Using Algebraic Specifications in Software Testing : a case study on the software of an automatic subway*

P. Dauchy M.-C. Gaudel B. Marre

Laboratoire de Recherche en Informatique
U.M.R. CNRS 8623
Université de Paris Sud / Orsay, Bât. 490
91405 ORSAY CEDEX
FRANCE

Résumé

Ce rapport présente une expérience utilisant une spécification formelle d'un logiciel critique comme base pour la dérivation de jeux de tests. Le logiciel était la partie embarquée du système de conduite d'un métro automatique. Il a été spécifié formellement en utilisant des techniques algébriques. Deux modules de la spécification ont été utilisés en entrée de l'outil de sélection de jeux de tests LOFT, qui est fondé sur la surréduction et d'autres aspects de programmation logique avancée. Cette expérience a été très fructueuse : elle établit la praticabilité de l'approche sur des exemples industriels ; elle révèle quelques difficultés tout en donnant une idée de la façon de les résoudre.

Abstract

This paper reports an experiment on the use of a formal specification of a critical software as a basis to derive test data sets. The software was the on board part of the driving system of an automatic subway. It was formally specified using algebraic techniques. Then two modules of the specification were used as input of the LOFT test selection tool which is based on narrowing and some other aspects of advanced logic programming. The experience was very fruitful: it established that the approach is practicable on industrial examples; it pointed out some difficulties and gave some hints on how to solve them.

*Part of this work was presented in 4^{th} European Software Engineering Conference, Springer-Verlag L.N.C.S. 550.

Introduction

Formal methods are generally advocated for critical software because they make it possible to prove the correctness of the software with respect to some formal specification. However, these methods also provide some new possibilities for guiding testing: some criteria can be defined on the texts of formal specifications in the same way as coverage criteria or data flow criteria are defined on the texts of programs. Besides, the formal specification generally allows some sort of symbolic execution: this can help the procedure which decides the success or the failure of a test experiment, the so-called oracle. Moreover, such approaches provide a formal framework for the study of "black-box testing", i.e. testing methods which are only based on the specification.

When there is an obvious underlying graph in the formalism, as for instance in Petri nets or automata, there are quite natural criteria [23], [12]. But in other cases, specific criteria must be established. In the case of algebraic specifications, several proposals have been done in [13], [18], [4]. More recently, we have presented in [2] a theory of software testing and a tool, named LOFT, which assists the selection of test data sets from algebraic specifications. LOFT is an acronym for LOgic for Functions and Testing; the underlying mechanisms are narrowing and some other aspects of advanced logic programming. All these proposals are discussed in the next section.

In the area of protocols, and more generally of distributed systems, a significant amount of work has been done around LOTOS, mainly on the part concerning processes [5], [24]; besides, a formal approach to testing distributed systems is presented in [6]; although the kinds of properties to be tested are different, this theory exhibits some interesting similarities with the one in [2], giving good hope for a unified theory of testing based on formal specifications.

Even in the area of real-time programming, where testing is considered to be especially difficult, some results, based on the TRIO specification language are reported in [20].

These methods look very attractive, but they still need to be experimented on realistic case studies. This paper reports such a case study: the software was the on board part of the driving system of an automatic subway.

The basis of the experiment was an algebraic specification of the safety control part of the system [10] written in the PLUSS specification language [3], [15]. Then, using the LOFT system, some automatic test data selections were performed on two modules of the specification, namely the control of the opening of the doors and the overspeed detection. It turned out that the first experiment, the one on the doors control, was easy to perform and led to pertinent results. However, the second experiment, on the overspeed detection, raised some problems and led us to some refinements in the use of the selection tool.

The plan of the paper is the following: the next section presents and discusses some related works on testing from algebraic specifications; then section 2 presents the informal and formal specifications of the case study; section 3 presents the testing method and the LOFT system; section 4 reports on the experiments with the DOORS module of the specification; section 6 does the same on the OVERSPEED module; then some general conclusions and new perspectives are given.

1 Related works on testing and algebraic specifications

The specifications we use are structured positive conditional algebraic specifications, as described in [14]. Algebraic specifications are a way of defining the behaviour of the abstract data types involved in the specified software. These data types are described by the properties (axioms) of their operations. Specifications are structured in specification modules. Generally, a module defines only one abstract data type.

More formally, the body of a specification module is a pair $SP = (\Sigma, Ax)$; the signature Σ is a finite set S of *sort* (i.e. data type) names plus a finite set of *operation* names with arity in S; the

properties of the operations of S are defined by the set Ax of *axioms* of the form:

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

where v_i, w_i, v and w are Σ-terms with typed variables (i.e. well-formed compositions of operations of Σ with typed variables). The formula $(v_1 = w_1 \wedge \ldots \wedge v_n = w_n)$ is called the *precondition* of the axiom.

Among the operations of the signature, a subset of *generators* is singled out; any ground term can be proved (using equational reasoning) equal to a term built from the generators only. The other operations are called *defined operations*. We only consider specifications with no equation between generators, thus any two different terms built from generators denote different values.

The structure of a specification is expressed by the "use" construction. A specification module is a body, as above, together with a list of used modules. All the operations and sorts declared in a *used specification module* are known in the specification modules which use it. The "use" construction is transitive, i.e. if a module SP uses SP_1 which uses SP_2, then SP uses SP_2. We will see later on that, in our approach, the way a specification is structured influences the test strategy: modules are tested in a bottom-up way; every used module is assumed to be already tested.

Several proposals have been done for using algebraic specifications as a basis for testing since the pioneering work of Gannon, McMullin and Hamlet [13]. This work described a system, called DAISTS, for testing a program against equations: the left-hand side and the right-hand side of the equation are compiled into two sequences of procedure calls of the program; then these sequences are exercised on some user-supplied test data and the two results are compared. It is interesting to note that in this framework, the oracle problem, i.e. the problem of deciding the success or the failure of a test, turns out to be the problem of deciding whether two values are equivalent or not, with respect to some abstraction.

The problem of generating relevant test set from algebraic specifications using logic programming was first addressed in [4]. One of the ideas was to transform positive conditional axioms into equations using some resolution of the preconditions; testing a program was then performed following the DAISTS scheme; moreover, some automatic generation of the test data was proposed along the following principle: the definition of the test set was stated by means of some *uniformity hypotheses* or *regularity hypotheses*. Informally, a uniformity hypothesis is an assumption that it is enough to have one value of a subdomain in the test set, and for a regularity hypothesis that it is enough to have those terms of size less than a given limit. Uniformity hypotheses were applied to sorts defined in the used modules; regularity hypotheses were applied to sorts defined in the current module.

In [9], Choquet suggested to improve the above approach by the use of logic programming with *constraints*: the preconditions of conditional axioms are then considered as constraints on the variables of the conclusion; this allows a cleaner and more efficient way of generating pertinent instantiations of these variables. At the same time, Wild [26] presented similar ideas for specifications analysis.

Some other works, which do not follow the DAISTS scheme, were reported by Jalote in [18] and [19].

In [18], axioms were equations and they were considered as rewrite rules. This is a classical way to make executable algebraic specifications, which is valid if the resulting term rewriting system is confluent and terminates. This executable specification was used as an oracle for some test data (a test data was a term on the signature). The values returned by the rewriting of these terms must be of type other than an abstract data type: the reason, which was not explicitely stated in the paper, is that deciding the equality of two representations of abstract objects is a complex problem; one way is to *observe* such objects via operations with results of a predefined type with an equality. The paper briefly addressed the problem of test data generation: it was suggested to exercise all the legal

combinations of operations such that their nesting level is less or equal than the maximum nesting level in the axioms of the specification.

The work reported in [19] is less directly related to algebraic specifications since the described test case generation uses only the signature of the data type. The paper presents a tool which produces all the well-typed terms with variables of a depth less or equal than a given limit. The initial values of the variables are user-supplied. It was experimentally found, on several classical abstract data types, that with proper initial values, depth 2 gives a good error detection power. There are no rules for selecting the initial values.

