The LearnLib in FMICS-jETI

Martin Leucker
Institut für Informatik
TU München
München, Germany
leucker@in.tum.de

Tiziana Margaria
Chair of Services and
Software Engineering,
University of Potsdam,
margaria@cs.uni-potsdam.de

Harald Raffelt, Bernhard Steffen
Chair of Programming Systems
University of Dortmund, Germany
harald.raffelt@cs.uni-dortmund.de
steffen@cs.uni-dortmund.de

Abstract

This paper explains

1 Introduction

One of the goals of FMICS, the ERCIM Working Group on Formal Methods for Industrial Critical Systems (FMICS) [?], is to transfer and promote the use formal methods technology in industry. The ongoing Verified Software Initiative Grand Challenge [?] offers a great opportunity to reach this goal, resulting in a more robust and solid software industry in Europe. The FMICS-jETI platform[1] concretizes the collective effort of the FMICS WG by offering a collaborative demonstrator of the FMICS techniques and tools based on the jETI technology[2]. FMICS-jETI provides as repository a collection of verification tools stemming from the activities of the FMICS working group and facilities to orchestrate them in a remote and simple way. At the same time FMICS-jETI itself is a contribution to the VSI repository and thus to the Grand Challenge.

The FMICS community develops since its inception methods, tools, and their applications to industrial critical systems. Our point of view thus stresses correctness at the model level, rather than at the software (more generally at the coding) level. An adequate repository should therefore contain not only analyzed or proven correct software, but principally *tools* (themselves software artifacts) that help establishing the correctness of the software in question starting from the requirements, specifications, and models. We are convinced that this perspective on modelling tools and even frameworks can be useful for the progress towards better software also for less critical application domains, like consumer IT products: they strike wherever *time to market* and *first time correct* are an issue.

[1]`http:jeti.cs.uni-dortmund.de/fmics/index.php`.
[2]`jabc.cs.uni-dortmund.de/plugins/jeti_en.html`.

Based on jETI [?], the new generation of ETI [?, ?], the core FMICS partners have set up a collaborative demonstrator that

- illustrates the applicability of the jETI technology for lightweight remote integration of tools into the repository
- shows how to provide tools to the repository, by registration and remote provision,
- demonstrates how to experiment with local and remote tools to solve cooperative verification tasks
- shows how to orchestrate different tools (possibly a mix of local and remote ones) which were not originally designed to cooperate, to address more complex case studies. This may require the availability of mediators, to cover semantic gaps between the tools.

It has been applied so far mostly to include verification tools based on model checking techniques, like GEAR by U. Dortmund/U. Potsdam [?], [?], and for applications of model checking to dataflow analysis, as in [?, ?] the and it is under application also for parallel model checking [?]

In this paper, we show how to extend the scope of the FMICS-jETI platform in three new directions that address the integration of *heterogeneous and legacy tools and technologies*. We are in fact integrating

- testing, as a technology - via our preexisting ITE platform [?] for model-based testing,
- CORBA as a platform for integration
- active model learning technologies, via the LearnLib [?] as a model extrapolation technique that uses the ITE to explore a black box system and CORBA as a communication mechanism, and
- third party applications built on top of the LearnLib, in this case Smyle [?], a tool that synthesizes design models by learning from examples which uses the LearnLib as learner.

These tools and techniques have been successfully used before outside the jETI technology. We are currently using this case study as a blueprint for guidelines on how to bring CORBA-compliant tools and complex, comunicating applications into (FMICS-)jETI.

In the following, we first present the current and the jETI-based architecture (in Sect. **??**), then we recall the essential description of the jABC/jETI platform (in Sect. **??**), before detailing on the ongoing integration of CORBA as interface description language (in Sect. **??**). We then briefly present the LearnLib (in Sect. **??**) and Smyle (in Sect. **??**) from the point of view of a typical FMICS-jETI user. We finally conclude in Sect. **??**.

Acknowledgements: We thank Benedikt Bollig, Joost-Pieter Katoen, and Carten Kern for many useful comments on Smyle [5].

2 jABC/jETI

The jABC framework [16, 1] is an environment for model-driven service orchestration based on lightweight process coordination. It has been used over the past 12 years for business process and service logic modelling in several application domains, including telecommunications, bioinformatics, supply chain management, e-commerce, collaborative decision support systems, as well as for software and system development. In this paper, we restrict us to the jABC facilities relevant to orchestration of an learning approach for synthesizing design models from example scenarios that are given as message sequence charts.

In jABC, orchestration and choreography of services happen on the basis of the processes they realize in the respective application domain. These processes embody the business logics, and are expressed themselves as (executable) process models.

Semantically, jABC models are control flow graphs with fork/join parallelism, internally interpreted as Kripke Transition Systems [24]. This provides a kernel for a sound semantical basis for description formalisms like BPNM, BPEL, UML activity diagrams, and dataflow graphs, and constitutes a *lingua franca* adequate for the analysis and verification of properties, e.g. by model checking [24]. BPNM and BPEL are considered different syntactic (visual) means for representing jABC models tailored for specific communities of users. In this context, we chose to privilege the abstract semantic view of the executable models over "syntactic" sugar, and therefore use only the jABC notation.

A service orchestration is largely generated automatically. The jETI framework (Java Electronic Tool Integration) [17, 32, 3] enhances the jABC to support seamless integration of remote services such as SOAP based web services [18] and CORBA applications [27]. An essential addon of the jETI framework is its ability to generate basic service types (called SIBs, Service Independent Building Blocks) from service interface descriptions.

- WDSL (Web Services Description Language) in case of SOAP bases web services, and
- IDL (Interface Definition Language) in case of CORBA applications

SIBs represent the atomic functionality of an involved service. Within jABC, domain-specific SIB palettes are shareable among projects, and organized in a project-specific structure and with project-specific terminology. This is a simple way for adopting or adapting to different ontologies within the same application domain. Domain-specific SIB palettes are complemented by a library of SIBs that offer basic functionality (e.g. SIBs for I/O or memory handling), control structures (as used here) or handling of data structures like matrices (e.g. in our previous bioinformatics applications [21]).

3 jETI Architecture Overview

jETI's tool integration philosophy addresses the major obstacle for a wider adoption, as identified during seven years of experience with tool providers, tool users and students: the difficulty to provide the latest versions of the state-of-the-art tools. Any tool integration process required on dedicated repository servers is too complicated for both the tool providers and the repository's support and maintenance team, making it impossible to keep pace with the development of new versions and a wealth of new tools. jETI's service-based remote integration philosophy overcomes this problems, because it replaces the requirement of "physical" tool integration by very simple registration and publishing. This allows the provisioning of tool functionalities in a matter of minutes: fast enough to be fully demonstrated during our presentation. Moreover, whenever the portion of a tool's API which is relevant for a new version of a functionality remains unchanged, version updating is fully automatic!

The realisation of this registration / publishing approach based on integration philosophy consists of tree leading actors as depicted in Fig. 1.

