from another cluster are instantiated. The only moment where it is useful to do that is when the second cluster eliminates a variable and allows path merging. Clusters related to the SAUT component variables include normally other clusters, for instance the cluster of the **Participant** A includes the clusters of all **SendingPorts** related to **Participant** A. Splitting the super-clusters delays only the instantiation of the corresponding BN variable.

The proposed solution is a *recursive dynamic program*: first you select the order between the super-clusters and then you select the order of the sub-clusters. This operation is repeated recursively until there are no more sub-clusters. The implementation of this algorithm can be compared to the creation of an oriented lattice where the nodes represent a set of instantiated **Interaction**s. The nodes are linked to a value that is an upper bound of the sub-circuit size and to an ordered list of **Interaction**s. Each lattice edge represents the choice of a cluster and, consequently, a partial order of instantiated **Interaction**s. Each edge bears 2 values that represent a lower bound and upper bound of the sub-circuit size that is specific to the **Interaction**s partial order.

The calculation of those values will be explained below, but they represent an ideal size of the sub-circuit (lower bound) and a boundary not to cross (upper bound) when trying to find the smallest sub-circuit. The process works recursively: with any set of **Interaction**s we first select the enclosing super-cluster. Those clusters must not be mutually included. The root node represents the initial state where none of the cluster **Interaction**s is instantiated and whose surface size equals 0.

The lattice is traversed breadth-first. A traversed node can be processed once all paths linking to it have been checked and it remains at least one path. As soon as it is the case for a node, we create one edge per remaining cluster that still contains non-instantiated **Interaction**s, and evaluate the edge lower bound and upper bound. The edges are created in an order such that the clusters with the largest number of non-instantiated **Interaction**s are evaluated first. When instantiating an **Interaction**, the size of the overlapping clusters containing the **Interaction** is reduced causing some clusters to end up included in other previously overlapping clusters.

Absorbed clusters are then processed recursively. Nodes are needed for each edge. If the node does not yet exist, it is created and the upper-bound value M is inserted as the node value SumM. When a node already exists we compare the edge upper-bound value (M) to the node value (SumM). The node value is replaced by the lower of the two values.

When all possible nodes have been processed then the algorithm looks for the non-checked edge with the lowest upper bound M.

To test an edge, the lower-bound value of the edge (m) must be less than the node value SumM otherwise the edge is removed. If m is lower that SumM then we recursively apply the entire algorithm to the set of **Interaction**s representing the cluster of the edge. Descending recursively causes re-evaluation of the upper bound and the lower bound of the edge, reducing the range between them and reducing the number of possible combinations for the partial order of the **Interaction**s in the cluster. When the recursive application ends, we obtain an exact value for the node and a total order of **Interaction**s for the cluster.

An example is sketched in Figure 44, where:

1 - the edge with the lowest upper bound is the edge *C* going from *AB* to *ABC*.

2 – after recursive testing the best value for instantiating the cluster *C* is v_c.

3 – the next best cluster to process recursively is B from AC to ABC the value v_B recovered is lower than v_C. The new value of the node ABC is v_B.

4 – the last edge A has a lower bound that is superior to v_B, the edge is removed. All paths going to ABC are tested the node ABC can be processed creating the new edges and evaluating them. The best partial order of interaction is the one provided by AC concatenated with the best ordered solution instantiating the cluster B.

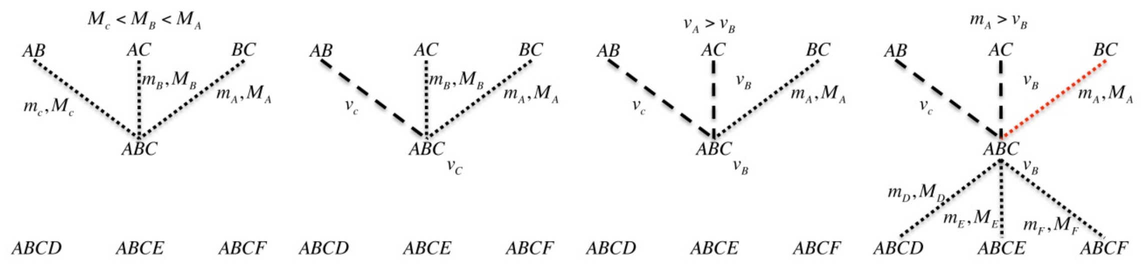


Figure 44: Checking the edges leading to the lattice node ABC.

The recursive action on an edge must respect the boundary imposed on the node value during the previous iterations to avoid processing useless cluster combinations. At the end, if the value obtained from the recursive processing is inferior to the node value, the node has a new order and a new value. When all paths ending on a same node have been removed or checked the node has an accurate value of the size of the sub-circuit for a specific order of **Interaction**s to instantiate. Then the node can be processed. At the end of the algorithm, the last node of the lattice returns the value of the size of the sub-circuit corresponding to a specific order of **Interaction** variable instantiations.

As shown in Figure 42, every node of the BDD corresponds to three nodes in the AC. So, after instantiating a group in a specific order, every layer of the BDD informs on the number of different function states after each **Interaction** instantiation. The layer width takes into account all of the instantiated **Interaction**s and the modification of the number of the unique MLF states are tracked. The number of leaves of the diagram represents the number of unique MLF states for the future selection of **Interaction,** i.e. each path leads to a leaf that reveal the information of the element not fully investigated. The elements can be faulty or unknown. Once all the instances belonging to a cluster of an element are instantiated the state of the element is known and does not influence the result of the instantiation of the remaining **Interaction**s. This fact allows saying that for any cluster of **Interaction**s, regardless of the order in which they are instantiated, we will have the same number of leaves. The layer width is always the same for a specific group of instantiated **Interaction**s regardless of their relation to each other and regardless of the order of instantiation, because solving the MLF for a group of **Interaction**s doesn't depend of the **Interaction** order.

The layer width is used to calculate the upper bound and the lower bound related to the selection of a cluster. The upper bound is the number of nodes for a random selected order of **Interaction**s of the cluster. It is obtained by the sum of the widths after each instantiation of **Interaction**s. The lower bound is an approximation of the minimum value of the size of the sub-circuit. When calculating the upper bound, the width (w_l) of the last instantiation is obtained and this value is used to calculate the lower bound. The width before instantiation of the cluster is used to calculate a lower estimation for the instantiation of the cluster. The last value that is needed to obtain the lower bound is the minimal width w_m. It represents the minimum number of MLF states that ideally would be present if all possible BN variables were instantiated.

Then the *transition surface,* which is the number of nodes that are needed when passing from the minimum layer width to the final width is calculated. Its equation is $S_t = \sum_{(\frac{w_l}{2^{n+1}} < w_m)} \frac{w_l}{2^{n+1}}$

The height n of the transition surface is obtained through the equation below:

$$\frac{w_l}{2^{n+1}} < w_m$$

Figure 45 shows how the lower bound becomes more and more accurate when iterating recursively.

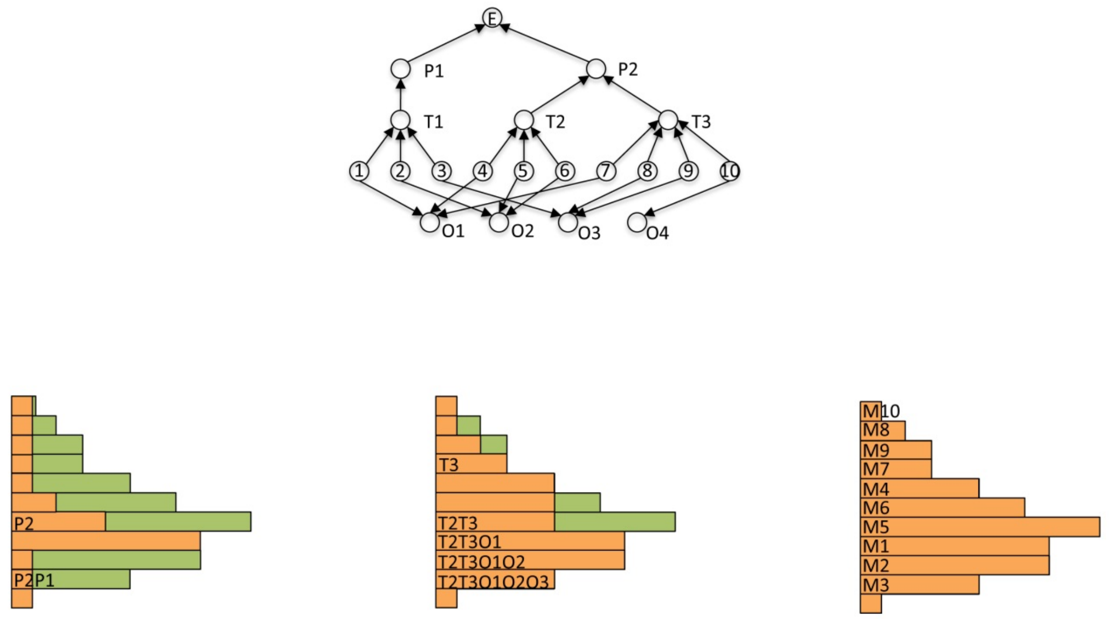


Figure 45: Example of the evaluation of the lower bound when iterate recursively.

Figure 46 gives illustrates an example of the lower bound calculation. Looking to the figure it is possible to distinguish three different sections of the orange surface when instantiating the cluster C:

- Surface 1 is function of the previous width.
- Surface 3 is the transition surface.
- Surface 2 is the minimum strip which is the $S_2 = w_m * 3 * (|m_c| - n - 1)$ m_c is the list of non-instantiated **Interaction**s of C.

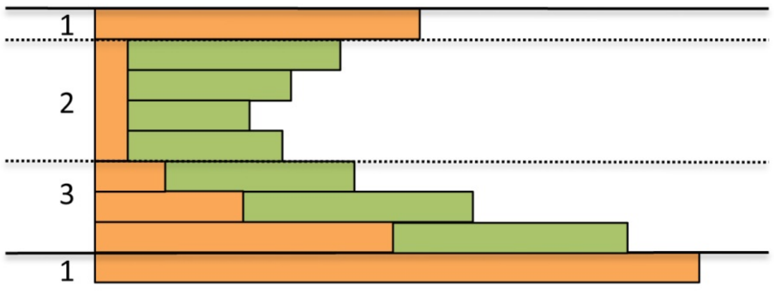


Figure 46: Lower bound calculation.

Roughly speaking, the idea behind the lower bound is to imagine the perfect situation where all the possible merging are done at the beginning of the cluster instantiation. This returns a number of MLF states that is inferior or equal to the actual circuit.

Building the AC with a given order of **Interaction**s is fairly simple. Figure 42 shows the mapping between the BDD and the AC, when instantiating an **Interaction**. Creating the AC consists in repeating the operation respecting the **Interaction** order and keeping track of the sub-circuit to avoid redundancies. But still there are a few more optimizations that can be provided to improve inference time.

Even though the previous algorithm avoids redundancy between the simplified MLFs, the same product of the Q evidence indicators can be performed several times. The algorithm looks at all multiplication nodes and counts the occurrences for the different grouping of the evidences indicators. Then for every repeated sub-circuit a unique instance is kept and the other nodes are linked to it. For every group of evidence indicators the number of nodes passes from m * s to m + s, where m is the number of the occurrences of the multiplication for the giving group and s the size of the group.

The second optimization is related to the place where the Q evidence indicators must be inserted. It was explained above that as soon as the evidence indicators could be factorized then their nodes should be inserted. It is true for the evidence indicators expressing non-faulty SAUT elements, but for the other evidence indicators, the nodes can be inserted at the same layer where there opposite are inserted ($\overline{m}$ for m). It does not change the topology of the circuit and doesn't change the method described above.

For every evidence indicator the number of links that are needed when inserting the nodes at the root of the sub-circuit is computed and compared with the number of links that are needed when inserting the nodes at the layer where the opposite evidence is inserted. The nodes should be inserted where the minimum links are required.

The first step of the construction of the Arithmetic Circuit by compilation of the virtual Bayesian Network performs the search for the best sequence of **Interaction**s. This first step is essential because the size of the Arithmetic Circuit depends heavily on it. The heuristics that is used in this first step is evaluated through experimental results presented in the section 7. Other heuristics will be investigated in the future (see the section of future works). The second step uses the **Interaction**s sequence to build a compact Arithmetic Circuit without sub-circuit redundancies. At the end of the initialisation phase, the Scheduler is equipped with an Arithmetic Circuit that allows it to behave as a Bayesian agent.

7. THE SCHEDULER BEHAVIOUR: DESCRIPTION AND EXPERIMENTAL RESULTS

The inf4sat algorithm of inference by compilation presented in the preceding section has been designed and implemented in compliance with some requirements issued from its utilisation as an inference engine for a scheduler of functional conformance test sessions on services architectures. The requirements are resumed below.

The first requirement is that the inf4sat algorithm shall be capable of generating and processing inference engines (ACs) able to schedule test sessions on large SAUT with big Test Suites. In other terms, compared with other existing solutions, the inf4sat algorithm shall be able to push further the boundaries of tractability.

The second requirement is that the AC compilation and the AC execution shall be asynchronous and that the inference engine, once compiled and generated (the AC Image, see section 4) shall be reusable without recompilation. A peculiarity of software and service testing is that the same Test Suite is executed on the same SAUT several times, in development (test-driven development) and maintenance (re-testing and regression testing).

The third requirement is that the each inference step shall cover all the SAUT and Test Suite elements, i.e. shall impact all the variables of the vBN/AC, re-computing the fault proneness of the SAUT structural and functional elements and the fault-exposing potential of each test sample. The global coverage of the inference enables its use in the formulation of generation requests by the Runner towards the Generator (see the section on future works.

The fourth requirement is that the inference step shall be fast, compared to the other existing solutions. A faster inference enables shorter and frequently repeated test sessions, which are needed by the Continuous Integration Testing approach.

This section firstly illustrates the concrete behaviour of the inference engine in the schedule/execute/arbitrate cycle. In the second part, some experimental results and comparisons with the other approaches are presented, in relationship with the requirements stated above.

SCHEDULER BEHAVIOUR DRIVEN BY THE INFERENCE ENGINE

The inference engine is an internal module of the Scheduler. To each execute/arbitrate cycle (with test samples as inputs and test verdicts as outputs) corresponds a schedule inference cycle (with test verdicts as inputs and test samples as outputs). The Scheduler manages its internal inference engine by setting prior probabilities on its DAG top variables (**Interaction**s) and assumptions/beliefs/observations (evidence realisations) on the other variables:

- the **System** bottom variable is *instantiated* with the *faulty assumption* before the first inference cycle – this behaviour implements the classic troubleshooting approach (see section 5, § 'Troubleshooting and probabilistic inference');
- the **Participant** variables (*is parent of* **System** - corresponding to SAUT structural element **Actual Component**) can be *instantiated* with a *faulty belief* at any inference cycle – there is no necessary initial or default instantiation;

- the **SendingPort** variables (*is parent of* **Participant** - corresponding to SAUT functional elements **Port**) can be *instantiated* with a *faulty belief* at any inference cycle - there is no necessary initial/default instantiation;
- the **InteractionType** intermediate variables (*is parent of* **SendingPort** - corresponding to TSD functional elements **Interaction Class**) can be *instantiated* with a *faulty belief* at any inference cycle - there is no necessary initial/default instantiation;
- the **Interaction** top variables (*is parent of* **InteractionType** and **TestSample** - corresponding to elementary oracles/outcomes) are initialised with a prior probability – the prior *fail* probability influences the fitness of the child **TestSample** – there is a default initialisation of the prior probabilities; these variables are instantiated with *pass/fail observations* following the feedback of the execute/arbitrate cycle that has observed their actual outcome and evaluated their local verdict;
- the **TestSample** bottom variables – before the execution of the corresponding test sample, they can be instantiated at any moment with a *fit belief* – there is not initial/default instantiation; at the beginning, and at any cycle, the *fit* belief distribution can be used in order to enter a statically computed *priority* following a number of criteria (examples of static prioritisation techniques for regression testing can be found in section 3, § 'Test case prioritisation'). When the corresponding **Test Sample** is executed, if at least one of the local verdicts is *fail*, then the inference calculates the value **1** for the **TestSample** *fit* dependent probability, otherwise (all the local verdicts are *pass*) the inference calculates the value **0** for the **TestSample** *fit* dependent probability.

As anticipated in the sections above, the collection of the evidence indicators $\boldsymbol{e}$ is initialised with the hypothesis that the **System** = *faulty*. In a test run cycle, the Scheduler selects one or more test samples that it suggests for execution, on the basis of a policy that takes into account the **TestSample** *fitness*. The Executor/Arbiter executes and arbitrates the test samples and returns collections of test verdicts (*pass* / *fail*), one for each monitored Oracle Interaction Token. The test verdicts of the observed Oracle Interaction Tokens are recorded as instantiations of the **Interaction** variables in the collection of the evidence indicators $\boldsymbol{e}$.

The AC engine utilises a "double traversal" algorithm [Darwiche 2003] to calculate: (i) $P(\boldsymbol{e})$ and $P(X, \boldsymbol{e} - X)$ in a first traversal and (ii) $P(X|\boldsymbol{e})$ in the second traversal, X being any variable not instantiated in the evidence $\boldsymbol{e}$.

