USING RULE-BASED ENGINE TO SUPPORT TEST VALIDATION MANAGEMENT OF COMPLEX SAFETY-CRITICAL SYSTEMS

Valentina Accili, Giovanni Cantone
Dip. di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, Via del Politecnico 1, 00133, Rome, Italy

Christian Di Biagio, Guido Pennella
MBDA-Italy spa, Via Tiburtina km 12,400, 00131 Rome, Italy

Fabrizio Gori
Eisys S.p.A., via Torre Rigata 5, 00131 Rome, Italy

Keywords: Automatic Validation, Validation Rule, Automatic Software Test, Distributed and parallel systems.

Abstract: Testing and validating software components in distributed architecture environments are critical activities for our reference company, where those activities have been performed in a non-automatic way up to now, so spending time and human resources. As a consequence, we were charged to design and construct a flexible system, the Automated Test Manager (ATM), for the automatic software testing and automatic validation of test results. In this paper we focus on the subsystem ATM-Console that handles the validation aspect of the ATM system. This subsystem reuses an Open Source Rule-based Engine, which is able to meet our purposes. Based on results from a case study, the paper reports that introducing the ATM-Console in field could very significantly improve the efficiency of test validation.

1 INTRODUCTION

The reference company for this paper develops safety-critical systems. We have are involved with the need of that company to improve the correctness, completeness, consistency, security, and quality of the software part of those systems.

Because the architecture utilized company-wide is based on multiple distributed and parallel subsystems, the test and validation process could be not deterministic and involve very heavy jobs, especially if testing is accomplished and managed without automatic supports.

Automated Test Manager (ATM) is our answer to some of those needs. ATM is a distributed software system designed and developed to automatically test and validate real distributed systems. ATM is built on two main subsystems, as in the followings:

- *ATM - Common Core* (Grillo, 2007), which aims to test automatically the interactions that occur between the real components of the system by simulating the behavior of some of those components.
- *ATM - Console*, which is in the focus for this paper. This subsystem takes care of the remote control of automated test sessions, and validation management. ATM-Console provides the function of an *at-once* validation configuration: through a user-friendly GUI it is possible to edit the Rules that will train the validation process, which a Rule-based Engine will execute (JBoss, 2006).

Based on the needs placed by the enactors of the current non-automatic software validation process, the goals of this paper are twofold: (i) To sketch on the subsystem ATM - Console, as developed to perform an automatic software test validation, and

Accili V., Cantone G., Di Biagio C., Pennella G. and Gori F. (2007).
USING RULE-BASED ENGINE TO SUPPORT TEST VALIDATION MANAGEMENT OF COMPLEX SAFETY-CRITICAL SYSTEMS.
In *Proceedings of the Second International Conference on Software and Data Technologies - PL/DPS/KE/WsMUSE*, pages 136-143
DOI: 10.5220/0001349901360143

(ii) To provide a remote control for the other ATM component, which is in the responsibility of performing the test.

In the remaining of the present paper, Section 2 analyzes previous work on automatic software validation, Section 3 describes the adopted method to perform automatic validation, Section 4 presents the architecture and the functionalities of the ATM-Console. Section 5 shows results from a case study, which involved the ATM-Console. Some final remarks and forward to future work conclude the paper.

2 RELATED WORK

Such as other authors (Liu, Yang, Wang, 2005) (Min, Yang, Wang, 2006), we propose an automatic expert system-like validation subsystem: the validation activity is based on a *Validation Knowledge Base*, which is divided in three parts:

- *Validation Data Knowledge Base*, which is built by parsing test results. Knowledge is arranged as an object-oriented knowledge representation (S. Walczak, 1998) (Liu, Yang, Wang, 2005).
- *Validation Rules Knowledge Base*, which is composed by validation rules as provided by end-users.
- *Validation Techniques Knowledge Base*, which is in the responsibility of determining whether the validation activity was or was not successful.