The jETI Tool Provider is the base of jETI approach. jETI supports tool providers in offering tools and services of various kinds. Besides web services and CORBA applications, which can be integrated using their native interfaces, jETI supports command line tools. In order to offer command line tool as remote services jETI comprises

- a *Tool Configurator*, which allows tool providers to register a new tool functionality just by filling our a simple template form, and

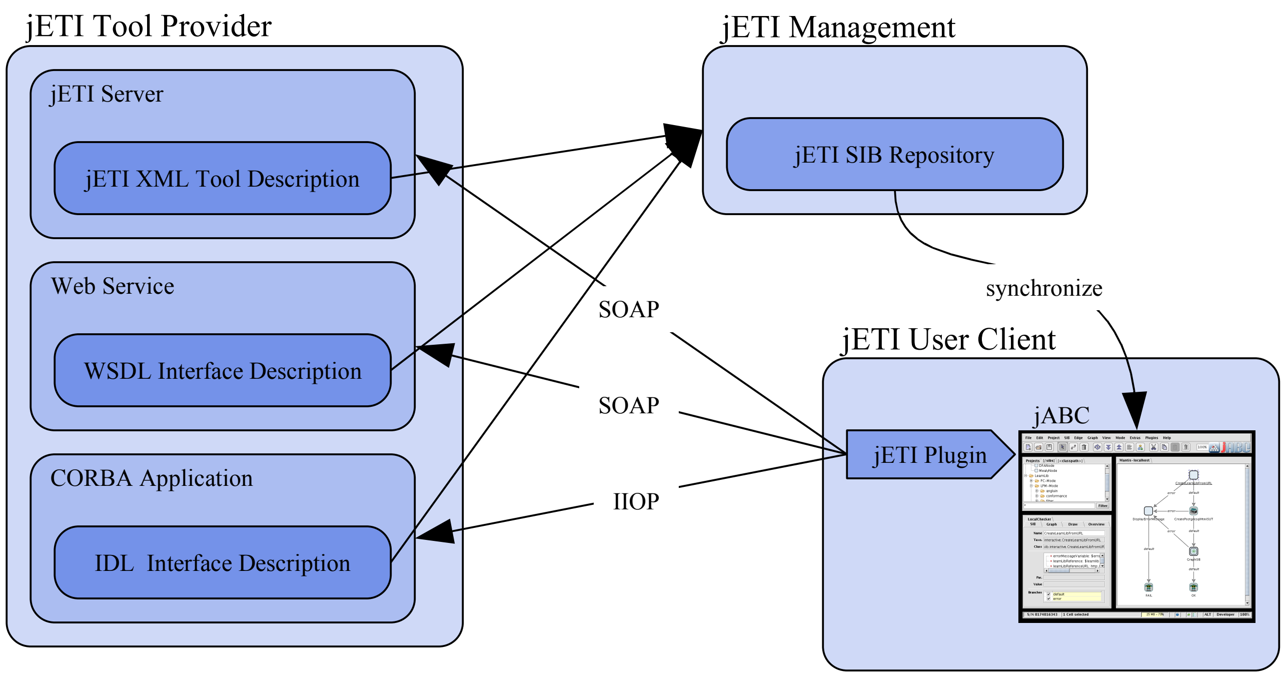


Figure 1. jETI Architecture

- a *Tool Executor*, which is able to steer the execution of the specified tools at the tool providers' site.

The jETI Component Server manages all the registered tool functionalities, including the corresponding version control. It also automatically generates appropriate jABC proxies from the interface definitions of registered tool. In the jABC context these proxies are called *service independent building block* (SIB). They represent atomic functionalities whereby complex applications can be composed. For the future, the component server is also intended to manage jETI users by means of authentication, authorisation, and accounting.

The jETI User Client loads the SIBs from the jETI Component Server and provides a flexible service development environment for orchestrating the tool functionalities. Depending on their goals and skill profile, users may just use the graphical coordination editor to experiment with the tools, or use the full support of his favourite integrated development environment (IDE) to really embed remote functionalities into normal programs.

3.1 Integration of CORBA Applications

Using CORBA applications inside the jETI client requires a valid IDL only file. IDL files comprise interface descriptions that specify of a set of possible operations a client may request through that interface. An interface provides a syntactic description of how a service is accessed via this set of operations. An example of an interface description is depicted in Fig. 2.

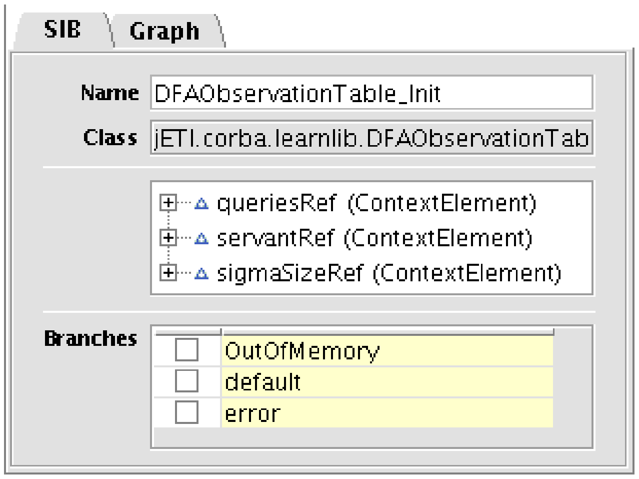
```
module learnlib {
  interface DFAObservationTable {
  void init(
    in unsigned long sigmaSize,
    out DFAQueryTree queries
  ) raises (OutOfMemory) ;

  /*  some more methods */
  };
};
```

Figure 2. CORBA Interface Defintion

jETI's SIB generator extracts the information about the interfaces and operations a CORBA application provides, and creates SIB accordingly. Since a SIB represents an atomic functionality, one SIB is generated for each operation. CORBA operations are similar to methods in object oriented programming languages. The may have input and output parameters, a return value, and they can throw exceptions for signaling error conditions. Parameters, as well as the return value, are handled as hierarchical SIB parameters: they enable the user to freely define where to store input and output values for the CORBA service, using the preexisting graphical user interface of the jABC. Fig. 3 shows the jABC view to the generated SIB.

Just as to call a method the invocation of CORBA operations requires an object that covers the state of the service. Therefor the SIB has an additional parameter: the reference to the object (servant).



Name: DFAObservationTable_Init
Class: jETI.corba.learnlib.DFAObservationTab

- queriesRef (ContextElement)
- servantRef (ContextElement)
- sigmaSizeRef (ContextElement)

Branches:

☐	OutOfMemory
☐	default
☐	error

Figure 3. jABC-View to the generated SIB

Exceptions are used change the normal flow of execution, therefor CORBA exceptions are mapped to SIB branches which steer the control flow in the jABC. Note that in the example the SIB has three branches. The "default" branch indicates normal execution of the service, the "OutOfMemory" branch represents the exception defined in the interface, and the "error" branch is used to cover all other failures, such as network communication problems. The complete source of the generated SIB is depicted in 4.

Currently the implementation does not analyze the IDL files directly. It first uses a standard tool to generate a Java client library and then analyzes that library in order to extract relevant information to generate the SIBs. This way the approach can easily be adapted to other remote invocation protocols such as RMI and SOAP. The disadvantage is, that it is not possible to transfer any API documentation from the IDL files into the SIB documentation.

3.2 Choreography

jABC originated in the context of the verification of distributed systems [24], therefore SLGs are inherently adequate as choreography models. The SIBs can physically run in a distributed architecture. They communicate directly or with a shared space (called the context). The SLGs are fully hierarchical: SIBs can themselves be implemented via SLGs. The macro mechanism described in [?] allows defining what communication actions of an SLG are visible to the environment (for choreography). Orchestration is as far as the jABC is concerned just a degenerate case of choreography.

```
public class DFAObservationTable_Init
implements Executable
{
public final String[] BRANCHES = {
    SIB.DEFAULT,
    SIB.ERROR,
    "OutOfMemory"
  };

public ContextElement servantRef;
public ContextElement sigmaSizeRef;
public ContextElement queriesRef;

public String trace(ExecEnvironment env)
{
  DFAObservationTableOperations servant;
  Integer sigmaSize;
  DFAQueryTreeHolder queries;

  // fetch input parameters
  servant=(DFAObservationTableOperations)
    env.get(servantRef);
  sigmaSize=(Integer)
    env.get(sigmaSizeRef);

  // prepare output parameter
  queries = new DFAQueryTreeHolder();

  // invoke the remote servant
  try {
    servant.init(sigmaSize, queries);
  } catch (OutOfMemory e) {
    return "OutOfMemory";
  }

  // store the output parameters
  env.put(queriesRef, queries.value);
  return SIB.DEFAULT;
}
```

Figure 4. SIB Source

3.3 Data Semantics

The static data semantics is captured automatically during the IDL-to-SIB import as the SIB parameters.