POLICIES

There are several parameterised scheduling policies that the scheduler can put in place: (i) Generic scheduling policies, (ii) Halting policies.

GENERIC SCHEDULING POLICIES

Once the inference is done, the Inference Engine supplies the Scheduler with the collection of remaining test samples with their *fitness* probability distribution. The Scheduler transmits to the Runner through notifyTestSchedOutcome the identifier of the next test sample to run. This test sample is chosen through the application by the Scheduler of a policy to the results (probability distributions) of the inference cycle. The Scheduler can put in place different configurable policies about the utilisation of these probability distributions. The three "extreme" generic policies: (i) *max-entropy policy,* (ii) *max-fitness policy*, (iii) *min-fitness policy*, constitute the *benchmarks* for other policies that can be considered through the usage of the Scheduler in specific situations.

The *max-entropy policy* can be described roughly as “the least informed the first”. The choice of the test sample will be done on the basis of the Shannon entropy [Khinchin 1957]. The Scheduler selects the test sample to be executed/arbitrated whose *fitness* probability distribution presents the maximum of entropy, i.e. the lowest amount of information (roughly speaking, the test sample whose variable has a *fit*/*misfit* distribution that is the nearest to fifty-fifty) and communicates its identifier to the Runner (that re-routes the information to the Executor). The entropy-based policy drives a breadth-first search that checks initially the elements of the services architecture under test about which the information is minimal.

The *max-fitness policy* can be described roughly as “the fittest the first”. This policy basically suggests test samples prioritised on the basis of their maximum *fit* probability. In this case the *fit* probability can be interpreted as a prioritisation measure of the test sample fault-exposing potential. The Scheduler chooses the test sample that has the maximum *fit* probability and notifies it to the Runner (that re-routes the information to the Executor).

The *min-fitness policy* can be described roughly as “the least fit the first”. This policy basically suggests test samples prioritised on the basis of their minimum *fit* probability. The Scheduler chooses the test sample that has the minimum *fit* probability and notifies it to the Runner that re-routes the information to the Executor.

GENERIC HALTING POLICIES

The Scheduler drives the test run cycle, by giving the appropriate directives to the Runner. It can put in place four different “generic” halting policies: (i) *n-fit-halt policy*, (ii) *n-misfit-halt policy*, (iii) *entropy-threshold-halt policy*, (iv) *no-halt policy*.

With the *n-fit-halt policy*, the scheduler suggest to stop the test session after the n-th **TestSample** dependent probability is calculated to **1**, i.e. after the n-th test sample arbitration with at least one *fail* local (Oracle Interaction Token) verdict.

With the *n-misfit-halt policy*, the Scheduler suggest to halt the test session after the n-th TestSample variable *fit* dependent probability is calculated to **0**, i.e. after the n-th test sample arbitration with all *pass* local (Oracle Interaction Token) verdicts.

With the *entropy-threshold-halt policy*, the Scheduler suggests to stop the test session when the entropy of a specific group of variables (the groups can be composed of variables of the same type, of variables of different types …) is lower than a given threshold.

With the *no-halt policy*, the Scheduler does not suggest any stop of the test session until the “end” of the test suite, i.e. while remaining non-executed test samples.

The Scheduler suggests stopping the session with a notifyTestSchedOutcome to the Runner with a halt directive and all the information it possesses in order to establish a meaningful test report (in particular, the probability distributions). If the Runner, which is a consumer of the Scheduling service, decides to execute other non-executed test samples and supplies their verdicts to the Scheduler, these verdicts are integrated as new *observations* in the inference engine and the inference step is executed. The Scheduler continues to supply all the reporting information at each schedule/execute/arbitrate cycle, modified by the new inference cycles suite to the new execution/arbitration cycles, until the reception of the abort command.

SCHEDULING BEHAVIOR EXAMPLES

This section offers a description of the scheduling behavior generated by the inference engine. In order to facilitate the understanding the “functional” behavior of the AC, this paragraph presents a simplified and more intuitive description of the inference engine states and transitions. Firstly, the inference engine is presented in its virtual Bayesian Network form (the AC representation states and behaviors have no intuitive semantics). Secondly, in a first picture, only the **System**, **Participant** and **Interaction** layers of the vBN are highlighted, i.e. the intermediate SAUT decomposition layer (**SendingPort** and **InteractionType**), representing the lower levels of functional coverage granularity, as well as, temporarily, the **TestSample** layer are not displayed. The exhibition of the intermediate **SendingPort** and **InteractionType** levels do complexify the picture without bringing useful information on the behavior.

Let’s remind that, at the beginning of the test session, the top DAG nodes (**Interaction**s) are initialized with prior probabilities. In each “normal” inference cycle (the last one identified by the halting policy is not considered), at the beginning the inference engine is informed of the local test verdicts and, at the end, it proposes a test sample.

For the purpose of the explanation let’s consider a simplified SAUT S () composed of 4 components p0, p1, p2 and p3. The component issues the interaction tokens m0,…,m9, “through” the intermediate structures (ports, interaction classes) whose variables (**SendingPort**s, **InteractionType**s) are not displayed. The interaction tokens are owned by four test samples O0, O1, O2 and O3.

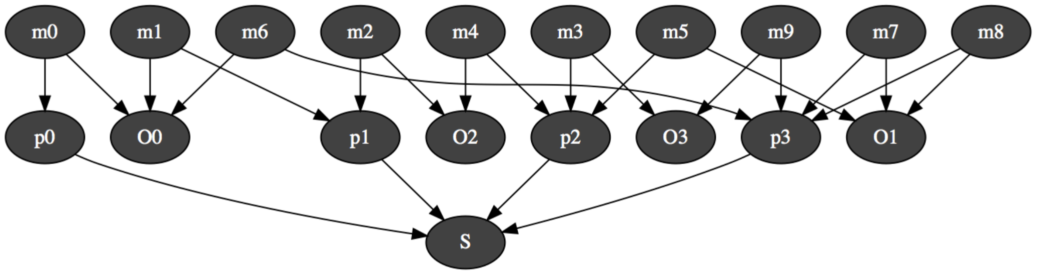
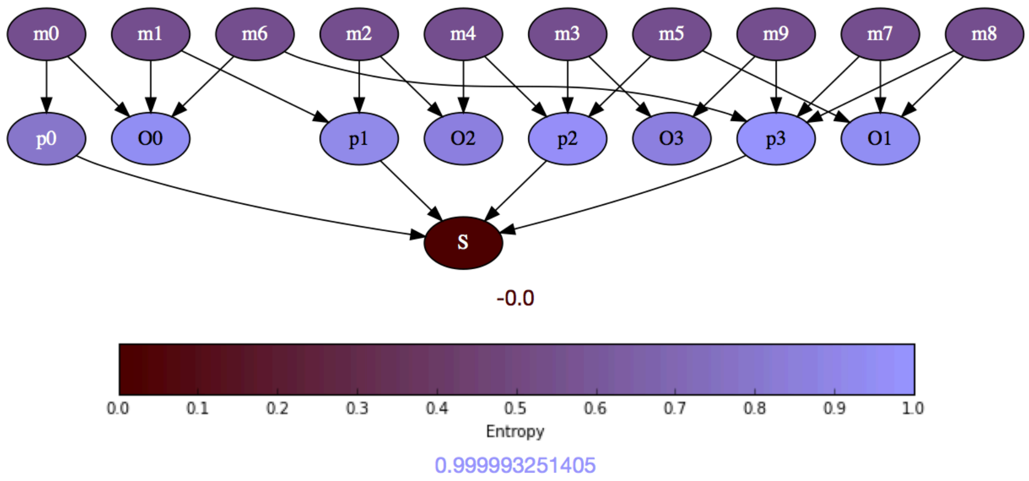


Figure 47. Simplified vBN S.

The first inference cycle is triggered by instantiating **System** to **1** (*faulty*) - the *assumption* that the SAUT is faulty. From the **Interaction** *fail* prior probabilities and the **System** *faulty* assumption, the inference engine calculates the **TestSample** *fitness* probabilities. Let’s suppose that the scheduling policy is the *max-entropy policy* (*“*the least informed the first”): given the **TestSample** fitness probability distributions, the Scheduler calculates the entropy of each test sample. The test samples are prioritized by their entropy. In this case the best candidate for the next test execution is the test sample O0 (see Figure 48).



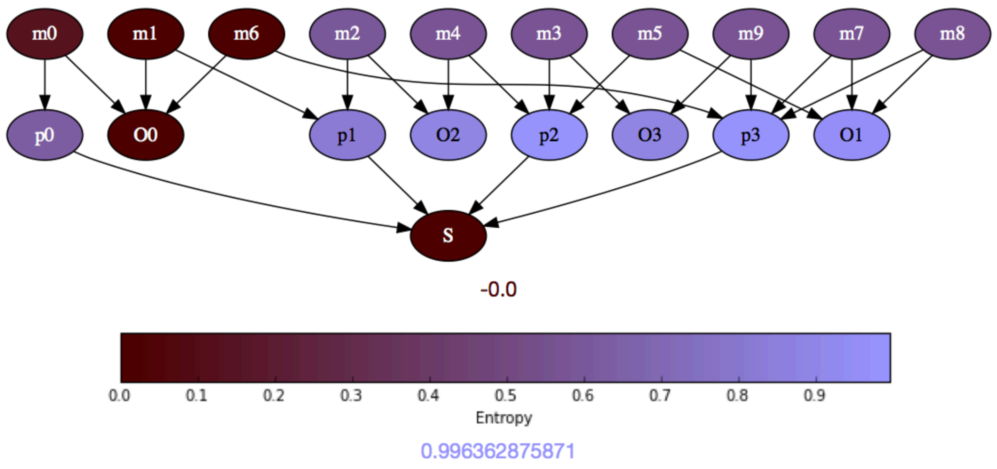
```
O0:0.9587452405     O1:0.958089863347     O2:0.876785386883     O3:0.875979077773
```

Figure 48. Prioritised test samples through the max-entropy policy.

The Scheduler sends to the Executor/Arbiter via the Runner the **test simple id** O0. The inference waits for the return from the Executor/Arbiter via the Runner of the global verdict, i.e. the collection of verdicts for m0, m1, m6. The Executor/Arbiter effectively runs O0 and returns the verdicts to the Scheduler (always through the Runner. The scheduler inserts the verdicts as *observations* in the inference engine and the inference step is triggered. The following scenarios are illustrated:

1) The test execution/arbitration returns the *pass* local verdict for all the m0, m1, m6 interaction tokens. The next test sample suggested by the *max-entropy policy* is O1.

```
showResults({'O0':OK,'m1':OK,'m6':OK})
```



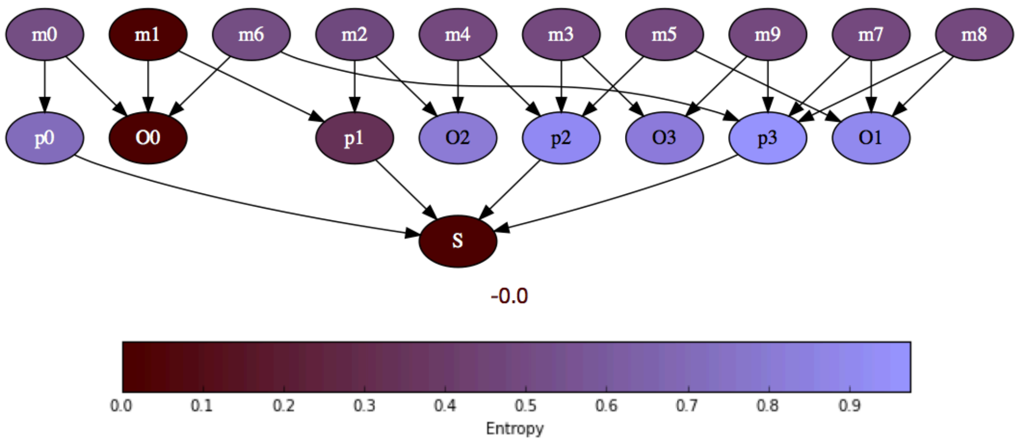
```
O1:0.973872694353     O2:0.897796383678     O3:0.896841723273     O0:-0
```

Figure 49. O0 passes and O1 is suggested.

2) The test execution/arbitration returns the *pass* for m0 and m6 and *fail* for m1. In order to bypass the single-fault assumption, m1 is inserted as being *pass* and m0 and m6 are not instantiated. A new prioritization of test

samples is proposed that is the specific result of the application of the *max-entropy policy* after the observation of m1 being observed *fail*.

```
showResults({'O0':NOK,'m1':NOK})
```



0.975392585873

```
O1:0.915734398671     O3:0.828236021682     O2:0.825143544483     O0:-0.0
```

Figure 50. Breadth-first (max coverage) search for failures knowing that there is a fault in p1.

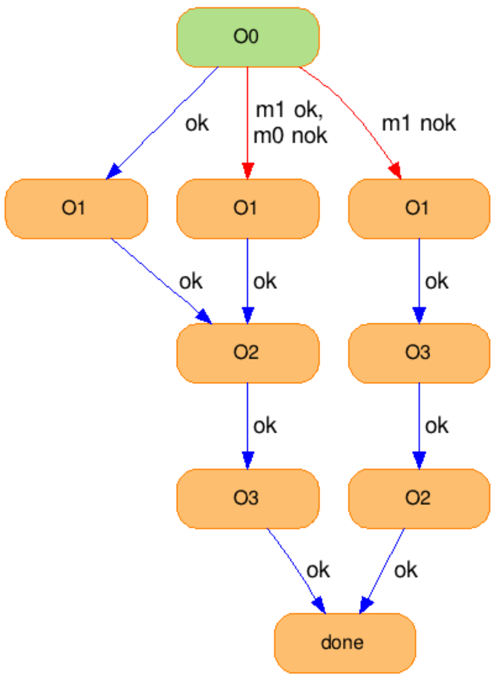
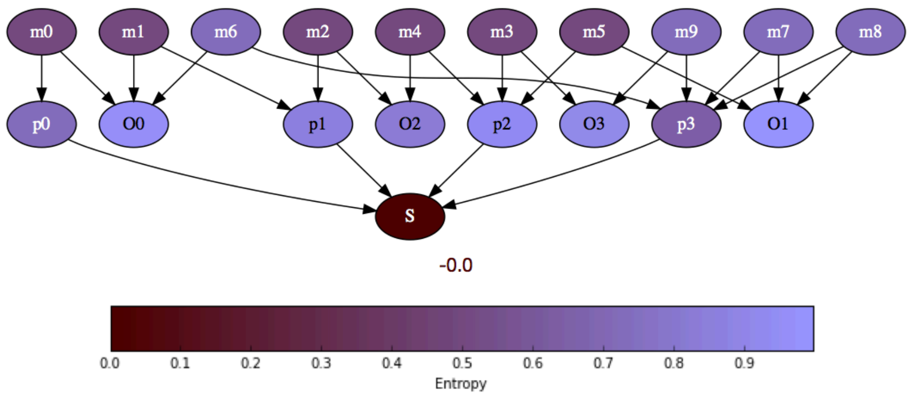


Figure 51. Sequence diagram of three path of test sample execution according to the returned verdicts.

In the activity diagram above three paths of test sample execution are presented, starting from different verdicts for O0 (the verdicts presented in Figure 49 and Figure 50, and another verdict not presented).

```
showResults({'p3':[1,5]})
```



0.997923876297

```
O1:0.997923876297     O0:0.974513839803     O3:0.923272617617     O2:0.842940822795
```

Figure 52. Entropy level on each component and proposed order of test samples after insertion of belief on p3.

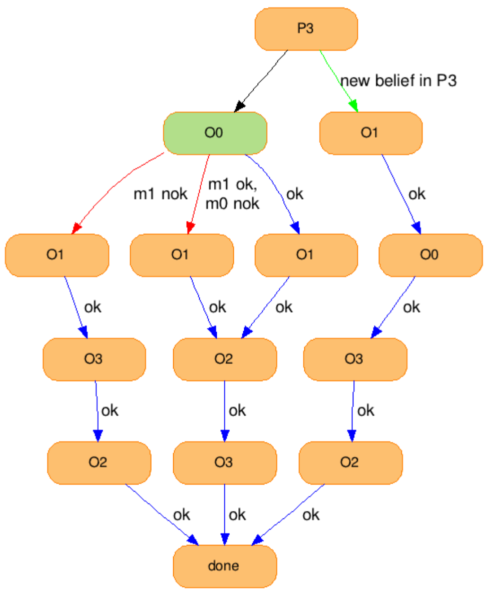


Figure 53. Activity diagram with regression testing test sample execution path

The Scheduler behavior in a regression testing scenario is sketched in Figure 52. A fault has been found in p3, the test session is halted and the AC Image is saved. The fault in p3 is fixed. A “new” test session starts, on the same SAUT and with the same test suite, and the scheduler upload the saved AC Image. The “new” test session is initialized by inserting version-specific information: the *belief* of functional reliability of the fixed p3. The new activity diagram showing the added execution path of test samples when modifying the belief of p3 is showed in Figure 53.

