

PATTERNS AT WORK: PREDEFINED PROPERTY SPECIFICATIONS FOR THE ANALYSIS OF DISTRIBUTED SYSTEMS

by

Hesham Hallal^{1,3}, Alexandre Petrenko¹, Sergiy Boroday¹, Andreas Ulrich²

¹ CRIM, 550 Sherbrooke West, Suite 100, Montreal, H3A 1B9, Canada,
{Hallal, Boroday, Petrenko}@crim.ca

² Siemens AG, Corporate Technology CT SE1, Otto-Hahn-Ring 6, 81730 München, Germany,
andreas.ulrich@siemens.com

³ Department of Electrical and Computer Engineering, McGill University, Montreal, Canada

Prepared for the 2004 Telelogic Americas User Group Conference

Abstract

PATTERNS AT WORK: PREDEFINED PROPERTY SPECIFICATIONS FOR THE ANALYSIS OF DISTRIBUTED SYSTEMS

The notion of patterns is increasingly being used in the design, test, and analysis of systems, e.g., in the Rational Suite (recently acquired by IBM) and the Bandera project (University of Kansas). In particular, patterns that encode system requirements and can be used in the analysis phase are getting more common in the field of software engineering. Here, we describe an approach that targets the problem of analyzing and testing distributed systems, and whose implementation -within the ObjectGEODE (OG) toolkit of Telelogic- relies on the concept of property specification patterns. The approach consists of using traces of distributed systems to regenerate a model of the system under test in the Specification and Description Language (SDL) and using the property specification language of ObjectGEODE, GOAL, to express properties of interest. The model checker of OG is then used to verify the system against the specified properties. To facilitate the task of developers/debuggers, we construct a library of parameterized property patterns in GOAL that can be used in the verification of various applications. We also discuss how the library of GOAL patterns and the possible manipulations on them can add to the usability of the language and the toolkit.

Author Biography

[Hesham Hallal¹, Alexandre Petrenko¹, Sergiy Boroday¹, Andreas Ulrich²]

[Hesham Hallal] has a Master's degree in electrical and computer engineering. He is currently pursuing his Ph.D. degree in electrical and computer engineering. His Ph.D. studies focus on the topic of testing and controlling distributed systems using formal methods. He has four years of engineering experience and four years of research and development experience. Currently he is a senior research officer at CRIM.

Alexandre Petrenko, Ph.D. in computer science, is a senior researcher and team leader of Distributed System Analysis Group at CRIM (Computer

Research Institut of Montreal). He sits on program committees of several international conferences such as TESTCOM, FORTE, ICNP, and FATES. He also gives invited talks at the main research institutes around the world. Alexandre has published over 150 papers and supervised many master and Ph.D. students. His research interests include formal methods and their applications in the distributed systems and computer networks.

Sergiy Boroday has a Ph.D. in testing theory. He has been working in research and development for more than nine years. Currently he is a senior research officer at CRIM. He is winner of the 3rd degree diploma at the Intl. Conference "Students for Peace and Scientific Progress" (Russia, 1992) and the best paper award in the FORTE/PSTV'99 (China, 1999). His research interests include formal methods, automata theory, testing theory and methodology, telecommunication protocols.

Andreas Ulrich received his Ph.D. in Computer Science from Magdeburg University, Germany in 1998. He then joined Siemens Corporate Technology Division, where he works as a Principal Engineer in the Department of Software & Engineering until now. His main task is to provide consultancy to Siemens' business units in the area of testing and quality assurance. He is also active in the research area of model-based system analysis, verification, and testing.

SCENARIO

Distributed systems are typically hard to design and analyze. In recent years, numerous approaches and techniques were the subject of research and development of many specialized groups in both academia and industry. Our group, the Distributed System Analysis group, at CRIM focuses on exploring the aspects of applying formal methods in the design and analysis of distributed applications. In an ongoing research project, the task has been to develop new technologies and tools to facilitate the development and testing activities of distributed systems. Consequently, a formal approach has been devised to check user-defined properties on execution traces collected by monitoring the applications during runtime. The developed approach relies on model checking, and it was implemented within the ObjectGEODE (OG) environment of Telelogic.

