Exhaustive test sets for algebraic specifications

Marc Aiguier\(^1\), Agnès Arnould\(^2\), Pascale Le Gall\(^1\) and Delphine Longuet\(^3\)∗

\(^1\) CentraleSupelec, MAS, Châtenay-Malabry, France
\(^2\) Université de Poitiers, XLIM UMR7252, Futuroscope, France
\(^3\) Université Paris-Sud, LRI UMR8623, Orsay, France

SUMMARY

In the context of testing from algebraic specifications, test cases are ground formulas chosen amongst the ground semantic consequences of the specification, according to some possible additional observability conditions. A test set is said to be exhaustive if every program \(P\) passing all the tests is correct and for every incorrect program \(P\), there exists a test case on which \(P\) fails. Since correctness can be proved by testing on such a test set, it is an appropriate basis for the selection of a test set of practical size. The largest candidate test set is the set of observable consequences of the specification. However, depending on the nature of specifications and programs, this set is not necessarily exhaustive. In this paper, we study conditions to ensure the exhaustiveness property of this set for several algebraic formalisms (equational, conditional positive, quantifier-free and with quantifiers) and several test hypotheses. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Specification-based testing; Algebraic specifications; Exhaustiveness; Observability.

1. INTRODUCTION

In the framework of black-box testing, specification-based testing has shown its efficiency to state conformance of programs with respect to their specifications. When specifications are given with a formal text provided with mathematical semantics (i.e. formal specification), both test case generation and evaluation of test executions can be automated [1, 2]. Testing from algebraic specifications, a family of formalisms used to specify programs through the data types they manipulate, has already been extensively studied [1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]. In this context, the basic test hypothesis is to suppose that programs and test cases can respectively be modeled by \(\Sigma\)-algebras and formulas. Hence, the interpretation of test cases is defined by the notion of formula satisfaction. Such formulas link input test data to expected results using the functions of the specification.

∗Correspondence to: Bât. 650, Univ. Paris-Sud 11, 91405 Orsay Cedex, France. E-mail: longuet@lri.fr

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls [Version: 2010/05/13 v2.00]
As the submission of test cases has to yield a verdict, the formulas that represent test cases are all formulas that can be interpreted by a computation of the program as “true” or “false”. These “executable” formulas are called observable and define a subset \( \text{Obs} \) of the whole set of formulas. In the framework of algebraic specifications, observable formulas are ground formulas whose equations are of some given sorts, called observable sorts. A sort is said to be observable if it is equipped with an equality predicate in the programming language used to implement the program under test.

Actually, one of the most widely recognised problems related to testing from algebraic specifications is the so-called oracle problem. This concerns the difficulty of comparing term values of non-observable sort computed by the system under test [10, 15]. In practice, only the sorts of the specification that correspond with built-in types of the program are provided with a reliable decision procedure, hence are considered to be observable. The notion of observable contexts has been introduced to systematically observe non-observable sorts through successive applications of operations leading to a result of observable sort [16, 17, 18]. Therefore, equations of non-observable sort are converted into a family of equations built by surrounding terms that occur in the initial equation with the same observable context. This has been particularly used for object-oriented software testing where, by assumption, object states are encapsulated [7, 14, 19].

Since test cases are defined up to observability issues, the notion of correctness is closely related to observability assumptions. Roughly speaking, correctness is reached when there is a (possibly infinite) test set that verifies that any correct (resp. incorrect) program successfully (resp. unsuccessfully) passes the test. Since any program that satisfies all test cases has to be considered correct, correctness is defined according to an observational approach similar to those used to define specification refinement [20, 21, 22, 23, 24]: a concrete specification is said to be a refinement of an abstract specification if any algebra of the concrete specification is observationally equivalent to an algebra of the abstract specification. Here, by analogy, we say that a program \( P \) is correct with respect to a specification \( SP \) if it is equivalent to an algebra of \( SP \), up to the observable formulas in \( \text{Obs} \) [1, 9]. Let us point out that by default, the model class defined by \( SP \) is not restricted to a unique model (such as the initial one or the terminal one [25]). Thus, we follow a loose semantics approach, i.e. the model class associated to \( SP \) can contain several models, each of them likely to represent a different program. The interest is to be able to accept programs that possibly implement some extra functionalities or properties (contrarily to other approaches considering terminal semantics [14, 19]).

As usual, \( SP^* \) denotes the set of all semantic consequences of \( SP \), that is, all formulas satisfied by all models of \( SP \) (see Sect. 2 for the formal definition). The notion of correctness implies that \( P \) must satisfy all observable semantic consequences of \( SP \), i.e. all formulas of \( SP^* \cap \text{Obs} \). Such set is actually the largest set of test cases which are both satisfied by all \( SP \)-algebras and executable by any tested program able to interpret formulas in \( \text{Obs} \). Thus, \( SP^* \cap \text{Obs} \) is successful on any correct program. This property corresponds to the so-called unbiased property [1]. On the other hand, when this set ensures the correctness of any successful program \( P \), then it is said to be exhaustive. The existence of such an exhaustive test set means that the considered specification \( SP \) is testable via \( \text{Obs} \). With such an exhaustive test set, correct programs cannot be rejected or dually, for any incorrect program, there exists at least a test case whose execution would lead to a failure. Hence, exhaustive
Exhaustive test sets, when they exist, are appropriate to start the process of selecting test sets of reasonable size [3, 26, 27].

Exhaustiveness has been often either circumvented by introducing white-box testing [11, 12], assumed obvious [1] or established for restrictive cases [14]. Besides, when specifications used for testing are not equipped with a loose semantics and characterize only one model, the exhaustiveness of the generated test set always holds [8]. However, following a loose approach to specification, depending on the nature of $SP$, of the set $Obs$ and of programs, $SP^* \cap Obs$ is not necessarily exhaustive.

**Contribution.** In this paper, we investigate the exhaustiveness of $SP^* \cap Obs$ with respect to the nature of $SP$, the set $Obs$ and testing hypotheses on programs. Four main families of algebraic specifications have been widely studied by the community: equational, conditional, quantifier-free and general specifications. Such families differ from one another with respect to the form of their axioms, which are, respectively equational, conditional, quantifier-free and first-order formulas. We study the conditions on specifications and programs under which the exhaustiveness of $SP^* \cap Obs$ is ensured. We show that the conditions on specifications are sensible because they are syntactical and can be checked automatically. On the contrary, conditions on programs are semantic, and so, are often difficult or even impossible to check. Moreover, we show that the form of tests has a strong influence on the conditions on programs.

In this paper, we identify the right sets, namely exhaustive sets, from which it is reasonable to test. These sets are thus simply identified without being operated to make test case selection. We then propose some results in connection with the above four families of algebraic specifications and with some additional hypotheses either on signatures (with constructors or not, with non-observable sorts or not, etc.) or on programs. In most cases, exhaustive sets are infinite, and then cannot be directly used as test cases sets. The exhaustiveness property only ensures the relevance of sets from which the selection of test sets of reasonable size can be initiated. Hence, this property outlines the upper bound of test effectiveness we can achieve. Therefore, this paper is theoretical in nature, and does not investigate the question of test case selection.

In practice, although algebraic specifications are little used in industry, several works report on case studies dealing with testing from algebraic specifications (such as testing object-oriented programs or test case selection tools [28, 29, 30]). Mostly, these works are based on equational or conditional specifications for which the initial test set is by construction exhaustive. Of course, if such an exhaustive test set does not exist, one can still select tests from a randomly generated large suite of tests. But in this case, whatever the considered selection process, some defaults will be missed.

**Structure of the paper.** The paper is organized as follows. In Section 2, we recall basic definitions and notations about algebraic specifications. Section 3 gives the main definitions of the formal testing framework over which our work is built. In Section 4, we start by studying the exhaustiveness result for specifications whose axioms are simple equations and test cases are ground equations of some observable sorts. In Section 5, we extend the exhaustiveness property to conditional and quantifier-free first-order specifications. We show that to ensure the exhaustiveness of $SP^* \cap Obs$, a strong condition has to be imposed on programs: the initiality condition. In Section 6, we then
study two ways to weaken and to remove this condition on programs. In Section 7, we consider the largest class of specifications (i.e. specifications whose axioms can be any first-order formulas with quantifiers) while test cases are again ground first-order formulas.

2. PRELIMINARIES

We recall here the basic definitions and notations for algebraic specifications. Such specifications are widely used for describing so called standard programs, i.e., programs that manipulate complex data types with simple control structures [9, 31, 32, 33].

**Syntax.** An (algebraic) signature \( \Sigma = (S, F) \) consists of a set \( S \) of sorts and a set \( F \) of function names each one equipped with an arity in \( S^* \times S \). In the sequel, a function \( f \) with the arity \((s_1 \ldots s_n, s)\) is denoted by \( f : s_1 \times \ldots \times s_n \rightarrow s \).

Given a signature \( \Sigma = (S, F) \) and an \( S \)-indexed set of variables \( V = (V_s)_{s \in S} \), \( T_\Sigma(V) = (T_\Sigma(V)_s)_{s \in S} \) is the \( S \)-indexed set of terms with variables in \( V \), freely generated from variables and functions in \( \Sigma \). \( T_\Sigma = (T_\Sigma(s))_{s \in S} \) denotes the \( S \)-indexed set \( T_\Sigma(\emptyset) \) of ground terms.

A signature \( \Sigma \) is said sensible for a sort \( s \in S \) if \( T_{\Sigma_s} \) is not empty.\(^1\) A substitution is a family of mappings \( \rho = \{ \rho_s : V_s \rightarrow T_{\Sigma_s}(V)_s \}_{s \in S} \). A substitution is said ground when its co-domain is restricted to ground terms. Substitutions are canonically extended to terms with variables.

\( \Sigma \)-equations are formulas of the form \( t = t' \) with \( t, t' \in T_\Sigma(V)_s \) for \( s \in S \). A \( \Sigma \)-formula is a first-order formula built on \( \Sigma \)-equations, connectives \( \neg, \land, \lor, \Rightarrow \), and quantifiers \( \forall \) and \( \exists \). \text{For}(\Sigma) \) is the set of all \( \Sigma \)-formulas. A quantifier-free \( \Sigma \)-formula is a \( \Sigma \)-formula without quantifiers; variables of quantifier-free formulas are implicitly universally quantified. A conditional \( \Sigma \)-formula is a \( \Sigma \)-formula of the form \( \alpha_1 \land \ldots \land \alpha_n \Rightarrow \alpha_{n+1} \) where each \( \alpha_i \) is a \( \Sigma \)-equation (\( 1 \leq i \leq n + 1 \)). For the particular case \( n = 0 \), the formula is called unconditional equation.