Our approach substantially differs from the others known from the literature (Liu, Yang, Wang, 2005) (Min, Yang, Wang, 2006). We perform automatic test and validation on *real* distributed and parallel systems to verify if their components run properly. Instead those other approaches validate distributed *simulation* systems, i.e. they test the credibility of their simulation model; in fact, they also provide an automatic tool, but use it to compare the outputs from a simulator with the corresponding outputs from the real system.

The originality of this paper consists in the novelty of using a Rule-based Engine as the "Validator*"* (Liu, Yang, Z. Wang, 2005): given the Validation Data Base and the Validation Rules Base, our ATM-Console is able to detect which validation rules are actually verified by matching the given Validation Rules with Validation Data.

3 A RULE-BASED ENGINE FOR AUTOMATED VALIDATION

It involves complex verifications, like checking the periodical transmission of a message, to enact automated validation of functional tests in distributed, interactive, and real time systems. So our decision was to use a Rule-based Engine to achieve the automatic validation activity.

A Rule Engine can be viewed as a sophisticated interpreter of logical implications: in fact, it evaluates and executes rules that are expressed in terms of *if-then* statements.

The power of those rules lies both in their ability to separate knowledge from its implementation logic, and in the fact that we can change those rules without having to act on artifacts in source code.

It dictates a Java runtime API for rule engines, the specification for the Java Rule Engine API (JSR 94) (Sun, 2005), as developed by the Java Community Process (JCP) program. In fact, such an API provides a simple means to access a rule engine from an up to date Java Platform.

Drools (JBoss, 2006), Fair Isaac Blaze Advisor (Fair Isaac, 2007), ILOG JRules (ILOG, 2007), and Jess (Sandia, 2007) are instances of JSR 94 compliant rule engines. ATM-Console adopts JBoss' Drools (Dynamic Rule Object-Oriented Language System) rule engine (JBoss, 2006), because it is a solid inference engine, completely Open Source, well documented, and with an Eclipse plug-in, which is useful for debugging.

Drools uses the Rule Based approach to implement an Expert System; more correctly, it is classified as a Production Rule System, which focuses on knowledge representation to express propositional and first order logic in a concise, non-ambiguous, and declarative manner. The "brain" (JBoss, 2007) of such a Production Rules System is an *Inference Engine*. In order to infer conclusions, which result in actions, this inference engine matches facts, the data, against the Production Rules (also called Productions or just Rules in the followings). A Rule, in Drools, is a twofold structure with Left (LHS) and Right (RHS) Hand Sides.

The syntax for a rule is:

```
when
    <condition>
then
    <action>;
```

When the condition is met, the action is executed. As already mentioned, first order logic is

used for representing a `condition`. Multiple conditions (and/or actions) can be utilized in a `when` construct (and its `then` part).

The inference engine performs Pattern Matching, i.e. the process of matching new or existing facts against rules. There is a number of algorithms that inference engines can use for pattern matching, including: Rete (Forgy, 1982), Treat (Miranker, 1990), Leaps (Don Batory, 1994). The Drools tool implements both Leaps and Rete, but ATM-Console is based on the latter (Forgy, 1982).

The *Production Memory* stores the rules, and the *Working Memory* stores the facts that the inference engine matches against. Facts are asserted into the Working Memory where they may then be modified or retracted.

The reference environment supports a large number of rules and facts, and conflicting rules could occur. *Agenda* is the name that Rete gives to the component that manages the execution order of the conflicting rules, if any is active, by using a *Conflict Resolution Strategy*. When a Rule is activated, it is placed onto the Agenda for potential future execution. It is called the *Consequence* of a rule, the set of actions that the rule's RHS includes.

The engine operates recursively in a 2-phases mode:

- *Working Memory Actions*, which define where most of the work takes place. There are two loci for such an Action: the `main` Java application process, and the Consequence set. Once a Consequence has finished or the main Java application process invokes the method `fireAllRules()`, the engine switches to the next phase.
- *Agenda Evaluation*, which attempts to select a rule to fire, whether such a rule is not found yet, but it might still exit; otherwise it switches the phase back to Working Memory Actions, and the process begins again and proceeds until the Agenda is empty.