These parameters, additional semantic properties attached to the SIBs, possibly imported from an ontology, and the SIB branch labels are visible to the model checker [16], which allows automatically proving global compliance constraints on the business logic of an SLG. These constraints are expressible in mu-calculus and its derivatives, a family of modal (temporal) logics.

Additionally, arbitrary relations between data elements can be provided as local checking expressions, with the expressiveness of Java. This facility allows expressing and checking pre and post conditions.

4. LearnLib

4.1 Classical Automata Learning

Machine learning deals in general with the problem of how to automatically generate system descriptions. Besides the synthesis of static soft- and hardware properties, in particular invariants [13], [25], [8], the field of *automata learning*, also called regular extrapolation [15] or regular inference [12], is of particular interest for soft- and hardware engineering [11], [23], [35], [28], [10].

We have used automata learning techniques in a number of contexts, e.g. to automatically construct models of Web applications as demonstrated in [29] and to enhance incomplete specifications of biological systems [20].

Automata learning tries to construct a deterministic finite automaton that matches the behavior of a given target automaton on the basis of observations of the target automaton and perhaps some further information on its internal structure. The interested reader may refer to [15, 31, 33] for our view on the use of learning. Here we only summarize the basic aspects of our realization, which is based on Angluin's learning algorithm L^* from [2].

Definition A deterministic finite automaton (DFA) is a tuple $\mathsf{M} = (S, s_0, \Sigma, \delta, F)$ where

- S is a finite nonempty set of *states*,
- $s_0 \in S$ is the *initial state*,
- Σ is a finite *alphabet*,
- $\delta : S \times \Sigma \to S$ is the *transition function*, and
- $F \subseteq S$ is the set of *accepting states*.

Intuitively, a DFA evolves through states $s \in S$, and whenever one applies an input symbol (or action) $a \in \Sigma$, the machine moves to a new state according to $\delta(s, a)$. A word $q \in \Sigma^*$ is accepted by the DFA if and only if the DFA reaches an accepting state $s_i \in F$ after processing the word starting from its initial state.

L^*, also referred to as an *active* learning algorithm, learns deterministic finite automata by *actively* posing *membership* queries and *equivalence* queries to the target automaton in order to extract behavioral information, and by refining successively an own hypothesis automaton based on the answers. A membership query tests whether a string (a potential run) is contained in the target automaton's language (its set of runs), and an equivalence query compares the hypothesis automaton with the target automaton for language equivalence, in order to determine whether the learning procedure was (already) successfully completed. In this case the experimentation can stop.

In its basic form, L* starts with the one state hypothesis automaton that treats all words over the considered alphabet (of elementary observations) alike and refines this automaton on the basis of query results iterating two steps. Here, the dual way of how L* characterizes (and distinguishes) states is central:

- from *below*, by words reaching them. This characterization is too fine, as different words may well lead to the same state.
- from *above*, by their future behavior wrt. a dynamically increasing set of words. These future behaviors are essentially bit vectors, where a '1' means that the corresponding word of the set is guaranteed to lead to an accepting state and a '0' captures the complement. This characterization is typically too coarse, as the considered sets of words are typically rather small.

The second characterization directly defines the hypothesis automata: each occurring bit vector corresponds to one state in the hypothesis automaton.

The initial hypothesis automaton is characterized by the outcome of the membership query for the empty observation. Thus it accepts any word in case the empty word is in the language, and no word otherwise. The learning procedure (1) iteratively establishes local consistency, after which it (2) checks for global equivalence.

Local Consistency This first step (also referred to as automatic *model completion*) again iterates two phases: one for checking wether the constructed automaton is *closed* under the one-step transitions, i.e., each transition from each state of the hypothesis automaton ends in a well defined state of this very automaton. And one for checking *consistency* according to the bit vectors characterizing the future behavior as explained above, i.e., whether all reaching words with an identical characterization from above possess the same one step transitions. If this is not the case, a distinguishing transition is taken as an additional distinguishing future in order to resolve the inconsistency, i.e., the two reaching words with different transition potential are no longer considered to represent the same state.

Global Equivalence After local consistency has been established, an equivalence query checks whether the language of the hypothesis automaton coincides with the language of the target automaton. If this is true, the learning procedure successfully terminates. Otherwise the equivalence query returns a counterexample, i.e., a word which distinguishes the hypothesis and the target automaton. This counterexample gives rise to a new cycle of modifying the hypothesis automaton and starting the next iteration.

In any practical attempt of learning legacy systems, the equivalence tests can only be approximated, but membership queries can be answered by testing the target systems [15, 31].

LearnLib is a library of tools for automata learning. It is implemented in C++ and tested under Linux and Solaris, and it currently consists of 150 classes and almost 50.000 lines of code. Originally, LearnLib has been designed to systematically build finite state machine models of real world systems. In the meantime, it also became a platform for experimenting with different learning algorithms and to statistically analyze their characteristics in terms of learning effort, run time and memory consumption. As shown in Fig. 5, LearnLib consists of three libraries:

- The *automata learning* library contains the basic learning algorithms,
- the *filter* library provides several strategies to reduce the number of queries, and
- the *approximative equivalence queries* library is based on the generation of conformance test suites for the conjectures of the learning algorithms.

4.2 Analysis and Profiling of Learning Algorithms

Dieser Abschnitt sollte stark gekrzt werden.

Different learning algorithms have different profiles: they differ in the way they proceed to gain structured knowledge about an unknown system. Mostly they differ in the number of membership- and equivalence queries, but also in the size of their queries. In order to analyze these differences and to find out more about how learning algorithms perform in practice we have built a configurator platform and a profiling tool, which allow us to experiment with several learning algorithms and configurations and to collect statistics about their performance under controlled and reproducible experimental conditions. The graphical user interface of the configurator is shown in Fig **??** in a configuration similar to the one we used in [22] to analyze second order effects among optimizations (described in detail in Sec. 5.1).

A LearnLib configuration is similar to a data flow graph: It specifies how the membership- and equivalence queries generated by one of the LearnLib learning algorithms are passed through other components of the LearnLib, for example through optimization filters. Considering Fig **??**, on top of the graph the DFA_Angluin node executes Angluin's

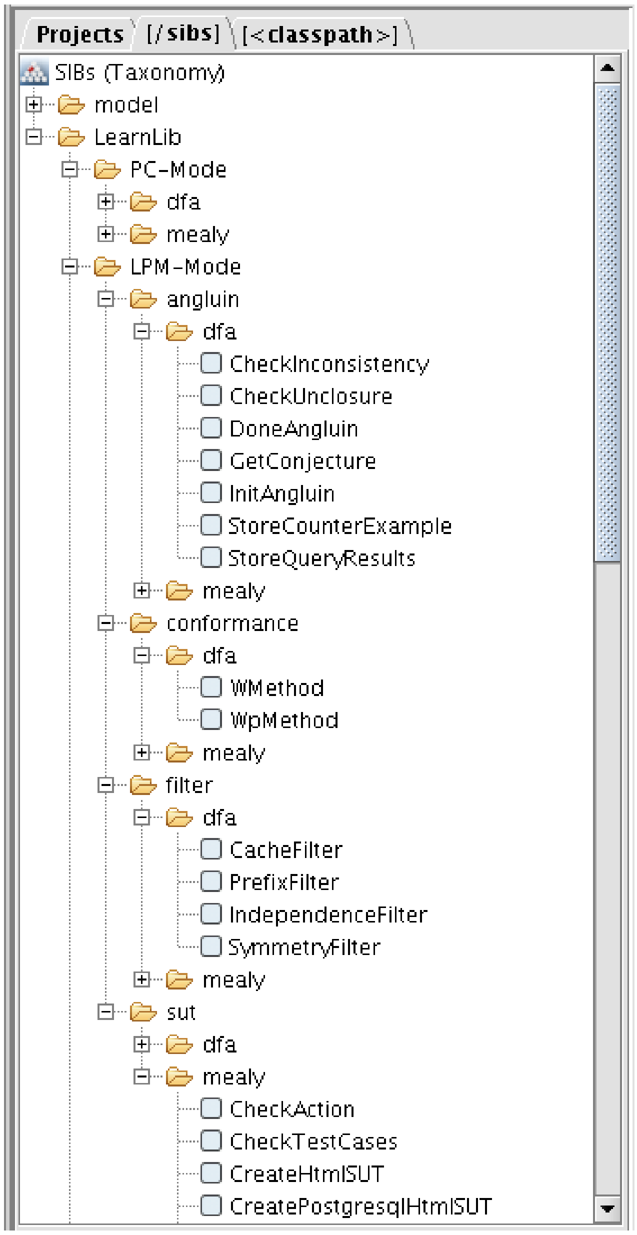


Figure 5. LearnLib: Components and Applications

algorithm in the DFA version. All the membership queries (MQ) it generates are passed to the (here 6) filter configurations, whose efficiency and performance we want to investigate.

