Testing Prototypes Validity to Enhance Code Reuse[+]

Didier Buchs[†], Alioune Diagne[*] & Fabrice Kordon[*],

[†] Swiss Federal Institute of Technology
1015 Lausanne - Switzerland

[*] LIP6, Université P. & M. Curie,
4 place Jussieu ,75252 Paris - France

E-mail: Didier.Buchs@di.epfl.ch, Alioune.Diagne@lip6.fr, Fabrice.Kordon@lip6.fr

Abstract

The complexity of distributed systems is a problem when designers want to evaluate their safety and liveness. Often, they are built by integration of existing components with newly developed ones. Actually, it is valuable to handle the integration of external pieces of software in the specification and testing activities. However, it is difficult to validate them formally unless doing reverse-engineering (which is a heavy procedure).

This paper proposes to use structured formal specifications to generate a reasonable set of tests that evaluate behavior of software components in order to get an answer to both questions.

To do so, we use the description of components' external behavior and express it using the OF-Class formalism (an encapsulation of Colored Petri Nets). Test patterns are generated using an appropriate formalism, HML logic, and exploit various hypotheses corresponding to users' testing procedure

1. Introduction

The complexity of distributed systems is a problem when designers want to evaluate their safety and liveness. So their expected properties (basic ones like deadlock freeness or domain dependent ones) must be known and verified during the specification of the solution and traced to the implementation by means of tests.

Formal description techniques like Petri nets are a potential solution to this problem. Structuration rules and/or association with structured representations compensate for their lack of structuration **[3, 10, 2]**. They have also been proved to be valuable basis for testing **[1]**. In the remainder of the paper, we consider Petri nets associated with object-oriented concepts for specification and an object-oriented layout for implementation.

Often, distributed systems are built by integration of existing components with newly developed ones. Actually, it is valuable to handle the integration of external pieces of software in the specification and testing activities. However, it is difficult to validate them formally unless doing reverse-engineering (which is a heavy procedure).

Components of a distributed system can be formally described by the following characteristics :

1) what they guarantee to other components i.e. what are their local **properties**,
2) what they require from the other components i.e. what are their **expectations**,
3) the means (methods or operations) by which correct interactions with the environment are handled, i.e. the **internals**.

These internals do not need to be known for reused components at the specification phase. The behavior of such elements can be abstracted through their expectations and properties to enable verification of the system **[5]**. For instance, one kind of expectation we consider is the sequences of operation invocations that can be safely and reliably supported by the external pieces of software. Properties and expectations define a "formal signature" for the components. Code may then be derived from newly developed components and integrated with existing parts. Such an approach raises two questions :

1) are stated properties and expectations of all the components verified at the different stages of the life-cycle? This enforces traceability of a component from specification downto implementation,
2) what happens when stated properties and expectations are not verified? It is a robustness criteria for the component (see section 2.).

Formal verification and tuned tests generation provide an answer to question (1) above. Then, we may not care about question (2) if the system structurally avoids such unverified expectations (e.g. closed system built once for all). However, the reuse of an external part may not meet all its expectations. The behavior of a component in case of expectation violation needs therefore to be known. It comes along as a characterization that can be attached to the component as well as the test cases for its evaluation.

This paper proposes to use structured formal specifications to generate a reasonable set of tests that evaluate behavior of software components in order to get an answer to both questions.

First, we identifies key problems in our testing approach before a description of the context of our study. Then, we present our solution and apply it to a small example before a short conclusion.

2. Key Problems

In this paper, we consider the verification, validation and

+ This work has been jointly performed during an exchange between the Swiss Federal Institute of Technology in Lausanne and the Université P. & M. Curie.

tests for distributed systems where newly developed components are integrated with existing ones. A specification of a system is valid if all the components are able to guarantee their properties while handling the interactions with their environment as long as their expectations are met. This means that *"nothing bad would happen in the system"* and *"everything good and expected will eventually happen"* whenever the environment respects the expectations. It is the nominal behavior. Once these expectations are no longer met, the system or its components have two possible behaviors :

1) detect that the expectations have been violated and raise some exception. This means that it is **robust** against illegal behavior from the environment,
2) go out of the nominal behavior so that *"something bad happened"* or *"some expected other thing never happens"*.