The trace analysis approach, which is summarized in the diagram in Figure 1, could be outlined as follows. The distributed system is instrumented in a way that events executed by the distributed processes are collected. Such events typically denote the send and receive of messages, local actions and others. A trace is produced that includes all the events collected during a system run, and an appropriate analysis tool can be applied to check the trace against some user-defined properties. A large body of work on developing various tools to visualize traces already exists, see [4] for a summary. The main goal is to facilitate the efforts of the designer or tester for locating and correcting bugs by filtering out unrelated information and by offering a proper visualization of traces. The analysis is performed manually either online (simultaneously with the system execution) or post mortem. Another group of methods targets the analysis phase by offering means to verify certain properties in the distributed system under test (SUT). Practically, when it comes to developing a tool for checking properties in execution traces, one would tend to reuse general-purpose model checkers, normally reliable, highly sophisticated, and versatile analysis tools. Therefore, given that a proprietary monitoring tool provides an event trace that contains sufficient information about the main characteristics of a SUT. Then, for an available model checker, ObjectGEODE in our case, we had only to develop a front-end tool that produces a formal model and a property description as required by the model checker. The model is written in the Specification and Description Language (SDL), and the properties are written in the property specification language GOAL, specific to ObjectGEODE. The results of the verification include a verdict whether the property was met in the model of the system and examples to show the execution sequences that satisfied/violated the property.