To this aim, the DFA_fork component forwards copies of the queries to six different filter combinations, which eliminate redundancies according to several criteria, as described in detail in Sec. **??**. Finally, the queries which could not be filtered are passed to the system under test for execution by the DFA_SUT_Simulator. This way we analyze the comparative impact of the individual filters while learning a certain model.

For studying the generic behavior of the filters, automatically generated models are most appropriate as they provide us with any required number of example systems. Moreover, the Learnlib allows us to generate models of a particular profile, concerning e.g. the number of accepting or rejecting states, the branching degree, or language features like prefix closure.

5 Automata Learning with the LearnLib

The main library of the LearnLib contains several variants of Angluin's algorithm. Angluin assumes an omniscient oracle (called teacher), which answers to the following two kinds of questions (as explained in Sect. 4):

- *Membership queries* ask whether a certain word is accepted by the finite state machine. This kind of query can be directly answered for real systems via testing.
- *Equivalence queries* ask for checking whether the current conjecture is (already) equivalent to the finite state machine. These queries should be answered either with *Yes* or a *counter example*.

Equivalence queries for 'black box' finite state machines are in general undecidable. Thus one has to live with approximations like e.g. variations of conformance testing, as shown in Fig. 5(right) which lists the conformance testing routines currently available in the LearnLib.

The following two sections describe the two modes offered by the LearnLib to flexibly deal with the wealth of available options: a *pre-configuration mode* (PC-Mode), which allows the user to pre-configure an optimized learning setting (see Sec. 5.1), and a *learning process modelling mode* (LPM-Mode), which enable the user to control the entire learning process, comprising the context-specific choice of optimizations, strategies of search, as well as the setting of interaction points for a truly interactive learning process (see Sec. 5.2).

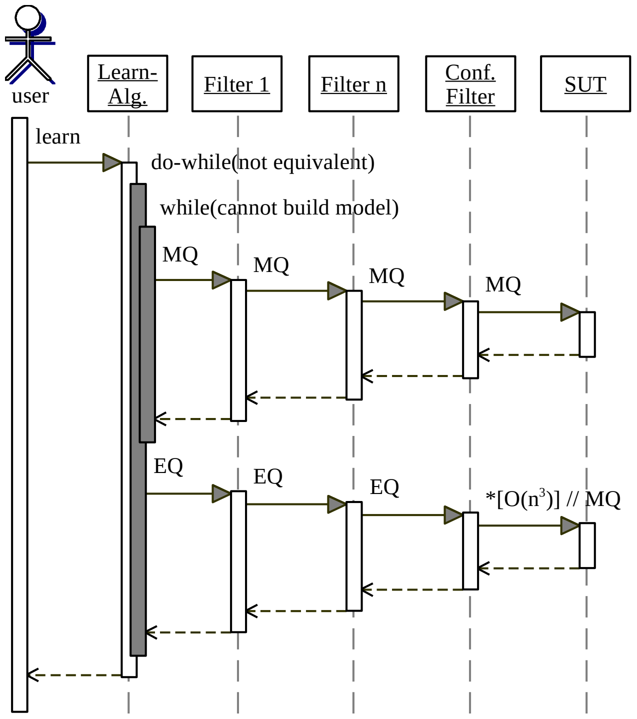


Figure 6. Pre-Configuration Mode

5.1 Pre-Configuration Mode

Dieser Abschnitt sollte stark gekrzt werden.

In the PC-mode the user graphically specifies a configuration that defines a number of LearnLib experiments carried out in parallel in terms of the chosen learning algorithm, its global selection strategy for membership queries, and the experiment-specific choices of filter chains.

Fig. **??** shows the use of the LearnLib as integrated tool to the jABC environment [16]. As we see, a configuration is a data flow graph-like structure which describes how membership- and equivalence queries are passed through components of the LearnLib. This is done by combining the functionalities of the library of components, shown in the upper left frame, to configurations in the canvas on the right. The user of the LearnLib can combine

- a learning algorithm
- a directed acyclic graph of query filters, to take advantage of structural knowledge about the system,
- a general strategy how to select membership queries, and

- an interface to a system under test

Configuration design is done, in the usual jABC style, by dragging library components to the canvas, and then connecting them by edges that specify how queries are passed and how the library components should interact. Additionally each component may have parameters, which are set via the inspector in the lower left panel. In Fig. **??** the parameters of the framed upper left *independence filter* are shown in the lower left panel, which currently indicates that action 2 is independent with actions 4, 5, and 6. The independence filter, described in detail in Section **??**, is applicable when the system contains pairs of independent actions.

Currently the user can choose between Angluin's method for learning DFAs [2] and our version for learning Mealy machines [33]. To take advantage of knowledge about the analyzed system's structure, the user can specify a chain of filters, discussed in detail in Sect. **??**, which are used to reduce the number of membership- and equivalence queries to the oracle (resp. the system under test). The chain of filters must terminate with an oracle or a system under test interface, which answers all unfiltered queries.

Additionally, the user can choose a general strategy of how membership queries are selected, which steers the algorithm either in a more depth or more breadth oriented way. This is particularly interesting in the context of enforcing consistency and closedness, where one typically has a variety of options (see also Sect. **??**). The available strategies so far are

- *random:* choose randomly,
- *fast:* take the first possible alternative,
- *long:* prefer alternatives leading to long membership queries,
- *short:* prefer alternatives leading to short membership queries,
- *cheap:* prefer alternatives producing membership queries that can be answered by the chain of filters, and
- *expensive:* prefer alternatives which produce membership queries that require the membership oracle.

In general the pre-configuration mode of the learn algorithms work as depicted in Fig. 6. An inner loop continues to generate and ask membership queries until the algorithm is able to build a valid hypothesis model. All these membership queries are sent through the chain of filters in order to suppress redundant queries. Queries remaining unfiltered are passed to the membership oracle. This either leads to a direct check (in case of the simulation mode, where the target model is known), or to a test run of the target system (this is the way membership queries are answered in our real life scenarios).

Whenever the learning algorithm has collected enough information to build a valid hypothesis model, this model is subject to an equivalence query, the bottleneck of the learning procedure. Except for the case of simulation, where the target model is known, we have to approximate equivalence queries by means of membership queries. Particularly suitable are here methods adopted from conformance testing (see Sect. 5.3). They help to systematically search for distinguishing execution traces by means of testing, i.e., by posing appropriate membership queries. Thus our filters can also be applied here.

As soon as one of the membership queries detects a discrepancy, the corresponding trace is given to the learning algorithm as a counterexample to improve the hypothesis model, and the next iteration of the learning algorithms begins. This continues until the (approximate) equivalence oracle returns TRUE, signaling that we successfully learned the target system.

5.2 Learning Process Modelling-Mode

Dieser Abschnitt sollte stark gekrzt werden. Der Screenshot LearnLib-Interactive wird berarbeitet und an den Smyle Demo-Case angepasst. Mit Anderen Worten: Im ScreenShot wird der Hypothetische Smyle-Prozess dargestellt.
Nach dem Abschnitt Smyle wird diese Abbildung eingefgt und erklhrt Schrittweise den Process Modeling Mode, die Remote komponente und den smyle prozess. Ein groteil dieses Abschnittes kommt also erst zum Ende des Papers.