The approach we have applied in the experiments reported here is described at length in [2]. It provides a method and a tool for testing programs against algebraic specifications. Its main characteristics are:

- We follow the DAISTS scheme; our solution to the oracle problem is to only consider equalities between observable terms, i.e. of sort corresponding to a predefined type with equality in the programming language; equalities between non observable terms are systematically transformed into a set of observable ones via *observable contexts*.
- We consider positive conditional axioms and the tool is based on equational logic programming (more precisely, conditional narrowing) with constraints, plus some specific additional features.
- The test data selection is done from the so-called *observable exhaustive* test data set, which contains all the ground instantiations of the axioms, embedded, when they are not observable, into all the possible observable contexts; the selection of a finite subset is performed via uniformity hypotheses.
- Uniformity hypotheses are chosen by the user with the assistance of the tool which performs some case analysis on the axioms, either conditional or equational, through the use of constraints: thus, constraints are used in a more general way than in [9].

A more precise overview of this method and the tool is given in section 3.

2 The case study

This case study was suggested by INRETS (Institut National de Recherche sur les Transports et leur Sécurité); this research institute is the French duly authorized body for ground transportation systems safety certification. The first goal of the case study was the elaboration of an algebraic specification from the manufacturer's informal specifications [16].

The subject of the case study is an unmanned subway driven by a software located on board. The embedded driving system or *embedded unit* (EU) takes care of most safety functions. It receives from wayside equipment (WE) some information on its environment, and especially on the other trains. The wayside equipment is controlled by a central control office (CCO), as shown on Figure 1.

Let us first introduce a few points of terminology. The way is divided into *segments* on which the position of each train is given by its *abscissa*. An embedded measurement system computes periodically the abscissa and the speed of the train. Segments are designated by their number; for each segment, it is known whether it belongs to the wayside or not, and if the second case, whether there is a station on it. Furthermore, every train has a certain *itinerary* to follow and determines from this itinerary which halting point it must respect. The nearest train ahead is known as the *target*.

We present here two specification modules on which test data selection experiments were performed; these are the ones specifying the door monitoring module, whose function is to raise an alarm when a dangerous situation results from a door opening, and the overspeed detection module, which similarly

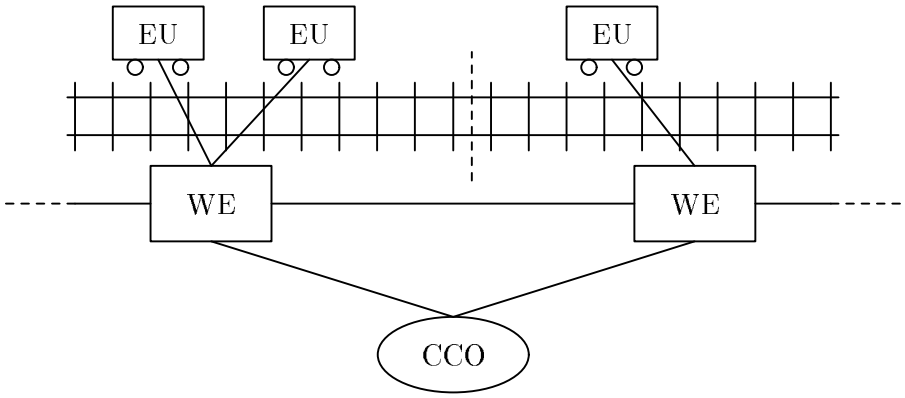

Figure 1: Automatic driving system organization

raises an alarm when the speed is too high. Both make use of a module called STATE which defines the data structure used to store the current values of the train parameters. The structure of the part of the specification we are interested in is represented on Figure 2.

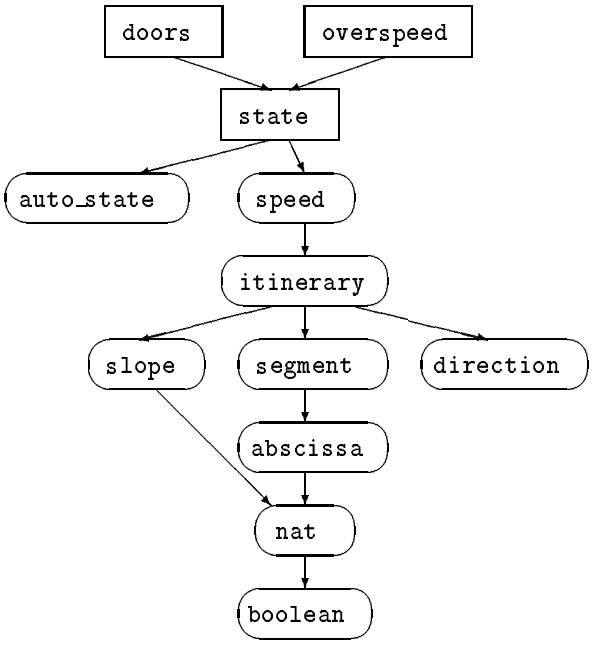

Figure 2: The specification modules and their dependencies

On this graph, the nodes are specification modules and the links represent the "use" relation.

The BOOLEAN and NAT modules are classical specifications of booleans and natural numbers, the latter being generated by 0 and a successor operation `s _ : nat → nat`.

The SPEED, SLOPE and ABSCISSA modules define the sorts of the same names by coercion of the naturals. A speed, slope or abscissa is thus given by its measure (in some arbitrary unit). Some constants are also defined in these modules.

The SEGMENT module defines the segments by coercion of naturals, since a segment is designated

by its number. It also defines four auxiliary operations. The boolean operations `sidetrack_segment` and `station_segment` test the nature of a segment; The operations `abs_min_pf` and `abs_max_pf` give the abscissas of the entry and exit points of the station platform (if there is a station on the segment).

The AUTO_STATE module enumerates the four states in the *doors automaton*, which will be presented later.

The DIRECTION module enumerates the possible directions, `forward` and `backward`.

The STATE module defines the data type used to store the current values of the train parameters. In the informal specification, these values are fetched from a global, implicit state. In order to build an algebraic specification, we need to make this state explicit. We chose to specify it as a "record" of a number of values. The only generator is the operation `cons` that builds a value of sort `state` from arguments of adequate sorts. Access operations return the values of the fields, each taking a state as argument: they are observers on the `state` sort.

To make the results simpler, we chose to particularize the specification of states according to the module under test; we kept only the fields which are relevant to this module.

2.1 The doors monitoring function

In order to ensure safe door openings, the following principles must be respected. Doors may not be open on the way side, except when the train is stopped on a sidetrack. Platform doors should not normally be open while the train is moving. However, to increase the passenger flow, the doors may be opened in station when the speed is less than some value `spd_open_doors`, thus anticipating the halting. But care must be taken to ensure safety:

- the train must not accelerate again beyond the speed limit above;
- it must stop before it has covered some distance limit `dx_max`;
- it must not pass the end of the station with its doors open.

When a forbidden situation happens, the alarm `al_doors` is set and emergency braking occurs.

In addition to the informal specification, an automaton describing possible "states" of the train w.r.t. the doors was provided; it is reproduced on Figure 3.

