Formal Aspects of Computing (1999) 3: 1–000

Testing Algebraic Data Types and Processes: A Unifying Theory

Marie-Claude Gaudel[1] and Perry R. James[2]

[1]Lab. de Recherche en Informatique,Université de Paris-Sud and CNRS, Orsay, France ;
[2]Dep. de Ciência da Computação, IME-Universidade de São Paulo, São Paulo, Brazil

Keywords: Black box testing, Formal testing, Full LOTOS

Abstract. There is now a lot of interest in program testing based on formal specifications. However, most the works in this area focus on one formalized aspect of the software under test. For instance, some previous works of the first author consider abstract data type specifications. Other works are based on behavioral descriptions, such as finite state machines or finite labelled transition systems. This paper begins by briefly recalling the principles of test data selection from algebraic data types specifications. Then, it transposes them to basic and full LOTOS. Finally it exploits this uniform framework and suggests a new integrated approach to test derivation from full LOTOS specifications, where both behavioral properties and data types properties are taken into account when dealing with processes.

1. Introduction

There is now a lot of interest in program testing based on formal specifications. However, most the works in this area focus on one formalized aspect of the software under test. For instance, some previous works consider abstract data type specifications [BGM91, Gau95]. Other works are based on behavioral descriptions, such as finite state machines [Cho78, FKG91, LY94] or finite labelled transition systems [dNH84, Hen88, Bri89, PF90, Tre92].

This paper suggests a unification of the concepts and terminology in this area on the basis of [BGM92]. Briefly, a notion of *exhaustive test set* is derived

Correspondence and offprint requests to: mcg@lri.lri.fr, perry.james@acm.org

from the semantics of the formal notation and from the definition of a correct implementation. Then a finite test set is selected via some *selection hypotheses*, which are chosen depending on

- some knowledge of the program,
- some coverage criteria of the specification,
- and ultimately cost considerations.

This paper begins in Section 2 by briefly recalling the application of this framework to algebraic data types [BGM91, Gau95]. Then, Section 3 generalizes this framework to the test-derivation method developed for basic and full LOTOS by Brinksma and his colleagues [Bri89, Eer94, Tre92]. However, in this method, the data types properties are not considered when selecting test data from the behavior part of the specification. Section 4 exploits the common framework stated for the two kinds of specification and suggests a new integrated method of test derivation from full LOTOS specifications. This approach is transposable to other specification languages where data types and processes coexist, such as SDL [CCI88] or Promela [Hol91].

2. Test Selection for Algebraic Specifications

Testing code against an algebraic specification consists of showing that the final system satisfies the axioms in the specification.

To create tests for a given axiom, the variables of the axiom are instantiated with values. To run such a test, the resulting expressions are evaluated. If the results satisfy the axiom then the test is *passed*, otherwise it is *failed*.

In [Gau95], Gaudel discusses the formal basis of testing based on an algebraic specification. Such a specification has two parts: a *signature* Σ=(S, F), where S is a finite set of *sorts* and F is a finite set of *operation names* over the sorts in S, and Ax, a finite set of *axioms*. These axioms are equations or positive conditional equations built from Σ-terms.

See for instance the specification of the *Message* data type in Figure 1. There a message is specified as a sequence of octets built by the $\_ . \_$ operation; there is a Size operation, and it is possible to Pack two messages together.

2.1. Testing a Data Type Implementation against its Specification

If SP is a specification and P is an implementation under test against SP, the implementation P has to provide some way to execute the operations of SP.

Let t be a ground Σ-term and t_p its computation by P. Given a Σ-equation $\tau = \tau'$, a *test* for this equation is any ground instance $t = t'$ of it. A *test experiment* of P against $t = t'$ consists of evaluating t_p and t'_p and comparing the resulting values. This comparison is the job of an *oracle*, that is, a process that can decide if the computed results are equivalent. Note that this decision is easy for simple data types, but far from being obvious in general. This point is discussed in [Gau95].

In the *Message* example, a possible test, inspired by the fourth equation is

$$\mathrm{Size}(A.B.C.\varepsilon) = \mathrm{Succ}(\mathrm{Size}(B.C.\varepsilon))$$

The corresponding test experiment consists in two computations and a comparison, namely

- computing the representation in P of A.B.C.ε, calling the Size_P function in P on it, and storing the result;
- similarly, calling Size_P on B.C.ε, and adding 1 to the result ;
- comparing the two results.

A program performing this experiment is called a *tester* and can be easily derived from the equation.

An *exhaustive test set* for a specification SP is the set $exhaustive(SP)$ of all the possible well-typed ground instantiations of all of the Σ-axioms. This notion

```
specification Example1 : exit

library Boolean, OctetString, NaturalNumber endlib

type Message is
  Octet, NaturalNumber, Boolean
  sorts
    Message
  opns
    ε :  -> Message
    _ . _ :  Octet, Message-> Message
    Pack :  Message, Message -> Message
    Size :  Message -> Nat
  eqns
    forall m1, m2: Message, o1: Octet
    ofsort Message
      Pack(ε, m1) = m1;
      Pack(b.m1, m2) = b.Pack(m1, m2);
    ofsort Nat
      Size(ε) = 0;
      Size(o1.m1) = Succ(Size(m1));
  endtype
behavior
   ...
where
   process Compact[inGate, outGate, control](Max: Nat) : exit :=
       control ? newMax:Nat[ newMax  > 0];
        ( Compact[inGate, outGate, control](newMax)
    [] control ? newMax:Nat[newMax = 0]; exit
    [] inGate ? x:Message;
         ( inGate ? y:Message[Size(x) + Size(y)>Max];
          outGate ! x ! y; Compact[inGate, outGate, control](Max)
       [] inGate ? y:Message[Size(x) + Size(y) <= Max];
          outGate ! Pack(x, y);
          Compact[inGate,outGate, control](Max)
         )
  endproc
endspec
```

Fig. 1. An Example Specification

is directly derived from the definition of the satisfaction of a set of axioms as stated for instance in [EM85].