In the LPM-Mode graphs, which are constructed just as in the PC-Mode, are used to model the entire learning process, which comprises the modelling of conditional or interactive behaviour. The nodes may now represent arbitrary statements, in particular including all atomic functionalities of the LearnLib, and the edges specify in which order and under which condition they are processed.

Fig. 9 depicts the control flow graph of a simple variation of Angluin's algorithm: here, the execution starts with connecting the graphical user interface to the LearnLib *ConnectToLearnLib*, before an interface to a system under test is created *CreatreSUInterface*. The SUT interface can be linked to a real system, but it can also represent a SUT simulator, which uses known models stored in a database. This means that a SUT interface can represent a number of systems of very different kind. Therefore in the next step it is checked whether there is a next SUT that should be analyzed (*HasNextSubject*).

After this first initialization steps the learning process is started by initializing Angluin's algorithm L^* [2]. The learning algorithm now generates a test suite, which must be executed by the SUT interface in the next step. The *QueryTestCase* component executes the traces contained in the test suite and records the response of the SUT. At this point the SUT interface may discover that the implementation offers more possibilities to be stimulated than currently specified. For example, a new input action may have arisen. This happens for example when learning web applications, since every new discovered web page leads to a new action for directly requesting that page. This special feature is handled by the two components connected to the *sizeChanged* branch. First the results of querying the SUT are stored in the learning algorithm and then the alphabet is updated. Afterwards the results of querying the SUT are returned as a basis for a user decision *UserInteraction* about the order of the two well-formedness checks *CheckClosure*) and (*CheckConsistency*. If the observations are both closed and consistent, L^* constructs a conjecture model, which is done in *GetConjecture*, otherwise the learning algorithm provides a new test suite and the main loop continues.

After the main loop, the conjecture can be visualised (*DrawMealyModelGraph*) and stored to a file (*SaveGraph*), before one enters the check for global equivalence. In this example this is done by generating and then executing a test suite (*CheckTestcases*) according to the Wp-Method [14]. If the conjecture does not conform to the SUT, a counter example is returned, and the learning algorithm continues. Otherwise L^* successfully terminates this learning task and continues with the next SUT (*HasNextSubject*).

The execution of this control flow graph can be interactively steered using the *Tracer*, which is able to execute these control flow graphs. In addition it provides useful debugging functionalities, which allow users to investigate the data exchanged between the nodes resp. atomic functionalities. This way the user can visualize at any time the sets of membership queries and intermediate finite state models generated by the learning process. It is also possible to automatically generate a stand-alone Java program which realizes the specified learning process.

Compared with the PC-mode, the interactive version provides more control on how the learning algorithm proceeds. Like in the PC-mode, learning happens in two alternating phases: constructing hypothesis automata and checking their equivalence with the target automaton. These phases alternate until the equivalence check, which is typically done via some version of conformance testing, is passed. The user however remains in control: at any time he may change the kind of filters used or decide which one of the proposed membership queries should be executed next.

During the first phase this typically happens in five successive steps:

1. Ask the learning algorithm to build a set of membership queries which are required for the learning process. For Angluin's algorithms there are two constraints which must hold before a hypothesis model is built: *closedness* and *consistency* (details can be found [2]). Closedness and consistency are also established via membership queries. In the LPM-mode, the user may influence the order in which membership queries are posed in order to accelerate the convergence.

2. Decide which filters should be used to filter out irrelevant queries. Chaining of filters is also supported.

3. Send the remaining membership queries to the oracle, which gives the missing answers.

4. Update the filters according to the gained information.

5. Analyze the result of the membership queries given to the learning algorithm, and decide where to continue the iteration.

This loop continues until the learn algorithm is able to construct a hypothesis model for the real system. Now the LearnLib user can use this conjecture for an equivalence query. This is only directly possible in the simulation case, where the target model is known. Otherwise, one typically approximates the equivalence queries by membership queries. Thus we may also profit from the filters here. The LPM-Mode supports this by

- allowing to choose specific sets of filters, which may be able to directly produce a counterexample on the basis of the structural assumptions, or

- to translate the conjecture into a (contex-specific) approximating conformance test.

In addition, at any time the user might present the LearnLib with particular execution traces, which he assumes to differentiate the current hypothesis model form the target system. This may drastically reduce the required number of equivalence queries, the true bottleneck of automata learning.

5.3 Equivalence Queries in the LearnLib

As mentioned before, it is impossible to decide equivalence queries if one is restricted to observe the input/output behavior of an unknown system. Therefore one has to resort to approximations of equivalence queries. It turns out that

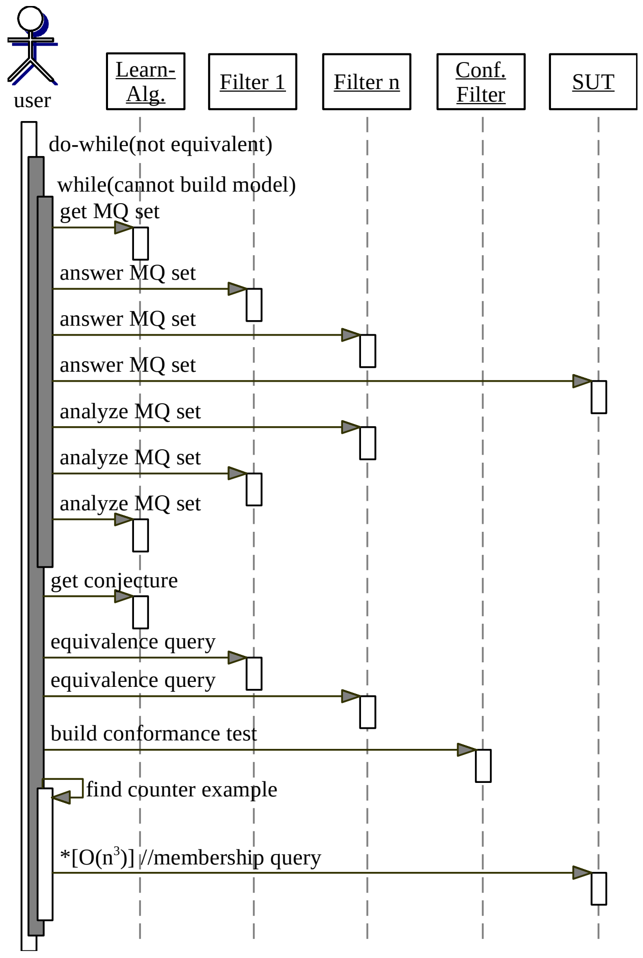


Figure 7. Learning Process Modeling Execution Model

methods from the field of conformance testing are particularly adequate to approximate equivalence queries [4].

The problem of conformance testing can be briefly described as follows [19]. Given

- a finite state machine M_S, which acts as known specification, and
- a black-box implementation M_I (typically representing just another finite state machine), providing testing capabilities only,

one wants to determine by testing whether M_I correctly implements or, as we say, *conforms* to M_S.

Of course also this problem is in general undecidable, but there are a number of practically relevant approaches, some of which, under certain circumstances, like e.g. restriction of the number of states of the black box implementation, are even complete.

Due to the restricted setting, the proposed alternatives only differ in the set of tests they produce. Thus conformance testing is closely related to test generation. Besides basic test suite generation algorithms like state cover set and transition cover set, the current version of the LearnLib supports also the W-Method [9] and the Wp-Method [14]. More advanced conformance test methods like the UIO-method [30] and the UIOv-method [34] are currently under development.

Wie werden Aequivalenz Queries in Smyle beachtet?

---

6. Smyle

Smyle, which stands for **S**ynthesizing **M**odels b**y** **L**earning from **E**xamples, is a tool for synthesizing design models by learning from example scenarios that are given as message sequence charts.