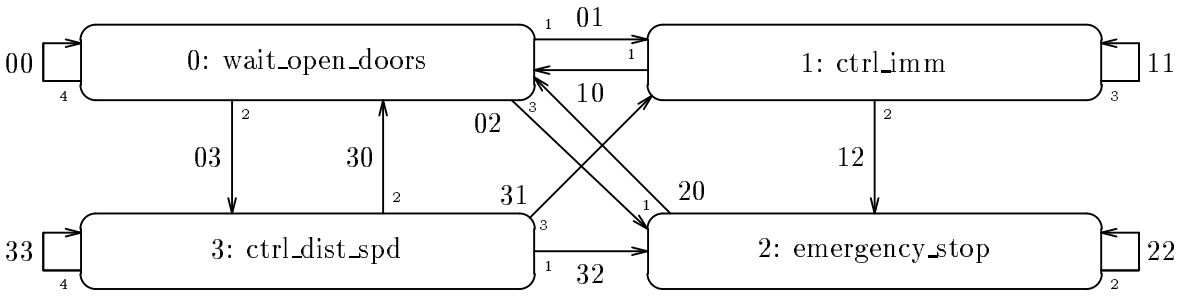

Figure 3: The doors automaton

The transitions are numbered according to their initial and final states. We give hereafter the respective firing conditions for each transition as originally given:

01 $\neg(closed\_pf \wedge closed\_way) \wedge on\_sidetrack \wedge (speed = 0)$

where `closed_pf` means that the platform-side doors are closed and `closed_way` that the way-side doors are closed;

03 $located \wedge in\_station \wedge (speed < spd\_open\_doors) \wedge closed\_way \wedge \neg closed\_pf$
where `located` means that the locating system works properly and `in_station` that all platform-side doors face the platform;

02 $\neg(closed\_pf \wedge closed\_way)$
$\wedge\ \neg(located \wedge in\_station \wedge (speed < spd\_open\_doors) \wedge closed\_way)$;

10 $closed\_pf \wedge closed\_way$;

12 $speed \neq 0 \vee \neg located \vee \neg(on\_sidetrack \vee closed\_way)$;

20 $closed\_pf \wedge closed\_way$;

32 $\neg(located \wedge (delta\_x < dx\_max)$
$\wedge\ (speed < spd\_open\_doors) \wedge in\_station \wedge closed\_way)$
where `delta_x` is the distance covered by the train since the doors opened;

30 $closed\_pf \wedge closed\_way$;

31 $speed = 0$.

Clearly, these conditions are neither mutually exclusive (see for instance conditions 10 and 12), nor exhaustive (see in particular condition 20).

The first problem is solved by introducing priorities (in small numbers on the figure). These priorities are taken into account when the conditions are translated into axioms; for instance, since transition 10 has priority on transition 12, the preconditions of the corresponding axioms are respectively 10 and $(10 \wedge 12)$

To address the second problem, four more transitions (00 ... 33), were added to the automaton; they may be fired when no state change can take place. These transitions were implicit in the automaton description we had; they were elaborated while writing the corresponding algebraic specification. The preconditions of the corresponding axioms are deduced from the firing conditions of all other transitions that may take place; for instance, the precondition of the axiom corresponding to the transition 11 is $(\neg 10 \wedge \neg 12)$.

We thus obtain a set of axioms describing the automaton; the three axioms listed here are the ones on which we will present the test data selection method. The variable `S` is of sort `state`.

```
01 : auto_state(S) = wait_open_doors & and(closed_pf(S), closed_way(S)) = false
 & on_sidetrack(S) = true & speed(S) = (0)
   ⇒ next_auto_state(S) = ctrl_imm

11 : auto_state(S) = ctrl_imm & and(closed_pf(S), closed_way(S)) = false
 & speed(S) = (0) & located(S) = true & or(closed_way(S), on_sidetrack(S)) = true
   ⇒ next_auto_state(S) = ctrl_imm

31 : auto_state(S) = ctrl_dist_spd & closed_pf(S) = false & closed_way(S) = true
 & located(S) = true & lt(delta_x(S), dx_max) = true & in_station(S) = true
 & speed(S) = (0)
   ⇒ next_auto_state(S) = ctrl_imm
```

The auxiliary operation `in_station` states whether all the doors of a train face the platform, as shown on Figure 4.

The axioms defining this operation are given below:

```
i1 : station_segment(segment(S)) = true & on_sidetrack(S) = false
 & lt(abs_min_pf(segment(S)), sub(abscissa(S), delta_door_1)) = true
 & lt(sub(abscissa(S), delta_door_6), abs_max_pf(segment(S))) = true
   ⇒ in_station(S) = true
```

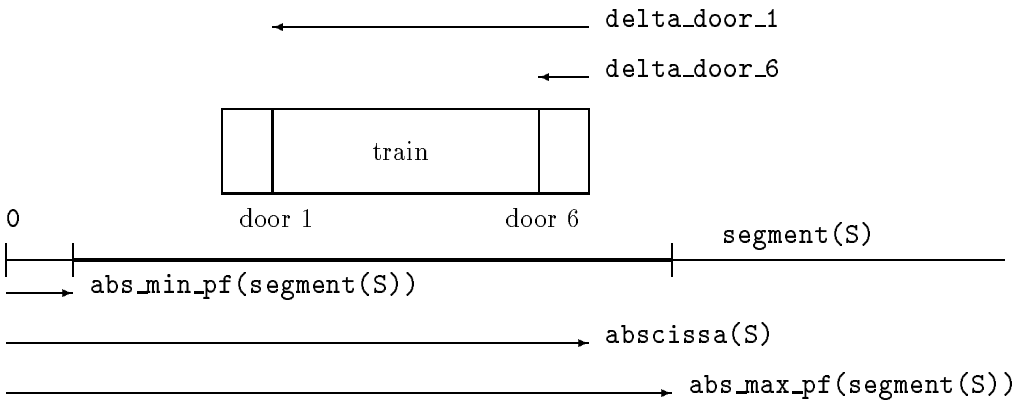

Figure 4: A train in station

```
i2 : station_segment(segment(S)) = false ⇒ in_station(S) = false
i3 : station_segment(segment(S)) = true & on_sidetrack(S) = false
 & lt(abs_min_pf(segment(S)), sub(abscissa(S), delta_door_1)) = false
   ⇒ in_station(S) = false
i4 : station_segment(segment(S)) = true & on_sidetrack(S) = false
 & lt(sub(abscissa(S), delta_door_6), abs_max_pf(segment(S))) = false
   ⇒ in_station(S) = false
```

The STATE module used by the DOORS module makes use of the following information:

- the boolean `located` is true if the locating system works correctly;
- the booleans `closed_pf` and `closed_way` are true respectively if the doors are closed on the platform side and on the way side;
- `auto_state` is the state in the doors automaton;
- `segment` and `abscissa` are self-explanatory;
- `abs_ctrl` stores the abscissa the train had on the last normal door opening in station; it is used to compute the distance the train covered since then.
- `direction` and `speed` are also self-explanatory.

2.2 The overspeed detection function

This function has the task of monitoring the train's speed and firing an alarm when this speed is too high. We present a simplified version of this function, taking only normal cases into account; when the train is on a sidetrack or is performing a change of direction, different actions may occur, which we will not describe here.

In normal cases, the alarm is set if the train's speed is greater or equal to the limit speed computed by the operation `limspd`:

```
al_ovspd(S) = ge(speed(S), limspd(S))
```

This limit speed is the smallest of four limit speeds depending respectively on the position of the first train ahead (the *target*), the point before which the train must stop, some authorizations from the ground based equipment and the geometry of the way. Thus, it can be specified by the following axiom:

`limspd(S) = min`$_4$`(target_spd(S), stop_spd(S), imposed_spd(S), way_spd(S))`

The first of these four limiting speeds is itself the greatest of two limiting speeds:

```
target_spd(S) = max(no_collision_spd(S), connect_spd(S))
```

The first one must ensure that the train can safely avoid collision with its target, and the second one allows it to connect with it if ordered so. Choosing the greatest value then allows to connect both trains at low speed.

For lack of space, we will not detail the axioms defining these operations. Suffice it to say that the first one is defined by table look-up, in function of the distance between the trains and of the slope of the way, while the second one checks the validity of the connecting command and has a limiting value if it is valid, `0` otherwise.

The function `stop_spd` determines the nearest stop point between the following three: the one associated to the itinerary (the next traffic light) and the entry points of the stations on the itinerary, if, for some reason, it is currently prohibited to enter them (for instance if a passenger has fallen on the way). It then uses table look-up to find an adequate speed in function of the distance between train and stop point and of the slope.

The function `imposed_spd` takes into account possible switch defects; it may allow the train to pass a defective switch, signalled by a ground-issued information, while limiting its speed as it does so, i.e. when it is on a certain interval surrounding the switch.

Finally, the function `way_spd` just does another table look-up to find which maximum speed is appropriate for the part of the way the train is on.