For existing components, we require an external description basically consisting of :

- how to use the available primitives i.e. what are the sequences allowed (expectations),
- what are the guarantee when using these primitives according to the expectations (properties).

No information at all is provided about the way primitives are implemented, as explained in the next section. The supplied information is valuable to check if the component is properly manipulated by other components of the system **[12]**.

This is necessary to take into consideration importation of software components that are already implemented. The external description (what it does and how to use it) of a component is basically an automata that can be expressed using Petri nets.

Figure 1 describes the behavior of a simple filesystem component which can run either in read-only or write-only mode. This component offers six operations (open, create, read, write and close) for which a possible correct execution is described by the following :

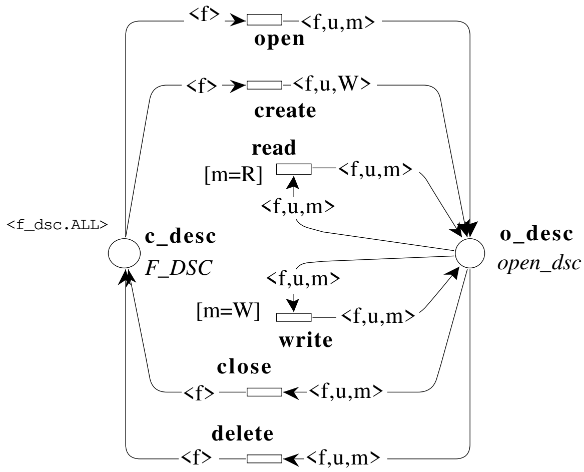
(create && (write)* || open && ((read)*||(write)*)) && (close||delete) [1]

The number of initial tokens in place **c_desc** corresponds to the available file descriptors in the system (i.e. the maximum number of files to be opened).

A specification like the one of Figure 1 allows a system designer to state the way a component runs. Such prerequisites must be respected by any element that use the specification component through its implementation. We call those *expectations* of the system. We note *Exp* a set of such expectations. It means that a client must satisfy the following relation:

$$client \models Exp$$

Based on such a description, we can state *properties* of the system. In the example of Figure 1, properties are "*no more than 64 files can be simultaneously opened*" or "*it is impossible to read in a write-only opened file and vice-versa*". We note *Prop* the set of properties attached to a component.




```
Class                           Var
f_dsc is 1..64;                 m in mode;
usr is 1..100;                  f in f_dsc;
mode is [R,W];                  u in usr;
Domain
open_dsc is <f_dsc,usr,mode>;
```

Figure 1 : *External description of the simple filesystem component by means of a Petri net.*

In the example of Figure 1, what happens if an application opens more than 64 files ? or if it writes in a file that is not yet opened ? In those cases, the component should signal an improper use exactly when the problem occurs. It means that the component also remain at least safe and perhaps reliable when :

$$client \not\models Exp$$

If $Prop \cup Exp$ can be formally verified using a formalism like Petri nets, automatic generation of tests are useful to check if the corresponding implementation also has similar characteristics. Evaluation of $client \not\models Exp$ enforces to specifically verify unexpected behaviors. Here, automatic generation of tests is also valuable to know how the component implementation behaves when its requirements are violated.