PERFORMANCE OF THE INF4SAT ALGORITHM

The performance of the inf4sat algorithm has been evaluated with an experimentation. A generator produces waves of 20 BN description files (corresponding to “dummy” SAUTs and Test Suites). Each wave includes bigger BNs. The vBNs are created in a top-down fashion in the sequence **System**, **Participant**s (at least two), **SendingPort**s (at least one per **Participant**), **InteractionType**s (at least one per **SendingPort**). A number of **TestSample**s is created, and, for each **TestSample**, enough **Interaction**s are created to guarantee that at least each **InteractionType** is linked to one **Interaction**. The dependencies between elements are randomly generated. The connection structure of two vBNs the same number of elements can be very different. Consequently, the two BN can represent two very different factorization of the joint probability law, hence not the same complexity for the inference effort. At each wave the number of elements is augmented for each BN.

To be able to test the performance of the inf4sat algorithm it is necessary to compare the results to alternative tools. For comparing compilation alternative the ACE package developed at UCLA[47] is used. Ace is a package coded in JAVA that includes a compiler that can transform a Bayesian network into an AC and a light evaluator for inference. The ACE compiler take as inputs .hugin/.net network format files and the evaluator uses a proprietary format for the evidences.

The performance test focuses on different characteristic of the outputs:

1. The number of AC nodes related to the amount of caching of subcircuits operations values.
2. The number of edges related to the number of operations and closely linked to the inference time.
3. The compilation time.

For the inference time comparison, it has been extended (beyond the comparison between inf4sat and Ace) to other classic inference algorithms. aGrUM, developed by the Decision TEAM of the LIP6 at the University UPMC[48], is a C++ library that permits the generation of BNs and provides multiple inference algorithms. The inference algorithms adopted for the performance test are Gibbs [Koller and Friedman 2009] and Lazy Propagation [Madsen and Jensen 1999]. Those algorithms were chosen over others because they implement global inference like inf4sat. aGrUM generates the BN from the specification files of inf4sat.

The configured evaluation environment offers to all the inference methods and algorithms, with and without compilation, exactly the same generated BNs.

This section presents the results of the performance test and the observations that can be made from them. This results concern the AC size and the inference time. The section ands with a discussion concerning the trade-off inference time versus compilation time.

[47] http://reasoning.cs.ucla.edu/ace

[48] http://agrum.lip6.fr

ARITHMETIC CIRCUIT SIZE

In this section, a comparison between the method investigated in this research and the state of the art concerning compilation size efficiency is presented. By compilation size efficiency, it is intended the research of the smallest size of the AC for a same BN.

A Multi-linear function (MLF) is unique and depends uniquely on the size and complexity of the BN (nodes and edges). Different methods of compilation can compile the same MLF into different ACs with different compression rates. Because the MLF is always the same, finding its smallest AC improves the inference calculation. In fact we know from the literature that the inference computation time is linear to the AC size.

inf4sat has been compared with the method proposed by the ACE tool. The ACE tool uses the technique of compilation developed by Darwiche and colleagues [Darwiche 2003].

The graphic presented in Figure 54 shows the dependence between the number of the BN elements and (i) the number of nodes (memory size), (ii) the number of edges (operations) of the MLF.

The first experimental result is that the inf4sat compilation method generates ACs with no standard deviation (the AC is always of the same size).

The compilation method of the ACE tool shows a higher number of nodes and edges with a clearly visible standard deviation, which is evidence of randomness in the compilation method. The standard deviation increases with the size of the BN.

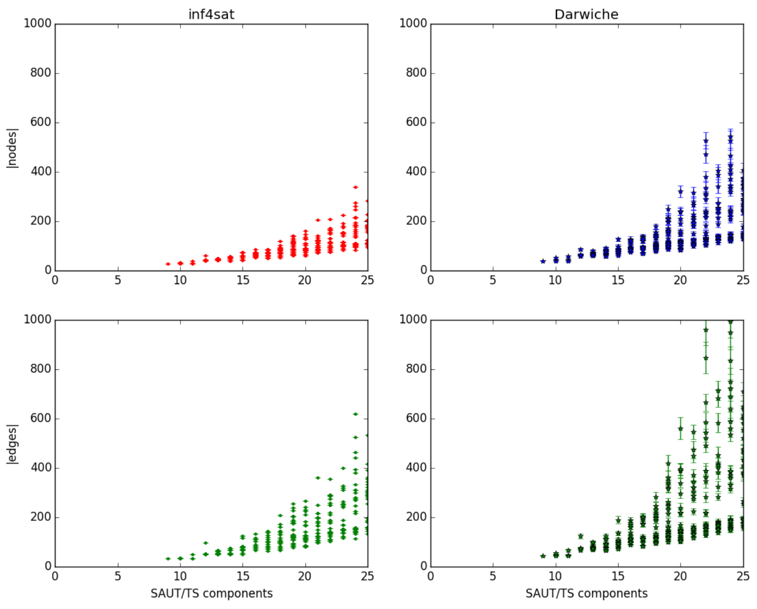


Figure 54. Comparison of compilation size results with variance.

In Figure 54 the standard deviation has been shown on a set of small BNs, since it is easier to see the difference. For the rest of the graphics presented the standard deviation, which is increasing with the problem size, will be omitted, because it makes the graphics unreadable. The graphic presented in the figure below shows the comparison of the generated AC sizes between inf4sat (exact AC size) and Darwiche (average AC size).

The comparison is made twofold for the AC nodes and the AC edges. The blue triangle represents the area where the numbers of objects generated are smaller for the ACE program than the inf4sat. The second observable result is that for a given BN the size result of the Darwiche algorithm is always higher both for AC nodes and AC edges. Figure 55 also shows that the inf4sat algorithm can process BNs that are practically untreatable for ACE (that crashes).

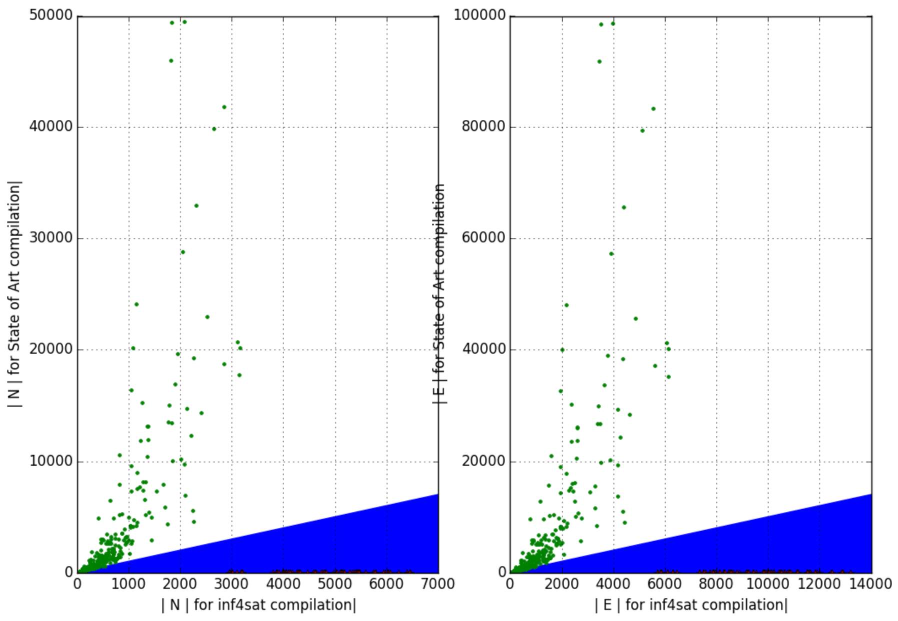


Figure 55. Comparison of compilation size results (II).

In summary, three results are revealed in this section concerning alternative compilation solution.

1. inf4sat exhibits a more efficient compression rate.
2. inf4sat always find a unique solution.
3. inf4sat is more robust and pushes further the limits of BN size.

INFERENCE SPEED

This section presents the results on inference time, issued from the comparison of the three previously mentioned approaches and the inf4sat inference algorithm.

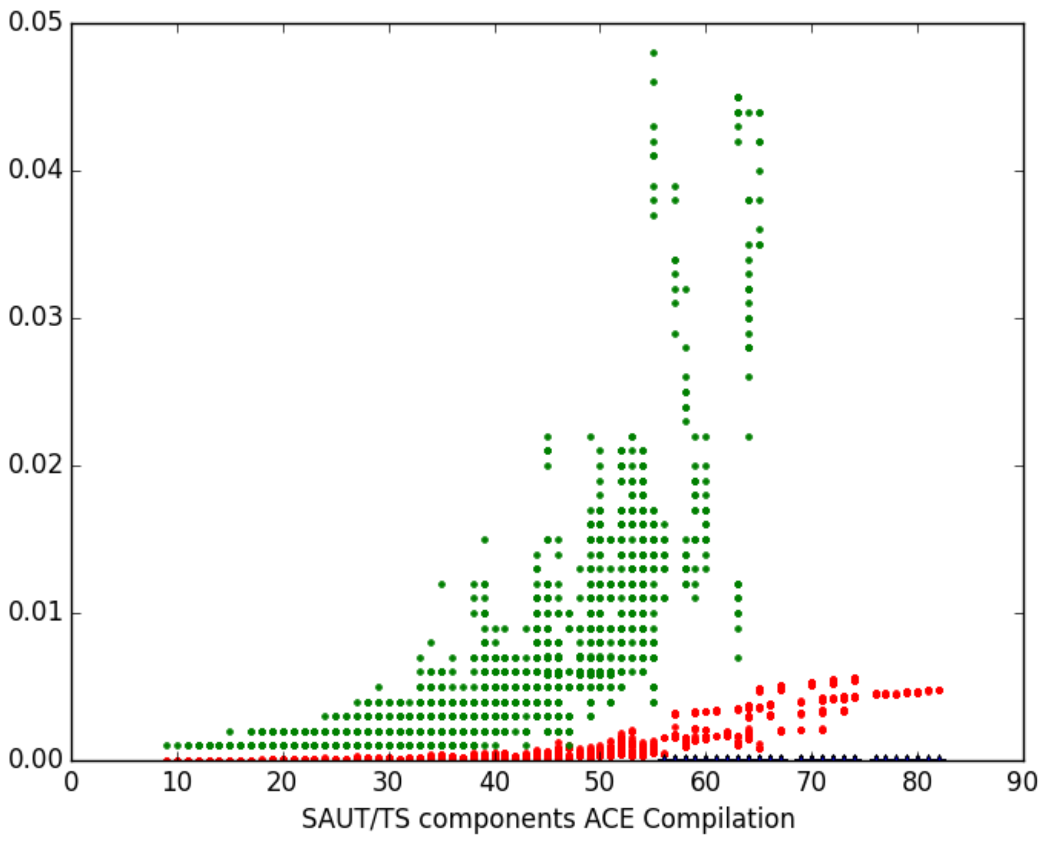


Figure 56. ACE AC evaluator inference time vs. inf4sat AC evaluator inference time (s).

In the figure above, the green dots represent the ACE AC evaluator inference time, and the red ones represent the inf4sat AC evaluator inference time. The blue dots near the bottom line represent the evaluation failures of the ACE evaluator (it crashes or cannot process the its generated AC).

The compression rate improvement of inf4sat with respect to the Darwiche method brings also an improvement of the average inference computation time. This is an expected result, because the inference computation time is linear to the AC size.

The robustness of the inf4sat algorithm compared to the ACE software can be asserted here since it is possible to see in the figure above that beyond 65 BN elements (SAUT/TS components) ACE is no more able to treat the complexity of the BN.

It is possible to compare the inference time using another format for the graphic. In the comparison between the two methods, which is illustrated in Figure 57, the inf4sat AC and the Darwiche AC are provided with the same evidences and this computation is repeated 20 times in order to have an average time for the Darwiche method.

For the same inference (BN + evidence), the inf4sat value is on the x-line (abscissa) and the Darwiche value is on the y-line (ordinate). The values show that for all inference computations the inf4sat value is better than the Darwiche value.

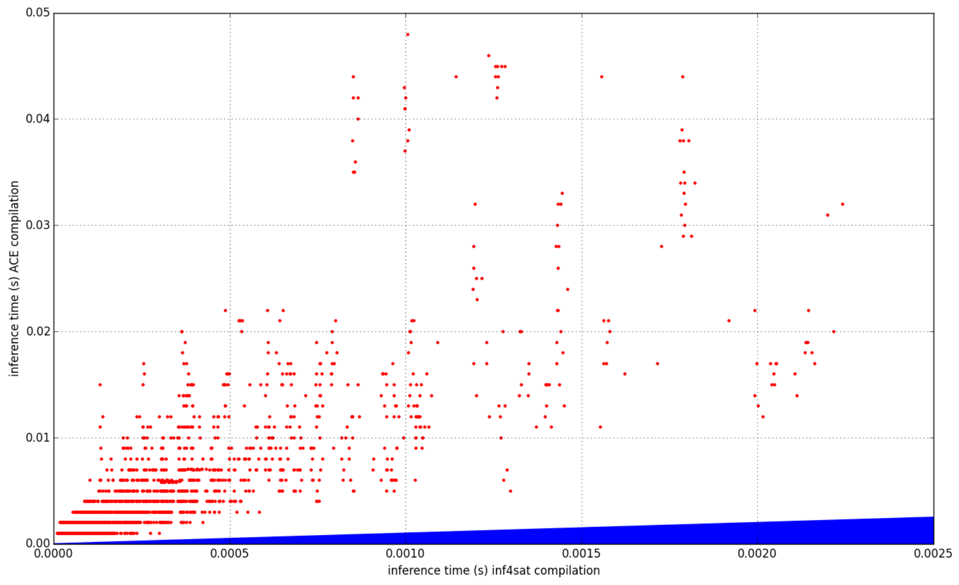


Figure 57. Comparison of the average inference computation time between inf4sat and Darwiche.

Moreover, the experimentation has been conducted also with the other “classical” inference methods such as:

- Gibbs sampling inference method - Figure 58.a
- Lazy propagation method - Figure 58.b

The graphic in Figure 58 shows a simple superposition for each “classic” inference algorithm (green dots) with the inf4sat inference algorithm (red dots). The blue dots near the bottom line represent the evaluation failures of the “classic” algorithm.

The results show directly that:

- the inference time results of the inf4sat algorithm are better than both classic algorithms, and
- the inf4sat algorithm is more robust and the classic algorithm.

Gibbs algorithm shows inferior performance but better robustness.

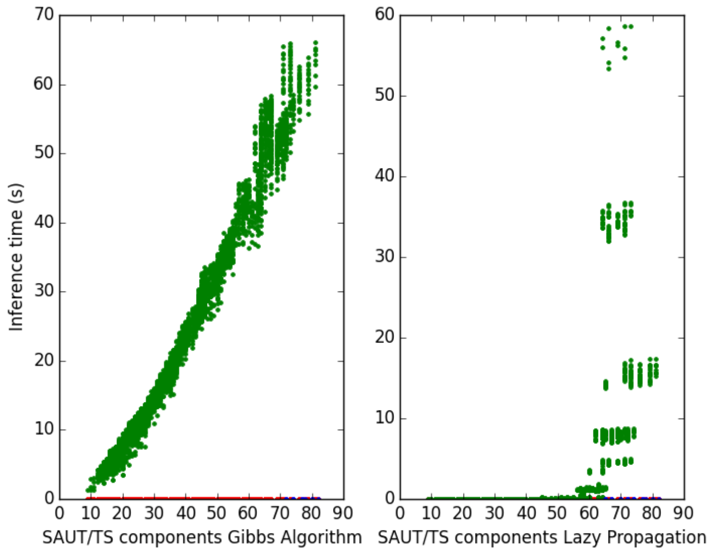


Figure 58. Classic inference algorithms inference time (s)

Figure 59 shows results in the same format than Figure 57. In each figure, the inf4sat value is on the x-line (abscissa) and the value of the competing classical method is on the y-line (ordinate). All the trials show that the inference computation time of the inf4sat method is by far better than the inference computation time of any "classic" method.

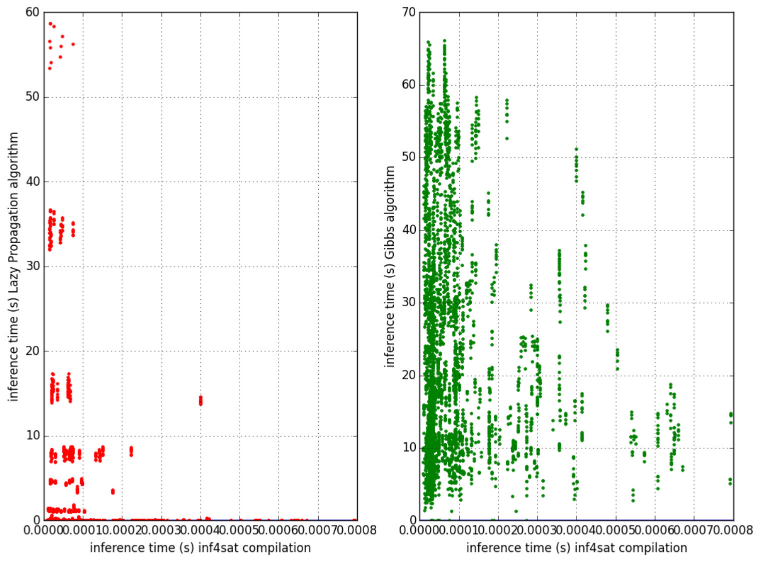


Figure 59. Comparison of the average inference computation time between inf4sat and Lazy Propagation algorithm (left) and Gibbs sampling algorithm (right).