The term "exhaustive" is inspired from Goodenough and Gerhart's pioneering paper [GG75].

However, an implementation's passing all of the tests in the set *exhaustive*(SP) does not necessarily mean that it satisfies SP, since *exhaustive*(SP) is exhaustive with regard to the values expressible in SP, but not necessarily to those computable by P. Therefore, P satisfies SP only if all the values computed by P can be reached by T_Σ, the ground terms that can be produced by the operations of Σ. By construction of *exhaustive*(SP), the success of all its tests ensures that an implementation P satisfies SP only if P defines a finitely generated Σ-algebra. This assumption on P is called the *minimal hypothesis* H_{min}. Practically, it corresponds to:

- The realizations of the operations of Σ by P are deterministic;

- P has been developed following reasonable programming techniques, ensuring encapsulation; in P any computed value of a data type must always be a result of a specified operation of this data type.

An implementation satisfying this *minimal hypothesis* is said to be Σ-testable.

2.2. Selection Hypotheses

Generally *exhaustive*(SP) is too large to be useful. It is possible to assume stronger hypotheses on the behavior of the implementation and reduce the number of tests necessary to show that it satisfies the specification. These kinds of hypotheses are known as *selection hypotheses*, and the two most commonly used are *uniformity* and *regularity* hypotheses.

A *uniformity hypothesis* is an assumption that the input space can be divided into sub-domains such that if a test set containing a single element from each subdomain is passed, then the test set *exhaustive*(SP) is also passed. An example for integers is "if the function works correctly for a negative value, for a positive value and for zero, then it will work correctly for all integers." This notion is similar to the reliability property of a partition criteria mentioned in [GG75].

A *regularity hypothesis* uses a size function from ground Σ-terms to $\mathcal{N}$ and has the form "if a set of tests made up of all the ground terms of size less than or equal to a given limit is passed, then *exhaustive*(SP) also is." An example of such a hypothesis for a list would be "if the add operation works correctly for all lists of length less than or equal to 4, then it will work correctly for all lists."

Various hypotheses can be formulated simultaneously about an implementation.

A test strategy is defined by the choice of selection hypotheses. Exposing the hypotheses makes clear the assumptions made on the implementation. A *test context* is a pair (H, T) of a set of hypotheses and one of tests. A test context is considered *valid* if H implies that if T is passed then *exhaustive*(SP) is as well. A context is considered *unbiased* if H implies that if *exhaustive*(SP) is passed then T is as well. Assuming H, validity guarantees that all incorrect implementations are rejected, and being unbiased guarantees that no correct implementation is rejected.

By construction, $(H_{min}, \textit{exhaustive}(SP))$ is both valid and unbiased. Another extreme example that is both valid and unbiased is $(H_{min} \wedge P \textit{ satisfies exhaustive}(SP), \emptyset)$, which indicates that if the implementation is assumed to be correct then no tests are needed. Interesting test contexts are those that are valid and unbiased, that is, those that are passed by all and only correct implementations. Weak hypotheses correspond to large test sets, and strong hypotheses correspond to small test sets. The goal is to make reasonable hypotheses stronger enough to reduce the set of tests to a tractable size. The selection of such hypotheses can be based on the formal specification, on some knowledge of the program, or in some of the characteristics of the system environment, see for instance [DGM93].

Remark 1: As it is defined here, for sake of brevity, $exhaustive(SP)$ contains useless tests, namely the tests corresponding to instantiations of conditional equations where the premise is false. Thus $exhaustive(SP)$ can be simplified. For an improved definition see [BGM91].

Remark 2: In [Bri89, Tre92, Eer94], a different terminology is used for notions similar to validity and unbias. A test is said to be *sound* if it only rejects incorrect implementations, which is the same as unbiased. A test set is said to be *exhaustive* if it rejects all incorrect implementations, and possibly more. It is said to be *complete* if all and only incorrect implementations are rejected, which is the same as valid and unbiased.

3. LOTOS Processes and Their Testers

3.1. A brief overview of LOTOS

LOTOS is a well-known specification language with a standardized definition [ISO89].

A LOTOS specification is made of two parts, a data type part and a behavior part. The data type part is an algebraic specification of some abstract data type (cf. Figure 1). The behavior part describes the behavior of the system as a behavior expression which is generally a composition of actions and processes; processes are also defined by a behavior expression (cf. Figure 1).

The basic component of a process is an *action*. An action may be observable or internal.

An *internal action* is noted i, and is ignored by the environment of the process.

An *observable action* corresponds to some communication via some *gate*. If g is a gate identifier, and exp a well formed expression then g!exp is an observable action which sends the value of exp to another process, which must be ready to receive such a value, on gate g. If s is a sort, and cond is a boolean expression, g?x:s[cond] is an observable action which receives on gate g a value v of sort s satisfying cond and assigns it to the variable x.

There are three main ways of composing actions and processes.