It is quite easy, based on a behavior specification like the one of Figure 1, to generate many primitive invocation sequences that either respect or violate expectations regarding a software component. However, it is of interest to obtain a reasonably sized benchmark which ensures a good coverage of potential problems. Hereafter are some valid and invalid sequences for illustration.

$$client \models \langle open\rangle\langle read\rangle\langle close\rangle$$
$$client \not\models \langle read\rangle\langle open\rangle\langle close\rangle$$

Under the hypothesis that the environment sticks to the expectations of a component, this one (noted *server* hereafter) should support the expectations and guarantee its properties :

$$server \vdash Exp \text{ and } server \models (Exp \Rightarrow Prop)$$

The first equation means that the specification of the component is compliant with its expectations and the second one means that under these expectations, it guarantees

(1) && stands for sequence and || for alternative.

its properties.

3. Context of the Study

The work presented in this paper relies on the MARS methodology **[7]**. MARS is a multi-formalism approach designed to offer a suitable representation for each description stage (from conception to verification and prototyping) of a distributed system.

3.1. Overview of the MARS Methodology

The MARS methodology proposes a frame to specify, evaluate and implement distributed applications : it defines a track that leads a system designer from the *conceptual description* (specification) to the *operational description* (implementation) from which programs are automatically generated.

The conceptual level is dedicated to the explicit definition and verification of safety and liveness properties. The operational level is more likely dedicated to implicit properties addressing the optimization and automatic production of the generated prototype.

Figure 2 illustrates steps of the MARS methodology. It relies on three formalisms :

- Well-formed Petri nets **[4]** fit all the formal needs. It is a potential target used to verify and compute properties of the system model;
- OF-Class (Object Formalism Class) **[5]** provides a conceptual description of the system. It contains information about the association of components, the way they behave and how they should be used. It may be transformed into a formal description;
- H-COSTAM (Hierarchical COmmunicating STAte Machine Model) **[9]** allows the designer to deal with operational aspects of his system. Such a description may be derived from the conceptual description by addition of information. It can also be transformed into a formal description and enables code generation.

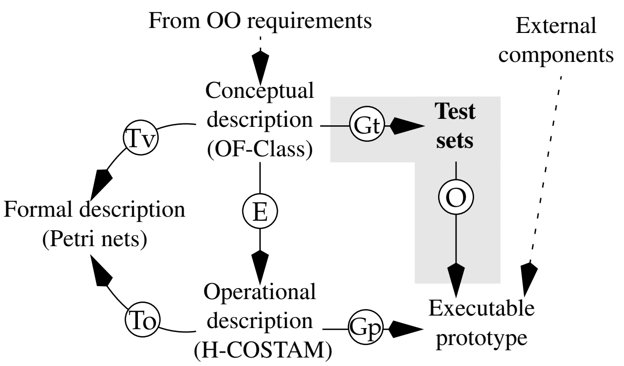


Figure 2 : *Overview of the MARS methodology.*

Three operations between these representations are characterized:

- Two transformations from respectively OF-Class into Petri nets (Tv in Figure 2) and H-COSTAM into Petri nets (To in Figure 2) enable the link with the formal representation. These transformations are different while they do preserve discrete properties. The result is a Petri net that express either functional relations (to extract conceptual properties of the system) or an operational description (to extract implementation characteristics). Transformation Tv aims to provide information about the safety and liveness of the system while transformation To focuses on the computation of characteristics for optimization purpose;
- Elicitation of the system is the transformation of a conceptual description into an operational one (E in Figure 2). This step should not be automatic like the two other ones. It should be performed once when the system attains a satisfactory level of confidence whith respect to expected properties. It can be considered as a list of questions that gradually clarify all the points of the implementation.
- Code generation is performed from the operational description (Gp in Figure 2). It may compute and use operational properties to optimize code generation. In the context of distributed systems, this operation must produce both a compilable program and a location proposal. In our methodology, a prototype is made of parts generated from the operational description plus *external components* that correspond to already existing pieces of software.

In MARS, any component description must identifiy its internal behavior (how it evolves) and its external description (how it must be used). For components modeled and evaluated at the conceptual level, the two descriptions are required. Only the external description has to be provided for external components.