A specification \( SP = (\Sigma, Ax) \) consists of a signature \( \Sigma \) and a set \text{Ax} of \( \Sigma \)-formulas called axioms. \( SP \) is said equational (resp. conditional, quantifier-free, first-order) if all axioms of \( SP \) are \( \Sigma \)-equations (resp. conditional, quantifier-free, first-order \( \Sigma \)-formulas).

**Semantics.** A \( \Sigma \)-algebra \( A \) is an \( S \)-indexed set \( A \) equipped for each \( f : s_1 \times \ldots \times s_n \rightarrow s \in F \) with a mapping \( f^A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s \). A \( \Sigma \)-morphism \( \mu \) from a \( \Sigma \)-algebra \( A \) to a \( \Sigma \)-algebra \( B \) is an \( S \)-indexed family of mappings \( \{ \mu_s : A_s \rightarrow B_s \}_{s \in S} \) such that for all \( f : s_1 \times \ldots \times s_n \rightarrow s \in F \) and all \( (a_1, \ldots, a_n) \in A_{s_1} \times \ldots \times A_{s_n} \), \( \mu_s(f^A(a_1, \ldots, a_n)) = f^B(\mu_{s_1}(a_1), \ldots, \mu_{s_n}(a_n)) \). \text{Alg}(\Sigma) \) is the category whose objects and morphisms are all \( \Sigma \)-algebras and all \( \Sigma \)-morphisms.

Let us note \( T_\Sigma \) the \( \Sigma \)-algebra of ground terms, where the \( S \)-indexed set is \( T_{\Sigma_s} \), equipped for each function \( f : s_1 \times \ldots \times s_n \rightarrow s \) with the mapping \( f^{T_\Sigma} : (t_1, \ldots, t_n) \mapsto f(t_1, \ldots, t_n) \). Given a \( \Sigma \)-algebra \( A \), we denote by \( \mu^A : T_\Sigma \rightarrow A \) the unique \( \Sigma \)-morphism that maps any \( f(t_1, \ldots, t_n) \) to \( f^A(t_1^A, \ldots, t_n^A) \). A \( \Sigma \)-algebra \( A \) is said reachable if \( \mu^A \) is surjective. \text{Gen}(\Sigma) \) is the full subcategory of \( \text{Alg}(\Sigma) \) whose objects are reachable \( \Sigma \)-algebras. Moreover, for any congruence \( \sim \) (i.e. any

\(^1\)A sufficient condition to ensure that a signature \( \Sigma \) is sensible for a sort \( s \in S \) is that \( \Sigma \) must contain at least a constant \( c : \rightarrow s \).
equivalence relation compatible with sorts and functions) over \( T_\Sigma \), \( T_\Sigma/\sim \) is the \( \Sigma \)-algebra defined for each \( s \in S \) by \( (T_\Sigma/\sim)_s = (T_\Sigma)_s/\sim \), and for each \( f : s_1 \times \ldots \times s_n \to s \in F \) and each \( t_1 \in (T_\Sigma)_s \), \( \ldots \), \( t_n \in (T_\Sigma)_s \) by \( f^{T_\Sigma/\sim}([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)] \) (where \([t] \) denotes the equivalence class of \( t \) for \( \sim \)).

Given a \( \Sigma \)-algebra \( A \), a \( \Sigma \)-valuation in \( A \) is a family of mappings \( \iota = \{ t_s : V_s \to A_s \}_{s \in S} \), that canonically extends to terms with variables. For a \( \Sigma \)-equation \( t = t' \), \( A \) satisfies \( t = t' \) for \( \iota \), denoted as \( A \models \iota, t = t' \), if \( \iota(t) = \iota(t') \) and \( A \) satisfies \( t = t' \), denoted as \( A \models t = t' \), if for every \( \Sigma \)-valuation \( \iota \) in \( A \), \( A \models \iota, t = t' \). The satisfaction of a \( \Sigma \)-formula \( \varphi \) by \( A \), denoted by \( A \models \varphi \), is inductively defined on the structure of \( \varphi \) from the satisfaction of \( \Sigma \)-equations of \( \varphi \) using the classical semantic interpretations of connectives and quantifiers.

Given \( \Psi \subseteq \text{For}(\Sigma) \) and two \( \Sigma \)-algebras \( A \) and \( B \), \( A \) is \( \Psi \)-equivalent to \( B \), denoted by \( A \equiv_\Psi B \), if and only if we have: \( \forall \varphi \in \Psi, A \models \varphi \iff B \models \varphi \).

Given a specification \( SP = (\Sigma, \text{Ax}) \), a \( \Sigma \)-algebra \( A \) is an \( SP \)-algebra if for every \( \varphi \in \text{Ax}, A \models \varphi \). \( \text{Alg}(SP) \) is the full subcategory of \( \text{Alg}(\Sigma) \) whose objects are \( SP \)-algebras. A \( \Sigma \)-formula \( \varphi \) is a semantic consequence of a specification \( SP = (\Sigma, \text{Ax}) \), denoted by \( SP \models \varphi \), if for every \( SP \)-algebra \( A \), we have \( A \models \varphi \). We denote by \( SP^* \) the set of semantic consequences of \( SP \).

All examples of specifications presented in this paper are developed using the Common Algebraic Specification Language CASL [34]. CASL is a general-purpose specification language which subsumes many existing specifications languages since it supports predicates, partial functions, subsorting, but also numerous specification libraries. It allows to specify software both constructively and more abstractly and to scale up by structuring specifications thanks to integration operators such as union, enrichment and renaming [35]. It is often recommended to use predicates instead of Boolean operations specified with equations. However, in the sequel, we systematically use Boolean operations over the Boolean sort \( \text{Bool} \) provided with the two usual constants \( \text{True} \) and \( \text{False} \) in order to directly comply with the framework of equation-based algebraic specifications.

3. TESTING FROM FORMAL SPECIFICATIONS

In this section, we give the main definitions of the formal testing framework over which our work is built [1, 9, 10], independently from the form of observable formulas.

3.1. Program correctness

Let \( SP \) be a specification built over a signature \( \Sigma \). Let \( Obs \subseteq \text{For}(\Sigma) \) be a set of observable formulas. As already said in the introduction, \( Obs \) is the set of all formulas eligible as test cases. Let \( P \) be a program defined as a \( \Sigma \)-algebra, i.e., \( P \) implements both sorts and functions of the specification.

The success of the submission of a test case \( \varphi \in Obs \) to \( P \) is then defined in terms of formula satisfaction.

**Definition 3.1** (Test case and test set)

A test case for \( SP \) is a formula \( \varphi \) in \( Obs \). If \( P \models \varphi \) (resp. \( P \not\models \varphi \)), we say that \( P \) passes on \( \varphi \) (resp. fails on \( \varphi \)) or equivalently that the submission of \( \varphi \) to \( P \) is a success (resp. a failure).

A test set \( T \) is a set of test cases. \( P \) passes \( T \) if \( \forall \varphi \in T, P \models \varphi \).
Following an observational approach [36], a system will be considered as a correct implementation of its specification if, as a model, it cannot be distinguished from a model of the specification. Since the program can only be observed through the observable formulas it satisfies, it is required to be equivalent to a model of the specification up to these observability restrictions.

**Definition 3.2 (Correctness)**
Let \( P \) be a \( \Sigma \)-algebra. \( P \) is correct for \( SP \) via \( \text{Obs} \), denoted by \( \text{Correct}_{\text{Obs}}(P, SP) \), if and only if there exists \( A \in \text{Alg}(SP) \) such that \( A \equiv_{\text{obs}} P \).

As we follow a loose semantics approach, i.e. \( \text{Alg}(SP) \) may contain several models, and so, several different programs can be correct with respect to \( SP \).

### 3.2. Exhaustive test sets

It is now possible to link the correctness of a system to the success of the test case submission. The first property requires that a test set does not reject correct systems. A test set that satisfies this property is called unbiased. Thus, if a system fails on an unbiased test set, it is proved to be incorrect. By definition, \( SP^* \cap \text{Obs} \) is the largest unbiased test set since a correct implementation should satisfy any such formula. Conversely, if a test set rejects any incorrect system (and perhaps correct ones), it is called valid. Then if a system passes a valid test set, it is necessarily correct.

Therefore, an ideal test set must have both the unbiased and validity properties. The success of the submission of this test set would actually prove the correctness of the system. According to the classical terminology [1, 4, 9, 10], such a test set is called exhaustive.

As we will see in Section 5, the existence of an exhaustive test set may depend on some hypotheses on the program under test. The notion of exhaustiveness is defined up to a model class, denoted generically by \( K \). In the most general case, \( K \) is simply the whole class \( \text{Alg}(\Sigma) \) when no additional test hypotheses are made on programs, except that it can be modelled by a \( \Sigma \)-algebra. In practice, depending on the knowledge one has of the program under test, some assumptions can be made. These conditions correspond to the notion of program hypotheses as introduced by Bernot et al. [1, 15]. The set \( K \) precisely abstracts all the system hypotheses.

**Definition 3.3 (Exhaustiveness)**
Let \( K \subseteq \text{Alg}(\Sigma) \) be a subcategory of algebras. A test set \( T \) is exhaustive for \( K \) with respect to \( SP \) and \( \text{Obs} \) if and only if:

\[
\forall P \in K, \ P \models T \iff \text{Correct}_{\text{Obs}}(P, SP)
\]

Formulas can be removed from an exhaustive test set \( T \), provided that they are redundant to other formulas of \( T \) or that they are implied by the set \( K \).

**Definition 3.4 (Equivalent test sets)**
Two test sets \( T \) and \( T' \) are equivalent with respect to \( K \), if and only if:

\[
\forall P \in K, \ P \models T \iff P \models T'
\]
In particular, if $T$ and $T'$ are such that $T \subseteq T'$ and $(\Sigma, T)^\bullet = (\Sigma, T')^\bullet$, then in practice, $T$ is preferred to $T'$. For example, this is the case if $T' \setminus T$ contains tautologies or more generally, if $T' \setminus T$ contains semantic consequences of $T$, i.e. $T' \setminus T \subseteq (\Sigma, T)^\bullet$.

Clearly, if two sets are equivalent with respect to $K$ and one of them is exhaustive for $K$, then this is also the case for the other one. Thus, tautologies can be removed from a test set without altering its error detection power.