The elicitation of requirements is the main initial phase in the typical software engineering development cycle. A plethora of elicitation techniques for requirement engineering exist. Popular requirement engineering methods, such as the Inquiry Cycle and CREWS [26], exploit use cases and scenarios to specify the system's requirements. Sequence diagrams are also at the heart of the UML. A scenario is a partial fragment of the system's behavior, describing the system components, their message exchange and concurrency. Their intuitive yet formal nature has resulted in a broad acceptance. Scenarios can be either positive or negative, indicating a possible desired or unwanted system behavior, respectively. Different scenarios together form a more complete description of the system behavior.

The following design phase in software engineering is a major challenge as it is concerned with a paradigm shift between the *requirement* specification—a partial, overlapping and possibly inconsistent description of the system's behavior—and a conforming *design model*, a complete behavioral description of the system (at a high level of abstraction). During the synthesis of such design models, usually automata-based models that are focused on intra-agent communication, conflicting requirements will be detected and need to be resolved. Typical resulting changes to requirements specifications include adding or deleting scenarios, and fixing errors that are found by a thorough analysis (e.g., model checking) of the design model. Obtaining a complete and consistent set of requirements together with a related design model is thus a highly iterative process.

The *Smyle modelling approach* (*SMA*, for short) is a novel methodology that is an important stepping stone towards bridging the gap between scenario-based requirement specifications and design models. The novel aspect of our approach is to exploit *learning* algorithms for the synthesis of design models from scenario-based specifications. Since message-passing automata (MPA, for short) [7] are a commonly used model to realize the behavior as described by scenarios, we adopt MPA as design model.

The technical heart of *SMA* is a procedure that interactively infers an MPA from a given set of positive and negative scenarios of the system's behavior provided as message sequence charts (MSCs). This is achieved by generalizing Angluin's learning algorithm for deterministic finite-state automata (DFA) (see Section **??**) towards specific classes of bounded MPA, i.e., MPA that can be used to realize MSCs with channels of finite capacity. Details can be found in [5].

An important distinctive aspect of *SMA* is that it naturally supports the *incremental generation* of design models. Learning of initial sets of scenarios is feasible. On adding or deletion of scenarios, MPA are adapted accordingly in an automated manner. Thus, synthesis phases and analysis phases, supported by simulation or analysis tools such as *MSCan* [6], complement each other in a natural fashion. Furthermore, on establishing the inconsistency of a set of scenarios, our approach mechanically provides *diagnostic feedback* (in the form of a counterexample) that can guide the engineer to evolve his requirements.

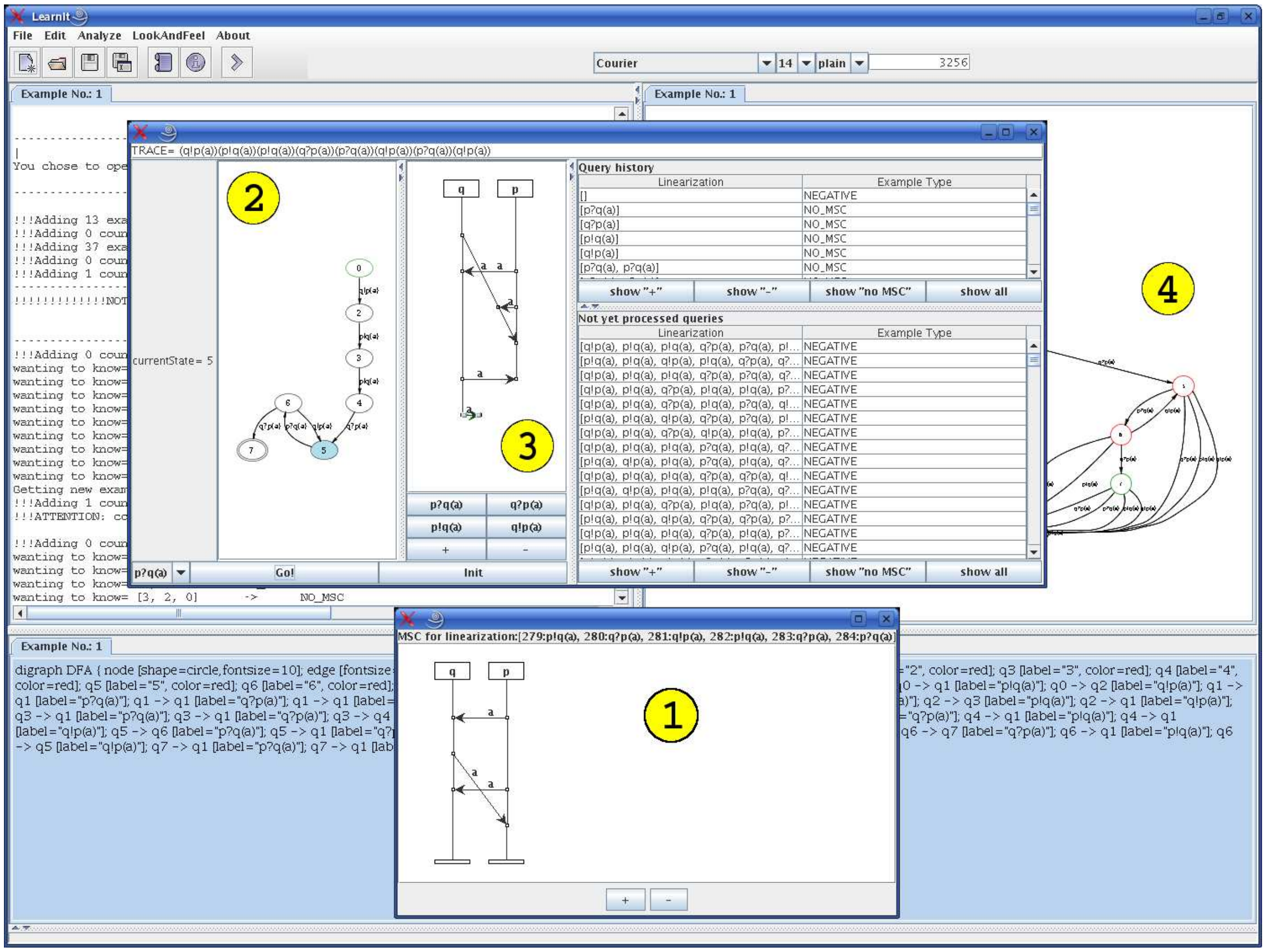
The SMA in detail Initially the user is asked to specify the learning setup. After having selected a language type (existentially/universally) and a channel bound B, the user provides a set of MSCs. These MSC specifications must then be divided into *positive* (i.e., MSCs contained in the language to learn) and *negative* (i.e., MSCs not contained in the language to learn). After submitting these examples, all linearizations are checked for consistency with respect to the properties of the learning setup. Violating linearizations are stored as negative examples. Now the learning algorithm starts. The *Learner* continuously communicates with the *Assistant* in order to gain answers to membership queries. This procedure halts as soon as a query cannot be answered by the *Assistant*. In this case, the *Assistant* forwards the inquiry to the user, displaying the MSC in question on the screen. The user must classify the message sequence chart as positive or negative (cf. Fig. 8 (`1`)).

The *Assistant* checks the classification for validity wrt. the learning setup. Depending on the outcome of this check, the linearizations of the current MSC are assigned to the positive or negative set of future queries. Moreover, the user's answer is passed to the *Learner* which then continues his question-and-answer game with the *Assistant*. If the `LearnLib` proposes a possible automaton, the *Assistant* checks whether the learned model is consistent with all queries that have been categorized but not yet been asked. If she encounters a counter-example, she presents it to the learning algorithm which, in turn, continues the learning procedure until the next possible solution is found. In case there is no further evidence for contradicting samples, a new frame appears (cf. Fig. 8 (`2,3`)). Among others, it visualizes the currently learned automaton (`2,4`) as well as a panel for displaying MSCs (`3`) of runs of the system described by the automaton. The user is then asked if she agrees with the solution and may either stop or introduce a new counter-example proceeding with the learning procedure.