3 Overview of the method and the LOFT system

The motivation of our approach is that it is possible to use the specification of a program *to define black-box test criteria in a formal framework.* These criteria allow us to test whether the properties specified in the specification are actually dealt by the program. Test data selection is mainly guided by the structure of the specification: for each property expressed by a formula (axiom) of the specification, test data are selected via strategies derived from hypotheses chosen by the user. These hypotheses offers the advantage to make explicit the relationship between testing and correctness. Moreover, it is possible to build well suited hypotheses for each formula by combining general schemas of hypotheses for which our tool (LOFT) provides automatic selection strategies.

General theoretical results are presented in [2]. We will only focus on the points which are relevant to the case study.

3.1 Foundations of the method

Let P be the program to be tested which is supposed to implement a specification module $SP = (\Sigma, Ax)$ (with some imported specification modules $SP_1, SP_2, \ldots, SP_n$). Testing P w.r.t. SP is only possible if the semantics of P and SP are expressible in a common framework. As we are interested in

dynamic testing, we have to execute P on a finite subset of its input domain and to interpret outputs w.r.t. *SP*. In order to consider P compatible with *SP*, we need first, to be able to compute the values of Σ-terms from P and secondly, to forbid that P handles "junk" values according to the operations. Indeed, if "junk" values existed, no test derived from *SP* could detect an error involving such a value. These two conditions mean that P implements all operations in Σ and exports only them[1] (and the operations imported from the *used* specification modules), and will constitute the first hypothesis we do on a program to be tested.

Let *Exhaust* be the set of all ground instances of the axioms obtained by replacing each variable by all ground terms of the right sort. We call a test each formula f in $Exhaust$. Thus, a test f is of the form: $t_1 = t'_1 \ \& \ \ldots \ \& \ t_n = t'_n \ \Rightarrow \ t = t'$. In order to check from P that f is satisfied or not (in the first case, we note then $success_P(f)$), we need:

- first, to compute in P the values t_{iP}, t'_{iP}, t_P and t'_P corresponding to t_i, t'_i, t and t';
- then, to verify that whenever $(t_{1P} =_P t'_{1P})$ and ... and $(t_{nP} =_P t'_{nP})$, we have also $(t_P =_P t'_P)$, where $=_P$ denotes a decision procedure of the equality in P.

Exhaust, besides its infiniteness (!), has some drawbacks. First, we would like, assuming that P defines a finitely generated Σ-algebra, $success_P(Exhaust)$ to be equivalent to the correctness of P but, in general, the existence of the decision procedure $=_P$ is not guaranteed. This problem of deciding the success of a test is often called the "oracle problem" [25]. In [2] a solution based on the notion of observability is proposed: by definition, for each observable sort s, P provides a correct procedure which decides if the evaluations of two terms of sort s are equal or not in P. With this notion of observable sorts, it is possible to built observable tests by surrounding the terms of an equation in a test with compositions of operations of observable sorts.

Since each test data set of this case study involves only equations between terms of an observable sort, we shorten the overview of the method by discarding the aspects which are specific to the "oracle problem".

Another drawback of *Exhaust* is that it contains conditional tests with false preconditions. It could seem useless to verify such formulas since they do not mean anything about the program correctness. It is shown in [2] that when such a conditional test has a precondition whose equalities are between terms of observable sorts (we call such a precondition an *observable precondition*), and when the precondition can be refuted using equational reasoning, then this conditional test can be discarded. Let us consider only specifications such that their conditional axioms have observable preconditions and such that each ground instance of these preconditions can be proved or refuted using equational reasoning (this is true for this case study). From the previous result, we eliminate from *Exhaust* all the conditional tests with a false precondition.

The last drawback of *Exhaust* is that the remaining conditional tests (whose preconditions are true) are in some sense redundant with the equational tests. Indeed, these conditional tests can be seen as conjunctions (since their preconditions are true), and their preconditions are already tested in *Exhaust* by other equational tests. Another result of [2] for the same form of specifications, allows us to discard the preconditions on the remaining conditional tests of *Exhaust* when some observational hypothesis H_{Obs} holds (H_{Obs} is related to the "oracle" correctness). Let us note $Mini_P$ the conjunction of the hypotheses "P defines a finitely generated Σ-algebra" and H_{Obs}. Let $EqExhaust$ be the set of all the ground instances of the conclusions of the axioms such that the instanciations validate their precondition, assuming $Mini_P$, the success of $EqExhaust$ is equivalent to the correctness of P w.r.t. *SP*.

It is clear that *EqExhaust* is often infinite or too large and we can only consider a finite subset of it. In order to cover all properties of *SP*, we select, for each axiom *ax* of *SP*, a finite subset T_{ax} of the

[1] We say then that P defines a finitely generated Σ-algebra.

subset $EqExhaust_{ax}$ of *EqExhaust* corresponding to *ax*. When we select T_{ax}, we make the following selection hypothesis:

$$success_P(T_{ax}) \Rightarrow success_P(EqExhaust_{ax})$$

Such hypotheses are usually left implicit. We prefer to bring them up since it seems sound first, to state hypotheses H and only after, to select a test set corresponding to them.

We require both following properties for a relevant pair (H, T_{ax}):

- under H, the test data set T_{ax} rejects all programs not validating *ax* (T_{ax} is *valid*):

$$(H \wedge (success_P(T_{ax}))) \Rightarrow success_P(EqExhaust_{ax})$$

- under H, the test data set T_{ax} does not reject programs validating *ax* (T_{ax} is *unbiased*):

$$(H \wedge (success_P(EqExhaust_{ax}))) \Rightarrow success_P(T_{ax})$$

If (H, T_{ax}) satisfies these properties and T_{ax} is finite, the success of T_{ax} is equivalent to the correctness of P w.r.t. *ax*.

It is shown in [2] that the pair ($Mini_P$, *EqExhaust*) satisfies *validity* and *unbias* properties, and, moreover, that the strategies and selection hypotheses we will use hereafter preserve these properties.

Another important point of the method is that the test selection can be done in a modular way, following the specification organization in modules. We can assume that when all the test data sets for the axioms of a given specification module SP_i have been selected, we have a sufficient test data set of the properties of the operations defined in this module. Furthermore, when one considers a module *SP* that uses a module SP_i, it seems useless to test again the properties of the operations of SP_i. Thus we can assume strong selection hypotheses for the variables occurring in the arguments of these imported operations in *SP*.

3.2 Some selection hypotheses and strategies

We describe here some general schemas of selection hypotheses and their corresponding selection strategies. We do not describe all the schemas mentioned in [2] but only those that are used for our study. Furthermore, since every axiom under test in our case study involves only equations between terms of sorts observable in most programming languages, the equality provided by the target program can be used for the decision of the success of a test.

Let us consider the axiom *al3* of the specification module `doors`:

```
next_auto_state(S) = ctrl_imm ⇒ al_doors(S) = false
```

We assume that we have already selected a test data set for each axiom of each used specification module. The sort of the variable S is `state`, defined in the used specification module `STATE`.

3.2.1 Uniformity hypothesis for a variable

Since the properties of all the operations of sort `state` have already been tested, it seems useless to test again these properties in the axiom *al3*. We first assume that the operations having S as an argument are correct (in the program) for every value of S. This hypothesis is a *uniformity hypothesis for the variable S of sort* `state`. It can be expressed as follows:

$$(\forall x_0 \in state)\ (al3(x_0) \Rightarrow (\forall S \in state)\ al3(S))$$

where *al3(x)* is the instance of the axiom *al3* obtained by replacing S with x.

Assuming that this formula is true in the program under test, we could select only one instance $al3(x_0)$ of the axiom for any term x_0 of sort `state` built from generators only. Since we know the signature and the generators, the corresponding selection strategy is very easy to implement. For instance, we could select the following term for x_0:

```
cons(false,false,true,ctrl_imm,(1),(0),(1),forward,(0))
```

which means that the train is not localized with platform side doors closed and and way side doors opened; it were previously in the automaton state `ctrl_imm`; its position is given by the segment `1` and the abscissa `0` and control abscissa `1`; it is going forward and its speed is null.

Unfortunately, this term does not satisfy the precondition of the axiom *al3*. From the specification, we can prove that `next_auto_state(`x_0`)` is `emergency_stop`, not `ctrl_imm`. Thus, we cannot select the test made of the instance of the conclusion of $al3(x_0)$.