In particular, if there exists an exhaustive test set $T$ for $K$ with respect to $SP$ and $Obs$, then $Obs \cap SP^\bullet$ is also an exhaustive test set, since it is by construction the largest unbiased test set that can be considered. Conversely, sometimes, there does not exist an exhaustive test set: in particular, $Obs \cap SP^\bullet$ is not exhaustive. Indeed, as we will see in the following, the existence of an exhaustive test set depends on some conditions on programs (and specifications). First, it depends on the observability of the program, i.e. on the set $Obs$ of observable formulas.

**Example 3.5**

Let us consider the following specification: it defines a sort $Elem$ with three constants $a$, $b$ and $c$, and it requires that either $a$ equals $b$ or $a$ equals $c$.

```plaintext
spec ABC =
  type Elem ::= a | b | c
  • a = b ▷ a = c
end
```

This specification has three different models: a model where $a = b$ but $a \neq c$; a model where $a = c$ but $a \neq b$ and a model where $a = b = c$. We consider only equations on sort $Elem$ to be observable, i.e. $Obs$ is the set of ground equations on sort $Elem$. To satisfy this specification up to this observability restriction, an implementation has to satisfy the same observable properties as one of these three models. Therefore, implementations where either $a = b$ or $a = c$ holds are different correct implementations of the same specification, and they do not share any property (except tautologies). However, neither $a = b$ nor $a = c$ are possible tests: an implementation where $a = c$ and $a \neq b$ will be rejected by the test $a = b$ and conversely, an implementation where $a = b$ and $a \neq c$ will be rejected by the test $a = c$, even if those two implementations are correct under this observability restriction. In fact, the set of observable equations which are consequences of the specification is $\{a = a, b = b, c = c\}$, so no exhaustive test set exists for this specification under these observability conditions. Of course, if $Obs$ was containing all ground formulas, then $Obs \cap SP^\bullet$ would contain $a = b ▷ a = c$ and would be exhaustive for $K = Alg(\Sigma)$.

3.3. **Observability issues**

Considering algebraic specifications, the set $Obs$ of observable formulas must verify some constraints so that its formulas can be submitted to the program under test. As explained in the introduction, most selection methods restrict to ground equations on some given sorts, called

---

‡ A formula $\varphi$ is a tautology iff $(\Sigma, \emptyset) \models \varphi$

§ The type construct in CASL is used as an abbreviation for the declaration of a sort with constructors. It does not imply any constraint on the values of the declared sort.
observable sorts. By testing hypothesis, the observable sorts are the sorts equipped with a reliable decision procedure. An observable sort may be provided by the programming language [1] or, by extension, defined by the user but extensively tested [6, 7]. Let us consider a subset of observable sorts \( S_{\text{obs}} \subseteq S \) and the following associated set of observable equations:

\[
\text{Obs}_{\text{Eq}}(S_{\text{obs}}) = \{ t = t' \mid \exists s \in S_{\text{obs}}, t, t' \in T_{\Sigma_s} \}
\]

In the sequel, we will adopt the following convention: the notation \( \text{Obs} \) in the expression \( \text{Obs}(S_{\text{obs}}) \) will be subscripted by an abbreviation designating the targeted set of observable formulas built over \( S_{\text{obs}} \). Thus, \( \text{Obs}_{\text{Eq}}(S_{\text{obs}}) \) indicates that observational formulas are equational.

Because test cases are only ground equations, a first condition requires that the set of ground terms is not empty. This can be easily obtained by imposing that specification signatures are sensible for any observable sort \( s \in S_{\text{obs}} \), i.e. there exists at least one ground term of sort \( s \). Since this property does not depend on the structure of the specification axioms, we will always suppose in the sequel of the paper that every specification signature is sensible for all observable sorts.

**Observable contexts.** A well-known way to circumvent the lack of an equality decision procedure is to replace an equality by a finite set of equalities obtained by context applications, provided that the resulting sort is an observable sort of the program \( P \) [36]. Thus, instead of directly considering a test of the form \( t = t' \) with \( t \) and \( t' \) of sort \( s \), one considers a set of tests of the form \( c[t] = c[t'] \) with \( c \) a context applicable to terms of sort \( s \) and yielding an observable sort.

Observable contexts are terms provided with a unique occurrence of a variable \( \Box \). Such contexts capture the testing practice which consists in applying to both sides of an equation the same observable context. Formally, they are defined as follows:

**Definition 3.6 (Contexts and observable contexts)**

Let \( \Sigma = (S, F) \) be a signature equipped with a subset \( S_{\text{obs}} \subseteq S \). Let us define the set of variables \( \Box \) by: \( \Box = \{ \{ \Box_s \} \}_{s \in S} \).

A \( \Sigma \)-context \( c \) is a term in \( T_{\Sigma_\Box} \) for \( s \in S \) with exactly one occurrence of the variable \( \Box_{s'} \) in \( \Box \) and this is the only symbol of \( \Box \) occurring in \( c \). The context \( c \) is then called of sort \( s' \), denoted by \( c : s' \). The application of a context \( c : s' \) to a term \( t \in T_{\Sigma(V)_{s'}} \), denoted by \( c[t] \), is the term obtained by substituting the term \( t \) for \( \Box_{s'} \).

A context \( c : s' \in T_{\Sigma_\Box} \) is an observable context if \( s' \in S \setminus S_{\text{obs}} \) and \( s \in S_{\text{obs}} \).

We can restrict ourselves to minimal observable contexts in order to deal with non-observable equations. An observable context is said to be minimal if it does not contain an observable context as strict subterm. If an observable context \( c \) has an observable context \( c' \) as strict subterm, then \( c[z] \) may be decomposed as \( c_0 c'[z] \) where \( c' \) is a context. This implies that for any terms \( t \) and \( t' \), for any \( \Sigma \)-algebra \( A \), \( A \models c[t] = c[t'] \) if \( A \models c'[t] = c'[t'] \). Both equalities being observable, the simplest one, \( c'[t] = c'[t'] \), suffices to infer whether \( c[t] = c[t'] \) holds or not. In the sequel, all the observable contexts will be considered to be minimal by default. Let us denote by \( \mu \text{Ctx} \) the set of all minimal observable contexts \( c : s \) defined on non-observable sorts in \( S \setminus S_{\text{obs}} \).
Example 3.7
Let us consider the two specifications of lists of natural numbers below. We assume that only natural numbers and Booleans are observable and lists are not.

**spec LISTOBservers1 =**
```plaintext
types Nat := 0 | s(Nat);
   List[Nat] ::= [] | _ :: _(Nat; List[Nat])
ops  head : List[Nat] → Nat;
     tail : List[Nat] → List[Nat]
∀x : Nat; L : List[Nat]
• head(x :: L) = x
• tail(x :: L) = L
end
```

**spec LISTOBservers2 =**
```plaintext
types Bool ::= True | False;
   Nat ::= 0 | s(Nat);
   List[Nat] ::= [] | _ :: _(Nat; List[Nat])
ops  isin : Nat × List[Nat] → Bool
∀x, y : Nat; L : List[Nat]
• isin(x, []) = False
• isin(x, x :: L) = True
• isin(x, L) = True ⇒ isin(x, y :: L) = True
end
```

Considering LISTOBservers1, an observable context for a term of sort List can be for instance head( ), head(tail(tail( ))), head(head(x :: [])) :: tail( ) or head(tail(m :: tail( ))), where x and m are ground terms of sort Nat. Minimal observable contexts are all terms of the form head(t) where t is a context of sort List that does not contain a strict sub-context with the operation head as top operation, i.e. a strict sub-context of the form head( . . ).

A list can also be observed through its elements, as in the specification LISTOBservers2. Observable contexts in this case can be for instance isin(n, ) or isin(n, x :: y :: ) where n, x and y are ground terms of sort Nat. Minimal observable contexts are all the terms of the form isin(x, t), where x is a ground term of sort Nat and t is a context of sort List.

3.4. Complete specifications

According to the form of the specification, the set \( SP^* \cap \text{Obs}_{Eq}(S_{obs}) \) can be too small to reasonably test programs from this specification.

Example 3.8
Let us consider the following minimal specification of lists, equipped with an operation reverse which reverses the order of the elements of a list. A very abstract way to specify this operation is as follows.

**spec REVERSE =**
```plaintext
types Nat := 0 | s(Nat);
   List[Nat] ::= [] | _ :: _(Nat; List[Nat])
ops  reverse : List[Nat] → List[Nat]
∀L : List[Elem]
• reverse([]) = []
• reverse(reverse(L)) = L
end
```

The only consequences of this specification are reverse([]) = [] and ground instances of reverse\(^2n\)(L) = L for \( n \geq 1 \). It means that any program that implements reverse with a function \( f \) such that \( f([]) = [] \) and \( f(f(L)) = L \) passes all tests in \( SP^* \cap \text{Obs}_{Eq}(S_{obs}) \). For example, the program that implements reverse with a function that only exchanges the first and the last elements of a list would be considered as correct. Therefore, this specification of reverse is too abstract to
lead to relevant test cases if the tester wants to target the usual \texttt{reverse} function that exchanges all elements, two by two, symmetrically positioned with respect to the middle of the list. A solution to this problem is to inductively specify \texttt{reverse} over the sort \texttt{List[Nat]}, thanks to its constructors \([\text{[]}\) and \([::]\) (see Example 3.10 for a constructor-based specification of the \texttt{reverse} function).

In fact, it is acknowledged that the reference test set that best reflects the practice of testing relies on a subset of constructors [1, 9]. Given a signature \(\Sigma = (S, F)\), its constructors define a subset \(C \subseteq F\). We denote by \(\Omega\) the signature \((S, C)\), and we consider the test set \(SP^* \cap \text{Obs}_\Omega\) where

\[
\text{Obs}_\Omega = \{ f(u_1, \ldots, u_n) = v \mid f \in F \land u_1, \ldots, u_n, v \in T_\Omega \}
\]

This set is well-suited for testing, provided that the constructors in \(\Omega\) allow denoting all data values. Specifications that satisfy such a condition are called complete.

**Definition 3.9 (Completeness)**
Let \(SP = (\Sigma, Ax)\) be a specification where \(\Sigma = (S, F)\) is a signature with constructors in \(\Omega = (S, C)\). \(SP\) is complete with respect to constructors, for short complete, if and only if:

\[
\forall t \in T_\Sigma, \exists v \in T_\Omega, SP \models t = v
\]

This condition, although more complicated than the condition of sensible specifications, can also be automatically checked sometimes when specifications have some special properties such as constructive specifications [37]. Moreover, complete specifications are rather easy to write, as shown in the following example. Some systems even allow only constructive specifications like the theorem prover assistant Isabelle [38].