The previous results show the efficiency of the inf4sat inference with respect with the other “compiled” and “non-compiled” methods. The cost of this improvement is in terms of larger compilation time.

In fact:

- Classic inference methods [Shenoy and Shafer 1988] [Zhang and Poole 1994] [Madsen and Jensen 1999] are slower in inference time and less robust in BN size tractability, but do not need any compilation time.
- The other AC compilation method [Darwiche 2003] is faster, because it does not seek for the optimal arrangement of the AC nodes and edges, but is less robust in BN size tractability,

The Arithmetic Circuit compilation of the inf4sat tool, i.e. the search for the best arrangement for the entwining optimal sub-circuits among all the possible configurations, is the most demanding task regarding memory allocation and processing power. But, once this compilation step is completed, the result can significantly improve the inference computation time and the AC size (that is related to the inference time).

The goal of this research is to build the smallest AC for the same MLF in the fastest possible way. It remains to be seen whether the benefit brought by the MLF optimised AC compilation to the scheduling engine is higher than the cost of the compilation time. This is the subject of the following section.

DISCUSSION

The results presented above highlight four points:

- The inf4sat compilation pushes the limits of BN representation to significantly large and complex BNs with respect to the classical BN methods, for which, beyond a certain size and complexity, the application of the Bayesian inference is practically intractable.
- The inf4sat compilation brings a big improvement of the inference computation time with respect to classical BN methods even for BNs that are tractable by these methods.
- The improvement of the AC size of the inf4sat compilation method brings a significant advantage in terms of AC size, size tractability and inference time even with respect to the ACE method.
- The cost of the advantages mentioned in the points above is in terms of AC compilation time, for which the inf4sat method is worse than all the others.

The question that can be asked now is if the cost of the compilation is worth the improvements in terms of size and complexity of the BNs that are tractable and in terms of inference computation time.

The inf4sat method satisfies the first goal of this research that is scheduling the test run cycle of relatively large test suites on large and complex SAUTs without significant degradation of the inference computation time.

The first argument that relativizes the compilation time cost is the *reusability* of the compiled AC. The Scheduler is able to store the *AC image* (the compiled form of the AC) at the end of initialisation step and at each schedule/execute/arbitrate cycle.

Re-testing and Regression testing are typically very interesting testing processes that can reuse advantageously the AC image. With *basic re-testing and regression testing*, the nth test session is run with exactly the same SAUT construction model and the same Test Suite on a service component architecture under test, where some components' implementations have been modified in order to fix some faults or improve their performance. By definition, there are no changes of the SAUT structural, functional and behavioural models (the service contracts), only of the service components' implementations, and the implementation changes always aim at ameliorating the compliance of the service components and of the overall architecture with the aforementioned models.

Basic re-testing and *regression testing* processes can advantageously utilise the AC image in order to improve the performance of the scheduled test run cycle. It suffices to change the input parameter of the initialisation operation (initTestSched) from the SAUT/TSD/TS files' URIs to the appropriate AC image file URI. In this case the AC is built in memory directly from the content of the AC image file. The cost of the AC image upload is linear to the AC size and negligible in any case.

Within basic re-testing and regression testing, the compilation step occurs only once, can occur asynchronously with the test session and its result is utilised in an unlimited number of test sessions.

Moreover, as the MIDAS platform is deployed on the cloud, the Scheduler initialisation step can be executed to the most powerful cloud virtual machine, and the Scheduler computations during the schedule/execute/arbitrate cycle could be downsized to a less powerful cloud virtual machine.

In order to assess the costs / benefits of the inf4sat AC compilation approach more objectively, the function $\frac{\Delta_c}{\Delta_i} = \mathrm{N}$ is defined where Δ_c is the variation of compilation time (loss) and Δ_i the variation of the average inference time (gain). N is the minimum number of iterations of inference computation necessary to justify the compilation effort. The comparison is effected between:

- inf4sat vs. Darwiche,
- Inf4sat vs. "classic" inference methods (Lazy Propagation and Gibbs sampling).

Note that in the case of the "classic" inference methods the variation in compilation time is the inf4sat compilation time since the time of generation of the BN is negligible.

The results are presented in the figures below. Within the domain of tractability of the three methods the ratio N is on average too high for inf4sat to be pertinent, especially for ACE and Lazy Propagation. One could argue that the ratio for the Gibbs sampling algorithm may be the only one that could be pertinently to replace. However, inf4sat offers a more robust solution that is sustainable beyond the size tractability limits of the other algorithms, that are too low for being applicable to the problem posed to this research work, the effective scheduling of test session of even small services architectures and test suites.

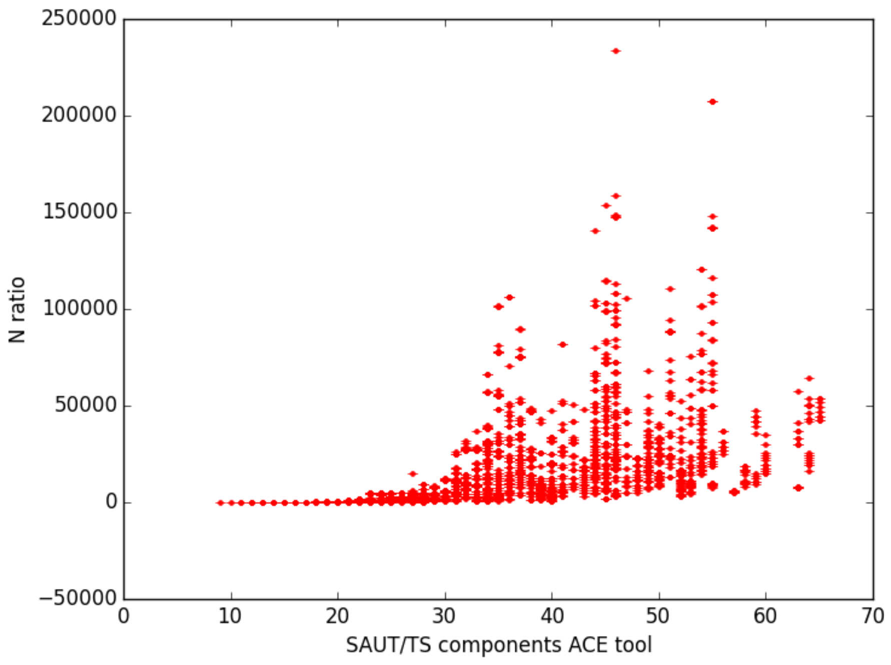


Figure 60. N ratio comparing to the ACE compiling and evaluating tool.

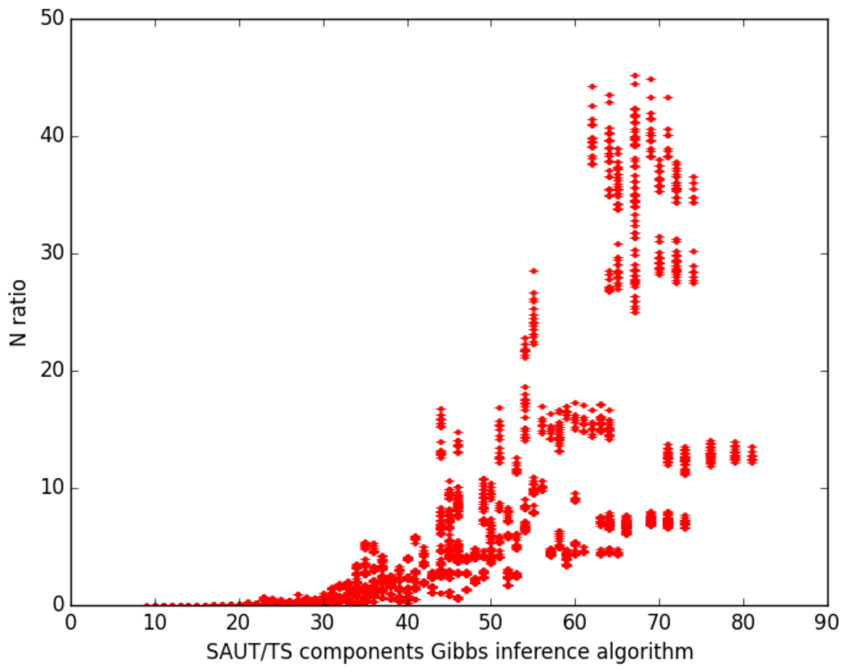


Figure 61. N ratio to compare to Gibbs inference performance

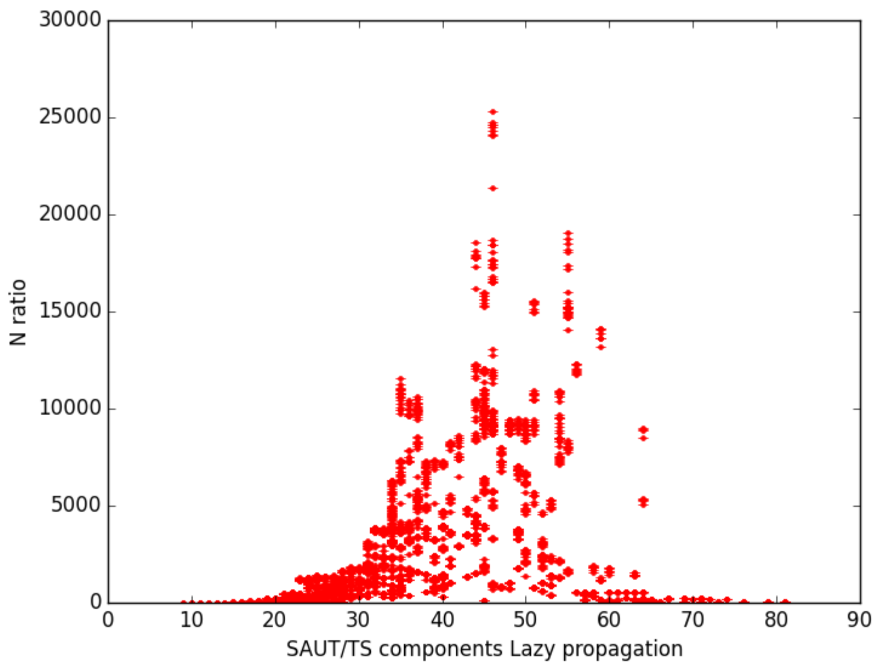


Figure 62. N ratio comparing to Lazy Propagation performance.

8. CONCLUSION AND FUTURE WORK

MAIN RESEARCH RESULTS

The main result of this research is a software tool that, on the basis of probabilistic inference, implements efficient dynamic scheduling services on cloud to be utilised by automated execution/arbitration systems that targets distributed services architecture. The scheduling services aim at improving the testing time efficiency (the *fault detection rate* - the precocity of the exposure of failures and of the detection of faulty elements that are the sources of the failures), at shortening the test session durations and at augmenting their frequency, in the context of a software engineering cycle that integrates the Test-Driven Development and the Continuous Integration Testing approaches.

INTELLIGENT DYNAMIC SCHEDULING OF TEST SESSIONS

Current research in SOA testing is mainly focused on test generation, and on orchestration and choreography testing. The research on testing of services architectures that practice *direct* (by program) composition, without any orchestration or choreography infrastructure, i.e. the majority of present and future real world applications (API economy, REST/JSON services), is still at its infancy. The test scheduling for efficient testing and troubleshooting requires a model-based testing approach, in particular a simple but powerful meta-model of the SAUT structure (descriptive model) and of the test system configuration (prescriptive model to be utilised by the test system). The current MBT research focuses on testing scenarios for test case generation but lacks independent structural and functional modelling of the SAUT.

In the domain of test run automation, an important step has been taken with the “invention” of TTCN-3 [Willcock et al. 2011]. TTCN-3 is an international standardised general programming language that has been designed with powerful traits that are specific for test automation. It allows complete automation by programming of the test execution and arbitration tasks. De Rosa and colleagues [De Rosa et al. 2013], including the author of this manuscript, have implemented a generic and complete model-based TTCN-3 framework for testing services and service compositions. This framework is able to integrate not only a formal representation of the test scenarios (message sequences) and of the test suite, but also a structural meta-model of the SAUT that is independent from any scenario representation meta-model and from any orchestration and choreography infrastructure. Starting from these formal representations, the framework implements the execution/arbitration automation of the scheduled test run cycle. This framework allows the scheduler to drive the dynamic scheduling of automated test execution and arbitration tasks. The same elements (SAUT structural model, test suite) allow configuring, generating and initialising the TTCN-3 executor/arbiter and the Bayesian agent that realises the dynamic scheduler. This execution/arbitration framework and the dynamic scheduler that results from this work are integrated in the MIDAS facility platform and realise what can be considered, in the author’s best knowledge, the first automated and dynamically scheduled execution arbitration framework for functional conformance testing of services architecture.

The SOA regression testing scientific literature is interesting because of its focus on practical questions about the cost, duration, effectiveness and efficiency of regression testing and re-testing with already existing test suites. The most interesting proposed solutions are static prioritisation techniques of the test suite based on test coverage of the SAUT and on the fault-exposing potential of the test case. Note that these techniques can be very greedy in terms of computational resources. The limitation of the approach is that the proposed heuristics and algorithms are twofold non contextual: they are not dynamic, i.e. they do not take into account successes and failures of the on-going test session, and they are not version-specific, i.e. they do not take onto account the specific characteristics of the version under test in the regression testing or re-testing history. The research presented in this manuscript introduces the dynamicity in the test suite prioritisation and allows considering the specific characteristics of an implementation version by means of *beliefs* about the *fault proneness* of the SAUT structural and functional elements that are taken into account by the probabilistic inference. A deeper focus on re-testing and regression testing and version-specific scheduling is a topic of future work.

In summary, this work introduces the dynamic scheduling of functional conformance test sessions for efficient testing and troubleshooting as a new research topic in the testing domain and proposes a generalised approach to the dynamic prioritisation of test cases based on probabilistic inference, beyond the current application of static prioritisation to regression testing and re-testing.

BAYESIAN REASONING METHODS FOR SERVICE TESTING

The application of Bayesian reasoning methods to the “reliable and efficient software testing and program analysis” [Namin and Sridharan 2010] is considered an “ideal research paradigm” and a promising future research thread on software testing.

It is surprising that the fundamental uncertainty of the testing activity and, above all of the black-box/grey-box testing (for instance, the high risk of false positives and false negatives that are highlighted in the examples presented in the preceding sections) has been recognised only very recently and is still taken into account only by a minority of the research approaches.

The seminal and pioneering works in this domain concern different aspects of the testing activity: test case generation, mutation testing, representing and measuring the reliability of program component (defect density, time to failure), software economics and metrics, decision support of the verification and validation activities, causal models for quality prediction, risk models, software quality models, modelling and updating developers’ beliefs regarding software components, and, last but not least, a pioneering work on test case static prioritisation for white-box testing [Mirarab and Tahvildari 2007] that interestingly consider interactive prioritisation incorporating feedback from testing (in other terms, dynamic scheduling) an promising topic for future research.

Bayesian troubleshooting is a well-established discipline that has been successfully applied in industrial applications for troubleshooting of physical devices. The research presented in this manuscript introduces troubleshooting models and techniques based on Bayesian Networks and in general applied to physical devices in the domain of software testing, where a fault is not a newly broken physical component, but the consequence of an error that has been introduced by the programmer from the beginning. The idea is to utilise similar troubleshooting methods to localise faults in services architectures.

In summary, this work is an original contribution to the utilisation of Bayesian network inference and Bayesian troubleshooting approaches to improve the time efficiency of testing of distributed services architectures through intelligent dynamic scheduling. The author of this manuscript believes that the problem of intelligent dynamic scheduling of functional conformance test sessions on services architectures is too complex for non-probabilistic inference methods and can be posed only from a probabilistic stance and solved only through probabilistic inference (and it is a “new” domain of the testing research for this reason).

BAYESIAN NETWORK MODEL FOR TEST SCHEDULING

An appropriate BN model (the definition of variables and of variable dependences) of the problem domain is a crucial element for the success of the probabilistic inference application. A BN model for the intelligent scheduling of test sessions on services architectures, with the objective of test time efficiency must take into account on one side the structural and functional decomposition of the SAUT at the lowest level of granularity in order to improve fault localisation and, on the other side, the structure of the test suite and its relationships with the aforementioned structural and functional decomposition in order to dynamically prioritise the test cases on the basis different criteria.

This work proposes an original, simple and universal Bayesian Network model of the problem domain (the grey-box functional conformance testing of services architectures). This model takes into account the structure of the services architecture, the service dependences and the functional architecture decomposition until the message type level of granularity, in terms of random variables and dependences between variables. The model is independent from: (i) the implementation technology of the service components, of the services and services operations, (i) the technology of the service interoperability platform[49] (WSDL/SOAP, REST/XML, REST/JSON – it applies even to services architecture in which all these technologies are present at the same time), (ii) the presence/absence of composition infrastructures such as orchestration and choreography. The BN model is also able to identify each individual test case and each individual message / oracle / local verdict, and to take into account as dependences their relationships with the architecture decomposition.