- Action prefixing : a; P, where a is an action and P a process or a behavior expression.
- Choice between possibly guarded behavior expressions : (P []Q) or (cond$_1$ $\rightarrow P$ []cond$_2$ $\rightarrow Q$)
- Parallel composition with synchronization on some specified gates : P |[g$_1$, ..., g$_n$]| Q, on all the gates: P||Q, or on no gate: P||| Q.

There is an identification of the state of a process with its behavior expression, i.e. a process P = a; P' is in state $a; P'$; after executing the a action, it will reach the state P'. The a action is said to be *executable* from the state $a; P'$.

Due to the choice construction, there may be several executable actions from a given state. In some case, it is convenient to use the notion of the *set of the next observable executable actions* of a state. In this set, internal actions are ignored and each of them is replaced by its following next observable actions.

Also due to the choice construction, there may be several states reachable after an action (or a sequence of actions). For instance, in
$P =$ (a; Q) [] (i; a; R)
two states are reachable after a, the one corresponding to Q and the one corresponding to R.

As an example of the use of the choice construct consider the last part of Figure 1. The *Compact* process begins by a choice. It may

- receive a positive value on the control gate, assign it to newMax and start again;
- receive a 0 on the control gate, and then terminate;
- receive a message on the inGate gate; then either it receives a second message such as the total size of the two messages is greater than Max and it resends them via the outGate gate; or the total size of the two messages is less than Max and it packs them and sends the resulting message via the outGate gate.

In a parallel composition with synchronization on gate g , for instance P |[g]| Q, it is possible to execute either an executable action of P not concerning gate g, or an executable action of Q not concerning gate g, or a common communication action via gate g with the *important constraint* that a such an action is executable only if it is executable from both P and Q and satisfies some synchronization conditions. More precisely, it is possible to synchronize the following couples of actions:

- g!exp and g?x:s[cond]: in this case, if the result r of exp is of sort s and satisfies the condition cond, then the *event* <g, r> occurs in both processes P and Q. Informally, the value r is assigned to x.
- g!exp_1 and g!exp_2 : in this case, if exp_1 and exp_2 have the same result r, <g, r> occurs in both processes P and Q.
- g?x_1:s[cond_1]and g?x_2:s[cond_2]: in this case, an event <g, r> occurs in both processes P and Q, where r is a value of sort s satisfying $\mathrm{cond}_1 \wedge \mathrm{cond}_2$. The value r is assigned to x_1 and x_2.

An important remark is that a process can block (deadlock). For instance, there is no executable action from the following states :
g!1; P |[g, h]| g?x:*Nat*[$x > 1$]; Q
g!v; P |[g, h]| h?x:s; Q
In the first example, the value 1 does not satisfy the condition $x > 1$. For the second example, the action g!v is *refused* by the process h?x:s; Q and the action h?x:s is refused by g!v; P.

There is a special observable action, known as stop, which leads to a state where there is no executable action.

The internal action i does not need to synchronize. For instance, in P|| i ; Q, the action i is executable whatever are the executable actions of P and leads to the state $P \,||\, Q$.

A sequence σ of events is *executable* by a process P if the successive events of σ can be successfully executed, possibly with interspersed internal actions. Such sequences are interesting since they correspond to observable behaviors of the process. The set of these sequences is called the *traces* of the process, and will be noted *traces*(P). It is closed under prefix, since when a sequence σ belongs to it, all its prefixes belong to it.

Coming back to the example, a trace of the *Compact* process is
<control, 25>, <inGate, H.E.L.L.O.ε>, <inGate, W.O.R.L.D.ε>,
<outGate, H.E.L.L.O.W.O.R.L.D.ε>

Remark: Actually, there is no significant difference between an event <g, r> and an action g!r. Of course, there is a distinction, since a sequence of events is a trace and a sequence of actions is a process. However, a convenient way to decide if a trace ($<\mathrm{g}_i, \mathrm{r}_i>, i = 1, ...n$) is executable by a process P, is to run the process P || $\mathrm{g}_1! \ \mathrm{r}_1$; ... ; $\mathrm{g}_n!\mathrm{r}_n$ and to observe whether this execution proceeds without blocking until the occurrence of event $<\mathrm{g}_n, \mathrm{r}_n>$. Thus in the rest of the paper we will identify the trace ($<\mathrm{g}_i, \mathrm{r}_i>, i = 1, ...n$) and the tester process $\mathrm{g}_1!\mathrm{r}_1$; ... ; $\mathrm{g}_n!\mathrm{r}_n$.

3.2. Testing Processes

A *test set* of a process P is a set of processes. The elements of this set are called *tests* or *testers*. A *test run* or a *test experiment* of a test T is the parallel execution of P and T: $P||T$ with full synchronization of the observable actions. Thus P and T must have the same observable actions to avoid meaningless deadlocks.

We have seen in Section 2 that the notion of a correct implementation with respect to an algebraic specifications is based on logical satisfaction. For processes it is based on simulation and/or containment of behaviors, behaviors being characterized by traces and deadlocks. There are several possible implementation relations for processes. They are discussed and compared in the literature (see for instance Chapter 3 of [Tre92]). In the works on test derivation, two relations are usually considered, the *testing preorder*, usually named the **red** relation, and its weaker version, namely the **conf** relation.

Definition: The **red** relation
I **red** $S \iff$ if a sequence of observable actions, σ, is executable by I and can lead to a state where all the actions of a set $\mathcal{A}$ are refused, then the sequence σ is also executable by S and can lead to a state where all the actions of $\mathcal{A}$ are refused.