As effect of a Working Memory Action, some rules may become fully matched and eligible for execution; a single Working Memory Action can result in multiple eligible rules. When a rule is fully matched, an *Activation* is created and placed onto the Agenda. Such an Activation keeps its references with that rule and the matched facts.

Conflict resolution is required when there are multiple rules on the agenda. As firing a rule may have side effects on the Working Memory, the rule engine needs to know in what order the rules should fire (for instance, firing rule A may cause rule B to be removed from the agenda).

The Drools engine supports two types of conflict resolution strategies: *Salience* and *LIFO*.

`Salience` is a subtype of the Natural numbers. It allows end-users to specify that a certain rule has higher priority than other ones. In a selection, higher salience rules are always preferred to lower priority rules. *LIFO* strategy is then applied to rules with the same priority.

The *LIFO* strategy enacts a last in, first out policy. It is based on the value of a counter, which Working Memory Actions assign to rules. Multiple rules, which receive the same counter value, are placed in a common *Agenda Group* if and only if the same Action created them. In case, they are randomly selected for execution.

The ATM-Console subsystem allows end-users to edit the LHS part of *Validation Rules* through a Java based Graphical User Interface. An end-user can both enter these *Rules* from the scratch, at every validation session, by editing the ATM-Console's Validation Rules Editing Form, and/or load them from an XML repository.

Every time an end-user edits a new rule, this enters the *XML Rules Repository*. As already mentioned, it is possible to define validation logic without affecting artifacts at the source level code, which makes dynamic and flexible the validation logic, in the user view.

Parsing files where the interactions between components are stored, as a result of a test, creates facts, which constitute the Working Memory.

In order to perform a Validation, the *Rule Set* and *Facts* are the only inputs needed by the Rule Engine.

4 ATM - CONSOLE

ATM – Console is a software subsystem which provides: (i) an instrument to remotely configure and control the ATM – Common Core: the emulating subsystem; (ii) a flexible and easy-to-use environment to configure a validation session for test results.

4.1 Architecture

The ATM-Console system consists of four macro-units:

1. **Test Configuration**: Manages the test configuration activity, by: (i) providing the end-user with the simulation configuration files, as available in the ATM-CC, (ii)

getting the files this user selects, and (iii) sending this selection to the ATM-CC.

2. **Test Control:** Manages the test phase, displays information concerning test progress, and shows operator-consent requests to end-user.
3. **Validation Configuration:** Manages the validation configuration activity during which the end-user edits the Validation Rules and/or selects them from the XML repository.
4. **Validation Control:** Builds both the Working Memory by parsing test results, and the Production Memory by utilizing the DRL (Drools Rule Language) session file, as written by the previous macro-unit; it also manages the validation task by running the Rule Engine.

4.2 Remote Control

An *ad-hoc*, RPC-based protocol allows the ATM-Console to remotely communicate with the ATM-CC subsystem. A proper interface exposes the ATM-Console's responsibilities, which consist in:

- Getting connection with the remote ATM-CC subsystem.
- Asking ATM-CC for the XML configuration file by specifying the involved ATM-CC's configuration directory.
- Communicating the selected configuration to the ATM-CC.
- Sending operator-commands (Start and Stop Simulation) to the ATM-CC.
- Asking for information about the progression of an ATM-CC simulation.
- Sending operator-consents, if any requested by the ATM-CC to set the remaining of a requested simulation.
- Asking the ATM-CC for, and getting test results.
- Communicating results from a validation session.

4.3 Usage for Test Validation

At the end of a test session, the operator can perform the *Validation Configuration*; by such a use case, as already mentioned, the ATM-Console allows the operator to edit Validation Rules from the scratch and/or load some of them from an XML Repository in which all previously defined validation rules are stored.

Rules that are present in the Repository are shown in an easy-to-read Rule Repository Table.

As we want to validate distributed system tests, objects of our validation are the interactions between the constituting components. These interactions are characterized by messages as exchanged between components. An example of validation *Rules* is:
*"Verify **if** the message X, with field constraints F1,F2,...Fn, has been sent on the connection Y within Z seconds."*

The end-user can set one or more test results files, in order to verify the presence of every expected message. All these files will be used to build and populate the *Validation Data Knowledge Base*, as the next section describes.