3.2. External Description of a Software Component

External description of a component is described at the conceptual level while its verification is performed at this stage of the methodology.

It is expressed using a structured language that declares *operations*, organizes them into *services* and provides a *usage pattern* for each service.

An operation is a procedure (with its input and output parameters) that appears to be atomic from a user's point of view. A service is a way to classify operations and logically group them. Services may share operations. The usage manual of a service defines how operations in the service should be operated.

The small file manager component (presented in Section 2.) contains five operations that are distributed over two services :

- **read-only** groups operations *open*, *read*, *close*, *delete* and has *open*&&(*read*)*&&(*close*||*delete*) for usage manual
- **write-only** groups operations *create*, *write*, *close*, *delete* and has *create*&&(*write*)*&&(*close*||*delete*) for usage manual.

The Petri net of Figure 1 is derived from such a specification. It is useful to define a behavioral signature of the software component. It can be used as well to generate testing sequences.

3.3. Introducing Testing Techniques in MARS

We introduce testing techniques in order to evaluate both validity and robustness of software components issued from formal specifications. This concerns :

- External components to check if they correspond to their external description;
- Newly developed components to verify that implementation choices introduced during the elicitation procedure (E in Figure 2) have not altered their robustness.

The extension of the MARS method we describe in this paper corresponds to the gray part Figure 2. Sets of tests are generated (Gt in Figure 2) from the formal description of selected components and applied (O in Figure 2) on the corresponding software implementation. As we will explain in Section 4., we mainly exploit the information contained in the external description of components.

3.4. Theoretical Grounds on Testing

In the following sections, we will shortly describe the theory of the test selection techniques for object oriented software. Interested readers can find more information on those techniques in **[1]** and **[11]**.

Functional testing is an approach to find errors in a program by verifying its functionalities, without analyzing the details of its code, but by using the specification of the system only. The goal is to find cases where a program does not satisfy its specification. It can be summarized as the equation:

$$(P \not\models_O T \Leftrightarrow P \not\models SP)$$

i.e. that the test set T applied on a program P will reveal that the program P does not implement correctly the specification SP. (This observation is performed through the help of an oracle, formally denoted by $\models_O$). Of course, the goal in selecting T is to uncover the cases where the program does not satisfy the tests, and thus reveal errors with respect to the specification.

Test selection is based on the knowledge of the properties of the specification language, which must be theoretically well founded. Usually, specification languages have a notion of formula representing properties that all desired implementations satisfy. Tests can be expressed using a common language, however it is not necessary to have the same language to express both specification properties and tests. The most interesting solution is to have a specification language well adapted to the expression of properties from an user point of view, and another language to describe test cases that can be easily applied to an oracle, as long as there is a full agreement between these two languages.

3.4.1. The Theory of Testing

The theory of testing is elaborated on specifications *Spec*, programs *Prog* and tests *Test*, and on adequate compatible satisfaction relationships between programs and specifications, $\models$, and between programs and tests, $\models_O$. This is defined by the following equation:

$$(P \models_O T_{SP} \Leftrightarrow P \models SP).$$

The equivalence relationship $\Leftrightarrow$ is satisfied when the test set T_{SP} is pertinent, i.e. valid (any incorrect program is discarded) and unbiased (it rejects no correct program). Of course, the exhaustive test set is pertinent.

However, a pertinent test set T_{SP} can only be used to test a program P if T_{SP} has a "reasonable" finite size. Limiting the size of a test sets is performed by sampling. In our theory, sampling is performed by applying hypotheses on the program P, making assumptions that the program react in the same way for some inputs.

Assuming that hypotheses H have been made on the program P, the following formula has to be verified for any selected test sets $T_{SP,H}$:

$$(P \text{ satisfies } H) \Rightarrow (P \models_O T_{SP} \Leftrightarrow P \models SP).$$

3.4.2. Practicable Test Context and Hypotheses

Thus, the test selection problem is reduced to applying hypotheses to a program until a test set of reasonable size can be selected. For that purpose, we build a test context, called practicable because it can be effectively applied to the oracle: Given a specification SP, a practicable test context $(H, T)_O$ is defined by a set of hypotheses H on a program under test P, a test set T of "reasonable" finite size and an oracle O defined for each element of T.