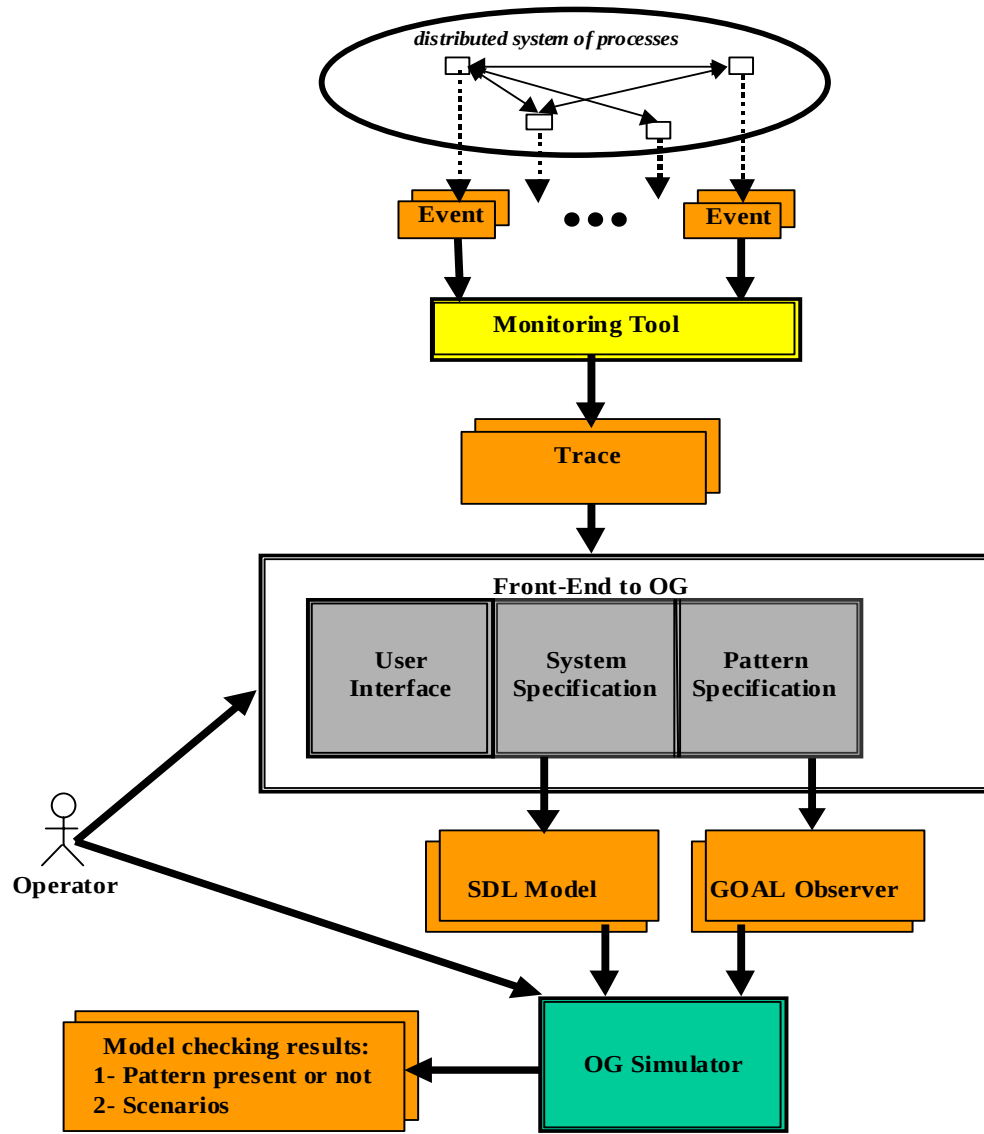


Figure 1: Workflow of the trace analysis approach.

OBJECTIVE/PROBLEM

Given the availability of OG, the implementation of the trace analysis approach reduces to the following two tasks:

- A. Model generation: building an SDL model from the collected trace. The SDL model of the tested system should reflect the following aspects:

structure, behavior, communication, and data. The structure of the system is modeled by the hierarchy: system/block/process/procedure statements in SDL. In our case, it is sufficient to define a system with a single block that is composed of several processes. The overall behavior of the system is modeled by the joint behavior of a set of communicating processes in SDL. These processes correspond to entities of the real system whose behaviors are recorded in the trace; thus making each of the processes linear, as in most cases we cannot identify any two states of it, so no cycles can be deduced. The representation of a process can be obtained by projecting the collected trace into the set of events it executes: send, receive, and local events and subsequently inserting states in between communication events, while representing local events as SDL tasks. The asynchronous communication between processes is achieved via signals with optional signal parameters (that represent exchanged data) and channels. Input signals to a process are stored in a queue before reading them. Thus, unknown delays in real communication channels of the distributed system are represented by those of input queues, associated to each SDL process. This means that in our framework, the whole communication media of the system is simply modeled with individual queues of SDL processes. To achieve this task, we have developed the front-end tool TRAYSIS (for TRACE ANALYSIS), Figure 2, which currently accepts logfiles with either synchronous or asynchronous communications. Its main task is to automate the process of producing an SDL specification from a logfile of events specified in XML syntax. The tool offers the following main features:

1. Conversion of message logfiles into event logfiles. The tool follows the approach described earlier in this section to convert a message logfile into a set of local traces of events corresponding to each process in the SUT. Each message is converted into an event record, which is uniquely identified and totally ordered within the issuing process.
2. Event logfile treatment and model construction. The tool checks the logfile of events. In case of missing communication events, the tool informs the user and converts the unmatched communication events into local events of the processes. The tool also checks for flaws in the logfile that would render the generated SDL model syntactically incorrect, e.g. the presence of SDL keywords in the logfile and the use of some illegal characters. Once a logfile is successfully checked, the tool generates the SDL model of the system. Causality cycles are then detected by OG during simulation of the generated SDL model as they cause deadlock prior to the termination of the processes.
3. Customized model generation. This feature enables the user to cope with large logfiles. The tool offers three types of filtering: process

filter, signal filter, and a filter that extracts a segment from the initial logfile.

4. Statistics about the SDL model. The tool offers listings of processes, variables, and signals occurring in the model to help the user better understand the generated model.



Figure 2: The TRAYSIS tool.