**Example 3.10**
The operation \texttt{reverse} can be completely specified with respect to the constructors of the sort \texttt{List[Nat]}, using the concatenation operation on lists in the following way:

```plaintext
spec REVERSECOMPLETE =
    types Nat ::= 0 | s(Nat);
    List[Nat] ::= [ ] | __::__(Nat; List[Nat])
    ops __@__ : List[Nat] \times List[Nat] \rightarrow List[Nat];
    reverse : List[Nat] \rightarrow List[Nat]
    \forall x : Nat; L, L'; L'' : List[Nat]
    • [ ]@L = L
    • (x :: L}@L' = x :: (L@L')
    • reverse([ ]) = [ ]
    • reverse(x :: L) = reverse(L}@(x :: [ ])
end
```

With a simple structural induction over terms, it is easy to show that \texttt{REVERSECOMPLETE} is complete.
4. GROUND EQUATIONS AS TEST CASES FOR EQUATIONAL SPECIFICATIONS

We first consider specifications where axioms are simple equations. In the presence of non-observable sorts, as we explained in the previous section, non-observable equations will be observed through observable contexts. Therefore, we want to consider the following test set:

\[ Exh_{SP}^{obs} = \{ c[\sigma(t)] = c[\sigma(t')] \mid t = t' \in Ax, \sigma : V \rightarrow T_{\Sigma}, c \in \muCtx \} \]

This set can easily be algorithmically generated, for instance, by using the algorithm proposed by Kong et al. [32].

**Proposition 4.1**

Let \( SP = (\Sigma, Ax) \) be an equational specification. \( Exh_{SP}^{obs} \) is exhaustive for \( K = Alg(\Sigma) \).

**Proof**

Suppose that \( P \models Exh_{SP}^{obs} \). Let us show \( Correct_{ObsEq}(S_{obs}) \) \((P, SP)\) for \( \mathcal{K} = Alg(\Sigma) \). Let \( T_{\Sigma}/\sim_P \) be the quotient of \( T_{\Sigma} \) where \( \sim_P \) is the congruence on \( T_{\Sigma} \) defined for every \( t, t' \in T_{\Sigma} \) by:

\[
  t \sim_P t' \iff \begin{cases} 
    P \models t = t' & \text{if } t = t' \in ObsEq(S_{obs}) \\
    \forall c : s \in \muCtx, P \models c[t] = c[t'] & \text{otherwise}
  \end{cases}
\]

By definition of \( \sim_P \), \( P \equiv_{obsEq(S_{obs})} T_{\Sigma}/\sim_P \). Let us show that \( T_{\Sigma}/\sim_P \in Alg(\mathcal{S}) \). Let \( t = t' \) be an axiom of \( Ax \) and \( \iota : V \rightarrow T_{\Sigma}/\sim_P \) be an interpretation. By structural induction on terms in \( T_{\Sigma} \), we can easily show that there exists a ground substitution \( \sigma : V \rightarrow T_{\Sigma} \) such that \( \iota = q_{\sim_P} \circ \sigma \) where \( q_{\sim_P} : T_{\Sigma} \rightarrow T_{\Sigma}/\sim_P \) is the quotient morphism. Two cases have to be considered:

1. \( s \in S_{obs} \). By definition of \( Exh_{SP}^{obs} \), \( \sigma(t) = \sigma(t') \in Exh_{SP}^{obs} \). By properties \( P \models Exh_{SP}^{obs} \) and \( P \equiv_{obsEq(S_{obs})} T_{\Sigma}/\sim_P \), we have \( T_{\Sigma}/\sim_P \models \sigma(t) = \sigma(t') \) and so \( T_{\Sigma}/\sim_P \models t = t' \).
2. \( s \notin S_{obs} \). By definition of \( Exh_{SP}^{obs} \), for every \( c : s \in \muCtx \), \( c[\sigma(t)] = c[\sigma(t')] \in Exh_{SP}^{obs} \). By property \( P \models Exh_{SP}^{obs} \) and by the definition of \( \sim_P \) for non-observable sort, we have \( \sigma(t) \sim_P \sigma(t') \). Therefore, \( T_{\Sigma}/\sim_P \models \sigma(t) = \sigma(t') \) and for any valuation \( \iota : V \rightarrow T_{\Sigma}/\sim_P \), \( T_{\Sigma}/\sim_P \models \iota t = t' \). That is \( T_{\Sigma}/\sim_P \models t = t' \).

We can conclude that any program that satisfies \( Exh_{SP}^{obs} \) is observationally equivalent to a model of \( SP \) (in this case, \( T_{\Sigma}/\sim_P \)).

Reciprocally, suppose that there exists \( A \in Alg(SP) \) such that \( A \equiv_{obsEq(S_{obs})} P \). Let \( t = t' \in Exh_{SP}^{obs} \). By hypothesis \( A \models t = t' \), then \( P \models t = t' \) as well. \( \square \)

By the inclusion \( Exh_{SP}^{obs} \subseteq SP^* \cap ObsEq(S_{obs}) \), we directly get:

**Corollary 4.2**

Let \( SP = (\Sigma, Ax) \) be a specification whose axioms are equations. \( SP^* \cap ObsEq(S_{obs}) \) is exhaustive for \( Alg(\Sigma) \) with respect to \( SP \) and \( ObsEq(S_{obs}) \).

**With constructors.** As we already saw in Section 3.4, the test set that best reflects testing practice given a specification \( SP \) is \( SP^* \cap ObsEq \). In the presence of constructors, exhaustiveness depends on specification completeness. To simplify the proof of this result, we consider that all sorts are observable (i.e. \( S = S_{obs} \)).
Theorem 4.3
Let \( SP \) be an equational specification, complete with respect to constructors. Then, \( SP^* \cap Obs_{\Omega} \) is exhaustive for \( K = Alg(\Sigma) \).

Proof
Let \( T_{eqconst} = \{ t = v \mid t \in T_\Sigma, v \in T_\Omega, SP \models t = v \} \). Let \( t = t' \in SP^* \cap Obs_{Eq}(S) \) (let us recall that we suppose \( S = S_{obs} \)). By Definition 3.9, there exists \( v \in T_\Omega \) such that \( SP \models t' = v \), and then \( SP \models t = v \) by transitivity. Hence, by definition, \( t' = v, t = v \in T_{eqconst} \). By both symmetry and transitivity, \( T_{eqconst} \models t = t' \).

Let us then show that for all \( t = v \in T_{eqconst} \), \( SP^* \cap Obs_{\Omega} \models t = v \). This is proved by induction on the structure of the term \( t \).

Base case: \( t \) is a constant of \( \Sigma \). Therefore, by Definition 3.9, there exists \( v \in T_\Omega \) such that \( SP \models t = v \), and then \( t = v \in SP^* \cap Obs_{\Omega} \).

Inductive step: \( t \) is of the form \( g(t_1, \ldots, t_n) \). By Definition 3.9, for every \( i, 1 \leq i \leq n \), there exists \( u_i \in T_\Omega \) such that \( SP \models t_i = u_i \) and so \( t_i = u_i \in T_{eqconst} \). Hence, by the induction hypothesis, \( SP^* \cap Obs_{\Omega} \models t_i = u_i \). By context passing, we have \( SP^* \cap Obs_{\Omega} \models g(t_1, \ldots, t_n) = g(u_1, \ldots, u_n) \). Two cases have to be considered:

1. \( g \in C \). Therefore, \( v = g(u_1, \ldots, u_n) \), and then \( SP^* \cap Obs_{\Omega} \models t = v \).
2. \( g \in F \setminus C \). By Definition 3.9, there exists \( v \in T_\Omega \) such that \( SP \models g(u_1, \ldots, u_n) = v \).

Hence, \( g(u_1, \ldots, u_n) = v \in SP^* \cap Obs_{\Omega} \), and then \( SP^* \cap Obs_{\Omega} \models t = v \) by transitivity.

Therefore, \( SP^* \cap Obs_{Eq}(S) \), \( SP^* \cap Obs_{\Omega} \) and \( T_{eqconst} \) are equivalent test sets. By Corollary 4.2, we have shown that \( SP^* \cap Obs_{Eq}(S) \) is exhaustive, so is \( SP^* \cap Obs_{\Omega} \) and \( T_{eqconst} \) as well.

This last result can be easily extended for a subset of observable sorts \( S_{obs} \) is considered. In this case, we consider the set \( Exh_{SP}^{\Omega} \) defined as \( Exh_{SP}^{obs} \) except substitutions \( \sigma \) that are restricted to map into terms in \( T_{\Omega} \).

5. GROUND EQUATIONS AS TEST CASES FOR MORE GENERAL SPECIFICATIONS

Many works have been done to select test cases defined by ground equations from conditional or quantifier-free specifications. From these works came out efficient algorithms and tools to select test case sets, all of them based on axiom unfolding methods[1, 3, 27]. However, as this will be shown in Sections 5.2 and 5.3, exhaustiveness can fail without an additional condition on programs: the initiality condition. The reason is (unlike in Section 4 and like in Example 3.5) that test cases have a more restricted form than specification axioms.

5.1. Initiality condition

To show the interest of the initiality condition, let us take the case of conditional specifications: we want to build test cases as unconditional equations from a specification that consists in conditional axioms. Intuitively, testing a conditional axiom \( a \Rightarrow b \) comes down to ensuring that, in the program,
a never holds when b does not. If a would hold but not b, the program would be incorrect. However, only instances of a that are consequences of the specification can be submitted to the program. Therefore, some instances of a that are satisfied by the program but not by the specification could correspond to instances of b that are not satisfied by the program. The program, although incorrect, would pass the test set of ground equational consequences of the specification, as we can see in the following example.