The intuition behind this definition is that, if after σ I may block on a set of actions, then S must also have the possibility of blocking on those actions. The implementation does not add deadlock possibilities.

Definition: The $exhaustive_{red}$ test set
Paraphrasing the above definition, the exhaustive test set is given by the following set of processes

$exhaustive_{red}(S) = \{\sigma;\ [\,]_{a_i \in \mathcal{A}}\ \mathrm{a}_i;\ \mathsf{stop} \mid \sigma \in \mathcal{L}^*, \mathcal{A} \subseteq \mathcal{L}\}$

where $\mathcal{L}$ is the set of all actions g!v where g is any observable gate of S, and v is any value of a sort s used by S.

This test set contains all the traces obtainable from $\mathcal{L}$ followed by (a choice in) all the sets of actions of $\mathcal{L}$.

Definition: Verdict for $exhaustive_{red}$
The verdict of a test experiment $I||T$, where $T = \sigma;\ [\,]_{a_i \in \mathcal{A}}\ \mathrm{a}_i;\ \mathsf{stop}$, is defined by

1. if σ is not executed by I then *success*
2. if σ is executed by I and some action in $\mathcal{A}$ is accepted by I after σ then *success*
3. if σ is executed by I and all of the actions in $\mathcal{A}$ are refused by I after σ then
   (a) if σ is executable by S and all of the actions in $\mathcal{A}$ are refused by S after σ then *success*
   (b) otherwise, *failure*

The first case corresponds to the detection of a deadlock before the end of σ. In the second case, the test experiment reaches one of the stops in T. The third case corresponds to the detection of a deadlock just after σ.

Remark: as numerous authors we assume that deadlocks are detected by some suitable time-out mechanism.

As mentioned above, the minimal hypothesis H_{min} must include the fact that S and I have the same sets of atomic observable actions. However, more is needed to ensure that the tests of $exhaustive_{red}(S)$ are all successful if and only if I satisfies the specification S, since a trace σ can lead to several states. In practice, the only way to cope with non-determinism, when testing with a black box strategy, is to repeat each experiment "a sufficient number of times". This corresponds to an hypothesis that the non determinism of the implementation is such that after a number p of executions of the same experiment, all the possible choices in the implementation are covered. This number depends on the length of σ since some choices can be done at any action of the trace. For instance, if c is known as the maximum number of choices for any action in I, then $p(\sigma)$ must be chosen greater or equal to $|\sigma| \times c$, its value depending on the way the non determinism is known to be balanced in the implementation.

H_{min} is the conjunction of the two above properties. We now require $p(\sigma)$ successful test experiments for each test $\sigma;[\,]_{a_i \in \mathcal{A}}$ a_i ; stop, for I to pass *exhaustive*$_{red}(S)$, .

Then, the test context (H_{min}, $exhaustive_{red}(S)$) is valid and unbiased since I **red** S implies the success of $exhaustive_{red}(S)$, assuming H_{min}, and vice versa.

We will see below that $exhaustive_{red}(S)$ contains useless tests. It can be reduced without loss of validity and unbias and without strenghthening of H_{min}.

Remark: The above minimal hypothesis is not always adequate. Other minimal hypotheses may arise, depending on the knowledge on the way non-determinism is implemented. For instance [BBP98] reports a case study where Ada 95 was used as implementation language. The used compiler implemented the select statement by a sequential choice. This was quickly understood during some preliminary tests, and the corresponding hypothesis was used for stating the exhaustive test set and defining the selection.

Another, weaker, notion of implementation correctness is the **conf** relation.

Definition: The **conf** relation.
I **conf** $S \Longleftrightarrow$ if a trace of S, σ, is executable by I and can lead to a state where all the actions of a set $\mathcal{A}$ are refused, then the sequence σ when executed by S can lead to a state where all the actions of $\mathcal{A}$ are refused.

Intuitively, this definition only considers sequences of actions which are traces of the specification. There is no requirement on what happens for other sequences of actions. Thus it is clear that I **red** S implies I **conf** S.

Definition: The *exhaustive*$_{conf}$ test set
The exhaustive test set with respect to **conf** is given by
exhaustive$_{conf}(S) = \{\sigma;\ [\,]_{a_i \in \mathcal{A}}\ \mathrm{a}_i;\ \mathsf{stop} \mid \sigma \in traces(S),\ \mathcal{A} \subseteq \mathcal{L}\}$
with the same verdict as above.

Assuming H_{min}, an implementation *I* passes this test set if and only if *I* **conf** *S*. The testing context (H_{min}, *exhaustive*$_{conf}(S)$) is valid and unbiased with respect to **conf**.

If the **red** relation is the correctness reference for an implementation, it is necessary to make a stronger hypothesis on the implementation to ensure unbias and validity of exhaustive$_{conf}(S)$ with respect to **red**.

Basically, such an hypothesis assumes the success of all the tests which belong to *exhaustive*$_{red}(S)$ and not to *exhaustive*$_{conf}(S)$. Namely, every *t* in

$$\{\ \sigma;\ [\,]_{a_i \in \mathcal{A}}\ \mathrm{a}_i;\ \mathsf{stop} \mid \sigma \notin traces(S),\ \mathcal{A} \subseteq \mathcal{L}\ \}$$

is passed by *I*. In [Pha94] this kind of testing is called *robustness testing*, thus we call this hypothesis *robustness hypothesis*.