The selection of a pertinent test set T of "reasonable" size is made by successive refinements of a possibly not practicable initial test context $(H_0, T_0)_O$ which has a pertinent test set T_0 (but not of "reasonable" size), until obtaining a practicable test context $(H, T)_O$:

$$(H_0, T_0)_O \leq \ldots (H_i, T_i)_O \leq (H_j, T_j)_O \ldots \leq (H, T)_O.$$

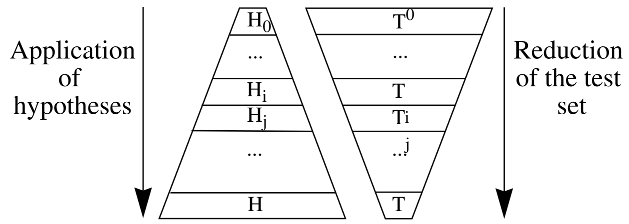


Figure 3 : *Iterative refinement of the test context*

At each step, the preorder refinement context $(H_i, T_i)_O \leq (H_j, T_j)_O$ is such that:

- The hypotheses H_j are stronger than the hypotheses H_i, $H_j \Rightarrow H_i$.
- The test set T_j is included in the test set T_i
- If P satisfies H_j then $(H_j, T_j)_O$ detects no more errors than $(H_i, T_i)_O$
- If P satisfies H_j then $(H_j, T_j)_O$ detects as many errors than $(H_i, T_i)_O$

Therefore, if T_i is pertinent then T_j is pertinent.

Since the exhaustive test set is pertinent, we can use it for the initial context T_0.

4. Proposal and Example

Our proposal integrates testing techniques in the MARS methodology according to the objectives identified in Section 3.3. In this section, we describe the mechanisms of our technique and then apply them to the small example presented Section 3.1.

4.1. Testing Behaviors and Properties in OF-Class

4.1.1. Expressing Tests with HML Formulae

For the specification language, the tests can be expressed with the HML Logic introduced by Hennessy-Milner **[8]**. HML formulae built using the operators Next (<_>), And (^), Not (¬), T (always true constant), and the events Event (SP) of the specification SP ∈ Spec, are noted HML_{SP}. An advantage of this approach is to have an observational description of the valid implementation through the tests. A test is a formula which is valid, invalid or raise an exception (failure). It must be experimented in the program (i.e. a correct implementation behaves similarly to the specifications and detects usage problems).

An elementary test for a program under test P ∈ Prog and a specification SP ∈ Spec can be defined as a couple <Formula, Result> where:

- Formula ∈ HML_{SP} : (ground) temporal logic formula.
- Result ∈ {true, false, failure}: value showing whether the expected result of the evaluation of Formula (from a given initial state) is true, false or generates a component failure.

A test <Formula, Result> is successful if Result reflects the validity of Formula in the labeled transition system modeling the expected behavior of P. In all other cases, a test <Formula, Result> is a fail. It is important to note that the test definition will allow the test procedure to verify that a non-acceptable scenario cannot be produced by the program (for instance, to read in a file that is not opened). For tests expressed using HML_{SP}, we can define the exhaustive test set $Exhaust_{SP, Ho} \subseteq$ Test such that:

$Exhaust_{SP, Ho}$ = {<Formula, Result> ∈ HML_{SP} × {true, false, failure} | (SP $\models$ Formula and Result = true) or (SP $\not\models$ Formula and no exception are raised and Result = false) or (SP $\not\models$ Formula and an exception is raised and Result = failure)}.