The previous hypothesis was not appropriate. In fact, the hypothesis we make is that it is sufficient to test *al3* for any instance such that its precondition is satisfied. This hypothesis is a *uniformity hypothesis on the domain* of the precondition of *al3*.

3.2.2 Uniformity hypothesis on a domain

For the axiom *al3*, this *uniformity hypothesis* can be expressed as follows:

$$(\forall x_0 \in \mathcal{D}om(next\_auto\_state(S) = ctrl\_imm))\ \ (al3(x_0)\ \Rightarrow\ (\forall S \in state)\ al3(S))$$

Assuming that this formula is true in the program under test, we can select only one instance $al3(x_0)$ of the axiom, for one arbitrary term x_0 which denotes some value of the validity domain of the precondition.

From the form of our specifications, the kind of domains on which uniformity hypotheses can be made are defined by conjunctions of equations $u_1 = v_1\ \&\ldots\&\ u_n = v_n$ where u_i, v_i are terms built from the signature.

In order to implement the selection strategy associated with a uniformity hypothesis on the domain of this conjunction, we need an equational resolution procedure enabling to compute *any* of its solutions. Such a resolution procedure exists for our form of axioms: the conditional narrowing proposed in [11, 17]. Given such a procedure, we can compute any solution of the conjunction of equations defining a uniformity domain. LOFT (LOgic for Functions and Testing) [21] is a tool we designed to provide such a narrowing procedure plus some control mechanisms enabling the automatization of selection strategies.

For the axiom *al3*, we can select as element of the domain of the equation `next_auto_state(`x_0`) = ctrl_imm` the non-ground solution:

```
x0 = cons(true, true, false, ctrl_imm, (0), Train_Abs, Ctrl_Abs, Dir, (0))
```

where `Train_Abs`, `Ctrl_Abs`, `Dir` are variables of the appropriate sorts. This solution corresponds to one of the cases described by the axiom *11* which says that when the train is localized, stopped, in the `ctrl_imm` state, either platform doors or way doors are opened, and either way doors are closed or the train is on a sidetrack segment, its next state is `ctrl_imm`. Such a solution can be computed by narrowing the equation

```
next_auto_state(x0) = ctrl_imm
```

Since we do not want to select always the same solution of a given equation, we made it possible in LOFT to use a "random" strategy for the choice of the axiom to be applied during this narrowing. This enables us to compute the solutions in a non-deterministic order. Thus, the first solution of an equation is not always the same, in opposition to the classical control in logic programming.

In order to select a ground instance of the conclusion of the axiom *al3*, we can make uniformity hypotheses on the variables occurring in x_0. The corresponding selection strategy can be implemented by narrowing a *typing equation* of the form `is_a_state(x0) = true` with the random choice strategy previously described.

The axioms defining such boolean typing operations are automatically generated by LOFT (using the generators of the signature). For example, LOFT generates the following axiom for the typing operation `is_a_state`:

```
is_a_boolean(A) = true & is_a_boolean(B) = true & is_a_boolean(C) = true &
is_a_auto_state(D) = true & is_a_segment(E) = true & is_a_abscissa(F) = true &
is_a_abscissa(G) = true & is_a_direction(H) = true & is_a_speed(I) = true
⇒ is_a_state(cons(A,B,C,D,E,F,G,H,I)) = true
```

where `A, B, ..., I` are variables of the appropriate sorts. The axioms defining the other typing functions are also automatically generated from the signature.

For the axiom *al3*, one possible test selected by LOFT is:

```
al_doors(cons(true,true,false,ctrl_imm,(0),(3),(1), backward,(0))) = true
```

which corresponds to the case where the train is localized with platform side doors opened and way side doors closed; it is in the automaton state `ctrl_imm`; its position is given by the segment `0` at abscissa `3` and control abscissa `1`; it is going forward and its speed is null.

Note that *under the hypotheses* previously made, this test is itself a *valid* and *unbiased* test data set for the axiom *al3*.

3.2.3 Coverage of a validity subdomain

We can remark that the coverage of the cases where the precondition is true is very poor. The above test covers only one of the cases described in the axiom *11*. Since this coverage is unsatisfying, we need weaker hypotheses in order to cover other interesting cases.

Three axioms describe the transitions leading to the state `ctrl_imm`: *01, 11, 31*. In order to cover the validity domain of the precondition of the axiom *al3*, we want to cover each firing condition described in these axioms. We can reach this coverage by applying uniformity hypotheses on the domains of the preconditions of the axioms *01, 11, 31*. Such a decomposition into uniformity subdomains can be done by *unfolding* the defined operations occurring in the domain definition. This unfolding mechanism is well-known in the area of program transformation [8]. In our case, the unfolding mechanism consists in using the axioms to replace equals by equals in order to uncover interesting subdomains.

Let us illustrate this unfolding on the decomposition of the validity domain of *al3*. By unfolding the definition of the operation `next_auto_state` in the precondition, we get the following definitions of three subdomains:

```
01 : auto_state(S) = wait_open_doors & and(closed_pf(S), closed_way(S)) = false
 & on_sidetrack(S) = true & speed(S) = (0)

11 : auto_state(S) = ctrl_imm & and(closed_pf(S), closed_way(S)) = false
 & speed(S) = (0) & located(S) = true & or(closed_way(S), on_sidetrack(S)) = true

31 : auto_state(S) = ctrl_dist_spd
 & closed_pf(S) = false & closed_way(S) = true & located(S) = true
 & lt(delta_x(S), dx_max) = true & in_station(S) = true & speed(S) = (0)
```

If these subdomains seem "too big" to apply uniformity hypotheses, we can go further in the decomposition and unfold e.g. the boolean operations `and`, `or` which are defined by their truth ta-

bles. This leads to much more uniformity subdomains on which we can assume weaker uniformity hypotheses.

Another way to define the previous decomposition strategy is to say that we do no want to unfold the occurrences of the operations different from `next_auto_state`, `and` and `or` in the subdomains obtained by the unfolding of their occurrences.

Since it tries to apply each relevant axiom at each occurrence of a defined operation, the narrowing of the definition of a domain is itself an implementation of decomposition by unfolding. All we need is to enrich narrowing with a way of expressing the conditions under which it should be stopped.

LOFT provides some control specifications allowing the user to define the conditions under which the unfolding of an operation is forbidden. This control allows to specify some elementary uniformity subdomains: such subdomains are defined by forbidding the unfolding of some equations. This is expressed with "do_not_unfold" declarations of the form

`do_not_unfold(f(`t_1 ... t_n`) = t) :- <condition>.`

where `f` is a defined operation and the t_i and `t` are terms built from variables and generators. These declarations affect the resolution procedure during the narrowing of a subgoal containing the definition of the domain to be decomposed. When the procedure chooses an occurrence to be narrowed, it consults the "do_not_unfold" declarations corresponding to the operation at this occurrence. If one of them can be applied at this occurrence, i.e. if the equation argument of the "do_not_unfold" matches the occurrence and `condition` is satisfied, the resolution procedure chooses another occurrence in the subgoal. If all occurrences of defined operations are blocked by "do_not_unfold" declarations, the current subgoal is returned together with the current computed substitution of the variables of the initial goal. This pair *(substitution, not unfoldable subgoal)* defines a uniformity subdomain of the domain of the initial goal according to the elementary uniformity subdomains defined by the "do_not_unfold" declarations.

We will later see some examples of "do_not_unfold" declarations that will be used to implement some decomposition strategies.

3.2.4 LOFT features

LOFT is essentially a parameterized narrowing procedure for our form of structured algebraic specifications. It allows the user to implement selection strategies corresponding to the selection hypotheses described below. Furthermore, LOFT can be used to prototype our form of structured algebraic specifications. Notice also that LOFT provides all the mechanisms useful for the automatization of Jalote's approach [18, 19].