INFERENCE BY COMPILATION IMPROVEMENT

The Scheduler puts into operation a probabilistic inference engine whose main job is to propose iteratively, in a test session, the test cases to be executed on the basis of the verdicts of the past test runs. The dynamic prioritisation of the test cases, that takes incrementally into accounts the results (verdicts) of the test runs as observations, aims at proposing for execution first the test cases that *fit* with different criteria. The dynamic (re-)computation of this *fitness* is based on the results of the probabilistic inference cycle that are evaluated on the basis of different scheduling policies.

However, the application of probabilistic inference to dynamic scheduling of test sessions raises significant computational complexity problems. This work implements an original “inference by compilation” algorithm (inf4sat) that is compliant with the Bayesian network inference.

Compared to its “competitors” (“classic” Bayesian network and Arithmetic circuit inference algorithm), inf4sat is more robust, less consuming in memory size of the compiled Arithmetic Circuit, improves the inference

[49] The scheduler is utilised today on the MIDAS platform on SOAP services architecture. This is a constraint of the MIDAS facility : the BN model is completely independent from the service technology utilised by the service components.

speed and pushes further the practical tractability limits of complex models. Conversely, it is not as good as its competitors in compilation speed, but this trade-off (compilation speed vs. inference speed) is completely consistent with the proposed *modus operandi* of the algorithm, which allows compiling once, saving the AC image and running fast (without any compilation time) a large number of frequent test sessions on the same AC image, particularly for re-testing and regression testing in Test-Driven Development and Continuous Integration Testing Environments.

TESTING COSTS

The dynamic scheduler is a software module that is deployed on the MIDAS testing facility platform on cloud and offers to the other components of the MIDAS platform its scheduling functionalities as a programmable service, through a "generic" API. The cloud deployment of the testing facility lowers substantially (at least of one of order of magnitude) the global cost of the computing resources needed for testing and troubleshooting, guarantees the scalability and elasticity of these resources and, finally, reduce to zero their marginal cost. Of the three cost components of testing and troubleshooting, *per se* and in relationship with the design and development activities of the software engineering cycle - equipment expense, labour effort, time-to-market – the first one is decreasing quickly compared to the others that are becoming critical. Hence, a significant reduction of the testing and troubleshooting costs can be obtained only through: (i) the extreme automation of all the testing tasks (labour effort), (ii) the shortening of the test sessions (time to market) and (iii) their automated, event-driven integration in the software engineering environments that improves their reactivity and augments their frequency (both).

CONCLUSION ON THE REAL-WORLD CASE STUDIES

One of the fundamental problems of the research about SOA testing is the lack of real-world case studies. In the SOA testing research review of Bozkurt and colleagues [Bozkurt et al. 2013], on 177 publications reviewed in 2010, only four research experiments are conducted on targets that can be considered real-world services and services architectures.

There are many plausible reasons for this lack of real-world case studies. The first is probably the fact that, in spite of the hype on SOA, only very recently the "API economy" phenomenon is going to spread the service orientation approach. The second reason is that SOA testing is complex and hard to put in place in real contexts, from the technical and organisational points of view. The third reason is that, roughly speaking, testing is a *hot topic*: businesses are reluctant to communicate about their testing practices and to expose their systems as targets of research approaches and tools for testing.

A specific characteristic of the MIDAS project is the planned availability of two real complex services architectures (the *Pilots*), respectively in the logistic (supply chain management) and health domains, as real-world case studies of the MIDAS facility. These services architectures have been designed and implemented in the first two years of the MIDAS project and are now (Autumn 2014) operational.

This Scheduler is a generic test scheduling tool that aims at being applied to several different testing approaches, strategies, contexts, through the definition and implementation of focused scheduling and halting policies. It is integrated in the MIDAS platform and its utilisation within test sessions on the MIDAS Pilots is on going. The author is confident that the Scheduler usage in relationships with different testing objectives and approaches on two real world case studies will confirm and improve the main technical choices of the proposed solution and will produce interesting policies about the utilisation of the probabilistic approach to dynamic scheduling.

FUTURE WORK

In this paragraph we list some topics of future research on the basis of the results presented in this manuscript.

TEST REPORT

A very important function of the Scheduler is to supply the Runner with meaningful information that accompanies the Scheduler halt notification (see section 7, § 'Generic halting policies'). The general idea is that every actor of the scheduled test run cycle (the Executor/Arbiter and the Scheduler) supplies to the Reporter (via the Runner) the information that can contribute to build a meaningful and comprehensive report about the test session. The goal of the test report is to shorten the debugging/fixing cycle. The problem is how to relate and supply not only core information such as test verdicts and faulty elements, but also information such as the *faulty index* of components, ports, message types and the fault-exposing potential of the not-yet-executed test samples. This could help the debug/fix team to localise and identify the implementation modification to be done.

ENHANCED RE-TESTING AND REGRESSION TESTING PROCESSES

An enhanced re-testing and/or regression testing process is able to utilise the dynamic prioritisation of test cases implemented by the Scheduler in order to maximise the value of the regression Test Suite for a given SAUT construction model.

Generally speaking, there are two kinds of prioritisation techniques for regression testing: (i) general test case prioritisation and (ii) version-specific test case prioritisation, and two general criteria for prioritisation: (i) prioritisation for improving the test coverage, and (ii) prioritisation on the basis of the fault-exposing potential of the test cases.

In the approach of this research the Scheduler:

- is model-based, i.e. it has a deep and detailed knowledge of the structure of the SAUT and of the Test Suite - it is aware of the impact of each element of each test sample on the SAUT structure;
- is able to perform sound probabilistic reasoning about the elements of the SAUT structure on the basis of prior probabilities;
- is able to acquire new evidences - not only *observations* such as the test verdicts, but also *beliefs* on the *faulty index* of the components, ports and interaction classes and on the fault-exposing potential of the not yet executed test samples.

In the current scheduler version, the tools that allow driving the inference process are the *prior probability settings* of the top random variables (the **Interaction**s) and the *beliefs* on the other variables that are processed as evidence realisations. The prior probability settings are parameters of the initialisation, i.e. of the compilation process, whereas beliefs can be entered and changed at each inference cycle. In the future the use of the different modalities of driving the inference process for re-testing and regression testing will be investigated.

Version-specific prioritisation in black-box or grey-box testing requires the availability of some information coming from the maintenance and the development teams. For instance, enhanced regression testing processes for implementation maintenance require that a more cooperative *test/debug/fix cycle* be put in place between the testing team and the debugging/fixing team. The idea is that the debugging/fixing team should accompany the deployment on the SAUT of new component builds with information about the failures revealed in the past test session that the new deployed builds are intended to correct. The investigation to be conducted should concern the best way of utilising this information in the new regression testing or re-testing session in terms of *beliefs*.

THE NOISY-OR APPROACH

In the actual vBN model of the Scheduler utilises OR nodes. This implies that, if an **Interaction Token** fails, its **Interaction Class** owner is *faulty*, the **Port** that owns the **Interaction Class** is *faulty* and so on. Conversely, if none of the **Interaction Token**s of an **Interaction Class** is *faulty*, the **Interaction Class** is not *faulty*. A less rigid approach considers the hypothesis that the test coverage of the test cases is incomplete. A solution for the less rigid approach could be the use of Noisy-OR nodes instead of the OR nodes [Pearl 1988] (see section 5). The parameters could be fitted to take into consideration the degree of exposure of the component's sub-component. The adoption of the Noisy-OR approach would not require a deep modification of the model and the implementation of the inference engine.

EVIDENCE DRIVEN GENERATION

In fact the idea behind a fully automated workflow for functional testing is to use the scheduler not only to drive the test among a set of existing test cases but also drive the generation of new test cases. With such a tool, the interested party need only to provide the structural, functional and behavioural description of the SAUT (SAUT Constitution and PSM models) for it to be tested.

The scheduler with the use of the Noisy-OR nodes is able to infer the faulty state of the different interaction types of the respective service. The Test Generator can generate instantiated test cases in accordance with the state machine (PSM) diagrams representing the SAUT behaviour (PSM) [De Rosa et al. 2014b]. Creating a test case corresponds to generate the interaction paths from the collection of SAUT PSM, and, for a path, to generate an instance of the path (a test sample) by generating the instances of messages of the path. For instance, the scheduler can specify the interaction types for which it wants new test samples.

LAYERED PARTITIONING TROUBLESHOOTING

Once the modifications described in the sections above are implemented, the Scheduler can be modified to become an enhanced troubleshooting tool. Knowing that the cost of compilation is not negligible and depends on the size of the SAUT/TS model, the troubleshooting problem posed by a large SAUT and a big Test Suite could be non-tractable with a direct compilation of the complete model. The solution to the problem is in the consideration that a distributed system is a system and can be considered as a black box. By partitioning the participant in regions and ignoring the wires linking the components inside the same region, the new SAUT model and the new TS become tractable in terms of vBN generation and AC compilation.

During each test cycle, if the evidence of faulty behaviour within an opaque region is revealed, the Scheduler can modify the partitioning of the participants by splitting faulty opaque region into two (dichotomously) opaque regions and, conversely, can regroup low faulty index regions by selecting the ports that are believed to work properly and ignoring them.

At the end of multiple cycles of test execution and test generations, the arrangement of the component partitioning is such that high faulty-index components are isolated and the rest are aggregated. In addition the Test Suites are tailored to focus on the outgoing and ingoing message of the high faulty-index structural and functional elements of the SAUT. For re-testing and regression testing if the test session is suspended and some faults are fixed, the suspended AC Image can be resumed.

ANNEXES

THE SCHEDULER INTERFACE

TESTSCHEDSERVICE_V1_4_0.WSDL

```
<wsdl:definitions xmlns:tns="http://www.midas-
project.eu/Core/API/TestSchedService" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:tst="http://www.midas-project.eu/Core/API/TestSchedTypes"
name="TestSchedulingService"
targetNamespace="http://www.midas-project.eu/Core/API/TestSchedService">
<wsdl:types>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:tst="http://www.midas-project.eu/Core/API/TestSchedTypes"
xmlns:tsit="http://www.midas-project.eu/Core/API/TestSchedInstantiatedTypes"
targetNamespace="http://www.midas-project.eu/Core/API/TestSchedTypes"
elementFormDefault="qualified" attributeFormDefault="unqualified">
<xs:import namespace="http://www.midas-project.eu/Core/API/TestSchedInstantiatedTypes"
schemaLocation="TestSchedInstantiatedTypes_v1_4_0.xsd"/>
<xs:simpleType name="DeclineMsg">
<xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="TestSchedMethodId">
<xs:restriction base="xs:anyURI"/>
</xs:simpleType>
<xs:simpleType name="InternalStatus">
<xs:restriction base="xs:NCName">
<xs:enumeration value="done"/>
<xs:enumeration value="drop"/>
<xs:enumeration value="inputError"/>
<xs:enumeration value="serverError"/>
</xs:restriction>
</xs:simpleType>
<xs:simpleType name="RequestStatus">
<xs:restriction base="xs:NCName">
<xs:enumeration value="undertake"/>
<xs:enumeration value="done"/>
<xs:enumeration value="decline"/>
<xs:enumeration value="drop"/>
<xs:enumeration value="fail"/>
</xs:restriction>
</xs:simpleType>
<xs:simpleType name="TaskStatus">
<xs:restriction base="xs:NCName">
<xs:enumeration value="done"/>
<xs:enumeration value="declined"/>
<xs:enumeration value="dropped"/>
<xs:enumeration value="failed"/>
</xs:restriction>
</xs:simpleType>
<xs:simpleType name="TestTaskId">
<xs:restriction base="xs:anyURI"/>
</xs:simpleType>
<xs:simpleType name="FileId">
<xs:restriction base="xs:anyURI"/>
```

```
</xs:simpleType>
<xs:simpleType name="TestContextName">
<xs:restriction base="xs:anyURI"/>
</xs:simpleType>
<xs:simpleType name="TestDirectiveId">
<xs:restriction base="xs:anyURI"/>
</xs:simpleType>
<xs:simpleType name="TransSpecId">
<xs:restriction base="xs:anyURI"/>
</xs:simpleType>
<xs:simpleType name="ModelId">
<xs:restriction base="xs:anyURI"/>
</xs:simpleType>
<xs:complexType name="TestModelMetadata">
<xs:sequence>
<xs:element name="modelId" type="tst:ModelId" minOccurs="0"/>
<xs:element name="directiveId" type="tst:TestDirectiveId" minOccurs="0"/>
<xs:element name="contextName"
type="tst:TestContextName" minOccurs="0" maxOccurs="unbounded"/>
<xs:element name="sourceModelId" type="tst:ModelId" minOccurs="0"/>
<xs:element name="transSpecId" type="tst:TransSpecId" minOccurs="0"/>
<xs:element name="targetModelId" type="tst:ModelId" minOccurs="0"/>
<xs:element name="sourceFileId" type="tst:FileId" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
<xs:element name="InitTestSchedIn" type="tst:InitTestSchedInMsgType"/>
<xs:complexType name="InitTestSchedInMsgType">
<xs:sequence>
<xs:element name="taskId" type="tst:TestTaskId"/>
<xs:element name="methodId" type="tst:TestSchedMethodId"/>
<xs:element name="metadata" type="tst:TestModelMetadata" minOccurs="0"/>
<xs:element ref="tsit:initTestSchedInputInfoset"/>
</xs:sequence>
</xs:complexType>
<xs:element name="InitTestSchedOut" type="tst:InitTestSchedOutMsgType"/>
<xs:complexType name="InitTestSchedOutMsgType">
<xs:sequence>
<xs:element name="reqStatus" type="tst:RequestStatus"/>
<xs:element name="reqDeclineMsg" type="tst:DeclineMsg" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
<xs:element name="NotifyTestSchedInitOutcomeIn"
type="tst:NotifyTestSchedInitOutcomeInMsgType"/>
<xs:complexType name="NotifyTestSchedInitOutcomeInMsgType">
<xs:sequence>
<xs:element name="taskId" type="tst:TestTaskId"/>
<xs:element name="methodId" type="tst:TestSchedMethodId"/>
<xs:element name="taskStatus" type="tst:TaskStatus"/>
<xs:element ref="tsit:initTestSchedOutputInfoset" minOccurs="0"/>
<xs:element ref="tsit:initTestSchedDeclineInfoset" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
<xs:element name="NotifyTestSchedInitOutcomeOut"
type="tst:NotifyTestSchedInitOutcomeOutMsgType"/>
<xs:complexType name="NotifyTestSchedInitOutcomeOutMsgType">
<xs:sequence>
<xs:element name="status" type="tst:InternalStatus"/>
</xs:sequence>
</xs:complexType>
<xs:element name="TestSchedIn" type="tst:TestSchedInMsgType"/>
<xs:complexType name="TestSchedInMsgType">
<xs:sequence>
```

```
<xs:element name="taskId" type="tst:TestTaskId"/>
<xs:element name="methodId" type="tst:TestSchedMethodId"/>
<xs:element name="metadata" type="tst:TestModelMetadata" minOccurs="0"/>
<xs:element ref="tsit:testSchedInputInfoset"/>
</xs:sequence>
</xs:complexType>
<xs:element name="TestSchedOut" type="tst:TestSchedOutMsgType"/>
<xs:complexType name="TestSchedOutMsgType">
<xs:sequence>
<xs:element name="reqStatus" type="tst:RequestStatus"/>
<xs:element name="reqDeclineMsg" type="tst:DeclineMsg" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
<xs:element name="NotifyTestSchedOutcomeIn" type="tst:NotifyTestSchedOutcomeInMsgType"/>
<xs:complexType name="NotifyTestSchedOutcomeInMsgType">
<xs:sequence>
<xs:element name="taskId" type="tst:TestTaskId"/>
<xs:element name="methodId" type="tst:TestSchedMethodId"/>
<xs:element name="taskStatus" type="tst:TaskStatus"/>
<xs:element ref="tsit:testSchedOutputInfoset" minOccurs="0"/>
<xs:element ref="tsit:testSchedDeclineInfoset" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
<xs:element name="NotifyTestSchedOutcomeOut" type="tst:NotifyTestSchedOutcomeOutMsgType"/>
<xs:complexType name="NotifyTestSchedOutcomeOutMsgType">
<xs:sequence>
<xs:element name="status" type="tst:InternalStatus"/>
</xs:sequence>
</xs:complexType>
<xs:element name="TestSchedAbortIn" type="tst:TestSchedAbortInMsgType"/>
<xs:complexType name="TestSchedAbortInMsgType">
<xs:sequence>
<xs:element name="taskId" type="tst:TestTaskId"/>
<xs:element name="methodId" type="tst:TestSchedMethodId"/>
</xs:sequence>
</xs:complexType>
<xs:element name="TestSchedAbortOut" type="tst:TestSchedAbortOutMsgType"/>
<xs:complexType name="TestSchedAbortOutMsgType">
<xs:sequence>
<xs:element name="reqStatus" type="tst:RequestStatus"/>
<xs:element name="reqDeclineMsg" type="tst:DeclineMsg" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
<xs:element name="NotifyTestSchedAbortOutcomeIn"
type="tst:NotifyTestSchedAbortOutcomeInMsgType"/>
<xs:complexType name="NotifyTestSchedAbortOutcomeInMsgType">
<xs:sequence>
<xs:element name="taskId" type="tst:TestTaskId"/>
<xs:element name="methodId" type="tst:TestSchedMethodId"/>
<xs:element name="taskStatus" type="tst:TaskStatus"/>
</xs:sequence>
</xs:complexType>
<xs:element name="NotifyTestSchedAbortOutcomeOut"
type="tst:NotifyTestSchedAbortOutcomeOutMsgType"/>
<xs:complexType name="NotifyTestSchedAbortOutcomeOutMsgType">
<xs:sequence>
<xs:element name="status" type="tst:InternalStatus"/>
</xs:sequence>
</xs:complexType>
</xs:schema>
</wsdl:types>
<wsdl:message name="InitTestSchedInMsg">
```

```
<wsdl:part name="content" element="tst:InitTestSchedIn"/>
</wsdl:message>
<wsdl:message name="InitTestSchedOutMsg">
<wsdl:part name="content" element="tst:InitTestSchedOut"/>
</wsdl:message>
<wsdl:message name="TestSchedInMsg">
<wsdl:part name="content" element="tst:TestSchedIn"/>
</wsdl:message>
<wsdl:message name="TestSchedOutMsg">
<wsdl:part name="content" element="tst:TestSchedOut"/>
</wsdl:message>
<wsdl:message name="NotifyTestSchedInitOutcomeInMsg">
<wsdl:part name="content" element="tst:NotifyTestSchedInitOutcomeIn"/>
</wsdl:message>
<wsdl:message name="NotifyTestSchedInitOutcomeOutMsg">
<wsdl:part name="content" element="tst:NotifyTestSchedInitOutcomeOut"/>
</wsdl:message>
<wsdl:message name="NotifyTestSchedOutcomeInMsg">
<wsdl:part name="content" element="tst:NotifyTestSchedOutcomeIn"/>
</wsdl:message>
<wsdl:message name="NotifyTestSchedOutcomeOutMsg">
<wsdl:part name="content" element="tst:NotifyTestSchedOutcomeOut"/>
</wsdl:message>
<wsdl:message name="TestSchedAbortInMsg">
<wsdl:part name="content" element="tst:TestSchedAbortIn"/>
</wsdl:message>
<wsdl:message name="TestSchedAbortOutMsg">
<wsdl:part name="content" element="tst:TestSchedAbortOut"/>
</wsdl:message>
<wsdl:message name="NotifyTestSchedAbortOutcomeInMsg">
<wsdl:part name="content" element="tst:NotifyTestSchedAbortOutcomeIn"/>
</wsdl:message>
<wsdl:message name="NotifyTestSchedAbortOutcomeOutMsg">
<wsdl:part name="content" element="tst:NotifyTestSchedAbortOutcomeOut"/>
</wsdl:message>
<wsdl:portType name="TestSchedConsumerPortType">
<wsdl:operation name="notifyTestSchedInitOutcome">
<wsdl:input name="TestSchedInitOutcomeInNotification"
message="tns:NotifyTestSchedInitOutcomeInMsg"/>
<wsdl:output name="TestSchedInitOutcomeOutNotification"
message="tns:NotifyTestSchedInitOutcomeOutMsg"/>
</wsdl:operation>
<wsdl:operation name="notifyTestSchedOutcome">
<wsdl:input name="TestSchedOutcomeInNotification"
message="tns:NotifyTestSchedOutcomeInMsg"/>
<wsdl:output name="TestSchedOutcomeOutNotification"
message="tns:NotifyTestSchedOutcomeOutMsg"/>
</wsdl:operation>
<wsdl:operation name="notifyTestSchedAbortOutcome">
<wsdl:input name="NotifyTestSchedAbortOutcomeInMsg"
message="tns:NotifyTestSchedAbortOutcomeInMsg"/>
<wsdl:output name="NotifyTestSchedAbortOutcomeOutMsg"
message="tns:NotifyTestSchedAbortOutcomeOutMsg"/>
</wsdl:operation>
</wsdl:portType>
<wsdl:portType name="TestSchedProviderPortType">
<wsdl:operation name="initTestScheduling">
<wsdl:input name="InitTestSchedInRequest" message="tns:InitTestSchedInMsg"/>
<wsdl:output name="InitTestSchedOutResponse" message="tns:InitTestSchedOutMsg"/>
</wsdl:operation>
<wsdl:operation name="requestTestScheduling">
<wsdl:input name="TestSchedInRequest" message="tns:TestSchedInMsg"/>
```

```
<wsdl:output name="TestSchedOutResponse" message="tns:TestSchedOutMsg"/>
</wsdl:operation>
<wsdl:operation name="abortTestSched">
<wsdl:input name="TestSchedAbortInMsg" message="tns:TestSchedAbortInMsg"/>
<wsdl:output name="TestSchedAbortOutMsg" message="tns:TestSchedAbortOutMsg"/>
</wsdl:operation>
</wsdl:portType>
<wsdl:binding name="TestSchedConsumerPortTypeSOAPBinding"
type="tns:TestSchedConsumerPortType">
<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="notifyTestSchedInitOutcome">
<soap:operation style="document"/>
<wsdl:input name="TestSchedInitOutcomeInNotification">
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output name="TestSchedInitOutcomeOutNotification">
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="notifyTestSchedOutcome">
<soap:operation style="document"/>
<wsdl:input name="TestSchedOutcomeInNotification">
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output name="TestSchedOutcomeOutNotification">
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="notifyTestSchedAbortOutcome">
<soap:operation style="document"/>
<wsdl:input name="NotifyTestSchedAbortOutcomeInMsg">
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output name="NotifyTestSchedAbortOutcomeOutMsg">
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:binding name="TestSchedProviderPortTypeSOAPBinding"
type="tns:TestSchedProviderPortType">
<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="initTestScheduling">
<soap:operation style="document"/>
<wsdl:input name="InitTestSchedInRequest">
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output name="InitTestSchedOutResponse">
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="requestTestScheduling">
<soap:operation style="document"/>
<wsdl:input name="TestSchedInRequest">
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output name="TestSchedOutResponse">
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="abortTestSched">
<soap:operation style="document"/>
<wsdl:input name="TestSchedAbortInMsg">
```

```
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output name="TestSchedAbortOutMsg">
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="TestSchedulingConsumerService">
<wsdl:port name="TestSchedulingConsumerSOAPPort"
binding="tns:TestSchedConsumerPortTypeSOAPBinding">
<soap:address location="http://www.midas-
project.eu/Core/API/TestSchedService/TestSchedulingConsumerSOAPPort"/>
</wsdl:port>
</wsdl:service>
<wsdl:service name="TestSchedulingProviderService">
<wsdl:port name="TestSchedulingProviderSOAPPort"
binding="tns:TestSchedProviderPortTypeSOAPBinding">
<soap:address location="http://www.midas-
project.eu/Core/API/TestSchedService/TestSchedulingProviderSOAPPort"/>
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

MYTESTSCHEDMETHODTYPES_1_4_0.XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.midas-
project.eu/TestMethod/PartnerNames/MyTestSchedMethodTypes" elementFormDefault="qualified"
attributeFormDefault="unqualified">
       <!-- Replace the following complex types with ones your test scheduling method expects
*    as input / output / decline parameters if any
-->
       <xs:complexType name="MyInitTestSchedMethodInputInfoset"/>
       <xs:complexType name="MyInitTestSchedMethodOutputInfoset"/>
       <xs:complexType name="MyInitTestSchedMethodDeclineInfoset"/>
       <xs:complexType name="MyTestSchedMethodInputInfoset"/>
       <xs:complexType name="MyTestSchedMethodOutputInfoset"/>
       <xs:complexType name="MyTestSchedMethodDeclineInfoset"/>
</xs:schema>
```