Since σ is not a trace of *S*, a test experiment $t\|\ I$ is a success only if the premise in the definition of **red** does not hold, i.e, either σ is not executable by *I* or for all $\mathcal{A} \subseteq \mathcal{L}$, at least one action in $\mathcal{A}$ is executable after σ. This last property cannot be true for all subsets of $\mathcal{L}$, since it is obviously false for the empty set. Thus the robustness property reduces to the fact that any trace which is not a trace of *S* is not executable by *I*, i.e.

$$H_{robust} = traces(I) \subseteq traces(S)$$

This leads to a simplification of *exhaustive*$_{red}(S)$ into
exhaustive'$_{red}(S)$ = *exhaustive*$_{conf}(S) \cup \{\sigma\ ;\ \mathsf{stop} \mid \sigma \notin traces(S)\}$
where the verdict on the tests of *exhaustive*$_{conf}(S)$ is unchanged, and the verdict on the unspecified traces is a success if the experiment blocks before the end of the trace, and a failure otherwise.

It is also possible to simplify *exhaustive*$_{conf}(S)$, and thus *exhaustive*$'_{red}(S)$. Let us consider a test

$$T = \sigma;\ [\,]_{a_i \in \mathcal{A}} \mathrm{a}_i;\ \mathsf{stop}$$

where σ is a trace of *S*. If all of the actions in $\mathcal{A}$ are refused by *S* after σ then, from the definition of the verdict, the test experiment $T\|I$ is a success for any implementation *I*, and thus it is useless. It means that the only useful tests are those where at least one action of $\mathcal{A}$ is always executable after σ.

exhaustive'$_{conf}(S) = \{\sigma; [\,]_{a_i \in \mathcal{A}} \mathrm{a}_i\ ;\ \mathsf{stop} \mid \sigma \in traces(S), out_S(\sigma) \cap \mathcal{A} \neq \emptyset\}$

The definition of $out_S(\sigma)$ is a bit tricky since, as mentioned above, there may be several states reachable after a trace. Let us note "*S* after σ" this set of states. Since we only consider cases where a deadlock is impossible, we require there to be at least one action of $\mathcal{A}$ that is executable in any of these states. Let us call $exec(s)$ the set of observable actions executable from state s. We get

$$out_S(\sigma) = \cap_{s \in S\ after\ \sigma} exec(s)$$

Here, the verdict of a test experiment $T\|\ I$ is a success when the execution reaches one of the stops in *T* and a failure otherwise.

Such tests are called *must tests* in [Hen88]. In [Tre92], they are associated with the property " after σ must $\mathcal{A}$". Our *exhaustive'*$_{conf}(S)$ is the same as $\Pi^{tr}_{conf}(S)$ which is shown to be complete, i.e. valid and unbiased, in [Tre92].

The transposition of the notions of uniformity and regularity hypotheses, presented in Section 2, to conformance testing of communication protocols has been studied in Phalippou's thesis [Pha94]. Moreover, Phalippou proposed some new kinds of selection hypotheses such as *independence* hypothesis or *fairness* hypothesis. We do not address these last points here.

Regularity hypotheses have been implicitely used for a long time to deal with processes with infinite behaviors. When a S is not terminating, for instance because of a recursive definition, *traces*(S) contains infinite traces, and thus *exhaustive'*$_{conf}(S)$ contains arbitrary long testers. It means non terminating test experiments, which are not practically acceptable. A natural selection hypothesis is to assume that: When all the test experiments with testers of "length" less than a given limit are successful, then all the test experiments are successful. The "length" may be defined as the number of observable actions or, more naturally in our opinion, as the number of recursive calls performed. Another possibility is to make a uniformity hypothesis on this number, that is to test for an arbitrary number of recursive calls.

Uniformity hypotheses can be stated in a way to ensure that the control structure of the process under test is covered. Namely, for each variable occuring in a process, some relevant values are chosen in such a way that every possible syntactical path of the process is exercised.

As an example, some testers for *Compact*(7) could begin by

- control!4; ...
- control!0; ...
- inGate!H.E.L.L.O.ε ; inGate!W.O.R.L.D.ε ; outGate!H.E.L.L.O.ε!W.O.R.L.D.ε ; ...
- inGate!H.E.L.ε ; inGate!W.O.ε ; outGate!H.E.L.W.O.ε ; ...

Here, there is a first uniformity hypothesis on the parameter Max, for which an arbitrary *Nat* value is chosen, namely 7. Then the first test corresponds to a uniformity hypothesis on the sub-domain newMax > 0, and 4 is chosen. The third test corresponds to the sub-domain Size(x) + Size(y) $>$ Max, and the last one to Size(x) + Size(y) $\leq$ Max.

The computation of the suitable sub-domains can be supported by a symbolic execution tool for LOTOS specifications.

This approach corresponds to the test derivation procedures described by Eertink in [Eer94]. In the next section we suggest some way of finding out weaker uniformity hypotheses by using not only the syntactic definition of the processes, but also the properties of the used data types.

4. Test Selection for full LOTOS Specifications

A first approach to test selection from a full LOTOS specification is to use independently the two approaches presented in Sections 2 and 3. It means that a test context is built for the data type part of the specification, and another one for the behavior part. This approach is illustrated below on the specification shown in Figure 1.