4.1.2. Test Selection

From a practical point of view, the reduction process is implemented as a selection process: to each reduction hypothesis on the program corresponds a constraint on the test set. Indeed, the exhaustive test set can be defined as a couple < f, r > where f is a HML_{SP} formula with variables universally quantified. The aim of the test selection becomes the reduction of the level of abstraction of f by constraining the instantiation of its variables. The various techniques that can be applied could not be described here, they can be found in **[1]** and **[11]**.

4.1.3. Operational Test Selection

The concrete implementation of the test selection can be performed by means of a logic programming engine if a complete axiomatization of the specification language can be given in Horn clauses. In the test selection tool, a modified resolution mechanism based on random choice of the resolvant clause is used.

4.1.4. The Oracle

Once a test set has been selected, its elements are executed on the program under test. Then the results collected from this execution are analyzed. It is the role of the oracle to perform the analysis, i.e. to decide the success or the failure of the test set.

The oracle O is a partial decision predicate of a formula in a program P. The problem is that the oracle is not always able to compare all the necessary elements to determine the success or the failure of a test; these elements are said to be non-observable. This problem is solved using the oracle hypotheses H_O which are part of the possible hypotheses and collect all power limiting constraints imposed by its implementation.

The failure result (see Section 4.1.1.) has been elaborated to detect a component failure during execution time. At the oracle level, it corresponds to no response : the component is out and can event not answer to signal a problem. At the specification level, we consider its a deadlock state (i.e. without successor).

4.1.5. The Incremental Test Selection Process

We use the structure of the dependencies among classes to determine a test selection process in which the whole specification is tested class by class. In case of mutual dependencies, a linearization is proposed in order to determine sequence of test application. Previously tested classes require less care, so stronger hypothesis can be considered while the class of interest need weaker hypothesis.

4.2. Example

Let us come back to the filesystem example sketched in Section 2. which models a basic file system six operations (create, open, read, write close and delete). Hereafter, we provide a partial description expressed using our specification model : OF-Class.

Expectations are stated in the interface by means of offered services. They allow to define coherent viewpoints on the component involving just the appropriate part of the operations. For instances the **readers** offered service define the behavior allowed for components accessing the file in

read-only mode. This enables the definition of clients classes and thus the use of a component in discrete contexts without facing its whole complexity.

In this simple file system, we have two kinds of proof obligations :

1) basic properties like reliability for servers. The component must ensure that each request to one of its operations is eventually followed by a result
2) the second one is domain-dependent. The numbers of file-descriptors should always be less than 64.

The properties depicted in (1) are implicit and always verified. They ensure the correctness of the specification. The reader can refer to **[6]** for more information about such implicit basic properties and the specification model in general. The properties in (2) are explicitly stated in the specification and verified.

```
filesystem ISA OFCLASS[2]
   DECLARATION {
      type f_dsc is 1..64;
   }
   MACRO-LEVEL
      EXPORTS {
         service readers
            operations {
               f_dsc : open (char name);
               char  : read (f_dsc fd);
               void  : close (f_dsc fd);
            }
            manual {
            open && (read)* && (close || delete)}
         #definition of service writers (omitted)
      }
   MICRO-LEVEL
      RESOURCES {
         #local resources declaration (omitted);
      }
      INSTANCES {
         inst1 ;
      }
      OPERATIONS {
          # here are defined a part of the internals
          # i.e. the algorithms for the operations
         void : close (f_dsc fd)
         {
            # actions to close the file (omitted)
         }
      }
   ENSURES { # invariant to be respected
      Alw(card(files)+card(opened_files)=64)
   }
ENDOFCLASS
```

4.2.1. Structural Constraints

As said before, test selection consists in applying successive constraints in order to implement strategies intended to implement hypothesis on the programs under test. For our example we are going to illustrate the possible tests that can be selected for the filesystem class. We will first use general hypothesis not related to the example and then instanciate them to it.