TESTSCHEDINSTANTIATEDTYPES_V1_4_0.XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:mtsmt="http://www.midas-project.eu/TestMethod/PartnerNames/MyTestSchedMethodTypes"
targetNamespace="http://www.midas-project.eu/Core/API/TestSchedInstantiatedTypes"
elementFormDefault="qualified" attributeFormDefault="unqualified">
<!--
Replace the namespace and the file name containing the types needed for your test scheduling
method
-->
<xs:import
namespace="http://www.midas-project.eu/TestMethod/PartnerNames/MyTestSchedMethodTypes"
schemaLocation="MyTestSchedMethodTypes_v1_4_0.xsd"/>
<!--
Change the types of the following elements with ones defined within the file you imported
```

```
-->
<xs:element name="initTestSchedInputInfoset" type="mtsmt:MyInitTestSchedMethodInputInfoset"/>
<xs:element name="initTestSchedOutputInfoset"
type="mtsmt:MyInitTestSchedMethodOutputInfoset"/>
<xs:element name="initTestSchedDeclineInfoset"
type="mtsmt:MyInitTestSchedMethodDeclineInfoset"/>
<xs:element name="testSchedInputInfoset" type="mtsmt:MyTestSchedMethodInputInfoset"/>
<xs:element name="testSchedOutputInfoset" type="mtsmt:MyTestSchedMethodOutputInfoset"/>
<xs:element name="testSchedDeclineInfoset" type="mtsmt:MyTestSchedMethodDeclineInfoset"/>
</xs:schema>
```

BIBLIOGRAPHY

[1] Aichernig, B. K., & Salas, P. A. P. (2005). Test case generation by OCL mutation and constraint solving. In *Fifth International Conference on Quality Software (QSIC 2005)*. IEEE.

[2] Allen, F. E. (1970). Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, (pp. 1-19). New York, NY, USA: ACM.

[3] Allen, D., & Darwiche, A. (2003). New advances in inference by recursive conditioning. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, UAI'03, (pp. 2-10). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[4] Askarunisa, A., Abirami, A. M., & MadhanMohan, S. (2010a). A test case reduction method for semantic based web services. In *Computing Communication and Networking Technologies (ICCCNT), 2010 International Conference on*, (pp. 1-7). IEEE.

[5] Askarunisa, A., Abirami, A. M., Punitha, K. A. J., Selvakumar, B. K., & Arunkumar, R. (2010b). Sequence-based techniques for black-box test case prioritisation for composite service testing. In Computational Intelligence and Computing Research (ICCIC), 2010 IEEE International Conference on, (pp. 1-4). IEEE.

[6] Askarunisa, A., Punitha, K. A. J., & Abirami, A. M. (2011). Black box test case prioritisation techniques for semantic based composite web services using OWL-s. In *Recent Trends in Information Technology (ICRTIT), 2011 International Conference on*, (pp. 1215-1220). IEEE.

[7] Athira, B., & Samuel, P. (2010). Web services regression test case prioritisation. In *Computer Information Systems and Industrial Management Applications (CISIM), 2010 International Conference on*, (pp. 438-443). IEEE.

[8] Bacchus, F., Dalmao, S., & Pitassi, T. (2003). Value elimination: Bayesian inference via backtracking search. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, UAI'03, (pp. 20-28). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[9] Bai, X., & Kenett, R. S. (2009). Risk-Based adaptive group testing of semantic web services. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, vol. 2, (pp. 485-490). IEEE.

[10]Baker, P., Bristow, P., Jervis, C., King, D., & Mitchell, B. (2003). Automatic generation of conformance tests from message sequence charts. In E. Sherratt (Ed.) *Telecommunications and beyond: The BroaderApplicability of SDL and MSC*, vol. 2599 of *Lecture Notes in Computer Science* , (pp. 170-198). Springer Berlin Heidelberg.

[11]Bartolini, C., Bertolino, A., Lonetti, F., & Marchetti, E. (2011). Approaches to functional, structural and security SOA testing. Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions, (p. 381).

[12]Bartolini, C., Bertolino, A., Marchetti, E., & Polini, A. (2009). WS-TAXI: A WSDL-based testing tool for web services. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*. IEEE.

[13]Bartolini, C., Bertolino, A., Elbaum, S., & Marchetti, E. (2009). Whitening SOA testing. In Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIG-

SOFT symposium on the foundations of software engineering, ESEC/FSE ’09, (pp. 161–170). New York, NY, USA: ACM.

[14]Beck, K. (2003). Test-driven development: by example. Addison-Wesley Professional.

[15]Becker, A., Bar-Yehuda, R., & Geiger, D. (2000). Randomized algorithms for the loop cutset problem. *Journal of Artificial Intelligence Research*, *12*.

[16]Bertolino, A., Marchetti, E., & Muccini, H. (2005). Introducing a reasonably complete and coherent approach for model-based testing. *Electronic Notes in Theoretical Computer Science*, *116*.

[17]Bertolino, A., De Angelis, G., & Polini, A. (2011). (role)CAST: A framework for on-line service testing. In J. Cordeiro, & J. Filipe (Eds.) *7th International Conference on Web Information Systems and Technologies (WABIST 2011)*.

[18]Besson, F. M., Leal, P. M. B., Kon, F., Goldman, A., & Milojicic, D. (2011). Towards automated testing of web service choreographies. In Proceedings of the 6th International Workshop on Automation of Software Test, AST ’11, (pp. 109–110). New York, NY, USA: ACM.

[19]Besson, F. M., Moura, P., Kon, F., & Milojicic, D. (2012). Rehearsal: a framework for automated testing of web service choreographies. In Brazilian Conference on Software: Theory and Practice.

[20]Bézivin, J. (2005). On the unification power of models. *Software and Systems Modeling*, *4* (2), 171-188.

[21]Bichier, M., & Lin, K. J. (2006). Service-oriented computing. *Computer*, *39* (3), 99-101.

[22]Bobbio, A., Portinale, L., Minichino, M., & Ciancamerla, E. (2001). Improving the analysis of dependable systems by mapping fault trees into Bayesian networks. Reliability Engineering & System Safety, 71(3).

[23]Bodlaender, H. L. (1993). A tourist guide through treewidth. *Acta Cybernetica*, *11* , 1-23.

[24]Bozkurt, M., Harman, M., & Hassoun, Y. (2013). Testing and verification in service-oriented architecture: a survey. *Softw. Test. Verif. Reliab.*, *23* (4), 261-313.

[25]Bozkurt, M. (2013). Cost-aware Pareto optimal test suite minimisation for service-centric systems. In Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, GECCO '13, (pp. 1429-1436). New York, NY, USA: ACM.

[26]BPEL_1_1 (2003). Business Process Execution Language for Web Services Version 1.1.

[27]BPEL_2_0 (2007). Web Services Business Process Execution Language Version 2.0. OASIS Standard 11 April 2007. Organization for the Advancement of Structured Information Standards.

[28]Breese, J. S., & Heckerman, D. (1996). Decision-theoretic troubleshooting: A framework for repair and experiment. In Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence. Morgan Kaufmann Publishers Inc.

[29]Brian Arthur, W. (2011). The second economy. *McKinsey Quarterly*.

[30]Briand, L. C., Labiche, Y., & Sun, H. (2003). Investigating the use of analysis contracts to improve the testability of object-oriented code. *Softw: Pract. Exper.*, *33* (7), 637-672.

[31]Brucker, A., Krieger, M., Longuet, D., & Wolff, B. (2011). A Specification-Based test case generation method for UML/OCL. In J. Dingel, & A. Solberg (Eds.) Models in Software Engineering, vol.

6627 of Lecture Notes in Computer Science, chap. 33, (pp. 334–348). Berlin, Heidelberg: Springer Berlin Heidelberg.

[32]Bucchiarone, A., Melgratti, H., & Severoni, F. (2007). Testing service composition. In *Proceedings of the 8th Argentine Symposium on Software Engineering (ASSE'07)*.

[33]Candea, G., Bucur, S., & Zamfir, C. (2010). Automated software testing as a service. In Proceedings of the 1st ACM symposium on Cloud computing. ACM.

[34]Canfora, G., & Di Penta, M. (2009). Service-Oriented architectures testing: A survey. In Software Engineering. Springer-Verlag.

[35]Chalin, P. (2006). Are practitioners writing contracts? In M. Butler, C. Jones, A. Romanovsky, & E. Troubitsyna (Eds.) *Rigorous Development of Complex Fault-Tolerant Systems*, vol. 4157 of *Lecture Notes in Computer Science* , (pp. 100-113). Springer Berlin Heidelberg.

[36]Chavira, M., & Darwiche, A. (2005). Compiling Bayesian networks with local structure. In *IJCAI*, vol. 5, (pp. 1306-1312).

[37]Chavira, M., Darwiche, A., & Jaeger, M. (2006). Compiling relational Bayesian networks for exact inference. *International Journal of Approximate Reasoning*, *42* (1-2), 4-20.

[38]Chavira, M., & Darwiche, A. (2007). Compiling Bayesian networks using variable elimination. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*.

[39]Chen, L., Wang, Z., Xu, L., Lu, H., & Xu, B. (2010). Test case prioritisation for web service regression testing. In *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*, (pp. 173-178). IEEE.

[40]Coolen, F. P. A., Goldstein, M., & Wooff, D. A. (2007). Using bayesian statistics to support testing of software systems. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, *221* (1), 85-93.

[41]Cooper, G.F. (1987). *Probabilistic inference using belief networks is np-hard.* Technical Report KSL-87-27, Medical computer Sience, Stanford University, Stanford, California.

[42]Cooper, G. F., Horvitz, E. J., & Heckerman, D. E. (1988). A method for temporal probabilistic reasoning. Tech. rep., Technical Report KSL 88-30, Medical Computer Science, Stanford University, Stanford, CA.

[43]Cooper, G. F. (1990). The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, *42* (2-3), 393-405.

[44]Cooper, G., Horvitz, E., & Curry, R. (1998). Conceptual design of goal understanding systems: Investigation of temporal reasoning under uncertainty. *Decision Theory & Adaptive Systems Group, Microsoft Research. Microsoft Corp. Redmond, WA*.

[45]Cozman, F. G. (2000). Generalizing variable elimination in Bayesian networks. In *Workshop on Probabilistic Reasoning in Artificial Intelligence*, (pp. 21-26).

[46]Cozman, F. G. (2004). Axiomatizing noisy-OR. In *ECAI*, vol. 16, (p. 979).

[47]Dagum, P., & Luby, M. (1993). Approximating probabilistic inference in bayesian belief networks is NP-hard. *Artificial Intelligence*, *60* (1), 141-153.

[48]Darwiche, A. (1998). Dynamic jointrees. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, UAI'98, (pp. 97-104). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[49]Darwiche, A. (2001). Recursive conditioning. Artificial Intelligence, 126(1-2), 5–41.

[50]Darwiche, A. (2003). A differential approach to inference in Bayesian networks. Journal of the ACM (JACM), 50(3).

[51]Darwiche, A. (2009). *Modeling and Reasoning with Bayesian Networks.* Cambridge University Press.

[52]Darwiche, A. (2010). *Inference in Bayesian networks: A historical perspective*. In Hector Geffner Rina Dechter and Joe Halpern, editors, Heuristics, Probability and causality. A tribute to Judea Pearl. College publications.

[53] Dechter, R. (1996). Bucket elimination: A unifying framework for probabilistic inference. In Proceedings of the 12th Annual Conference on Uncertainty in AI, pages 211-219.

[54]Dechter, R. (1998). Bucket elimination: A unifying framework for probabilistic inference. In M. Jordan (Ed.) Learning in Graphical Models, vol. 89 of NATO ASI Series, (pp. 75–104). Springer Netherlands.

[55]Dechter, R., & Pearl, J. (1989). Tree clustering for constraint networks. Artificial Intelligence, 38(3), 353–366.

[56]De Rosa, F., Maesano, A. P., & Maesano, L. (2012). Service contract clauses as business rules. In*24th International Conference on Software & System Engineering and their Applications (ICSSEA '12), Paris - Oct. 23-25 (2012).*

[57]De Rosa, F., Maesano, A. P., & Maesano, L. (2013). TTCN4SOA™: a TTCN-3 architecture and framework for SOA testing. In *1st User Conference on Advanced Automated Testing*. UCAAT 2013.

[58]De Rosa, F., Hillah, L. M., Maesano, A. P., & Maesano, L. (2014a). SAUT construction specification - service component architecture for services architecture under test (SCA4SAUT). Tech. Rep. D3.3 - SCA4SAUT V.1.1, MIDAS. EC FP7 Project # 318786.

[59]De Rosa, F., Hillah, L. M., Maesano, A. P., & Maesano, L. (2014b). Service component protocol state machine specification - state chart XML for protocol state machine (SCXML4PSM). Tech. Rep. D3.3 - SCXML4PSM V.1.0, MIDAS. EC FP7 Project # 318786.

[60]De Rosa, F., Hillah, L. M., Maesano, A. P., & Maesano, L. (2014c). Test suite definition / test suite specification (TSD/TS). Tech. Rep. D3.3 - TSD/TS V.0.8, MIDAS. EC FP7 Project # 318786.

[61]De Rosa, F., Maesano, A. P., & Maesano, L. (2014d). TTCN-3 library for service functional test specification (TTCN-3_Lib). Tech. Rep. D4.X - TTCN-3_Lib V.0.5, MIDAS. EC FP7 Project # 318786.

[62]Dias Neto, A. C., Subramanyan, R., Vieira, M., & Travassos, G. H. (2007). A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, (pp. 31-36). New York, NY, USA: ACM.

[63]Dietz, J. L. G. (2010). Enterprise Ontology: Theory and Methodology. Springer Publishing Company, Incorporated, 1st ed.

[64] Díez, F. (1996). Local conditioning in Bayesian networks. *Artificial Intelligence*, *87* (1-2), 1-20.

[65]Di Penta, M., Bruno, M., Esposito, G., Mazza, V., & Canfora, G. (2007). Web services regression testing. In L. Baresi, & E. Di Nitto (Eds.) *Test and Analysis of Web Services*, (pp. 205-234). Springer Berlin Heidelberg.

[66]Do, H., & Rothermel, G. (2005). A controlled experiment assessing test case prioritisation techniques via mutation faults. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, (pp. 411-420). IEEE.

[67]Do, H., & Rothermel, G. (2006). On the use of mutation faults in empirical assessments of test case prioritisation techniques.*Software Engineering, IEEE Transactions on*, *32* (9), 733-752.

[68]Dong, W. (2008). Test case reduction technique for BPEL-based testing. In Electronic Commerce and Security, 2008 International Symposium on, (pp. 814-817). IEEE.

[69]Draper, D. L. (1995). Clustering without (thinking about) triangulation. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, UAI'95, (pp. 125-133). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[70]Elbaum, S., Malishevsky, A. G., & Rothermel, G. (2002). Test case prioritisation: a family of empirical studies. *Software Engineering, IEEE Transactions on*, *28* (2), 159-182.