Example 5.1
Let us consider the specification REVERSECOMPLETE of Example 3.10 (with natural numbers), to which we add a new axiom to specify the property for a list to be a palindrome. This new specification is shown below:

```
spec PALINDROME =
  types Bool ::= True | False;
  Nat ::= 0 | s(Nat);
  List[Nat] ::= [] | x :: List[Nat];
  ops _@_ : List[Nat] × List[Nat] → List[Nat];
  reverse : List[Nat] → List[Nat];
  palindrome : List[Nat] → Bool
  ∀x : Nat; L, L', L'' : List[Nat]
  • []@L = L
  • (x :: L)@L' = x :: (L@L')
  • reverse([]) = []
  • reverse(x :: L) = reverse(L)@(x :: [])
  • reverse(L) = L ⇒ palindrome(L) = True
end
```

Let us suppose a programming environment with lists as a built-in type. In this programming language, a list of n elements x₁, ..., xₙ is encoded by the finite sequence [x₁, ..., xₙ]. Two lists of elements of a same type [x₁, ..., xₙ] and [y₁, ..., yₚ] are equal if, and only if if n = p and for every i, 1 ≤ i ≤ n, xᵢ = yᵢ. In this programming environment, let us suppose two programs P₁ and P₂ that implement all the operations of the specification PALINDROME. P₁ is the program people have usually in mind, i.e. P₁ implements the operations as follows:

- `_[_@_|` puts the element at the head of the list, i.e. `_[_@_|`: (x₁, ..., xₙ) → [x, x₁, ..., xₙ],
- `reverse` reverses the elements of the list, i.e. `reverse`: [x₁, ..., xₙ] → [xₙ, xₙ₋₁, ..., x₂, x₁],
- `_[@_|` concatenates the two lists in argument, i.e. `_[@_|`: ([x₁, ..., xₙ], [y₁, ..., yₚ]) → [x₁, ..., xₙ, y₁, ..., yₚ], and
- `palindrome` checks that the list in argument is a palindrome, i.e.

```
palindromeP₁ : [x₁, ..., xₙ] → \text{True if } \forall i, 1 \leq i \leq n/2, xᵢ = xₙ₋i₊₁ \text{ False otherwise}
```

P₂ is the program that manipulates sorted lists, i.e. we suppose a total order ≤ on the elements of lists and lists are sorted with respect to this order. The operations are then implemented as follows:
• \( \_ :: \_ \) puts the element at the correct position in the list, i.e. \( \_ :: \_ P_2 : (x, [x_1, \ldots, x_n]) \mapsto [x_1, \ldots, x_i, x, x_{i+1}, \ldots, x_n] \) such that \( x_1 \leq x_2 \leq \ldots x_i \leq x \leq x_{i+1} \leq \ldots \leq x_n \),
• \( \text{reverse} \) is the identity, i.e. \( \text{reverse} P_2 : [x_1, \ldots, x_n] \mapsto [x_1, \ldots, x_n] \), since the resulting list has to be sorted and then is in the same order than \( [x_1, \ldots, x_n] \),
• \( \_@\_ \) merges the two lists with respect to the order, and
• \( \text{palindrome} \) is implemented as in \( P_1 \), i.e. \( \text{palindrome} P_2 : [x_1, \ldots, x_n] \mapsto \begin{cases} \text{True} & \text{if } \forall i, 1 \leq i \leq n/2, x_i = x_{n-i+1} \\ \text{False} & \text{otherwise} \end{cases} \)

Indeed, \( P_1 \) satisfies all the axioms of the specification \( \text{PALINDROME} \). Surprisingly, \( P_2 \) meets the axioms of the specification \( \text{REVERSE} \). The reason is that sorting lists has been delegated to the constructor \( \_ :: \_ \). Unlike \( P_1 \), the program \( P_2 \) does not satisfy the last axiom of the specification \( \text{PALINDROME} \). Indeed, \( \text{reverse}(L) = L \) holds for all lists, in particular for lists which are not palindromes. Hence, the program \( P_2 \) is incorrect (it does not satisfy the conditional axiom). However, it passes all the tests in \( SP^* \cap \text{Obs} \). The reason is the only equations \( \text{reverse}(L) = L \) that are satisfied by all specification models are those satisfied by the initial one. Hence, \( \text{reverse}(L) = L \) is a consequence of the specification only for the lists \( L \) that are palindromes, and then the only tests for \( \text{palindrome} \) are \( \text{palindrome}(L) = \text{True} \) (when \( L \) is a palindrome).

A solution to this problem is to impose that the program under test does not satisfy more instances of axiom premises than the specification does: we say that the program is initial on these equations. Roughly speaking, this means that, on these equations, the program behaves like the initial algebra (and so like the specification).

**Definition 5.2 (Initiality)**

Let \( SP = (\Sigma, Ax) \) be a specification where \( \Sigma = (S, F) \) and \( P \in \text{Alg}(\Sigma) \) be a program. Let \( t = t' \) be a ground \( \Sigma \)-equation. \( P \) is initial on \( t = t' \) for \( SP \) if, and only if we have:

\[
P \models t = t' \iff SP \models t = t'
\]

In the case of conditional specifications, the exhaustiveness of \( SP^* \cap \text{Obs}_{Eq}(S_{obs}) \) relies on the condition of initiality, that will be imposed on programs on the ground instances of the axiom premises of the specification. Imposing initiality prevents a program from satisfying all the premises of an axiom without satisfying its conclusion, while this conclusion is not a consequence of the specification. We will see in Section 5.3 how this condition is generalised in the case of quantifier-free first-order specifications.

In Sections 5.2 and 5.3, the exhaustiveness results need this condition. However, this condition is often difficult or impossible to check because of its semantic nature. This is why we will see in Section 6 how in some situations, this condition of initiality can be relaxed.

### 5.2. Conditional specifications

Conditional equations are the most used and studied specification formalism in the framework of testing from algebraic specifications. As already explained, the test selection strategies such as the axiom unfolding method have been introduced to guide test selection from these specifications [1,
3, 9, 13]. Given a specification \( SP \) and an equation \( f(x_1, \ldots, x_n) = y \) where each \( x_i \) and \( y \) are variables, most of these selection methods consist in making a partition of the set of ground substitutions \( \sigma \) such that \( SP \models f(\sigma(x_1), \ldots, \sigma(x_n)) = \sigma(y) \). This partition is computed by deriving constraints on \( \sigma \) from the axioms of the specification, which is natural with axioms as conditional equations. The resulting equations become the test cases which will be submitted to the program under test.

As explained above, the exhaustiveness of \( SP^* \cap Obs_{Eq}(S_{obs}) \) is not straightforward and requires the initiality condition on all the ground instances of the axiom premises in \( SP \).

**Theorem 5.3**

Let \( SP = (\Sigma, Ax) \) be a conditional specification and \( K \) be the class of programs \( P \) which are initial on all the ground instances of any \( \Sigma \)-equation occurring in the axiom premises in \( Ax \). Then, \( SP^* \cap Obs_{Eq}(S_{obs}) \) is exhaustive for \( K \).

**Proof**

Let \( P \) be a program in \( K \) (as defined in the theorem) such that \( P \models SP^* \cap Obs_{Eq}(S_{obs}) \).

Let us show that \( Correct_{Observe}(S_{obs})(P, SP) \). Let us consider the following congruence defined on \( T_\Sigma \):

\[
\begin{align*}
\text{if } t = t' & \implies \{ P \models t = t' \} \\
& \forall \nu : s \in \mu Ctx, P \models c[t] = c[t'] \text{ and } SP \models t = t' \text{ otherwise}
\end{align*}
\]

By construction, \( P \models \text{Observe}(S_{obs}) T_\Sigma/\sim_P \). Let us show that \( T_\Sigma/\sim_P \) belongs to \( Alg(SP) \). Let \( \varphi = t_1 = t_1' \land \ldots \land t_n = t_n' \Rightarrow t = t' \) be an axiom of \( Ax \). Let \( \iota : V \rightarrow T_\Sigma/\sim_P \) be an interpretation such that \( T_\Sigma/\sim_P \models t_i = t_i' \) for every \( i = 1, \ldots, n \). We have already stated that there exists a ground substitution \( \sigma : V \rightarrow T_\Sigma \) such that \( \iota = q_{\sim_P} \circ \sigma \) where \( q_{\sim_P} : T_\Sigma \rightarrow T_\Sigma/\sim_P \) is the quotient morphism. Then, let us show that \( SP \models \sigma(t_i) = \sigma(t_i') \). Two cases have to be considered:

- \( t_i \) and \( t_i' \) are of observable sort. Therefore, by definition of \( \sim_P \), \( T_\Sigma/\sim_P \models \sigma(t_i) = \sigma(t_i') \) implies that \( P \models \sigma(t_i) = \sigma(t_i') \). As \( P \) is initial on all ground instances of the \( \Sigma \)-equations which occur in premises of axioms in \( Ax \), we can conclude \( SP \models \sigma(t_i) = \sigma(t_i') \).
- \( t_i \) and \( t_i' \) are not of observable sort. By definition of \( \sim_P \), we directly have that \( SP \models \sigma(t_i) = \sigma(t_i') \).

Thus from the axiom \( t_1 = t_1' \land \ldots \land t_n = t_n' \Rightarrow t = t' \), and from \( SP \models \sigma(t_i) = \sigma(t_i') \) for all \( i \) in \( 1..n \), we get \( \sigma(t) = \sigma(t') \in SP^* \).

- if \( t \) and \( t' \) are of observable sort, then \( \sigma(t) = \sigma(t') \) belongs to \( SP^* \cap Obs_{Eq}(S_{obs}) \), and \( P \models \sigma(t) = \sigma(t') \), that is \( T_\Sigma/\sim_P \models \sigma(t) = \sigma(t') \).
- if \( t \) and \( t' \) are not of observable sort, then for all contexts \( c : s \in \mu Ctx, c[\sigma(t)] = c[\sigma(t')] \) belong to \( SP^* \cap Obs_{Eq}(S_{obs}) \). As \( SP \models \sigma(t) = \sigma(t') \), then by definition of \( \sim_P \), \( T_\Sigma/\sim_P \models \sigma(t) = \sigma(t') \).

In both cases, \( T_\Sigma/\sim_P \) satisfies the considered axiom of \( SP \). Then, \( P \) is observationally equivalent to \( T_\Sigma/\sim_P \), which is a model of \( SP \). We get \( Correct_{Observe}(S_{obs})(P, SP) \) for \( P \) initial on all ground instances of equations occurring in the premises of axioms in \( Ax \).

The opposite implication of exhaustiveness is obvious. \( \square \)
5.3. Quantifier-free first-order specifications

Testing from algebraic specifications has been studied for a larger class of specifications, namely quantifier-free first-order specifications [26, 27], where a test case selection algorithm have been proposed. In the latter work, a result of exhaustiveness is also established. Here, we generalize this result by distinguishing observable and non-observable sorts. This requires to extend the initiality condition to such formulas.

We first need to consider a notion of positiveness in quantifier-free first-order formulas, similarly to Machado [12]. Roughly speaking, this condition states that non-observable equations only occur at positive positions. Intuitively, an equation is said to be at a positive position in a formula \( \varphi \) if in the disjunctive normal form of \( \varphi \), the equation is not preceded by a negation.