- B. Property Specification. Writing the property descriptions in GOAL for trace verification. As in the case of many formal verification techniques, the property specification mechanism is essential to the success of the trace analysis approach. In our case, we use the GOAL language of OG, which is an automata based language that is similar to SDL, but has some syntactic and semantic differences [1]. Properties are expressed in GOAL as observers [3], where an observer implements in fact a finite automaton with accepting states [7]. Observers are usually described in terms of entities (objects, signals etc.) of the SDL model of the tested system. This makes communication signals directly accessible for observation while other model data, e.g., variables and states, are accessible to observers using probes, which represent pointers to SDL entities. In addition, GOAL allows the declaration of two types of designated states: *success* states and *error* states. Entering success states (error states) indicates that the system

respects (violates) the property expressed in the observer. The use of the success/error convention is completely up to the user and has no formal meaning.

Actually, the task of specifying properties presents the following two problems:

1. The range of properties that are of interest to the analysis of distributed systems; in particular, those expressible in GOAL.
2. The degree of automation that can be introduced to the mechanism of property specification in our framework.

We discuss our approach to solve these two problems in the following section.

SOLUTION

In this section, we discuss the properties of distributed systems that can be verified with our approach and the automation process.

In general, we focus on properties that can be seen as requirements for the system to fulfill. These properties are normally tested against a model of the system or a representation of the system behavior as is the case in trace analysis. To address this issue we turn to the properties that are believed to be mostly used in practical applications. The existing research in the field has explored a wide range of properties that can be sought in distributed systems. On a first level, such properties can be classified into two types: state based and event based. Event based properties allow detecting simple atomic or composite events in the behavior of the SUT. State based patterns, on the other hand, formulate assertions on the state variables of the processes in the SUT. Meanwhile, the work of Dwyer et al. at Kansas State University to build a repository of specification patterns -well understood, but imprecise conceptions of system behavior- shows an effort to cover both types of properties [2], [6]. This online repository includes patterns that are commonly occurring in specifications of distributed and reactive systems, which reminds us of the notion of design patterns in software engineering. Each pattern in the repository models the essential aspects of a particular behavior of the system. Meanwhile, the repository is not static, so new patterns can always be added. Finally, to make the specification of patterns portable, a mapping to several formalisms (LTL, CTL, GIL, QRE, and INCA Queries) has already been provided in [6]. Mappings to other formalisms have also been developed by other researchers.

For our project, we plan to build a repository of typical and frequently used specification property templates in GOAL based on the existing repository. This means that we relieve the developer from the task of writing specifications from scratch. In addition, we will have a dynamic library of GOAL observers that can be expanded whenever new properties are considered. In the following, we describe how we built our library of GOAL observers and interfaced it to the trace analysis approach. First however, we give an overview of the existing repository of property patterns.

The Pattern Repository

The patterns in the repository are described in terms of the types of system behaviors they model (intents) and scopes of system behavior, where a property should hold. The repository consists of two main groups of patterns:

1. *Occurrence patterns*. They describe the occurrence (or absence) of an event (state)/sequence of events (states) in the system computation. There are four occurrence patterns:
 - Absence expresses the freedom of a portion of a system's execution from certain events or states.
 - Universality describes a portion of a system's execution that contains only events (states) where a desired property is met.
 - Existence expresses the presence of a certain event or state in a system's execution.
 - Bounded Existence describes the presence of a specified number of events or states in a system's execution.
2. *Order Patterns*. They express the ordering of several events that can exist in the computation under test. There are four order patterns:
 - Precedence expresses the relationship between a pair of events (states) where the occurrence of the first is a necessary condition for the occurrence of the second.
 - Response describes a cause-effect relationship between a pair of events (states).
 - Chain Precedence describes the precedence relation between two sequences of events (states).
 - Chain Response expresses the response relation between two sequences of events (states). The first sequence is considered as the chain stimulus, and the second is considered as the chain response.

The above classification can be summarized in the diagram in Figure 3.