[71]Endo, A. T., & Simao, A. S. (2010). A systematic review on formal testing approaches for web services. In *Proceedings of the 4th Brazilian Workshop on Systematic and Automated Software Testing (SAST'10)*, (p. 89).

[72]Endo, A. T., Linschulte, M., da Silva Simão, A., & Do Rocio Senger De Souza, S. (2010). Event- and Coverage-Based testing of web services. In *Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on*, (pp. 62-69). IEEE.

[73]ETSI EG 202 810 (2010). Methods for Testing and Specification (MTS); Automated Interoperability Testing; Methodology and Framework. ETSI EG 202 810 V1.1.1 (2010-03). European Telecommunications Standards Institute.

[74]ETSI ES 201 873-9 V4.5.1 (2013-04). Methods for testing and specification (MTS);the testing and test control notation version 3; part 9: Using XML schemas with TTCN-3. Tech. Rep. RES/MTS-201873-9 T3ed451XML, European Telecommunications Standards Institute.

[75]Fay, A., & Jaffray, J.-Y. (2000). A justification of local conditioning in Bayesian networks. *International Journal of Approximate Reasoning*, *24* (1), 59-81.

[76] Fenton, N., Krause, P., & Neil, M. (2002). Probabilistic modelling for software quality control. *Journal of Applied Non-Classical Logics*, *12* (2), 173-188.

[77]Fielding, R. T., & Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, *2* (2), 115-150.

[78]Floss, B., & Tilley, S. (2013). Software testing as a service: An academic research perspective. In 2013 IEEE Seventh International Symposium on Service-Oriented System Engineering, (pp. 421–424). IEEE.

[79]Garey, M. R., & Johnson, D. S. (1979). Computers and intractability: a guide to the theory of NP-completeness. *WH Freeman & Co., San Francisco*.

[80]Gras, J. J., Gupta, R., & Perez-Minana, E. (2006). Generating a test strategy with bayesian networks and common sense. In *Testing: Academic and Industrial Conference - Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, (pp. 29-40). IEEE.

[81]Grant, K., & Horsch, M. (2005). Conditioning graphs: Practical structures for inference in Bayesian networks. In S. Zhang, & R. Jarvis (Eds.) *AI 2005: Advances in Artificial Intelligence*, vol. 3809 of *Lecture Notes in Computer Science* , (pp. 49-59). Springer Berlin Heidelberg.

[82]Grant, K. (2010). Efficient indexing for recursive conditioning algorithms. In *FLAIRS Conference*.

[83]Gray, J. (1981). The transaction concept: Virtues and limitations. In *VLDB*, vol. 81, (pp. 144-154).

[84]Hao, J. X., & Orlin, J. B. (1994). A faster algorithm for finding the minimum cut in a directed graph. *Journal of Algorithms*, *17* (3), 424-446.

[85]Harel, D., & Politi, M. (1998). *Modeling Reactive Systems with Statecharts: The Statemate Approach*. New York, NY, USA: McGraw-Hill, Inc., 1st ed.

[86]Heckel, R., & Lohmann, M. (2005). Towards contract-based testing of web services. *Electronic Notes in Theoretical Computer Science*, *116* , 145-156.

[87]Heckel, R., & Mariani, L. (2005). Automatic conformance testing of web services. In M. Cerioli (Ed.) *Fundamental Approaches to Software Engineering*, vol. 3442 of *Lecture Notes in Computer Science* , (pp. 34-48+). Springer Berlin Heidelberg.

[88]Heckerman, D., Breese, J., & Rommelse, K. (1994). Troubleshooting under uncertainty. *Communications of the ACM*, (pp. 121-130).

[89]Heckerman, D., Breese, J. S., & Rommelse, K. (1995). Decision-theoretic troubleshooting. *Commun. ACM*, *38* (3), 49-57.

[90]Heckerman, D., & Breese, J. S. (1996). Causal independence for probability assessment and inference using bayesian networks. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, *26* (6), 826-831.

[91]Hedayat, A.S., Sloane, N.A.J. and Stufken, J. (1999) Orthogonal Arrays: Theory and Applications, Springer, New York.

[92]Herbold, S., De Rosa, F., Harms, P., Hillah, L. M., Maesano, A. P., Maesano, L., John, C., Schneider, M., & Wendland, M. F. (2013). MIDAS models and methods for the automated generation of functional, security, and usage-based test cases. Tech. Rep. D3.1, EC FP7 - MIDAS Project # 318786.

[93]Hierons, R. M., Bogdanov, K., Bowen, J. P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A. J. H., Vilkomir, S., Woodward, M. R., & Zedan, H. (2009). Using formal specifications to support testing. *ACM Comput. Surv.*, *41* (2), 1-76.

[94]Huang, H. Y., Liu, H. H., Li, Z. J., & Zhu, J. (2008). Surrogate: A simulation apparatus for continuous integration testing in service oriented architecture. In *Services Computing, 2008. SCC &#039;08. IEEE International Conference on*, vol. 2, (pp. 223-230). IEEE.

[95]Ilieva, S., Pavlov, V., & Manova, I. (2010). A composable framework for test automation of Service-Based applications. In Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the, (pp. 286–291). IEEE.

[96]ISTQB GTB (2013). ISTQB/GTB standard glossary for testing terms. v2.2. Tech. rep.

[97] Jensen, F. V. (1988). Junction trees and decomposable hypergraphs. *JUDEX Re search Report, Aalborg, Denmark*.

[98]Jensen, F. V., Kjærulff, U., Kristiansen, B., Langseth, H., Skaanning, C., Vomlel, J., & Vomlelová, M. (2001a). The SACSO methodology for troubleshooting complex systems. *AI EDAM*, *15* , 321-333.

[99]Jensen, F. V., Skaanning, C., & Kjærulff, U. (2001b). The SACSO system for troubleshooting of printing systems. In *Scandinavian Conference on Artificial Intelligence - SCAI '01*, (pp. 67-79). IOS Press.

[100] John, C., Wendland, M. F., De Rosa, F., Hoffmann, A., Maesano, L., & Schneider, M. (2013). TTCN-3 state of the art and selected tools. Tech. Rep. D4.1, EC FP7 - MIDAS Project # 318786.

[101] Jordan, M. I. (Ed.) (1998). *Learning in Graphical Models*, vol. 89. Springer.

[102] JSON (2013). The JSON data interchange format. Tech. Rep. Standard ECMA-404 1st Edition / October 2013, ECMA.

[103] Khan, T., & Heckel, R. (2011). On Model-Based regression testing of Web-Services using dependency analysis of visual contracts. In D. Giannakopoulou, & F. Orejas (Eds.) Fundamental Approaches to Software Engineering, vol. 6603 of Lecture Notes in Computer Science , (pp. 341-355). Springer Berlin Heidelberg.

[104] Khinchin, A. I. (1957). *Mathematical foundations of information theory* (Vol. 434). Courier Dover Publications.

[105] Kim, J. H., & Pearl, J. (1983). A computational model for causal and diagnostic reasoning in inference systems. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'83, (pp. 190-193). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[106] Kjærulff, U. (1990). Triangulation of graphs - algorithms giving small total state space. Tech. rep.

[107] Koller, D. and Friedman, N. (2009) *Probabilistic Graphical Models: Principles and Techniques.* MIT Press.

[108] Lauritzen, S. L., & Spiegelhalter, D. J. (1990). Readings in uncertain reasoning. chap. Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems, (pp. 415-448). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[109] Lepar, V., & Shenoy, P. P. (1998). A comparison of Lauritzen-Spiegelhalter, Hugin, and Shenoy-Shafer architectures for computing marginals of probability distributions. In Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI-98, (pp. 328–337). Morgan Kaufmann.

[110] Lewis, J. P. (2001). Limits to software estimation. *SIGSOFT Softw. Eng. Notes*, *26* (4), 54-59.

[111] Li, B., Qiu, D., Ji, S., & Wang, D. (2010). Automatic test case selection and generation for regression testing of composite service based on extensible BPEL flow graph. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, (pp. 1-10). IEEE.

[112] Li, B., Qiu, D., Leung, H., & Wang, D. (2012). Automatic test case selection for regression testing of composite service based on extensible BPEL flow graph. Journal of Systems and Software, 85 (6), 1300-1324.

[113] Li, Z., Sun, W., Jiang, Z. B., & Zhang, X. (2005). BPEL4WS unit testing: Framework and implementation. In Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on. IEEE.

[114] Li, Z. J., Tan, H. F., Liu, H. H., Zhu, J., & Mitsumori, N. M. (2008). Business-process-driven gray-box SOA testing. IBM Systems Journal, 47 (3), 457-472.

[115] Liu, H., Li, Z., Zhu, J., & Tan, H. (2007). Business process regression testing. In B. Krämer, K.-J. Lin, & P. Narasimhan (Eds.) *Service-Oriented Computing – ICSOC 2007*, vol. 4749 of*Lecture Notes in Computer Science* , (pp. 157-168). Springer Berlin Heidelberg.

[116] Liu, H., Li, Z., Zhu, J., Tan, H., & Huang, H. (2009). A unified test framework for continuous integration testing of SOA solutions. In*Web Services, 2009. ICWS 2009. IEEE International Conference on*, (pp. 880-887). IEEE.

[117] Madsen, A. L., & Jensen, F. V. (1999). Lazy propagation: A junction tree inference algorithm based on lazy evaluation. Artificial Intelligence, 113(1-2), 203–245.

[118] Maesano, A. P., & De Rosa, F. (2011). Towards stochastic inference driven SOA testing - Bayesian networks for services architecture testing (BN4SAT). In IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE 2011), Nov 29 - Dec 2, 2011 Hiroshima, Japan.

[119] Maesano, A. P., De Rosa, F., Maesano, L., & Wuillemin, P. H. (2011). Steps towards model-based, inference-driven SOA testing. In 23rd International Conference on Software & System Engineering and their Applications (ICSSEA ’11), Paris - Nov 29 - Dec 1 (2011).

[120] Maesano, A. P., De Rosa, F., Herbold, S., Harms, P., Maesano, L., & Wendland, M. F. (2013). Methods and tools for the intelligent planning and scheduling of test campaigns. Tech. Rep. D5.1, EC FP7 - MIDAS Project # 318786.

[121] Maesano, L., & De Rosa, F. (2010). simpleSOAD® 2.0 - architecture & governance. In 22nd International Conference on Software & Systems Engineering and their Applications (ICSSEA 2010) - Paris Dec 7-9, 2010.

[122] Maesano, L., De Rosa, F., Harms, P., Herbold, S., Hoffman, A., John, C., Schneider, M., & Wenland, M. F. (2013a). Requirements for automatically testable services and services architecture. Tech. Rep. D2.1, EC FP7 - MIDAS Project # 318786.

[123] Maesano, L., De Rosa, F., Hoffmann, A., John, C., Lettere, M., Schneider, M., & Wendland, M. F. (2013b). Architecture and specifications of the MIDAS framework and platform. Tech. Rep. D2.2, EC FP7 - MIDAS Project # 318786.

[124] Maesano, L. (2013). Automated testing as a service on cloud. In *25th International Conference on Software & Systems Engineering and their Applications*. ICSSEA '13 - Invited talk.

[125] Maesano, L., De Rosa, F., García, L., Barcelona, M. A., Di Bona, S., Lettere, M., & Guerri, D. (2014). Specifications and adaptation of the selected pilot systems for the MIDAS platform. Tech. Rep. D7.1, MIDAS Project.

[126] Marconi, A., Pistore, M., & Traverso, P. (2006). Specifying data-flow requirements for the automated composition of web services. In SEFM 2006. Fourth IEEE International Conference on Software Engineering and Formal Methods, 2006. IEEE.

[127] Mayer, P., & Lübke, D. (2006). Towards a BPEL unit testing framework. In*Proceedings of the 2006 Workshop on Testing, Analysis, and Verification of Web Services and Applications*, TAV-WEB '06, (pp. 33-42). New York, NY, USA: ACM.

[128] Mei, L., Chan, W. K., & Tse, T. H. (2008). Data flow testing of service-oriented workflow applications. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, (pp. 371-380). IEEE.

[129] Mei, L., Chan, W. K., & Tse, T. H. (2009a). Data flow testing of service choreography. In Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE '09, (pp. 151–160). New York, NY, USA: ACM.

[130] Mei, L., Chan, W. K., Tse, T. H., & Merkel, R. G. (2009b). Tag-Based techniques for Black-Box test case prioritisation for service testing. In *Quality Software, 2009. QSIC '09. 9th International Conference on*, (pp. 21-30). IEEE.

[131] Mei, L., Zhang, Z., Chan, W. K., & Tse, T. H. (2009c). Test case prioritisation for regression testing of service-oriented business applications. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, (pp. 901-910). New York, NY, USA: ACM.

[132] Mei, L., Chan, W. K., Tse, T. H., & Merkel, R. G. (2011). XML-manipulating test case prioritisation for XML-manipulating services. Journal of Systems and Software, 84 (4), 603-619.

[133] Mei, L., Zhai, K., Jiang, B., Chan, W. K., & Tse, T. H. (2012). Preemptive regression test scheduling strategies: A new testing approach to thriving on the volatile service environments. In Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual, (pp. 72-81). IEEE.

[134] Mei, L., Cai, Y., Jia, C., Jiang, B., & Chan, W. K. (2013a). Prioritizing structurally complex test pairs for validating WS-BPEL evolutions. In Web Services (ICWS), 2013 IEEE 20th International Conference on, (pp. 147-154). IEEE.

[135] Mei, L., Cai, Y., Jia, C., Jiang, B., & Chan, W. K. (2013b). Test pair selection for test case prioritisation in regression testing for WS-BPEL programs. International Journal of Web Services Research (IJWSR), 10 (1), 73-102.

[136] Meyer, B. (1992). Applying 'design by contract'. Computer, 25(10), (pp. 40–51).

[137] Meyer, B. (2008). Seven principles of software testing. *Computer*, *41* (8), 99-101.

[138] Mirarab, S., & Tahvildari, L. (2007). A prioritisation approach for software test cases based on bayesian networks. In M. Dwyer, & A. Lopes (Eds.) *Fundamental Approaches to Software Engineering*, vol. 4422 of*Lecture Notes in Computer Science* , (pp. 276-290). Springer Berlin / Heidelberg.

[139] Namin, A. S., & Sridharan, M. (2010). Bayesian reasoning for software testing. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, (pp. 349-354). New York, NY, USA: ACM.

[140] Nguyen, C. D., Marchetto, A., & Tonella, P. (2011). Change sensitivity based prioritisation for audit testing of webservice compositions. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, (pp. 357-365). IEEE.