**Notation.** Using the standard numbering of tree nodes by strings of natural number, a position in a formula \( \varphi \) is a string \( \omega \) on \( \mathbb{N} \) which represents the path from the root of \( \varphi \) to the sub-formula at that position.

**Definition 5.4**
The property for a \( \Sigma \)-equation \( t = t' \) in a formula \( \varphi \) to be positive (resp. negative) at a position \( \omega \), is defined as follows:

- if \( \varphi \) is of the form \( u = v \) then \( t = t' \) is positive at \( \omega \) in \( \varphi \) iff \( \omega = \varepsilon, u = t \) and \( v = t' \), where \( \varepsilon \) denotes the empty word.
- if \( \varphi \) is of the form \( \varphi_1 \land \varphi_2 \) or \( \varphi = \varphi_1 \lor \varphi_2 \) then \( \omega = i.\omega' \) with \( i = 1, 2 \), and \( t = t' \) is positive (resp. negative) at \( \omega \) in \( \varphi \) iff \( t = t' \) is positive (resp. negative) at \( \omega' \) in \( \varphi_i \),
- if \( \varphi \) is of the form \( \neg \varphi_1 \) then \( \omega = 1.\omega' \), and \( t = t' \) is positive (resp. negative) at \( \omega \) in \( \varphi \) iff \( t = t' \) is negative (resp. positive) at \( \omega' \) in \( \varphi_1 \), and
- if \( \varphi \) is of the form \( \varphi_1 \Rightarrow \varphi_2 \) then \( \omega = i.\omega' \) with \( i = 1, 2 \), and \( t = t' \) is positive (resp. negative) at \( \omega \) in \( \varphi \) iff
  - if \( i = 1 \) then \( t = t' \) is negative (resp. positive) in \( \varphi \) at \( \omega' \).
  - otherwise, \( t = t' \) is positive (resp. negative) in \( \varphi \) at \( \omega' \).

A \( \Sigma \)-equation is positive (resp. negative) in \( \varphi \) if, and only if it is positive (resp. negative) at position \( \varepsilon \) in \( \varphi \).

In Theorem 5.3, the initiality condition is imposed on ground instances of the premises of conditional axioms, which are the negative equations in these axioms. Following this observation, the initiality condition will be imposed on all the negative equations of first-order axioms, in order to get the result of exhaustiveness for quantifier-free first-order specifications.

**Theorem 5.5**
Let \( SP = (\Sigma, Ax) \) be a quantifier-free first-order specification. Let \( K \) be the class of programs \( P \) which are initial on all ground instances of any negative \( \Sigma \)-equation occuring in axioms of \( Ax \). Then, \( SP^* \cap \text{Obs}_{Eq}(S_{obs}) \) is exhaustive for \( K \).

**Proof**
We only prove the "only if" part of the exhaustiveness property because the if part is obvious. Let us show that \( \text{Correct}_{\text{Obs}}(S_{obs})(P, SP) \). By following the proof of Proposition 4.1, we define the congruence \( \sim_P \) such that \( P \equiv_{\text{Obs}}(S_{obs}) T_\Sigma/\sim_P \).
Let \( P \) be a program such that \( P \models SP^* \cap \text{Obs}_{E}^{q}(S_{\text{obs}}) \). Let \( \varphi \) be an axiom of \( Ax \). Let \( \iota : V \rightarrow T_{\Sigma}/\sim_{P} \) be an interpretation. As in the previous proofs, \( \iota \) can be factorized as follows: \( \iota = q_{\sim_{P}} \circ \sigma \) where \( \sigma : V \rightarrow T_{\Sigma} \) is a ground substitution and \( q_{\sim_{P}} : T_{\Sigma} \rightarrow T_{\Sigma}/\sim_{P} \) is the quotient morphism. Let us denote by \( Tr(\sigma(\varphi)) \) the set of ground formulas obtained from \( \sigma(\varphi) \) by replacing every \( \Sigma \)-equation \( t = t' \) if non-observable and in negative position by \( c[t] = c[t'] \) for every \( c : s \in \mu Ctx \).

First, let us show by structural induction on ground \( \Sigma \)-formulas the property \( P(\sigma(\varphi)) \) defined by:

\[
(\forall \psi \in Tr(\sigma(\varphi)), T_{\Sigma}/\sim_{P} \models \psi) \iff T_{\Sigma}/\sim_{P} \models \sigma(\varphi)
\]

- **Base case**: \( \varphi \) is a \( \Sigma \)-equation \( t = t' \) with \( t, t' \in T_{\Sigma} \). Here, two cases have to be considered:
  
  1. \( s \in S_{\text{obs}} \). \( Tr(\sigma(\varphi)) \) is the singleton \( \{ \sigma(t) = \sigma(t') \} \). But, \( \sigma(t) = \sigma(t') \in SP^* \cap \text{Obs}_{E}^{q}(S_{\text{obs}}) \). We then have \( T_{\Sigma}/\sim_{P} \models \sigma(\varphi) \).
  
  2. \( s \notin S_{\text{obs}} \). Therefore, \( Tr(\sigma(\varphi)) = \{ c[\sigma(t)] = c[\sigma(t')] \mid s \in \mu Ctx \} \). By the property that \( P \models obs(S_{\text{obs}}) T_{\Sigma}/\sim_{P}, \) we have for every \( c \in \mu Ctx \) that \( P \models c[\sigma(t)] = c[\sigma(t')] \). By the definition of \( \sim_{P} \), we then have that \( \sigma(t) \sim_{P} \sigma(t') \), and then \( T_{\Sigma}/\sim_{P} \models \sigma(\varphi) \).

- **Inductive step**: Let us handle the more complicated case where \( \varphi \) is \( \lnot \varphi_1 \). By definition, \( Tr(\sigma(\varphi)) = \{ \lnot \psi_1 \mid \psi_1 \in Tr(\sigma(\varphi_1)) \} \).

  - Let us suppose that \( T_{\Sigma}/\sim_{P} \not\models \lnot \varphi_1 \). \( \sigma(\varphi_1) \) being a ground formula, we then have \( T_{\Sigma}/\sim_{P} \not\models \sigma(\varphi_1) \). By the induction hypothesis and the fact that \( P \) is initial on all ground equations in negative position in \( \sigma(\varphi_1) \), we can write that for every \( \psi_1 \in Tr(\sigma(\varphi_1)) \), \( T_{\Sigma}/\sim_{P} \not\models \psi_1 \), and then \( T_{\Sigma}/\sim_{P} \not\models \lnot \psi_1 \).

  - Let us suppose that for every \( \lnot \psi_1 \in Tr(\sigma(\varphi)) \), \( T_{\Sigma}/\sim_{P} \models \lnot \psi_1 \). \( \psi_1 \) being a ground formula, we have \( T_{\Sigma}/\sim_{P} \not\models \psi_1 \). By the induction hypothesis, we then have \( T_{\Sigma}/\sim_{P} \not\models \sigma(\varphi_1) \), and then \( T_{\Sigma}/\sim_{P} \models \lnot \varphi_1 \).

The cases of the other propositional connectives are simpler and are left to the reader.

Let us now show by structural induction on ground formulas that for every ground \( \Sigma \)-formula \( \varphi \) for which \( P \) is initial on all its equations in negative position for \( SP \), we have:

\[
SP \models \varphi \iff \forall \psi \in Tr(\varphi), P \models \psi
\]

The proof is appreciably similar to the previous one.

- **Base case**: Directly from definitions and hypothesis.
- **Inductive step**: Here also we propose to handle the case where \( \varphi \) is \( \lnot \varphi_1 \). By definition, \( Tr(\varphi) = \{ \lnot \psi_1 \mid \psi_1 \in Tr(\varphi_1) \} \).

  - Let us suppose that \( SP \models \varphi \). Therefore, we have \( SP \not\models \varphi_1 \). Hence, by the induction hypothesis, we have for every \( \psi \in Tr(\varphi_1) \) that \( P \not\models \psi \), and then \( P \models \lnot \psi \).

  - Let us suppose that for every \( \lnot \psi_1 \in Tr(\varphi) \), \( T_{\Sigma}/\sim_{P} \models \lnot \psi_1 \). \( \psi_1 \) being a ground formula, we have \( T_{\Sigma}/\sim_{P} \not\models \psi_1 \). By the induction hypothesis and the fact that \( P \) is initial on all ground equations in negative position in \( \varphi_1 \), we can write that for every \( \psi_1 \in Tr(\varphi_1) \), \( T_{\Sigma}/\sim_{P} \not\models \psi_1 \), and then \( T_{\Sigma}/\sim_{P} \models \lnot \psi_1 \).
As previously, the cases of the other propositional connectives are simpler and are left to the reader.

Hence, to finish the proof, we know by the previous result that for every axiom \( \varphi \) and every interpretation \( \iota \), \( P \models \psi \) for every \( \psi \in \text{Tr}(\sigma(\varphi)) \) where \( \iota = q_{\sim_p} \circ \sigma \). We can then conclude, by the property \( P(\sigma(\varphi)) \), that \( T_{\Sigma}/_{\sim_p} \models \sigma(\varphi) \), and then \( T_{\Sigma}/_{\sim_p} \models \iota \varphi \).

6. GETTING AROUND INITIALITY

We saw that when dealing with more expressive specifications than equational ones, the very strong property of initiality has to be imposed on programs to obtain the exhaustiveness of \( SP^* \cap \text{Obs}_{Eq}(S_{obs}) \). Since we are testing programs as black boxes, this condition may be impossible to verify on the program under test. Therefore, we study here how to weaken or to remove the initiality condition on programs.

6.1. Structured specifications

All the specifications we gave so far are what we call flat specifications, meaning that they are specifications of a single software module. However, for the description of large systems, it is convenient to compose specifications in a modular way [39]. The specification of a large system is generally built from small specifications of individual modules, that are composed by making their union and enriching the resulting specification with new features in order to get new (larger) specifications, that are themselves composed and so on.

Example 6.1
We specify sets of natural numbers equipped with the standard operations like union, intersection, membership and size. The specification \( \text{Set} \) is built over both specifications of Booleans and natural numbers. We first make the union of these two specifications (with the \textbf{and} operator in CASL). Then we enrich the obtained specification (with the \textbf{then} operator in CASL) by adding the new type \( \text{Set}[\text{Nat}] \) and new operations for this type, involving Booleans and natural numbers.