Additionally, the ATM-Console's Rule Editing Form gives end-users the possibility of binding multiple rules, so composing complex rules. Concerning this point, an example follows (where the keyword "since" means to test for the occurrence in the past):
*"Verify **if** the message X2, with field constraints set to F21,F22,...F2n, has been sent on the connection Y2 within Z2 (= Z1 + ΔT2) seconds **since** the message X1, with field constraints set to F11,F12,...F1m, was sent on the connection Y1 within Z1 seconds."*

Every rule the user edits is added to the Rule Set and finally the DRL rule is written.

4.4 Validation Data Knowledge Base

Let us focus now on the construction of the Drools' Working Memory. As already mentioned, we perform distributed systems test validation by checking the exchange of messages between system's components. The remote subsystem ATM-CC stores these messages in some files, which the ATM-Console gets from the ATM-CC at the beginning of a Validation session. Based on the contents of these files, the ATM-Console builds the *Validation Data Knowledge Base*, i.e. the Working Memory.

As JBoss Rule Engine requires, we model facts as "beans"; they are asserted into the Working Memory. Facts are thus Java objects of any kind, which a rule can access together with their attributes by proper access methods. The Rule Engine does not clone facts at all; it just utilizes object references. The ATM-Console creates objects by parsing test result files, and asserts facts in the Working Memory.

Because of the messages variety, it is variable the number of fields in a message, thus the attribute `fields` of the class `Message` has been represented as a `List of Fields`; this to take advantage of some features of the JBoss Rule Engine, as we will see in the next section. So message's fields are represented as objects to assert in the Working Memory.

4.5 Validation Techniques Knowledge Base

There are many methods to validate a test, including subjective validation methods and statistical validation methods. Subjective validation methods include Turing testing (Turing, 1950), sensitivity analysis (Archer, Saltelli, Sobol, 2006), and graph methods (Bollobas, 1998) (Jungnickel, 2003); they are easy to use, but depend on the subjectivity of the person who performs in the role of analyzer. Statistical validation methods include confidence intervals, hypothesis testing, and time series (Archer, Saltelli, Sobol, 2006); they are used to quantitatively compare real outputs with human-generated or simulation-generated outputs, and give "objective" conclusions, in the limits established by the statistical level of significance, and test power obtained (Wohlin, Petersson, and Aurum, 2003). Of course, our case takes in consideration real outputs and simulation-generated outputs.

As already mentioned, it is in the context of safety-critical distributed systems any of the application that we have to test. Consequently, it is strictly constrained the set of test validation criteria that we can apply to test-results. In our approach, in order to complete with success, a validation has to meet all the defined rules, i.e. to verify each end-user given rule, rather than statistical selected ones.

4.6 Knowledge Base for Validation Rules

Let us consider now the construction of the Production Memory, where Production Rules are stored.

A Production Rule System's Inference Engine is stateful and able to enforce truthfulness or "Truth Maintenance" (JBoss, 2006): in practice, logical expressions are declared to hold for actions, that is the state of an action depends on the inferences that still remain true; when it is no longer true the logical dependant action is undone.

There are two exemplar modes of execution for a Production Rule System: Forward Chaining, and Backward Chaining.

Backward Chaining is goal-driven: we start by providing a conclusion that the engine tries to satisfy. If it can't, then it searches for limited conclusions, i.e. sub-goals that help to satisfy an unknown part of the current goal. The engine continues to enact that process until either the initial goal is proven or there are no more sub goals to analyze. Prolog models a kind of a Backward Chaining engine.

Forward Chaining is data-driven: facts are asserted into the working memory, which results in one or more rules being concurrently true and scheduled for execution by the Agenda. The process starts when a fact is provided; this fact propagates, and the process ends in a conclusion. Drools is a Forward Chaining engine.