The main parameters of the narrowing procedure can be set by the following commands:

- `constraints(true/false)` allows the narrowing to stop when a subgoal only contains equations matching "do_not_unfold" declarations;
- `random_choice(true/false)` activates a random choice strategy for the choice of the axiom to be applied, instead of choosing the axiom according to the textual order (in the specification);
- `factorize(true/false)` allows to optimize narrowing by sharing different occurrences of the same subterm between equations in a goal;
- `rewrite(true/false)` turns on or off an additional simplification mechanism: the rewriting of subgoals between two steps of narrowing; rewriting uses axioms with a left-to-right orientation; additional rewrite rules can be defined in a special module attached to a specification module;
- `eager(true/false)` turns on or off eager narrowing (each occurrence of defined operation is narrowed); when the mode is not eager, narrowing takes into account axioms that do not contain

in their right hand-side some variables of their left hand-side, in order to discard terms matched by these variables;

- `max_depth( Value )` sets to `Value` the bound for the length of a narrowing proof; each time narrowing reaches this bound a warning is issued to the user and narrowing backtracks to the most recent axiom choice; this bounded narrowing ensures the termination of computations;
- `init-iter( Value )` sets to `Value` the initial bound for the length of a bounded narrowing proof; this bound grows automatically until `max_depth` when there is no complete narrowing proof of the given length; this strategy called *iterative deepening* is complete and more efficient than breadth-first search.

A narrowing request may be composed of the following elements:

- <*conjunction of equations*>: parameterized narrowing of these equations;
- `?()`: first solution of the constraints resulting from the parameterized resolution of the previous part of the request, with a random choice strategy for the axiom to be applied and no stop on constraints;
- `?(`<*conjunction of equations*>`)`: same as above, except that the equations are added to the constraint list;
- `nodisplay`: (at the end of a request) just give the number of solutions and the CPU time used by the narrowing; this is usefull to estimate the accurracy of a decomposition.

Futhermore, LOFT provides other features such as an automatic generator of typing functions (used to implement uniformity hypotheses) and canonical functions of interest for each sort (used for regularity hypotheses[2]), and a partial evaluator that computes a more efficient version of a specification module.

4 Application to the DOORS module

We have already sketched the selection of a test data set for the axiom *al3*. We give here all the hypotheses and strategies used to select a test data set for each of the four axioms defining the operation `al_doors`.

We want to decompose the domain of these axioms in such a way that every possible kind of train behaviour is covered by a test. In particular, we want to cover every transition, leading to an alarm or not. Remark that the axioms defining `next_al_doors` already exhibit the four possible next automaton states. Thus, we want to test all possible transitions leading to these states. Furthermore, since the firing conditions of these transitions cover many interesting cases, we also want to decompose them into smaller subdomains. The corresponding unfolding is described in the following "rules":

- First, unfold all occurrences of `next_auto_state`.
- Then, the introduced occurrences of `and` and `or`, which leads to the coverage of their truth tables; thus, each accessor of a boolean field in a `state` appearing in a boolean expression is decomposed in its two possible values.
- `in_station` is unfolded in the four domains described by its axioms (see section 2).
- The operation `lt` in `lt(speed(S), spd_open_doors)` is unfolded in three domains (lower, equal, greater) corresponding to the axioms defining `lt` on natural numbers. The special case where the speed is equal to the limit seems as interesting (if not more) as the other ones.

[2] See [2] for a description of the regularity hypothesis and its selection strategy.

We do not want to unfold the other operations which may occur. In particular, an inequality on segments or abscissas is not decomposed. This forbids to further decompose the subdomains reached by the unfolding of `in_station` and, in general, helps avoid enumerating abscissas or segments. This is done by use of the "do_not_unfold" declaration:

```
do_not_unfold(lt(A,B) = C).
```

where `A` and `B` are declared of sort `abscissa` and `C` of sort `boolean`.

The operation `delta_x` is not unfolded either, since its definition makes use of the train direction and we assume that the train behaviour does not depend on its direction. So we define a similar "do_not_unfold":

```
do_not_unfold(delta_x(S)=A).
```

Similarly, since `sidetrack_segment` and `station_segment` only enumerate the adequate segment numbers, we do not want to unfold them. This is specified by the following declarations:

```
do_not_unfold(sidetrack_segment(Sg) = B).
do_not_unfold(station_segment(Sg) = B).
```

We now have everything we need to implement the selection strategies. Note that all we said above just gives a means to decompose the validity domain of the axioms *al1* ... *al4* into uniformity subdomains. We still do not know what these subdomains are. They will be defined by the pairs (substitution, constraints) returned by the narrowing of the preconditions of these axioms.

Now let us show some actual examples of decompositions using the system. For the axiom *al3*, the described decomposition strategy is implemented by the narrowing of the following goal:

```
??- next_auto_state(S:state) = ctrl_imm.
```

where "`??-`" is the system prompt. We comment hereafter the result given by the system.

From the axiom *01*, we obtain the following uniformity subdomains:

```
S = cons(_v1:boolean,true,false,wait_open_doors,_v2:segment,
         _v3:abscissa,_v4:abscissa,_v5:direction,(0))
constraints = { sidetrack_segment(_v2) = true }

S = cons(_v1:boolean,false,true,wait_open_doors,_v2:segment,
         _v3:abscissa,_v4:abscissa,_v5:direction,(0))
constraints = { sidetrack_segment(_v2) = true }

S = cons(_v1:boolean,false,false,wait_open_doors,_v2:segment,
         _v3:abscissa,_v4:abscissa,_v5:direction,(0))
constraints = { sidetrack_segment(_v2) = true }
```

These correspond to the three false cases in the truth table of `and`.

From the axiom *11*:

```
S = cons(true,true,false,ctrl_imm,_v1:segment,_v2:abscissa,
         _v3:abscissa,_v4:direction,(0))
constraints = { sidetrack_segment(_v1) = true }

S = cons(true,false,true,ctrl_imm,_v1:segment,_v2:abscissa,
         _v3:abscissa,_v4:direction,(0))
```

```
constraints = { sidetrack_segment(_v1) = false }

S = cons(true,false,true,ctrl_imm,_v1:segment,_v2:abscissa,
         _v3:abscissa,_v4:direction,(0))
constraints = { sidetrack_segment(_v1) = true }

S = cons(true,false,false,ctrl_imm,_v1:segment,_v2:abscissa,
         _v3:abscissa,_v4:direction,(0))
constraints = { sidetrack_segment(_v1) = true }
```

Two of the preconditions of the axiom *11* offer possible decompositions:

```
and(closed_pf(S), closed_way(S)) = false
or(closed_way(S), on_sidetrack(S)) = true
```

If `closed_way(S) = true`, the first one has the only solution `closed_pf(S) = false` and the other one offers both values for `on_sidetrack(S)`. If `closed_way(S) = false`, we symmetrically get both values for `closed_pf(S)` and only one for `on_sidetrack(S)`. These are the four cases above.

Remark that the narrowings involving `on_sidetrack(S)` are blocked by a "do_not_unfold"; thus each equation involving this operation is returned as a constraint.

The last case comes from the axiom *31*. There is no decomposition: the equation `in_station(S) = true` has only one uniformity subdomain and the other defined operations are either deterministic or blocked by "do_not_unfold" declarations.

```
S = cons(true,false,true,ctrl_dist_spd,_v1:segment,(_v2:nat),
         _v3:abscissa,_v4:direction,(0))
 constraints = { delta_x(S) = _v5:abscissa, lt(_v5,(2)) = true,
                 station_segment(_v1) = true, sidetrack_segment(_v1) = false,
                 sub(_v2,33) = _v6:nat, lt((24),(_v6)) = true,
                 sub(_v2,3) = _v7:nat, lt((_v7),(96)) = true }
```

We must now select instances of `S` belonging to the uniformity subdomains defined by the answers above. The corresponding selection strategy is achieved by adding at the end of the previous goal the construction `?()`. This enables to compute the first solution of each set of constraints with `random_choice(true)`.