A set is either the empty set or a finite union of singletons. The union of two sets is specified only through its properties of associativity, commutativity, idempotence and the empty set being its neutral element. We use the shortcuts provided by the CASL language for standard properties of operations, like associativity, commutativity, idempotence and the existence of a neutral element.
spec Set =
  BOOL and NAT
then type Set[Nat] ::= ∅ | {}(Nat) | _∪ _∪(Set[Nat]; Set[Nat])
opts _∪ _∩ : Set[Nat] × Set[Nat] → Set[Nat], assoc, comm, idem, unit ∅;
    isin : Nat × Set[Nat] → Bool;
    size : Set[Nat] → Nat
∀ x, y : Nat; S, S', S'' : Set[Nat]

%% axioms for the membership operation
• isin(x, ∅) = False
• x = y ⇔ isin(x, {y}) = True
• (isin(x, S) = True ∨ isin(x, S') = True) ⇔ isin(x, S ∪ S') = True

%% axioms for the intersection operation
• S ∩ ∅ = ∅
• isin(x, S) = True ⇒ S ∩ {x} = {x}
• isin(x, S) = False ⇒ S ∩ {x} = ∅
• S ∩ (S' ∪ S'') = (S ∩ S') ∪ (S ∩ S'')

%% axioms for the size operation
• size(∅) = 0
• size({x}) = s(0)
• size(S ∪ S') = (size(S) + size(S')) − size(S ∩ S')
end

When dealing with such structured specifications, built along union and enrichment, initiality on programs can be stated by taking advantage of the specification structure. Indeed, as shown above, programs often implement data structures that are recursively built over elementary data structures provided by the target programming language. Hence, every program \( P \) can be seen as the enrichment of smaller programs \( P_i \) implementing these elementary data structures. Moreover, if \( SP_i = (\Sigma_i, Ax_i) \) is the specification of one of these programs \( P_i \), the model of \( P_i \) is often the initial algebra, i.e. it satisfies:

\[
∀ t, t' \in T_{\Sigma_i}, P_i \models t = t' \iff SP_i \models t = t'
\]

Finally, these elementary data types are often the only ones that are observable because they are provided with an implemented equality. When the specification \( SP \) that specifies \( P \) has the property not to generate junks (it is sufficiently complete over \( SP_i \) for all \( i \)) then \( P \) is initial for every equation \( t = t' \) of sort in \( S_j \).

Definition 6.2 (Sufficient completeness [31])
Let \( SP = (\Sigma, Ax) \) be a specification where \( \Sigma = (S, F) \), and let \( SP_0 = (\Sigma_0, Ax_0) \) where \( \Sigma_0 = (S_0, F_0) \) be a subspecification of \( SP \) (i.e. \( \Sigma_0 \subseteq \Sigma \) and \( Ax_0 \subseteq Ax \)). \( SP \) is said to be sufficiently
complete over $SP_0$ if and only if
\[ \forall s \in S_0, \forall t \in T_{\Sigma_s}, \exists t_0 \in T_{\Sigma_{s_0}}, SP \models t = t_0 \]

Intuitively, $SP$ is sufficiently complete over $SP_0$ if the new operations in $SP$ do not create new values of sort in $S_0$.

We can see that the structured specification $SET$ of Example 6.1 is sufficiently complete over $BOOL$ and $Nat$. It is sufficiently complete over $BOOL$ since the specification of the membership operation associates a Boolean term to any term of the form $isin(x, S)$ where $x$ is of sort $Nat$ and $S$ of sort $Set[Nat]$. It would not be sufficiently complete over $BOOL$ if we replaced for instance the second axiom of $isin$ with the implication $x = y \Rightarrow isin(x, \{y\}) = True$ only. Then the term $isin(0, \{s(0)\})$ of sort $BOOL$ would denote a new Boolean value. Similarly, the specification $SET$ is sufficiently complete over $Nat$ since the specification of the size operation associates a term of sort $Nat$ to any term of the form $size(S)$ where $S$ is of sort $Set[Nat]$. It would not be sufficiently complete over $Nat$ if we forgot for instance the first axiom for $size$. Then the term $size(\emptyset)$ of sort $Nat$ would denote a new natural, different from all the natural numbers that can be built from $0$ and the successor operation.

Considering that only the sorts of $S_i$ are observable, we can show that a program $P$ is initial when each $P_i$ is supposed to be initial and $SP$ is sufficiently complete over each subspecification $SP_i$.

**Proposition 6.3**

Let $SP = (\Sigma, Ax)$ be a specification where $\Sigma = (S, F)$, and let $SP_0 = (\Sigma_0, Ax_0)$ where $\Sigma_0 = (S_0, F_0)$ be a subspecification of $SP$ such that $SP$ is sufficiently complete over $SP_0$. Let $P \in Alg(\Sigma)$ be a program such that $P \models SP^* \cap Obs_{Eq}(S_0)$ and
\[ \forall s \in S_0, \forall t, t' \in T_{\Sigma_0}, P \models t = t' \iff SP_0 \models t = t' \]

Then, for every ground $\Sigma$-equation $t = t'$ of sort in $S_0$, $P$ is initial on $t = t'$ for $SP$.

**Proof**

Let $P \models t = t'$ with $t, t' \in T_{\Sigma_s}$ for $s \in S_0$. By hypothesis, there exist $u, v \in T_{\Sigma_{s_0}}$ such that $SP \models t = u$ and $SP \models t' = v$. Hence, $t = u, t' = v \in SP^* \cap Obs_{Eq}(S_0)$. By hypothesis, we deduce that $P \models t = u$ and $P \models t' = v$, and then by transitivity, $P \models u = v$. By hypothesis, we then have that $SP_0 \models u = v$ and then $SP \models u = v$, whence we deduce that $SP \models t = t'$.

Let $SP \models t = t'$ with $t, t' \in T_{\Sigma_s}$ for $s \in S_0$. By hypothesis, we obviously get that $P \models t = t'$.

Therefore, with a structured specification $SP$ sufficiently complete over its subspecifications $SP_i$, the test set $SP^* \cap Obs_{Eq}(\cup_i S_i)$ is exhaustive for any program $P$ which is initial on the equations built from $SP_i$ only.

**Corollary 6.4**

Let $SP = (\Sigma, Ax)$ be a specification where $\Sigma = (S, F)$, and let $SP_0 = (\Sigma_0, Ax_0)$ where $\Sigma_0 = (S_0, F_0)$ be a subspecification of $SP$.

If $SP$ is sufficiently complete over $SP_0$, then $SP^* \cap Obs_{Eq}(S_0)$ is exhaustive for the class of programs $P$ satisfying:
\[ \forall s \in S_0, \forall t, t' \in T_{\Sigma_s}, P \models t = t' \iff SP_0 \models t = t' \]
6.2. Ground conditional formulas as test cases

We can see that the initiality property on programs is needed when test cases have a more restricted form than axioms. On the contrary, we saw with Proposition 4.1 and Corollary 4.2 that $SP^* \cap \text{Obs}$ is exhaustive for any program when both axioms and test cases are unconditional equations. We propose in this paragraph to extend this result by studying the exhaustiveness result for conditional specifications, when test cases are ground conditional formulas. We will see that, here also, no condition on programs is needed to get the exhaustiveness result. On the other hand, a condition has to be imposed on specification. Indeed, it is well-known that if unobservable equations occur in the premises of axioms, some semantic problems may occur [1, 9]. Since a non-observable equation $t = t'$ can only be (partially) observed through observable contexts, the satisfaction of all the equations $c[t] = c[t']$ built from contexts $c \in \mu \text{Ctx}$ is not equivalent to the satisfaction of the equation $t = t'$. Then the premises of an axiom may be satisfied through contexts by the program under test without being fully satisfied.

Example 6.5

Let us add to the specification LISTOBSERVERS2 an axiom which prevents the operation of insertion at the head of a list from being idempotent:

\[
\text{spec LISTINSERT =}
\begin{align*}
\text{types Bool ::= True | False; } \\
\text{Nat ::= 0 | s(Nat);} \\
\text{List[Nat] ::= [ ] | _ :: _ (Nat; List[Nat])} \\
\text{op isin : Nat \times List[Nat] \rightarrow Bool} \\
\forall x, y : \text{Nat}; L : \text{List[Nat]} \\
& \quad \text{isin}(x, []) = \text{False} \\
& \quad \text{isin}(x, x :: L) = \text{True} \\
& \quad \text{isin}(x, L) = \text{True} \Rightarrow \text{isin}(x, y :: L) = \text{True} \\
& \quad x :: (x :: L) = x :: L \Rightarrow \text{True} = \text{False} \\
\end{align*}
\]

Here, only natural numbers and Booleans are observable, lists are observed through the membership predicate isin. If we apply contexts to build a test set from this specification, as we did for equational specifications, we obtain in particular the following formula:

\[
isin(n, n :: (n :: L))) =isin(n, n :: L) \Rightarrow \text{True} = \text{False}
\]

where $n$ is any natural. But this formula cannot be a test case, because it is not a semantic consequence of the specification. Actually, $isin(n, L) = isin(n, L')$ for all $n \in \mathbb{N}$ does not imply $L = L'$. As a consequence, the above formula is not a test for the property $n :: (n :: L) = n :: L \Rightarrow \text{True} = \text{False}$, since the program under test can pass the test for all possible values of $n$ without satisfying the property.

A sufficient condition to solve this problem is to impose that only observable equations occur in premises of axioms. Such specifications are called positive.
Definition 6.6 (Positive conditional specification)
A specification $SP = (\Sigma, Ax)$ with a set $S_{obs}$ of observable sorts is said positive if and only if all equations occurring in the premises of axioms in $Ax$ are observable: for every $t_1 = t'_1 \land \ldots \land t_n = t'_n \Rightarrow t = t'$ in $Ax$ and for every $i, 1 \leq i \leq n$, there exists $s$ in $S_{obs}$ such that $t_i, t'_i \in T_\Sigma(V)_s$.

Unlike initiality on programs, this condition can be easily and automatically checked on specifications.

Theorem 6.7
Let $SP = (\Sigma, Ax)$ be a positive conditional specification where $\Sigma = (S, F)$ has a set of observable sorts $S_{obs} \subseteq S$. Then, $SP^* \cap Obs_{Cond}(S_{obs})$ is exhaustive for $K = Alg(\Sigma)$.

Proof
We only prove the "only if" part of the exhaustiveness property because the if part is obvious.

Let us suppose that $P \models SP^* \cap Obs_{Cond}(S_{obs})$. Let us show that $Correct_{Obs_{Cond}(S_{obs})}(P, SP)$. By following the proof of Proposition 4.1, we define the congruence $\sim_P$ such that $P \equiv Obs_{Cond}(S_{obs}) T_\Sigma/\sim_P$.