A Drools' Production Rule has the following further attributes, many of which we already mentioned above:

- `salience`, which determines the priority of the rule in execution.
- `agenda-group`, which allows the user to partition the `Agenda`, so enhancing on the execution control: only rules in the focus group are allowed to fire.
- `auto-focus`: when a rule is activated, if the value of this attribute is true and the rule's agenda-group does not have focus then it is given focus to such a group, allowing that rule to potentially fire.
- `activation-group`: rules that belong to the same named activation-group will only fire exclusively. In other words, it cancels the activations of the remaining rules, so removing their chances to fire, the first rule that fires, in an activation-group.
- `no-loop`: when a fact is modified as a consequence of a rule activation, it may cause that rule to activate again, causing recursion. This attribute helps in managing such occurrences.

Rules are written in DRL files, the Drools preferred file format. DRL files eventually are simple text files in which multiple DRL rules, functions etc. are stored.

When the user has finished with editing, the Validation Configuration unit first converts rules from the input format to DRL, and then writes these DRL rules to a the *Rule File* for input to the Rule Engine.

Example 1 presents some details about a *Rule* in DRL language.

Example 1: A Rule in DRL language.

```
rule rule0
no-loop true
salience 103
when
   f0: Field(key == "VALUE",
             value == "REACHED")
   m : Message(msgType == "AAAA",
            fields contains f0)
   r:  Rule(ruleID == 1)
 then
   r.setVerified(true);
    modify(r);
end
```

In order to achieve the result wanted for every end-user given rule, the ATM-Console first creates a Java object Rule, which contains the message information that we want to verify, and then it builds an entry for that information in the DRL file.

Apart from the facts coming from test output files, the ATM-Console also asserts Rule objects in the Working Memory. The subsystem uses those objects to bind rules, so implementing dependence relationships.

As Example 2 shows, when a rule has been verified, the ATM-Console sets the `isVerified` object attribute to true, which propagates to linked rules, so activating their checking.

Example 2: Rule binding.

```
rule rule3
no-loop true
salience 4
when
   f0: Field(key == "VALUE",
             value == "START")
   m : Message(msgType == "AAAC",
              fields contains f0)
   r:  Rule(times: receivedTimes,
            ruleID == 8)
   r0: Rule(ruleID == 2,
            isVerified == true)
   eval(r.verifyBoundRuleTime(
          times, r0);
then
   r.setVerified(true);
   modify(r);
```

Additionally, concerning Rule's programming good practices, Example 2 shows that it should be as lower the `salience` of a rule as greater is the number of binds of that rule: i.e. the greater a rule depends on other rules, the lower should be its priority. By the way, let us also note that, in order to avoid recursion, the attribute `no-loop` was set to true both in Example 1 and Example 2.

Moreover, those examples show that using the following pattern, the ATM-Console can test whether there is an object (o) of class `Class` in the Working Memory, which attributes `a1` and `a2` are set as x and y, respectively:

```
o:Class (a1 == x, a2 == y)
```

Furthermore, Example 2 shows another important issue: based on the DRL, it is possible to invoke Java functions indirectly by the `eval` instruction, in case of LHS part of a rule, otherwise directly (rule's RHS).

Every change that can affect rules' checking has to be notified to the Working Memory, which contains all asserted facts, and this is made by invoking the method `modify(o)` for the modified object `o`.

If a fact isn't yet true, it can be removed from the Working Memory by calling the method `retract(o)`.

As already mentioned, fields are stored in Message object as a variable List, and thanks to the Drools' operator `contains`, the ATM-Console can easily verify if any of the fields of a Message's instance contains specified key and value.

5 CASE STUDY

This section describes a comparative study between the execution's modality of the validation activity, as currently many industrial sites adopt, the reference company for this paper included, and the modality introduced by the sub-system ATM-Console.

Because it is very high the complexity of the systems that we are addressing, like Command and Control systems or safety critical systems, many loops are performed on development and testing in the software life-cycle. In practice, testing and validation are continually repeated until the system satisfies all the defined requirements. Currently, the reference company validation activities are manually performed. Due to the confidential nature of data object of this study, the reference company does not allow to disclose details about the test results.