***Smyle* and the LearnLib** Currently, *Smyle*, which can be freely downloaded at `http://smyle.in.tum.de`, is is written in Java and makes use of the `LearnLib` library via its CORBA interface. Thus, the LearnLib basically used like a standard library in this case providing learning functionality. The main difference is the the learning functionality is not integrated into *Smyle* at compile time but at run time. While this requires to have an Internet connection to the LearnLib's location, this design choice has several advantages: *Smyle* immediately profits from ongoing improvements or bug fixes of the learning library—completely transparent to the user of *Smyle*. Furthermore, learning of large systems typically asks for machines with a lot of memory. Having the LearnLib running remotely shifts this issue to location that supports the LearnLib and can deal with this issue in a better manner than the typical user of *Smyle* on its local machine.

***Smyle* and jETI** As described above, the current structure of *Smyle* is *static* except of the access of the learning library which is carried out at run time. For reasons described below, it be interesting, to break-up *Smyle*'s static design and move to the jABC/jETI philosophy.

Figure 9



```
--------------------
|
You chose to ope
--------------------

!!!Adding 13 exa
!!!Adding 0 coun
!!!Adding 37 exa
!!!Adding 0 coun
!!!Adding 1 coun
--------------------
!!!!!!!!!!!!!NOT


--------------------
!!!Adding 0 coun
wanting to know=
wanting to know=
wanting to know=
wanting to know=
wanting to know=
wanting to know=
wanting to know=
wanting to know=
wanting to know=
Getting new exam
!!!Adding 1 coun
!!!ATTENTION: co

!!!Adding 0 coun
wanting to know=
wanting to know=
wanting to know=
wanting to know= [3, 2, 0]        ->        NO_MSC
```

TRACE= (q!p(a))(p!q(a))(p!q(a))(q?p(a))(p?q(a))(q!p(a))(p?q(a))(q!p(a))

Query history

Linearization	Example Type
[]	NEGATIVE
[p?q(a)]	NO_MSC
[q?p(a)]	NO_MSC
[p!q(a)]	NO_MSC
[q!p(a)]	NO_MSC
[p?q(a), p?q(a)]	NO_MSC

Not yet processed queries

Linearization	Example Type
[q!p(a), p!q(a), p!q(a), q?p(a), p?q(a), p!...	NEGATIVE
[p!q(a), p!q(a), q!p(a), p!q(a), q?p(a), q?...	NEGATIVE
[q!p(a), p!q(a), p!q(a), q?p(a), p?q(a), q?...	NEGATIVE
[q!p(a), p!q(a), q?p(a), p!q(a), p!q(a), p?...	NEGATIVE
[q!p(a), p!q(a), q?p(a), p!q(a), p?q(a), q!...	NEGATIVE
[p!q(a), p!q(a), q!p(a), q?p(a), p?q(a), p!...	NEGATIVE
[q!p(a), p!q(a), q?p(a), q!p(a), p!q(a), p?...	NEGATIVE
[q!p(a), p!q(a), p!q(a), p?q(a), p!q(a), q?...	NEGATIVE
[p!q(a), q!p(a), p!q(a), p?q(a), p!q(a), q?...	NEGATIVE
[q!p(a), p!q(a), p!q(a), q?p(a), q?p(a), q!...	NEGATIVE
[p!q(a), q!p(a), p!q(a), p!q(a), p?q(a), q?...	NEGATIVE
[q!p(a), p!q(a), q?p(a), p!q(a), p?q(a), p!...	NEGATIVE
[p!q(a), p!q(a), q!p(a), q?p(a), q?p(a), p?...	NEGATIVE
[q!p(a), p!q(a), p!q(a), q?p(a), p!q(a), p?...	NEGATIVE
[p!q(a), p!q(a), q!p(a), p?q(a), p!q(a), q?...	NEGATIVE

MSC for linearization:[279:p!q(a), 280:q?p(a), 281:q!p(a), 282:p!q(a), 283:q?p(a), 284:p?q(a)]

```
digraph DFA { node [shape=circle,fontsize=10]; edge [fontsize    ="2", color=red]; q3 [label="3", color=red]; q4 [label="4",
color=red]; q5 [label="5", color=red]; q6 [label="6", color=red];    0 -> q1 [label="p!q(a)"]; q0 -> q2 [label="q!p(a)"]; q1 ->
q1 [label="p?q(a)"]; q1 -> q1 [label="q?p(a)"]; q1 -> q1 [label=    a)"]; q2 -> q3 [label="p!q(a)"]; q2 -> q1 [label="q!p(a)"];
q3 -> q1 [label="p?q(a)"]; q3 -> q1 [label="q?p(a)"]; q3 -> q4    ="q?p(a)"]; q4 -> q1 [label="p!q(a)"]; q4 -> q1
[label="q!p(a)"]; q5 -> q6 [label="p?q(a)"]; q5 -> q1 [label="q?    q6 -> q7 [label="q?p(a)"]; q6 -> q1 [label="p!q(a)"]; q6
-> q5 [label="q!p(a)"]; q7 -> q1 [label="p?q(a)"]; q7 -> q1 [lab
```

Figure 8. *Smyle* **screenshot**

References

[1] jabc website. `http://www.jabc.de`, seen Apr. 2007.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2(75):87–106, 1987.

[3] A. Arenas, J. Bicarregui, and T. Margaria. The FMICS view on the verified software repository. In *Proc. Integrated Design and Process Technology (IDPT)*, San Diego (USA), June 26-29 2006. Society for Design and Process Science.

[4] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In M. Cerioli, editor, *Proc. of* 8^{th} *Int. Conf. on Fundamental Approaches to Software Engineering (FASE'05)*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer Verlag, April 4-8 2005.

[5] B. Bollig, J.-P. Katoen, C. Kern, and M. Leucker. Replaying play in and play out: Synthesis of design models from scenarios by learning. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *Lecture Notes in Computer Science*, Braga, Portugal, Mar. 2007. Springer.

[6] B. Bollig, C. Kern, M. Schlütter, and V. Stolz. MSCan: A tool for analyzing MSC specifications. In *TACAS 2006*, volume 3920 of *Lecture Notes in Computer Science*, pages 455–458. Springer, 2006.

[7] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. of the ACM*, 30(2):323–342, 1983.

[8] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proc. of the* 26^{th} *Int. Conf. on Software Engineering (ICSE'04)*, pages 480–490, Edinburgh, Scotland, May 2004.

[9] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978.

[10] J. E. Cook, Z. Du, C. Liu, and A. L. Wolf. Discovering models of behavior for concurrent systems. Technical report, New Mexico State University, Deppartment of Computer Science, August 2002. NMSU-CS-2002-010.