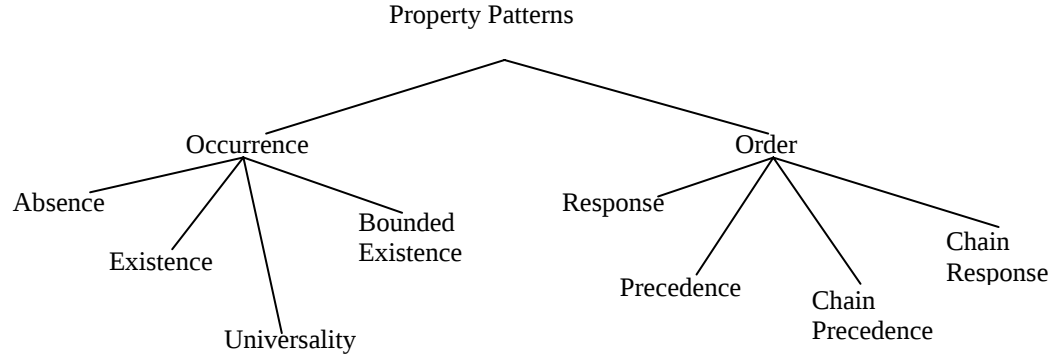


Figure 3: The pattern classification [6].

Scopes

Each pattern is defined over a scope that expresses the extent of the execution, over which the pattern must hold. Five basic kinds of scopes exist in the repository. A scope of a pattern can be:

- *global*: the complete execution,
- *before*: up to a given state/event in the execution,
- *after*: the part of the execution after a given state/event,
- *between*: any part of the execution from one given state/event to another given state/event, or
- *after-until*: like *between* but the designated part of the execution continues even if the second state/event does not occur.

In the repository, each scope is determined by specifying state/event delimiters for the pattern: the scope consists of all states/events beginning with the starting state/event and up to but not including the ending state/event [6].

Parameters

Each pattern in the repository is a parameterized expression that includes parameters to represent the actual predicates specifying the property and to express the scopes of the property. The parameterization of the patterns in the repository adds to its usability and portability to several application domains. To use the patterns from the repository, one has only to instantiate the parameters of the property pattern he needs and use it according to his specific context.

Mappings in the Repository

The patterns of the repository are expressed in several different languages to ensure portability and suitability to various verification environments. The patterns are mapped into temporal logics (CTL and LTL), quantified regular expressions (QRE), graphical interval logic (GIL), and inequality necessary

condition analysis language (INCA). The temporal logics are well known in academia, while extended regular expressions have a long success story in theory and in practice as well. The remaining formalisms are also being used in various verification frameworks. Figure 4 shows an example of the *existence* (scope: *global*) pattern with representations in the different formalisms in the repository.

<i>Existence Globally</i>	
LTL	$\langle \rangle (P)$
CTL	$AF(P)$
QRE	$[- P]^*; P; .^*$
GIL	$[\text{---} \diamond \text{---}]$ P
INCA	<pre>(defquery "global_existence_of_p" "nofair" (omega-star-less (sequence (interval :initial t :forbid '("P")) (interval :perpetual t :forbid '("P")))))</pre>

Figure 4: Existence pattern with global scope.

In the following, we discuss mapping the repository into a library of GOAL observers.

GOAL Pattern Library

The decision to use GOAL for property specification was partially dictated by the suitability of the SDL language for trace modeling and by the capabilities of the available SDL verifiers. However, the use of this automata based language in expressing the property specifications has its own advantages. While being at a lower level, automata allow one to specify more complex patterns than the other formalisms. Actually, logic formulae or regular expressions become difficult to understand for non-trivial properties (e.g. consider usual complaints on low readability of pattern based Perl programs). As an example, consider properties that involve counting, where the corresponding LTL or CTL expressions are quite lengthy and cumbersome. In addition, automata may be more appealing to regular users since they resemble usual programs and provide some visualization. On the other hand, in translating the existing repository patterns into GOAL observers, we have to decide which formalism (from the repository) to use as the source of patterns.