4.1. Algebraic Data Types

First, let us present the exhaustive test set for the data type part of this specification. For the data type *Message*, there are four axioms. Therefore *exhaustive* (*Message*) is the union of four test sets:
$exhaust_{ax1} = \{\mathrm{Pack}(\varepsilon,\ \mathrm{m}) = \mathrm{m} \mid \mathrm{m} \in \mathrm{T}_{Message}\}$
$exhaust_{ax2} = \{\mathrm{Pack}(\mathrm{o}_1.\mathrm{m}_1,\ \mathrm{m}) = \mathrm{o}_1.\mathrm{Pack}(\mathrm{m}_1,\ \mathrm{m}) \mid \mathrm{o}_1 \in \mathrm{T}_{Octet}, \mathrm{m}_1, \mathrm{m} \in \mathrm{T}_{Message}\}$
$exhaust_{ax3} = \{\mathrm{Size}(\varepsilon) = 0\}$
$exhaust_{ax4} = \{\mathrm{Size}(\mathrm{o}_1.\mathrm{m}) = \mathrm{Succ}(\mathrm{Size}(\mathrm{m})) \mid \mathrm{o}_1 \in \mathrm{T}_{Octet},\ \mathrm{m} \in \mathrm{T}_{Message}\}$,
where T_s is the set of ground terms of sort s.

Since there are no conditions on the variables in the specification, a first selection strategy would be to make a uniformity hypothesis on all the variables. This means that there are four tests, which are arbitrary instantiations of the axioms.

If for some reason more tests are desired, one possibility is to *unfold* Pack in the right hand side of the second axiom. The definition of Pack distinguishes between ε and $\mathrm{o}_1.\mathrm{m}_1$ as first arguments. This leads to two new tests, which are arbitrary ground instances of

$$\mathrm{Pack}(\mathrm{o}_1.\varepsilon,\ \mathrm{m}) = \mathrm{o}_1.\mathrm{m}$$
$$\text{and } \mathrm{Pack}(\mathrm{o}_1.\mathrm{o}'_1.\mathrm{m}'_1,\ \mathrm{m}) = \mathrm{o}_1.\mathrm{o}'_1.\mathrm{Pack}(\mathrm{m}'_1,\ \mathrm{m}).$$

Here, the uniformity sub-domains for the first operand of Pack are the messages of size 1 and the messages of size greater than 1.

Unfolding is a classical technique for discovering uniformity hypotheses from a specification [Mar95]. It can be automated by using a narrowing procedure and is briefly described below.

In presence of a conditional axiom such as

$$v_1 = w_1 \wedge \ldots \wedge v_n = w_n \Rightarrow v = w,$$

an obvious candidate for a uniformity sub-domain is the set characterized by the premisse of the axiom. It means that the axiom is tested once, with some arbitrary values satisfying the premisse. However, it is possible to use the other axioms of the specification to get weaker uniformity hypotheses, an more tests for this axiom.

Assume that there is an occurrence of a term $f(u)$ in the axiom above, that f is defined by some axioms, and that among them there is the axiom

$$vv_1 = vw_1 \wedge \ldots \wedge vv_m = vw_m \Rightarrow f(vv) = vw$$

The replacement of $f(u)$ by vw in the first axiom is conditioned by the validity of the premisse of the second one and the fact that $u = vv$ (modulo some renaming when necessary). More precisely, if we note $v'_1, \ldots, v'_n, w'_1, \ldots, w'_n, v', w'$ the terms obtained by replacing $f(u)$ by vw in $v_1, \ldots, v_n, w_1, \ldots, w_n, v, w$, we get the formula

$$v'_1 = w'_1 \wedge \ldots v'_n = w'_n \wedge u = vv \wedge vv_1 = vw_1 \wedge \ldots \wedge vv_m = vw_m \Rightarrow v' = w'.$$

This defines a new uniformity sub-domain which is a combination of two cases mentioned in the specification. If f is defined by p axioms, we get p test cases for the original axiom. Note that the terms of the new formula can be simplified, using other axioms. It is what we have done when unfolding Pack in the second axiom.

The same technique can be applied to the fourth axiom, unfolding the second occurrence of Size. Since Size is defined by two axioms, it gives two test cases, one where $m = \varepsilon$ and one where $m = o'_1.m'$.

Another testing strategy is to choose a regularity hypothesis on k, the number of *Octet*s inserted in a *Message*. In this case, the set of tests is the set of all the ground instances of the axioms where the *Message* variables are replaced by ground terms with at most k *Octet*s.

As mentioned earlier, a test context consists of a set of hypotheses and a test set. The selection hypotheses used above can be combined to adjust the size of the test set. Unfolding, regularity, random resolution of uniformity sub-domains, are supported by the LOFT system [Mar95] which has been experimented on several realistic case studies [DGM93, MTFW+92].

4.2. Processes

As mentioned in Section 3, in [Eer94], Eertink gives an algorithm that generates a set of symbolic test cases for full LOTOS processes. The method is recursive, determining actions that can be immediately performed by the process *P* under test followed by the actions that can be performed by *P* after these initial actions. Eerting gives several variants; here we follow Algorithm 4.16 of [Eer94].

As an example, we derive the test set TS(*Compact*(Max)).

Eertink uses the following notations

- The tester for an action with the form g?x:*type*[pred] is choice x:*type*[] [pred];→i;g!x, meaning that any action g!v where v is a value of type *type* satisfying the condition pred is executable. Using our notation of Section 3, it is the generalized choice $[\,]_{v:type,pred[v/x]}$ g!v among all the values v of type *type* satisfying pred.
- The tester for g!v is just g!v.
- The tester for exit is success ; stop, where success is a special action which indicates the success of the test.