[11] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *(TOSEM) ACM Transac-*

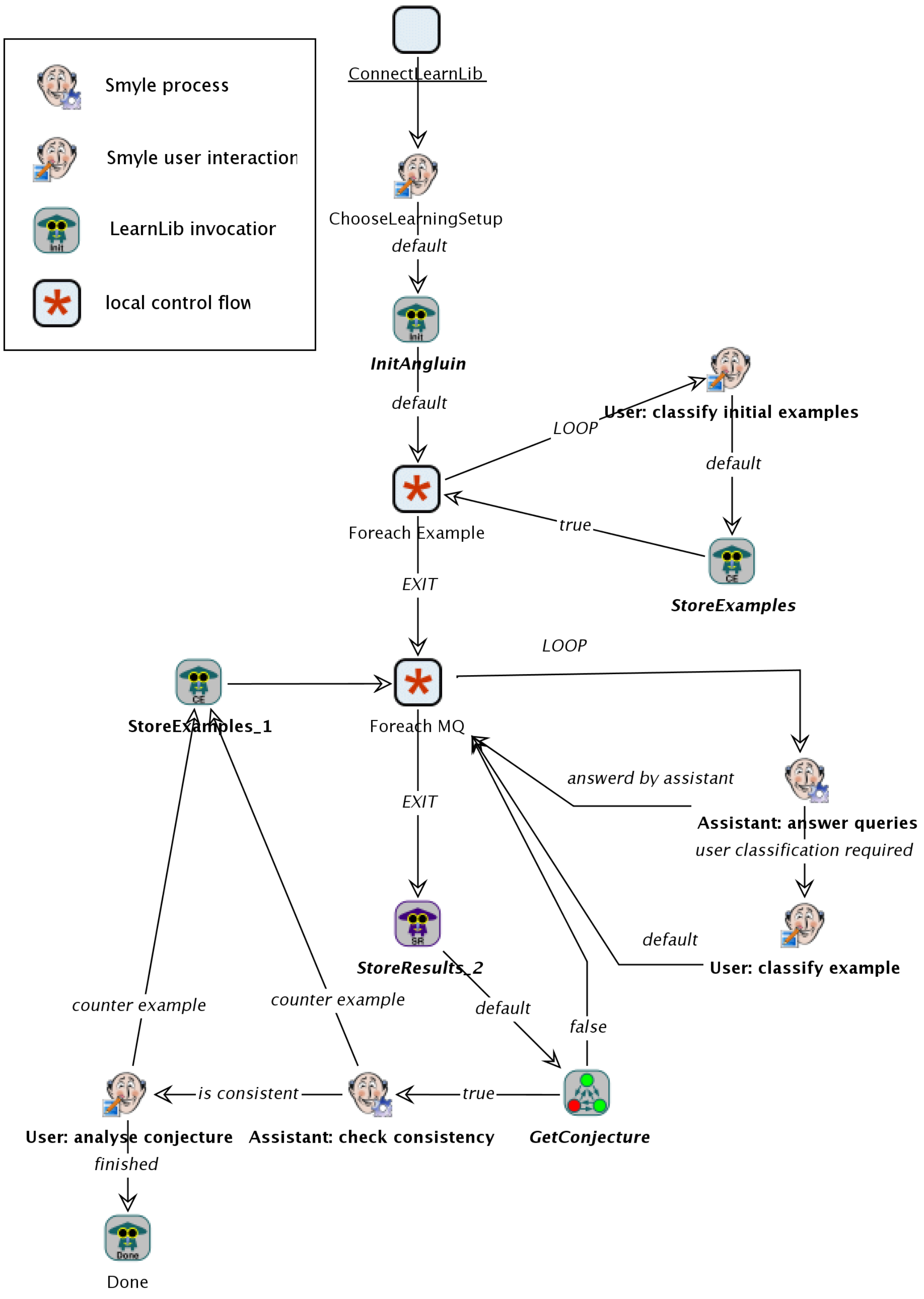


Figure 9. Learning Process Modeling Mode: Design of Smyle in jETI

tions on Software Engineering and Methodology, 7(3):215–249, 1998.

[12] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38:1332–1348, September 2005.

[13] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proc. of* 22^{nd} *Int. Conf. on Software Engineering (ICSE'00)*, pages 449–458, June 2000.

[14] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering.*, 17(6):591–603, 1991.

[15] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In H. W. R. Kutsche, editor, *Proc. of the* 5^{th} *IntĊonfòn Fundamental Approaches to Software Engineering (FASE'02)*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95, Heidelberg, Germany, April 2002. Springer-Verlag.

[16] S. Jörges, C. Kubczak, R. Nagel, T. Margaria, and B. Steffen. Model-driven development with the jABC. In *in Proc. of Haifa Verification Conference 2006 (HVC 2006)*, LNCS, Haifa, Israel, October 23-26 2006. IBM, Springer Verlag. hvc06.

[17] C. Kubczak, B. Steffen, and T. Margaria. The jABC approach to mediation and choreography. 2^{nd} *Semantic Web Service Challenge Workshop.*, June 2006.

[18] Y. Lafon. W3c web services activity. `http://www.w3.org/2002/ws/`, seen Apr. 2007.

[19] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines — a survey. *Proc. IEEE*, 84(8):1090–1126, 1996.

[20] T. Margaria, M. G. Hinchey, H. Raffelt, J. Rash, C. A. Rouff, and B. Steffen. Completing and adapting models of biological processes. In *Proc. of IFIP Conf. on Biologically Inspired Cooperative Computing (BiCC 2006), Santiago (Chile)*, 2006.

[21] T. Margaria, C. Kubczak, M. Njoku, and B. Steffen. Model-based design of distributed collaborative bioinformatics processes in the jabc. In 11^{th} *IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS 2006)*, pages 169–176, Stanford, CA, August 2006. IEEE CS Press. iccecs-bio06.

[22] T. Margaria, H. Raffelt, and B. Steffen. Analyzing second-order effects between optimizations for system-level test-based model generation. In *Proc. of IEEE International Test Conference (ITC'05)*. IEEE Computer Society, November 2005.

[23] L. Mariani and M. Pezzè. A technique for verifying component-based software. In *Proc. of Int. Workshop on Test and Analysis of Component Based Systems (TACoS'04)*, pages 17–30, March 2004.

[24] M. Müller-Olm, D. Schmidt, and B. Steffen. Model-checking: A tutorial introduction. In *SAS, 6th InT: Static Analysis Symposium*, LNCS N.1694, pages 330–354. Springer Verlag, Sept. 1999.

[25] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Procȯf the 2002 Int. Symposium on Software Testing and Analysis (ISSTA'02)*, pages 229–239, Rome, Italy, July 22–24, 2002.

[26] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *ICSE 2000*, pages 35–46. ACM, 2000.

[27] I. Object Management Group. Omg's corba website. `http://www.omg.org/corba/`, seen Apr. 2007.

[28] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In J. Wu, S. T. Chanson, and Q. Gao, editors, *Proc. of the Joint Int. Conference on Formal Description Techniques for Distributed System and Communication/Protocols and Protocol Specification, Testing and Verification FORTE/PSTV '99:*, pages 225–240. Kluwer Academic Publishers, 1999.

[29] H. Raffelt and B. Steffen. Learnlib: A library for automata learning and experimentation. In L. Baresi and R. Heckel, editors, *Proc of* 9^{th} *Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2006)*, volume 3922 of *Lecture Notes in Computer Science*, pages 377–380. Springer, 2006.

[30] Y. N. Shen, F. Lombardi, and A. T. Dahbura. Protocol conformance testing using multiple uio sequences. In *Proc. of the* 9^{th} *Int. Symposium on Protocol Specification, Testing and Verification*, pages 131–143. North-Holland, 1990.

[31] B. Steffen and H. Hungar. Behavior-based model construction. In S. Mukhopadhyay and L. Zuck, editors, *Proc. of the* 4^{th} *Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2575 of *Lecture Notes in Computer Science*, pages 5–19. Springer-Verlag, 2003.

[32] B. Steffen, T. Margaria, and R. Nagel. Remote Integration and Coordination of Verification Tools in jETI. In *Proc. of* 12^{th} *IEEE Int. Conf. on the Engineering of Computer Based Systems (ECBS 2005)*, pages 431–436, Greenbelt (USA), April 2005. IEEE Computer Soc. Press.

[33] B. Steffen, T. Margaria, H. Raffelt, and O. Niese. Efficient test-based model generation of legacy systems. In *Proc. of the* 9^{th} *IEEE Int. Workshop on High Level Design Validation and Test (HLDVT'04)*, pages 95–100, Sonoma (CA), USA, November 2004. IEEE Computer Society Press.

[34] S. Vuong, W. Chan, and M. Ito. The UIOv-method for protocol test sequence generation. In J. de Meer, L. Machert, and W. Effelsberg, editors, *Proc. of* 2^{nd} *Int. Workshop on Protocol Testing Systems (IWPTS'89)*, pages 161–175. North-Holland, 1989.

[35] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In A. Petrenk and A. Ulrich, editors, *Proc. of* 3^{rd} *Int. Workshop on Formal Approaches to Testing of Software (FATES'03)*, volume 2931 of *Lecture Notes in Computer Science*, pages 60–69. Springer Verag, 2004.