[141] Nielsen, T. D., Wuillemin, P. H., Jensen, F. V., & Kjærulff, U. (2000). Using ROBDDs for inference in Bayesian networks with troubleshooting as an example. In Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence. Morgan Kaufmann Publishers Inc.

[142] NIST-02-3 (2002). *The Economic Impact of Inadequate Infrastructure for Software Testing*. Planning Report 02-3. National Institute Of Standards & Technology.

[143] OCL 2 3 1 (2012). OMG Object Constraint Language (OCL) Version 2.3.1. formal/2012-01-01. Object Management Group.

[144] Palacios, M., García-Fanjul, J., & Tuya, J. (2011). Testing in service oriented architectures with dynamic binding: A mapping study. *Information and Software Technology*, *53* (3), 171-189.

[145] Papazoglou, M. P. (2003). Service-oriented computing: concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, vol. 0, (pp. 3-12). Los Alamitos, CA, USA: IEEE.

[146] Papazoglou, M. P., Traverso, P., Dustdar, S., & Leymann, F. (2008). Service-oriented computing: A research roadmap. *Int. J. Coop. Info. Syst.*, *17* (02), 223-255.

[147] Parnas, D. L., Clements, P. C., & Weiss, D. M. (1983). Enhancing reusability with information hiding. *Tutorial: Software Reusability*, (pp. 83-90).

[148] Pearl, J. (1982). Reverend Bayes on inference engines: a distributed hierarchical approach. In *Proceedings of the National Conference on Artificial Intelligence*, (pp. 133-136).

[149] Pearl, J. (1986). Fusion, propagation, and structuring in belief networks. *Artificial Intelligence*, *29* (3), 241-288.

[150] Pearl, J. (1988). Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[151] Periaswamy, S. V., & McDaid, K. (2006). Bayesian belief networks for test driven development. In *Proceedings of World Academy of Science, Engineering and technology*, vol. 11.

[152] Peyton, L., Stepien, B., & Seguin, P. (2008). Integration testing of composite applications. In *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, (p. 96). IEEE.

[153] Qiu, D., Li, B., Ji, S., & Leung, H. (2014). Regression testing of web service: A systematic mapping study. *ACM Comput. Surv.*, *47* (2).

[154] Rapaport, W. J. (2012). Intensionality vs. intentionality.
URL http://www.cse.buffalo.edu/\~{}rapaport/intensional.html

[155] Rees, K., Coolen, F., Goldstein, M., & Wooff, D. (2001). Managing the uncertainties of software testing: a Bayesian approach. Qual. Reliab. Engng. Int., 17(3), 191–203.

[156] Rehman, M. J., Jabeen, F., Bertolino, A., & Polini, A. (2007). Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability*,*17* (2).

[157] RLUS 1 0 1 (2011). OMG Retrieve, Locate, and Update Service (RLUS) Specification - Version 1.0.1. Object Management Group. Formal/2011-07-02.

[158] Robertson, N., & Seymour, P. D. (1986). Graph minors. II. Algorithmic aspects of tree-width. *Journal of algorithms*, *7*(3), 309-322.

[159] Rose, D. J., Tarjan, R. E., & Lueker, G. S. (1976). Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on computing*, *5*(2), 266-283.

[160] Rothermel, G., & Harrold, M. J. (1997). A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, *6*(2), 173-210.

[161] Rothermel, G., Untch, R. H., Chu, C., & Harrold, M. J. (2001). Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, *27* (10), 929-948.

[162] Ruth, M. E., & Tu, S. (2007a). A safe regression test selection technique for web services. In *Internet and Web Applications and Services, 2007. ICIW '07. Second International Conference on*, (p. 47). IEEE.

[163] Ruth, M. E., & Tu, S. (2007b). Towards automating regression test selection for web services. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, (pp. 1265-1266). New York, NY, USA: ACM.

[164] Ruth, M. E., Oh, S., Loup, A., Horton, B., Gallet, O., Mata, M., & Tu, S. (2007). Towards automatic regression test selection for web services. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, vol. 2, (pp. 729-736). IEEE.

[165] Ruth, M. E. (2008). Concurrency in a decentralized automatic regression test selection framework for web services. In Proceedings of the 15th ACM Mardi Gras Conference: From Lightweight Mash-ups to Lambda Grids: Understanding the Spectrum of Distributed Computing Requirements, Applications, Tools, Infrastructures, Interoperability, and the Incremental Adoption of Key Capabilities, MG '08. New York, NY, USA: ACM.

[166] Ruth, M. E. (2011). Employing Privacy-Preserving techniques to protect Control-Flow graphs in a decentralized, End-to-End regression test selection framework for web services. In Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on, (pp. 139-148). IEEE.

[167] Ruth, M., & Rayford, C. (2011). A privacy-aware, end-to-end, CFG-based regression test selection framework for web services using only local information. In Applications of Digital Information and Web Technologies (ICADIWT), 2011 Fourth International Conference on the, (pp. 13-18). IEEE.

[168] SCA_AM_V1_2 (2011). Service component architecture assembly model specification version 1.2. Tech. rep., OASIS. OASIS Committee Specification Draft 01.

[169] Schieferdecker, I., & Stepien, B. (2003). Automated testing of XML/SOAP based web services. In K. Irmscher, & K.-P. Fähnrich (Eds.) *Kommunikation in Verteilten Systemen (KiVS)*, Informatik aktuell, (pp. 43-54). Springer Berlin Heidelberg.

[170] Schieferdecker, I. (2012). Model-Based testing. *IEEE Software*, *29* (1), 14-18.

[171] SCXML (2014). State chart XML (SCXML): State machine notation for control abstraction. Tech. Rep. W3C Last Call Working Draft 29 May 2014, W3C.

[172] Shachter, R. D., D'Ambrosio, B., & Del Favero, B. A. (1990). Symbolic probabilistic inference in belief networks. In Proceedings of the eighth National conference on Artificial intelligence - Volume 1, AAAI'90, (pp. 126–131). AAAI Press.

[173] Shafer, G. (1996). Probabilistic expert systems, vol. 67. SIAM.

[174] Shafer, G. R., & Shenoy, P. P. (1990). Probability propagation. *Annals of Mathematics and Artificial Intelligence*, *2* (1-4), 327-351.

[175] Shafique, M., & Labiche, Y. (2010). A systematic review of model based testing tool support. Tech. rep., Technical Report SCE-10-04, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada.

[176] Shenoy, P. P., & Shafer, G. R. (1986). Propagating belief functions with local computations. *IEEE Expert*, *1* (3), 43-52.

[177] Shenoy, P. P., & Shafer, G. R. (1988). An axiomatic framework for bayesian and belief-function propagation. In *Proceedings of the Fourth Conference on Uncertainty in Artificial Intelligence (UAI1988)*.

[178] Shibata, Y. (1988). On the tree representation of chordal graphs. *Journal of Graph Theory*, *12*(3), 421-428.

[179] Singh, M. P., & Huhns, M. N. (2006). *Service-oriented computing: semantics, processes, agents*. John Wiley & Sons.

[180] Skaanning, C., Jensen, F., & Kjærulff, U. (2000). Printer troubleshooting using bayesian networks. In R. Logananthara, G. Palm, & M. Ali (Eds.) *Intelligent Problem Solving. Methodologies and Approaches*, vol. 1821 of *Lecture Notes in Computer Science* , (pp. 367-380). Springer Berlin Heidelberg.

[181] SoaML 1 0 1 (2012). Service Oriented Architecture Modeling Language (SoaML) Specification, Version 1.0.1. Object Management Group.

[182] SOAP_1_1 (2000). Simple object access protocol (SOAP) 1.1. Tech. rep. W3C Note 08 May 2000.

[183] Sprenkle, S., Sampath, S., Gibson, E., Pollock, L., & Souter, A. (2005). An empirical comparison of test suite reduction techniques for user-session-based testing of web applications. In Software Maintenance, 2005. ICSM&#039;05. Proceedings of the 21st IEEE International Conference on, (pp. 587–596). IEEE.

[184] Stefanescu, A., Wieczorek, S., & Kirshin, A. (2009). MBT4Chor: A Model-Based testing approach for service choreographies. In R. Paige, A. Hartman, & A. Rensink (Eds.) *Model Driven Architecture - Foundations and Applications*, vol. 5562 of *Lecture Notes in Computer Science* , (pp. 313-324). Springer Berlin Heidelberg.

[185] Stefanescu, A., Wendland, M. F., & Wieczorek, S. (2010). Using the UML testing profile for enterprise service choreographies. In*Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on*, (pp. 12-19). IEEE.

[186] Stepien, B., Peyton, L., & Xiong, P. (2008). Framework testing of web applications using TTCN-3. *International Journal on Software Tools for Technology Transfer*, *10*(4), 371-381.

[187] Suermondt, H. J., & Cooper, G. F. (1990). Probabilistic inference in multiply connected belief networks using loop cutsets. *International Journal of Approximate Reasoning*, *4* (4), 283-306.

[188] Takanen, A., Demott, J. D., & Miller, C. (2008). Fuzzing for software security testing and quality assurance [electronic resource]. Artech House.

[189] Tarhini, A., Fouchal, H., & Mansour, N. (2006a). Regression testing web services-based applications. In *Computer Systems and Applications, 2006. IEEE International Conference on.*, (pp. 163-170). IEEE.

[190] Tsai, W. T., Chen, Y., Cao, Z., Bai, X., Huang, H., & Paul, R. (2004). Testing web services using progressive group testing. In C.-H. Chi, & K.-Y. Lam (Eds.) *Content Computing*, vol. 3309 of *Lecture Notes in Computer Science* , (pp. 314-322). Springer Berlin Heidelberg.

[191] Tsai, W. T., Chen, Y., Paul, R., Huang, H., Zhou, X., & Wei, X. (2005). Adaptive testing, oracle generation, and test case ranking for web services. In *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, vol. 1, (pp. 101-106 Vol. 2). IEEE.

[192] Tsai, W. T., Zhou, X., Paul, R. A., Chen, Y., & Bai, X. (2007). A coverage relationship model for test case selection and ranking for multi-version software. In *High Assurance Systems Engineering Symposium, 2007. HASE '07. 10th IEEE*, (pp. 105-112). IEEE.

[193] Tsai, W. T., Xinyu, Z., Yinong, C., & Xiaoying, B. (2008). On testing and evaluating Service-Oriented software. *Computer*, *41* (8), 40-46.

[194] Tsai, W.-T., Zhou, X., Paul, R., Chen, Y., & Bai, X. (2009). A coverage relationship model for test case selection and ranking for multi-version software. In L.-J. Zhang, R. Paul, & J. Dong (Eds.) *High Assurance Services Computing*, (pp. 285-311). Springer US.

[195] UML_2_4_1 (2011). OMG unified modeling LanguageTM (OMG UML), superstructure version 2.4.1. Tech. Rep. formal/2011-08-06, Object Management Group.

[196] Utting, M., Pretschner, A., & Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, *22* (5), 297-312.

[197] UTP_1_2 (2012). UML testing profile (UTP) version 1.2. Tech. Rep. formal/2013-04-03, Object Management Group.

[198] Vinoski, S. (2002). Web services interaction models - Current practice. *Internet Computing, IEEE*, *6* (3), 89-91.

[199] Vogels, W. (2003). Web services are not distributed objects. *Internet Computing, IEEE*, *7* (6), 59-66.

[200] Wang, H., Wang, G., Chen, A., Wang, C., Fung, C. K., Uczekaj, S. A., & Santiago, R. A. (2006). Modeling bayesian networks for autonomous diagnosis of web services. In *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference (FLAIRS), Melbourne Beach, FL*.

[201] Wang, D., Li, B., & Cai, J. (2008). Regression testing of composite service: An XBFG-based approach. In *Congress on Services Part II, 2008. SERVICES-2. IEEE*, (pp. 112-119). IEEE.

[202] Weiglhofer, M., Aichernig, B. K., & Wotawa, F. (2009). Fault-based conformance testing in practice. *International Journal of Software and Informatics*, *3* (2-3).

[203] Werner, E., Grabowski, J., Troschutz, S., & Zeiss, B. (2008). A TTCN-3-based web service test framework. Software Engineering.

[204] Wieczorek, S., Stefanescu, A., & Grossmann, J. (2008). Enabling model-based testing for SOA integration. In Model-based Testing in Practice Workshop, (p. 73). Citeseer.

[205] Wieczorek, S., Kozyura, V., Roth, A., Leuschel, M., Bendisposto, J., Plagge, D., & Schieferdecker, I. (2009). Applying model checking to generate Model-Based integration tests from choreography models. In M. Núñez, P. Baker, & M. Merayo (Eds.) Testing of Software and Communication Systems, vol. 5826 of Lecture Notes in Computer Science, (pp. 179–194). Springer Berlin Heidelberg.

[206] Willcock, C., Deiß, T., Tobies, S., Keil, S., Engler, F., & Schulz, S. (2011). *An Introduction to TTCN-3*. John Wiley & Sons, Ltd, second ed.

[207] Whittaker, J. A., & Thomason, M. (1994). A Markov chain model for statistical software testing. *Software Engineering, IEEE Transactions on*, *20* (10), 812-824.

[208] Wooff, D. A., Goldstein, M., & Coolen, F. P. A. (2002). Bayesian graphical models for software testing. Software Engineering, IEEE Transactions on, 28(5), 510–525.

[209] WSDL_1_1 (2001). Web service definition language (WSDL) 1.1. Tech. Rep. W3C Note 15 March 2001, World Wide Web Consortium.

[210] Xiong, P., Probert, R. L., & Stepien, B. (2005). An efficient formal testing approach for web service with TTCN-3. In *Proceedings of the 13th International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2005), Split, Croatia*.

[211] XML_1_1 (2006). *Extensible Markup Language (XML) 1.1 (Second Edition)*. World Wide Web Consortium. W3C Recommendation 16 August 2006, edited in place 29 September 2006.

[212] Yedidia, J. S., Freeman, W. T., & Weiss, Y. (2001). Generalized belief propagation. In *Advances in Neural Information Processing Systems*, (pp. 689-695).

[213] Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritisation: a survey. *Softw. Test. Verif. Reliab.*, *22* (2), 67-120.

[214] Yu, Y., Huang, N., & Luo, Q. (2007). OWL-s based interaction testing of web Service-Based system. In *Next Generation Web Services Practices, 2007. NWeSP 2007. Third International Conference on*, (pp. 31-34). IEEE.

[215] Zhang, N. L., & Poole, D. (1994). A simple approach to Bayesian network computations. *7th Canadian Conference on Artificial Intelligence*, (pp. 171-177).

[216] Zhang, N. L., & Poole, D. (1996). Exploiting causal independence in Bayesian network inference. *J. Artif. Int. Res.*, *5* (1), 301-328.

[217] Ziv, H., & Richardson, D. J. (1997). Constructing bayesian-network models of software testing and maintenance uncertainties. In *Software Maintenance, 1997. Proceedings., International Conference on*, (pp. 100-109). IEEE.