These instances may contain variables. We make uniformity hypotheses on them. We can thus select any ground instances of them. This is achieved by appending a typing equation `?(is_a_state(S) = true)` to the previous goal. This typing equation will be solved with `random_choice(true)`. All the selection strategies associated to the hypotheses made are implemented through the narrowing of the following goal:

```
??- next_auto_state(S:state) = ctrl_imm,?(),?(is_a_state(S) = true).
```

This returns eight ground instances of `S`, belonging to the previous uniformity subdomains. From these instances, we build the following test data set for *al3*.

```
al_doors(cons(false,true,false,wait_open_doors,(0),(1),(2),backward,(0))) = false
al_doors(cons(false,false,true,wait_open_doors,(0),(0),(1),forward,(0))) = false
al_doors(cons(false,false,false,wait_open_doors,(1),(3),(0),backward,(0))) = false
al_doors(cons(true,true,false,ctrl_imm,(0),(0),(0),forward,(0))) = false
al_doors(cons(true,false,true,ctrl_imm,(9),(1),(1),backward,(0))) = false
al_doors(cons(true,false,true,ctrl_imm,(2),(0),(3),forward,(0))) = false
al_doors(cons(true,false,false,ctrl_imm,(0),(4),(0),backward,(0))) = false
al_doors(cons(true,false,true,ctrl_dist_spd,(7),(60),(0),backward,(0))) = false
```

Using the same unfolding strategy for the axioms *al1*, *al2* and *al3*, we get other uniformity subdomains. These are expressed by conjunctions of elementary uniformity subdomains defined by the "do_not_unfold" declarations. For the sake of brevity, we will not give them here, nor the test data sets they enable to build. The narrowing of the following requests implements the selection strategies used for their selection. The use of the `nodisplay` command allows to show only the number of selected ground instances.

For the axiom *al1*:

```
??- next_auto_state(S:state) = emergency_stop,?(),?(is_a_state(S) = true),nodisplay.
Number of solutions = 230
```

For the axiom *al2*:

```
??- next_auto_state(S:state) = wait_open_doors,?(),?(is_a_state(S) = true),nodisplay.
Number of solutions = 4
```

For the axiom *al3*:

```
??- next_auto_state(S:state) = ctrl_dist_spd,?(),?(is_a_state(S) = true),nodisplay.
Number of solutions = 2
```

The large number of cases selected for the axiom *al1* is essentially due to the unfolding of `next_auto_state` through the axioms *02* and *32*. The boolean operations occurring in their preconditions lead to an important combinatory, as can be seen, for instance, on the axiom *32*:

```
32 : auto_state(S) = ctrl_dist_spd
   & and(lt(delta_x(S), dx_max),
         and(lt(speed(S), spd_open_doors),
             and(in_station(S), and(closed_way(S), located(S))))) = false
 ⇒ next_auto_state(S) = emergency_stop
```

What is tested in these cases is all possible conjunctions of a number of dangerous circumstances, each of them leading to the alarm. These circumstances being *a priori* rather infrequent, it may seem unnecessary to test so much conjunctions, which are still less likely to happen. Depending on the required safety level, less extensive decompositions can be considered sufficient. They can be obtained in two ways. We can add "do_not_unfold" declarations on boolean operations in order to stop the decomposition sooner, but then the reached uniformity subdomains will not tell apart different dangerous circumstances; or we can rewrite the specification a little, in order to test dangerous circumstances separately. For instance, on the axiom *32*, if we wish to single out the situation where the locating system fails, we can write two axioms:

```
32a : auto_state(S) = ctrl_dist_spd & located(S) = false
 ⇒ next_auto_state(S) = emergency_stop

32b : auto_state(S) = ctrl_dist_spd & located(S) = true
      & and(lt(delta_x(S), dx_max),
            and(lt(speed(S), spd_open_doors),
                and(in_station(S), closed_way(S)))) = false
 ⇒ next_auto_state(S) = emergency_stop
```

Thus, we will cover the dangerous situation `located(S) = false` and all conjunctions involving the other conditions. This new expression of the transition *32* leads to half the number of test cases we

got before. We can even push this further and write as many axioms as we wish in order to express the basic unfolding we wish to obtain.

When faced to these test cases, the manufacturer reacted very positively. It gave him some integration testing scenarii which he had not planned to test.

Unfortunately, since the formal specification and the test selection were performed independently of the actual development, it turned out that the minimal hypotheses on the program did not hold (there were some differences of signature). Thus the test sets could not be used as such.

5 Application to the OVERSPEED module

The previous test data selection experiment turned out to give interesting results without too much toil. We will see here a less simple example.

The "do_not_unfold" declarations we introduce are similar to the ones chosen for the DOORS module. In particular, we forbid decompositions of auxiliary operations such as the numerical tables that occur frequently in this module (see subsection 2.2). As a matter of fact, unfolding these operations would lead to the enumeration of the different table entries.

The test of the four operations `target_spd`, `stop_spd`, `imposed_spd` and `way_spd` is straightforward; the operation `target_spd` leads to 40 cases, the operation `stop_spd` 36 cases, the operation `imposed_spd` 18 cases and the operation `way_spd` only one case. We will not give more details on the test data selection on these operations.

But problems occur when we try to test the axiom defining the operation `limspd`. Let us recall that it is defined by the following axiom:

```
limspd(S) = min4(target_spd(S), stop_spd(S), imposed_spd(S), way_spd(S))
```

A natural definition of the operation `min4` is as follows:

```
min4(A, B, C, D) = min(A, min(B, min(C, D)))
```

while the operation `min` can be defined by the following axioms:

```
lt(A, B) = true ⇒ min(A, B) = A ;
lt(A, B) = false ⇒ min (A, B) = B ;
```

With these definitions, if the operation `lt` is not unfolded, the unfolding of the operation `min4` yields the 8 following domains:

- C < D, B < C, A < B : the minimum is A ;
- C < D, B < C, A ≥ B : the minimum is B ;
- C < D, B ≥ C, A < C : the minimum is A ;
- C < D, B ≥ C, A ≥ C : the minimum is C ;
- C ≥ D, B < D, A < B : the minimum is A :
- C ≥ D, B < D, A ≥ B : the minimum is B ;
- C ≥ D, B ≥ D, A < D : the minimum is A ;
- C ≥ D, B ≥ D, A ≥ D : the minimum is D.

Each of the two axioms defining `min` was used for each occurrence of that operation. That decomposition is not the one we would like to obtain. When unfolding `min4`, our wish is to get four domains corresponding to the choice of each of the 4 arguments as the minimum.

This decomposition can be reached by defining the operation `min4` by the axioms:

```
lt(B, A) = false & lt(C, A) = false & lt(D, A) = false ⇒ min4(A, B, C, D) = A ;
lt(A, B) = false & lt(C, B) = false & lt(D, B) = false ⇒ min4(A, B, C, D) = B ;
```

```
lt(A, C) = false & lt(B, C) = false & lt(D, C) = false ⇒ min4(A, B, C, D) = C ;
lt(A, D) = false & lt(B, D) = false & lt(C, D) = false ⇒ min4(A, B, C, D) = D ;
```

However, this is still not sufficient to get a proper decomposition. The four domains we obtain by unfolding the operation `limspd` correspond to the choice of each of the four limiting speeds as the value of `limspd`. For instance, the first of these domains is defined by the following equations:

```
lt(stop_spd(S), target_spd(S)) = false
lt(imposed_spd(S), target_spd(S)) = false
lt(way_spd(S), target_spd(S)) = false
```

But further decompositions will operate, not only on the operation `target_spd`, but also on the three other arguments of `min4`. And since the final value of `limspd` will be `target_spd`, we are definitely not interested in the decompositions induced by the other operations. As a matter of fact, such decompositions would lead, for each of the axioms defining `min4`, to all possible intersections of the domains corresponding to the four limiting speeds. The number of these domains is the product of the numbers of domains for each limiting speed, thus $40 * 36 * 18 = 25920\ldots$

So we must use sharper hypotheses, and different ones for each of the axioms defining the operation `min4`. In the above case, we wish to obtain all the domains generated by the definition of the operation `target_spd`; and for each of these domains, we want the preconditions of the axiom leading to the choice of `target_spd` as minimal speed to be satisfied. But we are not interested in the way in which they are satisfied. Thus, the adequate hypothesis is a *uniformity hypothesis* on the domain defined by these preconditions. The desired testing strategy is thus: first perform decomposition of the operation `target_spd`, then add to the definition of each obtained domain the three equations corresponding to the comparisons with the other speeds, and return one arbitrary term in this domain. The corresponding request is as follows:

```
??- target_spd(S:state) = X, ?(lt(stop_spd(S), X) = false,
    lt(imposed_spd(S), X) = false, lt(way_spd(S), X) = false).
```

As a matter of fact, it can be written in a much more concise way:

```
??- target_spd(S:state) = X, ?(limspd(S) = X).
```

This request clearly shows that in the end, it was not necessary to rewrite the axioms defining `min4` in order to express the desired selection strategy. It can also be seen that constraints expressed through control declarations are a very powerful mechanism, but need some expertise in order to be fully exploited. Remark that in a context where `limspd` would occur in the decomposition, it would be more difficult to express a similar selection strategy for its occurrences.