In case of manual validation, the effort of validation activities depend on the following factors:

- *Human Knowledge* (HK): Validation implies the knowledge of the problem domain.
- *Analysis of Results* (AR): Validation always implies an analysis on test results

Practically, analyzing and correlating many thousands of complex messages are the activities to

perform for manual validation. Those messages are stored and eventually distributed on many files. Typically, each message has hundreds of fields to check.

Vice versa, it is independent on HK and AT, the automatic validation, as executed through the ATM-Console system. In fact, it describes test cases and procedures for qualification testing, the Software Test Description (STD), as defined in the MIL 498 standard (DOD, 1994). In practice, the STD reports on a list of indications, like:

- Do .... (Test object)
- Control... (Validation object), where such a control is like *Verify if X exists after/before/within Y*.

Let us note explicitly that a standard compliant ATM end-user is in the only responsibility of entering the STD information into the ATM GUI.

The STD document is arranged in a tree structure. The root level defines a list of *Formal Qualification Testing* (FQT); each FQT defines a set of *Tests Cases*. Each *Test Case* includes many *Checks*.

Referring to application systems as the industry nowadays typically develops, and taking into account average values, an STD contains 250 FQT(s), each FQT defines 20 test cases, each made up on 10 Checks.

In case of manual validation, let each test case be performed in the average period AT. This results from the sum of two times, AT_1 and AT_2:

- AT_1: Localization of the Meaning Information (e.g. e set of messages). Typically, the end user has to locate, inside the test result files (thousand of lines), the information that is meaningful for the test case.
- AT_2: "Meaning Delay". Time necessary to check the detected meaningful information.

The following factors influence AT_1:

- Total of the expected messages.
- Message throughput (average m/s).
- Duration test time.
- Triggers' complexity (boundary place).
- Number of source files.

The following factors influence AT_2:

- Verification difficulties.
- Number of field for each message.
- Message heterogeneity.
- Number of messages.

To the best of our knowledge, it is observed that on average, for a single human resource, AT_1 is 5 minutes, and AT_2 is 10 minutes.

The overall time to manually perform a typical STD verification activity follows:

$$AT_V = (AT_1 + AT_2) * AN_{FQT} * AN_{TC} \cong$$
$$\cong \ 8 ManMonths$$

where: AT_v indicates the average time to complete a validation activity, AN_{FQT} is the average number of the expected FQT for a validation activity, and AN_{TC} is the average number of expected Tests Cases for every FQT.

In case of ATM-Console-based automatic validation, only the time necessary to insert validation rules affects the validation time. For such insertion activity, we observed an average time of 10 minutes for every test case, which is made up on an average of 10 Checks of medium difficulty, and with several links between the various Checks.

As it has been previously mentioned, the reference company's software systems are subjected to repeated validation activities. Thus, ATM-Console introduces a remarkable improvement in terms of saving both time and human resources: in fact, starting from the second validation activity, in which the whole Rule Set concerning the various test cases is already available, we observed that the execution validation time duration is 20 seconds for each test case in the average.

Therefore, in case of automatic validation, it shows:

$$AT_{VAUT} = (AT_{ins} * AN_{TC} * AN_{FQT}) +$$
$$+ (AT_{TC} * AN_{FQT} * AN_{TC}) \cong 5 ManMonths$$

where: AT_{ins}=10 minutes is the average time for inserting the test case in the GUI form, and AN_{TC}=20, AN_{FQT}=250. Additionally, AT_{TC}=20 seconds indicates the average time to execute the automatic validation of such a test case.

Rather, for the subsequent validation activities, it disappears the last equation's first addend, which represents the insertion time of the Checks' characteristics, thanks to the ATM-Console's feature to hold the history of any test case. Therefore, it shows:

$$AT_{VAUT} = (AT_{TC} * AN_{FQT} * AN_{TC}) \cong$$
$$\cong 27 \quad ManHours$$

This outstanding result concerns just the validation activities, without taking care of the time for inserting rules, and reports a 97,8% improvement. Also taking in count the insertion of the validation rules, it registers a 25% improvement the time spent for automatic validation compared to the manual validation.