The set of initial actions of *Compact*(Max) is the set
{control?newMax:*Nat*[newMax > 0],
control?newMax:*Nat*[newMax = 0],
inGate?x:*Message* },
thus the test set contains the processes
{choice newMax:*Nat*[] [newMax > 0]→i;control!newMax; t_a,
choice newMax:*Nat*[] [newMax = 0]→i;control!newMax; t_b,
choice x:*Message*[]→i;inGate!x; t_c },
where
t_a = TS(*Compact*(newMax)),
t_b = success; stop,
t_c = TS(*Compact*(Max) **after** inGate?x:*Message*).

The process *Compact*(Max) **after** inGate?x:*Message* is
inGate?y:*Message*[Size(x)+ Size(y)> Max];
outGate!x!y; *Compact*[inGate, outGate, control](Max)
[]

```
    inGate?y:Message[Size(x)+ Size(y)<= Max];
      outGate!Pack(x,y); Compact[inGate, outGate, control](Max)
```

Applying the algorithm again to this new, shorter process gives the test set

```
t_c = { choice y:Message[] [Size(x)+ Size(y)> Max]→i;inGate!y; outGate!x!y;
            TS(Compact(Max)),
        choice y:Message[] [Size(x)+ Size(y)<= Max]→i;inGate!y;
            outGate!Pack(x,y); TS( Compact(Max))
      }
```

and so there are four processes in TS(*Compact*(Max)):

```
{ choice newMax:Nat[] [newMax > 0]→i;control!newMax;
      TS(Compact(newMax)),
  choice newMax:Nat[] [newMax = 0]→i;control!newMax; success; stop,
  choice x:Message[]→i;inGate!x ;
      choice y:Message[] [Size(x)+ Size(y)>Max]→ i; inGate!y;
      outGate!x!y; TS(Compact(Max)),
  choice x:Message[]→i;inGate!x ;
      choice y:Message[] [Size(x)+ Size(y)<= Max]→i;inGate!y;
      outGate!Pack(x, y); TS(Compact(Max))
}.
```

The second element of TS(*Compact*(Max)) is finite. In fact, there is only a single instantiation that will satisfy the predicate [newMax = 0].

The other three processes are infinite, and there are two sources of their non-finiteness. One of these is the recursive call to TS(*Compact*). Eertink notes that since test cases must be finite, it is reasonable to limit the size of the tests at some point, but he makes no comment on how this point should be determined or quantified. As said at the end of Section 3, a regularity hypothesis can be made on k, the number of times the recursive call is used to expand the test cases.

The other source of infiniteness is the presence of variables and parameters which are of infinite types. Eerting describes how to propagate the constraints on variables along a syntactic path of the process under test, and then to use a resolution procedure to get a set of values satisfying these constraints (modulo some problems due to unfeasible paths and non termination of the resolution procedure). This is an implicit way of assuming a uniformity hypothesis for the activation space of each path of the process under test, thus to have one test for each path (path coverage).

4.3. Using Data-Type Properties to Test Processes

However, in presence of complex guards and conditions, making systematic uniformity hypotheses on the predicates of the paths of the behavior expressions is not always desirable. A less brute-force way is to consider weaker uniformity hypotheses derived from the specification of the data types inspired from the techniques presented in Section 4.1.

For example, the fourth process in TS(*Compact*(Max)) includes the predicate [Size(x) + Size(y) <= Max]. Using standard unfolding of <=, as it is implemented in the LOFT system, this predicate can be broken into the two cases [Size(x) + Size(y) < Max] and [Size(x) + Size(y) = Max].

Then, the test set given at the end of Section 3 for *Compact*(7) becomes, for instance

- control!4 ; ...
- control!0 ; ...
- inGate!H.E.L.L.O.ε ; inGate!W.O.R.L.D.ε ; outGate!H.E.L.L.O.ε!W.O.R.L.D.ε ;...
- inGate!H.E.L.ε ; inGate!W.O.ε ; outGate!H.E.L.W.O.ε ; ...
- inGate!H.E.L.L.ε ; inGate!W.O.R.ε ; outGate!H.E.L.L.W.O.R.ε ; ...

The fifth test introduced above corresponds to the limit value for packing the messages.

Unfolding can also be used on the operations Size and Pack. For example, Size differentiates between messages with the forms ε and o.m. In the third and fourth processes of *TS*(*Compact*(Max)), Size occurs in the expression Size(x) + Size(y). Unfolding once the two occurrences of Size in this expression gives four cases.

For instance, the third test case of *TS*(*Compact*(Max)) is then replaced by the following three processes (the case where x and y are ε being insatisfiable):

- choice x:*Message* [] [x = ε]$\rightarrow$i ; inGate!x;
  choice o:*Octet*, y, y′:*Message* [] [y=o.y′$\wedge$ Size(x)+ Size(y)>Max]$\rightarrow$i;
  inGate!y; outGate!x!y; *TS*(*Compact*(Max)),
  which tests the case of a first empty message and a second non empty one of size greater than Max,
- choice o:*Octet*, x, x′:*Message* [] [x=o.x′]$\rightarrow$i ; inGate!x;
  choice y:*Message* [] [y = ε $\wedge$ Size(x)+ Size(y)>Max]$\rightarrow$i; inGate!y;
  outGate!x!y; *TS*(*Compact*(Max)),
  which tests the case of a first non empty message of size greater than Max and a second empty one , and finally
- choice o:*Octet*, x, x′:*Message* [] [x=o.x′]$\rightarrow$i ; inGate!x;
  choice o′:*Octet*, y, y′:*Message* [] [y=o′.y′$\wedge$ Size(x)+ Size(y)>Max]$\rightarrow$i;
  inGate!y; outGate!x!y; *TS*(*Compact*(Max)),
  which tests the case of two non empty messages of total size greater than Max.