Actually, the literature shows a variety of researches on the translation of LTL formulae into automata. However, the most challenging problem would be the determinization of the obtained automata, which may cause state explosion. Another limitation of automata based languages is that there are no means to express complicated branching logic relations (as in CTL). Therefore, we choose to go with QRE patterns to build our GOAL library. After all, regular expressions and automata have lots in common between them. On the other hand, notice that the repository does not map patterns in automata and there are only few preliminary works on this topic.

Building Library in GOAL

We use the QRE specifications in the repository to build our GOAL library of observer patterns. We implement a conversion tool that takes a modified form of the QRE specifications as input and produces GOAL specifications of observers. The produced observers are also parameterized, which means that the produced library can be used in any application domain. Then, we develop a user interface module that facilitates the management and use of the library within the trace analysis approach.

Conversion

The first module in our tool is responsible for the conversion of existing specifications of the property patterns, and any newly written QRE formulae into observers. In this module, we reuse an existing tool from XEROX to perform the actual conversion and we build an interface around it that enhances its usability. The produced observers can be seen as templates for property specification. Each template includes three items:

1. The textual GOAL representation. The real description of the observer template. Each template is commented to explain the use of parameters and to enhance legibility.
2. The graphical GOAL representation (the state diagram). A static figure of the observer state machine.
3. A help item that defines the property, its intent, scopes and how to use it. In addition, the help includes some hints as to where the property is known to be commonly used.

Library Interface

The second module of our tool includes two main functions:

1. The library manager: This function is responsible for the management of the library. It allows adding new templates to the library and modifying existing ones. The manager allows two options to library management. The user can choose to use the library as a file system, where the files of the observers sharing the same class (e.g., response, existence, etc.) are

stored in a separate directory. Alternatively, the user can choose to use the library as a database while the manager provides the appropriate interface to read, modify, and instantiate the existing template records. In addition, the user can add newly written templates.

2. The observer editor: This function allows the user to fill in his desired values for the existing parameters in the template. Here also the user can choose between two options. First, the user can select to replace all the parameters manually. Here the user should know SDL, GOAL, and the model of the system under test. He should be able to write correct probes and make relevant instantiations of the parameters. In the second option, the tool can read the SDL model of the analyzed system and extract the meaningful characteristics (signal names, process names, variable names, state names, etc.). Then, the user can simply select, from a list, the appropriate instantiations to the parameters of the desired template.

Figure 5 shows the user interface of the Property Manager tool, which includes the two modules (conversion and library management).

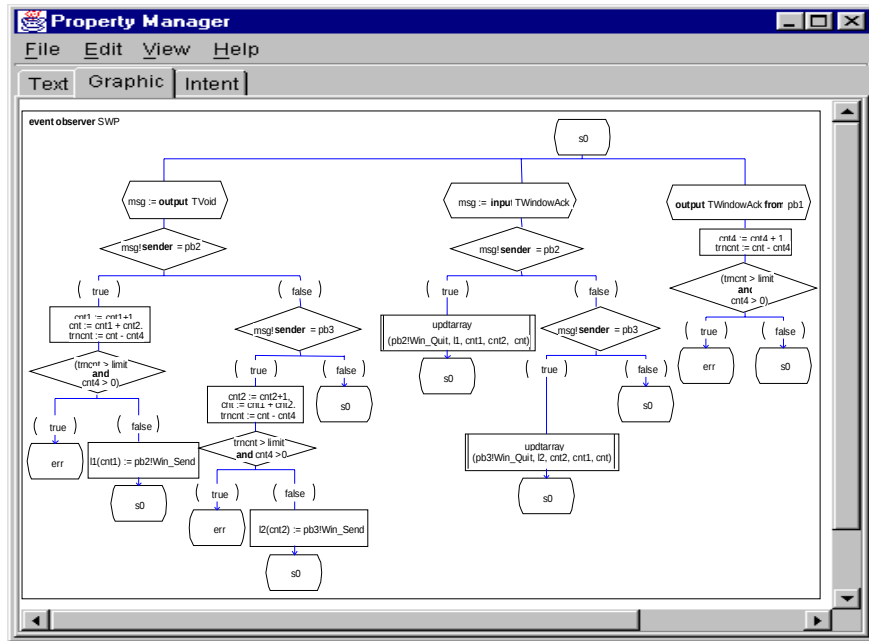


Figure 5: The user interface of the Property Manager tool.