Because the considerations above result from a case study, their validity is threaten and they need confirmation by extensive accreditation in field.

6 CONCLUSIONS AND FUTURE WORK

In order to configure the validation rules and execute the validation process of medium/large safety-critical systems by utilizing the power offered by an open-source Rule Based Engine, this paper presented the concept and architecture of a novel flexible subsystem that we designed and developed to remotely control the simulation of some application system components (one simulated component in the present version of the subsystem) and their interactions with the remaining real distributed components for failure identification.

Results from an industrial case study shows that the presented subsystem could provide great support in performing distributed systems testing, saving management and execution efforts that relate to tedious complex operations, and assuring a very interesting return on investment.

Using rule engines to perform test validation of safety-critical systems could have a large exploit in future, such as is happening in the application domain of business management. In such a scenario, it could deliver significant benefits the improvement of the presented subsystem by including validation criteria for test configuration and providing features to select the criterion to apply from a given set of those criteria.

ACKNOWLEDGEMENTS

Authors would like to thank people in the Division of Applied Software Research of the MBDA-Italy Spa for the support provided in analysis, design, and realization of the ATM-Console subsystem.

REFERENCES

Archer G., Saltelli A., Sobol I. M., 1997 *Journal of Statistical Computation and Simulation*. Taylor & Francis.

Bollobas B., 1998. *Modern Graph Theory*, Springer Verlag.

DOD, 1994. Software Test Description, http://www2.umassd.edu/swpi/DOD/MIL-STD-498/STD-DID.PDF.

Fair Isaac, 2007. *Blaze Advisor Online Documentation*. http://www.fairisaac.com/fic/en/product-service/product-index/blaze-advisor/, 29 March 2007.

Forgy C. L., 1982. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, North Holland Conference, pp. 17 – 37.

Grillo A., Cantone G, Di Biagio C., Pennella G., 2007. Automatic Test Management of Safety Critical Software: The Common Core - Behavioral Emulation of Hard-Soft Components, Proceedings of ICSOFT 2007 (to appear).

ILOG, 2007. JRules Online Documentation. http://www.ilog.com/products/jrules/, 29 March 2007.

JBoss, 2007 JBoss Rules Online Documentation http://labs.jboss.com/portal/jbossrules/docs.

Jungnickel D., 2003. *Graphs, Network and Algorithms*, Springer.

Liu F., M. Yang, Z. W.-S. Wang, 2005. Design and development of an expert system-like validation tool for distributed simulation systems., In *Fifth IEEE IC on Machine Learning and Cybernetics*, CS Press.

Min F.-Y., Yang M., Wang Z.-C., 2006. An intelligent validation system of simulation models. In *Fifth IEEE International Conference on Machine Learning and Cybernetics*.

Miranker D.P., 1990. TREAT: A New and Efficient Match Algorithm for AI Production Systems. In *Research Notes in Artifical Intelligence*. Pitman Publishing Ltd.

Don Batory, 1994. The LEAPS Algorithms. In *Technical Report 94-28*, Department of Computer Sciences, University of Texas at Austin.

Sandia Lab., 2007. *Jess Online Documentation*. http://herzberg.ca.sandia.gov/jess, 29 March 2007.

Sun, 2007. *Getting Started With the Java Rule Engine API (JSR 94): Toward Rule-Based Applications*. http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html.

Turing A., 1950 Computing machinery and intelligence. In *Mind*, vol. LIX, no. 236, pp. 433-460.

Walczak S., 1998. Knowledge Acquisition and Knowledge Representation with Class: the Object-oriented Paradigm. In *Expert Systems with Applications*, No. 15, pp.235 - 244.

Wohlin, C., Petersson, H., and Aurum, A., 2003. Combining data from reading experiments in Software Inspections. In *Juristo, N. and Moreno, A. (eds.) Lecture Notes on Empirical Software Engineering*, World Scientific Publishing.