Hypothesis: If a test $\langle f, r\rangle$ is successful for all instances of f having a number of events equal to a bound k, then it is successful for all possible instances of f. The number of events is computed recursively with the function *nb-events* as follows:

Thus the constraint $C \in \textit{CONSTRAINT}_{SP,\,X}$ is the predicate: $\textit{nb-events}\ (f) = k$

Strategy: The strategy used to solve the former constraint C generates all the $HML_{SP,\,XS}$ formulae with a number of events equal to k, without redundancy. With this strategy, only skeletons are generated and nothing is imposed by the specification. Later, free variables are supposed to be instantiated to events based on environment's operations. For instance, the constraint *nb-events (f)* = 2 produces the four following tests:

```
T0: <(not <V0> T) and (not <V1> T),result>
T1: <(not <V0> T) and (<V1> T), result>
T2: <(<V0> T) and (<V1> T), result>
T3: <<V0> <V1> T, result>
```

where the variables `V0` and `V1` are of type event.

4.2.2. Event Based Constraints

Another way to reduce the size of the test sets is to constrain the number of occurrences of a given operation in each test.

Hypothesis: If a test $\langle f, r\rangle$ is successful for all instances of f having a number of occurrences of a given operation m equal to a bound k, then it is successful for all possible instances of f.

The number of occurrences of a given operation m is recursively computed with the function

$$\textit{nb-occurrences}: HML_{SP,\,XS} \times \textit{Operations} \rightarrow \mathbb{N},$$

which is defined like the function *nb-events*.

The constraint $C \in \text{Constraint}_{SP,\,X}$ is the predicate nb-occurrences (f, m) = k.

Strategy: The strategy used to solve the former constraint C generates all the $HML_{SP,\,XS}$ formulae with a number of events based on the operation m equal to k. For example, let us consider a filesystem *fl* on which we assume:

nb-occurrences (*f,* open) = 1 (one occurrence of open) which leads to this kind of tests :

```
T:<<f1←fl.open(<str>)><c1←fl.read(<f2>)>
      <c2←fl.read(<f3>)>T, result>
```

where variables str is a string, variables c_1 and c_2 characters and variables f_1, f_2 and f_3 file descriptors.

Free variables should be instanciated in order to produce the finite set of applicable tests. *Uniformity hypothesis* can be used for that goal, producing for instance :

```
T:<<2←fl.open("xxx")><c1←fl.read(<3>)>
      <c2←fl.read(<4>)>T, result>
```

This test is obviously unsatisfied (it means that *result* should be false). Selecting tests in this way lead to bad coverage of the different specification cases.

4.2.3. Subdomain Decomposition

A better way to proceed is to first decompose the different tests by performing a so called *sub-domain decomposition* leading to two cases:

- satisfiable test

(2) Lines beginning with a # are comments

`T: <<f1←fl.open(<str>)> <c1←fl.read(<f1>)> <c2←fl.read(<f1>)>T, true>`

- exception case due to the fact that we are testing expectations

`T: <<f1←fl.open(<str>)> <c1←fl.read(<f2>)> <c2←fl.read(<f2>)>T, false>`

with $f_1 \neq f_2$

The next step is to apply the previously mentioned uniformity hypothesis for correctly instantiating variables.

4.2.4. Other Hypothesis

More hypotheses can be imagined from the above presented one. It must be noted that they are designed to reflect the usual test practices. Interested reader can consult **[11]** which explains a set of interesting hypotheses as well as how to implement them by a suitable logic programming engine.

4.2.5. Testing Expectations and Properties

The idea of differentiating expectation and properties is taken into account by having discrete interpretations of the unsatisfiable formulae (satisfiable formulae are interpreted in the same way for both kinds of specifications):

- *Expectations* must produce exceptions when applied (*result* = false)
- *Properties* must produce unsatisfiable behavior (*result* = false).