In the second test case above, it appears that some backward propagation of the second constraint on x must be performed as described by Eerting. The whole process, unfolding and propagation, could be supported by some tool integrating the functionalities of Eerting's symbolic simulator [Eer94] with Marre's LOFT system [Mar95].

The same unfolding of Size can be performed on the two sub-cases obtained above by unfolding <= in the fourth process of *TS*(*Compact*(Max)). It leads to four cases in the "< Max" case, and three cases in the "= Max" case, since then the case where both x and y are empty is unfeasible.

The success of these tests ensures that *Compact* behaves well in the nominal cases and in two kinds of special cases: when the total size of the messages is exactly Max, and when there are empty messages. It is worth noting that these cases are obtained systematically as soon as the decision is made of weakening the uniformity hypotheses on paths by unfolding once each occurrence of the operations <= and Size, using the axioms of the data type part of the Full LOTOS specification.

5. Conclusions

This paper brings two main contributions. The first one develops the generic framework for test derivation from formal specifications sketched in [BGM92]. This framework is first instantiated on algebraic data types, and then on the behavior part of LOTOS. In this last case, the exhaustive test set corresponds, after some simplification, to the must tests of Hennessy and then to the approach of the Brinksma's group.

The fact that this framework works on features as different as data types and processes confirms its generality.

The second contribution is the proposition of a new integrated test selection method from full LOTOS specifications, considering the characteristics of the data type part when treating the process part. This leads in a natural way to test cases corresponding to special subdomains of the guards and conditions occuring in a process description. This approach is clearly transposable to other languages where data types and processes coexist, for instance SDL or Promela.

Acknowledgement: This work was partially supported by the *DEVA Long Term Research Project 20072* of the ESPRIT program. Marie-Claude Gaudel thanks warmly the Instituto de Matematica e Estatistica (IME), Universidade de São Paulo, for its kind hospitality during the academic year 1996-97, and the members of the IFIP Working Group 1.3 for fruitful discussions.

References

[BBP98] S. Barbey, D. Buchs, and C. Peraire. Modeling the production cell case study using the Fusion method. Technical Report 98, EPFL-DI, 1998.

[BGM91] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: A theory and a tool. *IEE Sofware Engineering Journal*, 6, November 1991.

[BGM92] G. Bernot, M.-C. Gaudel, and B. Marre. A formal approach to software testing. In Maurice Nivat, Charles Rattray, Teodor Rus, and Giuseppe Scollo, editors, *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology*, Workshops in Computing, pages 243–253, London, May 22–25, 1992. Springer Verlag.

[Bri89] E. Brinksma. A theory for the derivation of tests. In Peter H. J. van Eijk, Chris A. Vissers, and Michel Diaz, editors, *The Formal Description Technique LOTOS: Results of the ESPRIT/SEDOS Project*, pages 235–247. Elsevier Science Publishers North-Holland, 1989.

[CCI88] CCITT. Specification and description lanuage (SDL). Recommendation Z.100, CCITT, 1988.

[Cho78] T. S. Chow. Testing software design modeled by finite-state machine. *IEEE Transactions on Software Engineering*, 4(3), 1978.

[DGM93] P. Dauchy, M.-C. Gaudel, and B. Marre. Using algebraic specifications in software testing: a case study on the software of an automatic subway. *Journal of Systems and Software*, 21-3:229–244, 1993.

[dNH84] R. de Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Sceience*, 34, 1984.

[Eer94] E. H. Eertink. *Simulation Techniques for the Validation of LOTOS Specification*. PhD thesis, Universiteit Twente, the Netherlands, March 1994.

[EM85] H. Ehrig and B. Mahr. *Fundamental of Algebraic Specification 1.* EATCS monographs on Theoretical Computer Science. Springer Verlag, 1985.

[FKG91] G. Fujiwara, S.and von Bochmann, F. Khendek, and A. Ghedamsi. Test selection based on finite-state models. *IEEE Transactions on Software Engineering*, 17(6), 1991.

[Gau95] M.-C. Gaudel. Testing can be formal, too. *Lecture Notes in Computer Science*, 915:82–96, 1995.

[GG75] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):158–173, june 1975.

[Hen88] M. Hennessy. *An algebraic theory of processes.* MIT Press, 1988.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[ISO89] ISO. LOTOS: A formal description technique based on the temporal ordering of observational behaviour. Technical Report 8807, International Standards Organisation, 1989.

[LY94] D. Lee and M. Yannakakis. Testing finite-state machines: state identification and verification. *IEEE Transactions on Computers*, 43(3), 1994.

[Mar95] B. Marre. LOFT: A tool for assisting selection of test data sets from algebraic specifications. *Lecture Notes in Computer Science*, 915:799–800, 1995.

[MTFW+92] B. Marre, P. Thévenod-Fosse, H. Waeselynck, P. Le Gall, and Y. Crouzet. An experimental evaluation of formal testing and statistical testing. In *SAFECOMP-92*, 1992.

[PF90] D. H. Pitt and D. Freestone. The derivation of conformance tests from LOTOS specifications. *IEEE Transactions of Software Engineering*, 16(12), 1990.

[Pha94] M. Phalippou. *Relations d'implantation et hypothèses de test sur des automates à entrées et sorties.* PhD thesis, Université de Bordeaux, France, 1994.

[Tre92] J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, Universiteit Twente, the Netherlands, December 1992.