We see that defining a selection strategy is not always a simple task. Let us insist that up to now, the encountered difficulties were by no means weaknesses of the method; they only resulted from the fact that with the currently implemented selection tool, the "do_not_unfold" declarations are globally defined for the modules. This experiment just revealed a situation where control must be done for particular occurrences during the narrowing.

If we try to identify the origins of the difficulty, we notice that in the axioms of `min`, some variables which are present in the preconditions and the left-hand side of the conclusion vanish in the right-hand side. Since `min` is used with arguments containing defined operations, this explains the uselessness of the decompositions of the instanciations of these variables. Thus, a convenient extension of LOFT would be to allow to mark variables as unfoldable or not when instanciated; the resulting decomposition strategy would be closer to case analysis based on the axioms as presented in section 3. A first step in this direction is the `eager(true/false)` mechanism (cf. subsection 3.2.4).

Conclusion

We have reported an experiment with the use of algebraic specifications and logic programming techniques for assisting black-box testing of an industrial software. We got some interesting feed-back concerning both the method and the tool.

We have got confirmation that the method is applicable to realistic cases; moreover, the results have turned out to be pertinent and in one case, they have led to new testing scenarios which were not planned in the original, intuition-guided, test plan.

As for any black-box testing method, it may be difficult in some case to find out what must be done, given a specification and a program to be tested against it; it is clear that further experiments are needed with this method in order to build an exploitable expertise, based upon both the form of the algebraic specification and the informal testing requirements. Another case study is currently performed [22].

The tool has played an essential role; it must be noted that it provides support for test data selection and, as well, for the determination of the expected results; besides, it makes it possible to compute the amount of testing required for a given strategy (via the use of the nodisplay option), which helps to find, by try and error, the unavoidable compromise between cost and exhaustivity. This tool can be used for different purposes: on the DOORS module, we used it to analyze the specification during its development: precisely, the tool was used to compare and test new versions of the specification module against the previous ones.

The method is not limited to algebraic specifications: a first discussion on its generalization was published in [1]; since then, the similarities we have noticed between our approach and Brinksma's approach for testing distributed systems have given some hints that they could be unified. Of course, the tool requires that the formal specification can be transformed into equational Horn clauses; it turns out that it is the case for some formalisms, for instance CO-OPN, a recent proposal for combining Petri nets with objects [7]; moreover, more general formal approaches, such as VDM or Z, could be considered, provided that the properties are written as equational Horn clauses.

Acknowledgments

This work has been supported by INRETS, by the Programme de Recherches Coordonnées "Programmation Avancée et Outils pour l'Intelligence Artificielle" and by the COMPASS ESPRIT Basic Research Action.

References

[1] G. Bernot, M.-C. Gaudel, and B. Marre. A formal approach to software testing. In *2^{nd} International Conference on Algebraic Methodology and Software Technology, to appear in LNCS*, Iowa City, May 1991. Springer-Verlag.

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

[3] M. Bidoit. *Pluss, un langage pour le développement de spécifications algébriques modulaires.* Thèse d'Etat, Université de Paris Sud, May 1989.

[4] L. Bougé, N. Choquet, L. Fribourg, and M. C. Gaudel. Test sets generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6(4):343–360, November 1986. Also: LRI report $n°240$.

[5] E. Brinksma. A theory for the derivation of tests. In *The Formal Description Technique LOTOS*, P.H.J. van Eijk, C.A. Vissers, and M. Diaz editors, pages 235–248. North-Holland, 1989.

[6] E. Brinksma. A formal approach to testing distributed systems. Technical report, SICS, Box 1263, S-164 28 Kista, Sweden, 1992.

[7] D. Buchs and N. Guelfi. Co-opn : A concurrent object oriented petri net approach. In *12th International Conference on theory and application of Petri Nets*, Aarhus, Denmark, 1991.

[8] R. Burstall and J. Darlington. A transformation system for developping recursive programs. *J.A.C.M.*, 24(1):44–67, 1977.

[9] N. Choquet. Test data generation using a PROLOG with constraints. In *Workshop on Software Testing*, pages 132–141, Banff, Canada, July 1986. IEEE catalog number 86TH0144-6.

[10] P. Dauchy and P. Ozello. Experiments with formal specifications on MAGGALY. In *Applications of Advanced Technologies in Transportation Engineering*, Yorgos J. Stephanedes and Kumares C. Sinha editors. American Society of Civil Engineers, August 1991.

[11] L. Fribourg. SLOG, a logic programming language interpreter based on clausal superposition and rewriting. In *International Symposium on Logic Programming*, Boston, U.S.A., July 1985.

[12] S. Fujiware, G.V. Bochmann, F. Kendhek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[13] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, July 1981.

[14] M.-C. Gaudel. Algebraic specifications. In *Software Engineer's Reference Book*, J. McDermid editor, chapter 22. Butterworth-Heinemann, 1991.

[15] M.-C. Gaudel. Structuring and modularizing algebraic specifications: the PLUSS specification language, evolutions and perspectives. In *9th Symposium on Theoretical Aspects of Computer Science, LNCS n^o 577*, pages 3–23, February 1992.

[16] M.-C. Guillaumin, C. Vernant, and S. Wartski. Spécifications de l'UGE sécuritaire, édition 11. MATRA TRANSPORT, November 1991.

[17] Hussmann. Unification in conditional-equational theories. Technical Report MIP-8502, Univ. Passau, January 1985. Short version in Proc. EUROCAL 85 Conf., Linz.

[18] P. Jalote. Specification and testing of abstract data types. In *Proc. COMPSAC'83, IEEE 7th International Computer Software and Applications Conference*, pages 508–511, November 1983.

[19] P. Jalote and M.G. Caballero. Automated test case generation for data abstractions. In *Proc. COMPSAC'88, IEEE 12th International Computer Software and Applications Conference*, pages 205–210, October 1988.

[20] D. Mandrioli, S. Morasca, and A. Morzenti. Functional test case generation for real-time systems. In *Proc. DCCA-3, 3rd IFIP Working Conference on Dependable Computing for Critical Applications*. Springer-Verlag, September 1992. Also: report n^o 92-008, Politecnico di Milano, Laboratorio di calculatori, Piazza Leonardo da Vinci 32, 20133-Milano, Italy.

[21] B. Marre. *Une méthode et un outil d'assistance à la sélection de jeux de tests à partir de spécifications algébriques*. Nouvelle thèse, Université de Paris Sud, march 1991.

[22] B. Marre, P. Thevenod-Fosse, H. Waeselynck, P. Le Gall, and Y. Crouzet. An experimental evaluation of formal testing and statistical testing. Technical Report 92.219, LAAS, Toulouse, France, June 1992.

[23] S. Morasca and M. Pezze. Using high-level Petri nets for testing concurrent and real-time systems. In *Real-time systems, theory and applications*, H. Zendan editor, pages 119–131. North-Holland, 1990.

[24] D.H. Pitt and D. Freestonei. The derivation of conformance tests from LOTOS specifications. *IEEE Transactions on Software Engineering*, 16(12):1337–1343, December 1990.

[25] E.J. Weyuker. On testing non testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[26] C. Wild. The use of generic constraint logic programming for software testing. In *Proc. 2nd ACM-IEEE Workshop on Software Testing, Verification, and Analysis*, 1988. Also: Comp. Sc. Technical Report Series, no 88-02, Dept of Computer Science, Old Dominion University, Norfolk, VA, February 1988.