For instance, another kind of test can be derived from properties :

`T:<<f1←fl.open(<str1>)> f2←fl.open(<str2>)> ..f65←fl.open(<str65>)>T, false>`

Where the following constraint is verified : $\forall$ i, j $\in$ [1..65] with i≠j, $f_i \neq f_j$. This test checks that the filesystem component rejects the opening of more than 64 files at a time.

In both cases (expectations or properties), it is expected to the component to signal a bad usage through the oracle. If it does not (*failure* detected by the oracle), we consider it has crashed and thus, is not reliable.

5. Conclusion

In this paper, we have proposed to enrich MARS, a Petri net based design and prototyping methodology for distributed systems. The enrichment introduces a way to evaluate the correspondence between specifications and generated prototypes. One of our goals is to check their robustness. Robust software components should be able to detect inappropriate use by other components and signal it.

To do so, we use the external description of components that describe their external behavior and express it using the OF-Class formalism. In this formalism, users may express a usage protocol by means of atomic operations (called expectations). They may also express properties on the system's components.

Test patterns are generated using an appropriate formalism (HML logic) and exploit various hypotheses corresponding to users' testing procedure. We generate tests sets for both nominal behavior of components and violation of their expectations.

6. References

[1] S. Barbey, D. Buchs, and C. Péraire, "A Theory of Specification-based Testing for Object-Oriented Software", In Proceedings of EDCC2, LNCS 1150, pages 303-320, Taormina, Italy, Oct. 1996.

[2] O. Biberstein, D. Buchs, and N. Guelfi, "Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism", In G. Agha and F. De Cindio, editors, Advances in Petri Nets on Object-Orientation, volume to appear of Lecture Notes in Computer Science. Springer-Verlag, 1997.

[3] P. Buchholz, "Hierarchical High Level Petri Nets for Complex System Analysis", Proceedings of the 15th International Conference on Application and Theory of Petri Nets (LNCS, spinger Verlag), Zaragoza, Spain, June 1994, LNCS vol. 815 PP. 119-138.

[4] G. Chiola, C. Dutheillet, G. Franceschini & S. Haddad, "On Well-Formed Colored Nets and their Symbolic Reachability Graph", High Level Petri Nets. Theory and Application. Edited by K. Jensen G.Rozenberg, Springer Verlag 1991

[5] A. Diagne & P. Estraillier, "Formal Specification and Design of Distributed Systems", International Workshop FMOODS'96, Paris, Mars 1996

[6] A. Diagne & P. Estraillier, "A Component-based Framework for for the Specification, Verification and Validation of Open Distributed Systems", Technical Report of the LIP6 Laboratory #1997/037.

[7] A.Diagne & F.Kordon, "From Formal Specification to Optimized Implementation of Distributed Systems : A Multi-Formalism Approach", Technical Report of the LIP6 Laboratory #97/039, December 1997.

[8] M. Hennessy & R. Milner, "Algebraic laws for non-determinism and concurrency", Journal of the ACM, 32(1):137-161, January 1985.

[9] F. Kordon & W. El Kaim, "H-COSTAM : a Hierarchical Communicating State-machine Model for Generic Prototyping", Proceedings of the 6th International Workshop on Rapid System Prototyping, N. Kanopoulos Ed, IEEE comp. Soc. Press 95CS8078, pp 131-138, Triangle Park Institute, June 1995

[10] C.A. Lakos, "From Colored Petri Nets to Object Petri Nets", Proceedings of the 16th International Conference on Application and Theory of Petri Nets (LNCS, spinger Verlag), Torino, Italy, June 1995, LNCS vol. 935, PP 278-297

[11] C. Péraire, S. Barbey, and D. Buchs, "Test Selection for Object-Oriented Software Based on Formal Specification", In Proceedings of PROCOMET98, N.Y, USA, 8-12 June. 1998.

[12] J. Sa, J. A. Keane & B. C. Warboys, "Software Process in a Concurrent, Formally-based Framework", In Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Beijing, China, October 1996, pages 1580-1585