RESULTS

We have discussed a trace analysis approach to test properties on distributed systems. Our approach is implemented within the ObjectGEODE environment, where a model of the SUT is built in SDL, the desired properties are specified in GOAL, and the verification is done using the simulator of OG. We reused an existing repository of property specification patterns to facilitate the task of the user to choose properties for verification.

Our implementation efforts included two front-end tools. The first, TRAYSIS, builds the SDL model from a given XML trace and makes the use of OG more transparent to the system tester or developer. The second front-end tool allows writing property specifications in GOAL in an easy and efficient way. The Property Manager converts existing QRE specifications of patterns into parameterized GOAL observer descriptions and gives the user control over the management and instantiation process.

Our experience indicates that a library of patterns for the property specification is quite useful for developers and testers of distributed systems. We have used our approach to test systems from different application domains including traffic control systems, telecommunication protocol based control systems, and mobile systems (GSM). The availability of the property library in GOAL proved essential to the success in detecting flaws and evaluating the performance of the tested systems. Furthermore, we have found that the verification mechanism of OG adds to the advantages of having the pre-defined library of observers. Actually, OG allows the user to verify several observers on the same SDL model in a sequential way.

Our plans include an investigation of the following problems:

1. Study the possible manipulations that could be applied to the observers of the library. Here, we will focus on the issue of combining patterns to form properties that are more complex.
2. The possibility to integrate the current library of GOAL observers in the OG environment so that it becomes accessible to the wide range of users of the toolkit.

References

1. B. Algayres, Y. Lejeune, E. Hugonnet, "GOAL: Observing SDL behaviors with GEODE", in *SDL'95 with MSC in CASE* (ed. R. Braek, A. Sarma), In *Proc. of the 7th SDL Forum*, Oslo, Norway, September, 1995, Elsevier Science Publishers B. V. (North Holland), pp. 359-372.
2. M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in Property Specifications for Finite-state Verification", In *Proc. of 21st International Conference on Software Engineering*, May 1999.
3. R. Groz, "Unrestricted Verification of Protocol Properties on a Simulation Using an Observer Approach", *Protocol Specification, Testing and Verification*, VI, Montréal, Canada, North-Holland, 1986, pp. 255-266.
4. Hallal, H., Boroday, S., Ulrich, A. and Petrenko, A. "An Automata-based Approach to Property Testing in Event Traces" In *Proc. of the IFIP TC6/WG6.1 XV International Conference on Testing of Communicating Systems (TestCom 2003)*. Sophia Antipolis, France, May 26-29, 2003.
5. Hallal, H., Petrenko, A., Ulrich, A. and Boroday, S. "Using SDL Tools to Test Properties of Distributed Systems". In *Proc. of the Formal Approaches to Testing of Software (FATES'01), Workshop of the International Conference on Concurrency Theory (CONCUR'01)*. pp. 125-140. Aalborg, Denmark, August 21-24, 2001.
6. Pattern Specification System, online repository: <http://www.cis.ksu.edu/santos/spec-patterns>.
7. Telelogic, "ObjectGEODE SDL Simulator Reference Manual", 1999.
8. A. Ulrich, H. Hallal, A. Petrenko, S. Boroday, "Verifying Trustworthiness Requirements in Distributed Systems with Formal Log-file Analysis". In *Proc. of the thirty-sixth Hawaii International Conference on System Sciences (HICSS-36)*, 2003.