Let $\varphi : t_1 = t'_1 \land \ldots \land t_n = t'_n \Rightarrow t = t'$ be a conditional axiom of $Ax$. Let $\iota : V \rightarrow T_\Sigma/\sim_P$ be an interpretation. As in the previous proofs, $\iota$ can be factorized as follows: $\iota = q_{\sim_P} \circ \sigma$ where $\sigma : V \rightarrow T_\Sigma$ is a ground substitution and $q_{\sim_P} : T_\Sigma \rightarrow T_\Sigma/\sim_P$ is the quotient morphism. Let us denote by $Tr(\sigma(\varphi))$ the set of ground formulas obtained from $\sigma(\varphi)$ by replacing the $\Sigma$-equation $t = t'$ if non-observable by $c[t] = c[t']$ for every $c : s \in \mu Clx$. From the hypothesis that $SP$ is positive, we have that $Tr(\sigma(\varphi)) \subseteq SP^*$, and then $Tr(\sigma(\varphi)) \subseteq SP^* \cap Obs_{Cond}(S_{obs})$. Because $SP$ is positive and by definition of $\sim_P$, we can easily show that:

$$(\forall \psi \in Tr(\sigma(\varphi)), T_\Sigma/\sim_P \models \psi) \iff T_\Sigma/\sim_P \models _\iota \varphi$$

We further know that for every formula $\varphi$, $Tr(\sigma(\varphi)) \subseteq SP^* \cap Obs_{Cond}(S_{obs})$. By hypothesis, we then have for every $\psi \in Tr(\sigma(\varphi))$ that $P \models \psi$. By the property that $P \equiv Obs_{Cond}(S_{obs}) T_\Sigma/\sim_P$, we deduce that $T_\Sigma/\sim_P \models \psi$. We conclude that $T_\Sigma/\sim_P \models _\iota \varphi$.

6.3. Ground first-order formulas as test cases
To extend the previous result in order to get the exhaustiveness result without imposing the initiality condition on programs, the family of selection criteria based on axiom unfolding has been extended to the class of axiomatic specifications whose axioms are quantifier-free first-order formulas [26]. Some works on specification-based testing [11, 12] have already considered a similar class of formulas. They propose a mixed approach combining black-box and white-box testing to deal with the problem of non-observable data types. From the selection point of view, they do not propose any particular strategy, except for substituting axiom variables by some arbitrarily chosen data. Following the specification-based testing framework [10], we showed that for every specification $SP$ whose axioms are quantifier-free formulas, $SP^* \cap Obs$ is exhaustive for any program without constraint when $Obs$ is the set of ground formulas over $\Sigma$ (no observability constraint is supposed, i.e. every sort is observable) [26, 27].

Here, as in Section 5.3, we propose to extend this result by considering a subset of observable sorts which, as already explained in Section 3, is a realistic assumption for most programs. Therefore,
given a subset $S_{\text{obs}}$ of observable sorts for a signature $\Sigma$, the set $\text{Obs}(S_{\text{obs}})$ contains all ground first-order formulas in which all equations are on observable sorts. Unlike the exhaustiveness result established in our previous work [26, 27] and as in the previous section, a constraint has to be imposed on specifications to obtain the exhaustiveness of $SP^\bullet \cap \text{Obs}(S_{\text{obs}})$. This constraint generalizes Definition 6.6 to quantifier-free first-order formulas and is similar to the one used by Machado [11]. Roughly speaking, this condition states that non-observable equations only occur at positive positions.

Definition 6.8 (Positive first-order specifications)
A first-order specification $SP = (\Sigma, Ax)$ with a subset of observable sorts $S_{\text{obs}}$ is said positive if and only if for all axioms $\varphi \in Ax$, all equations of non-observable sort are positive in $\varphi$.

Theorem 6.9
Let $SP = (\Sigma, Ax)$ be a positive quantifier-free first-order specification where $\Sigma = (S, F)$ has a set of observable sorts $S_{\text{obs}} \subseteq S$. Then, $SP^\bullet \cap \text{Obs}_{QF}(S_{\text{obs}})$ is exhaustive for $K = \text{Alg}(\Sigma)$.

Proof
We only prove the ”only if” part of the exhaustiveness property because the if part is obvious.

Let us suppose that $P \models SP^\bullet \cap \text{Obs}_{QF}(S_{\text{obs}})$. Let us show that $\text{Correct}_{\text{obs}_{QF}(S_{\text{obs}})}(P, SP)$. By following the proof of Proposition 4.1, we define the congruence $\sim_P$ such that $P \equiv_{\text{obs}_{QF}(S_{\text{obs}})} T_{\Sigma/\sim_P}$.

Let $\varphi$ be an axiom of $Ax$. Let $\iota : V \rightarrow T_{\Sigma/\sim_P}$ be an interpretation. As in the previous proofs, $\iota$ can be factorized as follows: $\iota = q_{\sim_P} \circ \sigma$ where $\sigma : V \rightarrow T_{\Sigma}$ is a ground substitution and $q_{\sim_P} : T_{\Sigma} \rightarrow T_{\Sigma/\sim_P}$ is the quotient morphism. Let us denote by $Tr(\varphi)$ the set of ground formulas obtained from $\varphi$ by replacing every non-observable $\Sigma$-equation $t = t'$ with $t, t' \in T_{\Sigma}$, by $c[t] = c[t']$ for every $c : s \in \mu\text{Ctx}$. From the hypothesis that all non-observable $\Sigma$-equations are positive in $\varphi$, we have that $Tr(\varphi) \subseteq P^\bullet$, and then $Tr(\varphi) \subseteq SP^\bullet \cap \text{Obs}_{QF}(S_{\text{obs}})$. By structural induction on $\varphi$, let us show the property $P(\varphi)$ defined by:

$$(\forall \psi \in Tr(\varphi), T_{\Sigma/\sim_P} \models \psi) \iff T_{\Sigma/\sim_P} \models _\iota \varphi$$

- Base case: $\varphi$ is a $\Sigma$-equation $t = t'$ with $t, t' \in T_{\Sigma}$. Here, two cases have to be considered:

1. $s \in S_{\text{obs}}$. Therefore, $Tr(\varphi)$ is the singleton $\{\sigma(t) = \sigma(t')\}$.
   - If $T_{\Sigma/\sim_P} \models \sigma(t) = \sigma(t')$ then $\sigma(t) \sim_P \sigma(t')$. We then conclude that $T_{\Sigma/\sim_P} \models _\iota t = t'$.
   - If $T_{\Sigma/\sim_P} \models _\iota t = t'$ then $\sigma(t) \sim_P \sigma(t')$. We then conclude $T_{\Sigma/\sim_P} \models \sigma(t) = \sigma(t')$.

2. $s \notin S_{\text{obs}}$. Therefore, $Tr(\varphi) = \{c[\sigma(t)] = c[\sigma(t')]| c : s \in \mu\text{Ctx}\}$.
   - Suppose that for every $c : s \in \mu\text{Ctx}$, $T_{\Sigma/\sim_P} \models c[\sigma(t)] = c[\sigma(t')]$. By the property that $P \equiv_{\text{obs}_{QF}(S_{\text{obs}})} T_{\Sigma/\sim_P}$, we have for every $c : s \in \mu\text{Ctx}$ that $P \models c[\sigma(t)] = c[\sigma(t')]$. By the definition of the congruence $\sim_P$, this means that $\sigma(t) \sim_P \sigma(t')$. We then conclude that $T_{\Sigma/\sim_P} \models _\iota t = t'$.
   - Suppose that $T_{\Sigma/\sim_P} \models _\iota t = t'$. By definition, this means that $\sigma(t) \sim_P \sigma(t')$. By definition of the congruence of $\sim_P$, we then have for every $c : s \in \mu\text{Ctx}$ that $P \models c[\sigma(t)] = c[\sigma(t')]$. By the property that $P \equiv_{\text{obs}_{QF}(S_{\text{obs}})} T_{\Sigma/\sim_P}$, we then conclude that for every $c : s \in \mu\text{Ctx}$, $T_{\Sigma/\sim_P} \models c[\sigma(t)] = c[\sigma(t')]$. 

• **Inductive step:** Let us handle the case where \( \varphi \) is \( \neg \psi_1 \). By definition, we have that \( Tr(\sigma(\varphi)) = \{ \neg \psi_1 | \psi_1 \in Tr(\sigma(\varphi_1)) \} \).

Suppose that for every \( \neg \psi_1 \in Tr(\sigma(\varphi)) \), \( T_{\Sigma/\sim_p} \models \neg \psi_1 \). By the property that all the \( \psi_1 \in Tr(\sigma(\varphi_1)) \) are ground formulas, this means that for every \( \psi_1 \in Tr(\sigma(\varphi_1)) \), \( T_{\Sigma/\sim_p} \models \neg \psi_1 \). By the induction hypothesis, we then have that \( T_{\Sigma/\sim_p} \not\models \neg \psi_1 \) whence we conclude \( T_{\Sigma/\sim_p} \models \varphi_1 \).

Suppose that \( T_{\Sigma/\sim_p} \models \varphi_1 \). This means that \( T_{\Sigma/\sim_p} \not\models \neg \varphi_1 \). By the induction hypothesis and because \( SP \) is positive, we then have that for every \( \psi_1 \in Tr(\sigma(\varphi_1)) \), \( T_{\Sigma/\sim_p} \not\models \psi_1 \). We then conclude that for every \( \psi_1 \in Tr(\sigma(\varphi_1)) \), \( T_{\Sigma/\sim_p} \models \neg \psi_1 \).

The cases of the other propositional connectives are simpler and are left to the reader.

Moreover, we know that for every formula \( \varphi \), \( Tr(\sigma(\varphi)) \subseteq SP^* \cap Obs_{QF}(S_{obs}) \). By hypothesis, we then have for every \( \psi \in Tr(\sigma(\varphi)) \) that \( P \models \psi \). By the property that \( P \equiv_{Obs_{QF}(S_{obs})} T_{\Sigma/\sim_p} \), we deduce that \( T_{\Sigma/\sim_p} \models \psi \). Finally, by the property \( P(\varphi) \), we conclude \( T_{\Sigma/\sim_p} \models \varphi \).  

---

**7. GENERAL FIRST-ORDER AXIOMS**

When dealing with more general first-order formulas, i.e. formulas with quantifiers, the existence of an exhaustive test set is no longer possible to ensure. The presence of an existential quantifier in an axiom actually prevents from building a relevant test set for this axiom.

The problem comes from the fact that the specification cannot be used to build the relevant data on which to execute the program. If the specification requires that there exists an element \( a \) verifying a certain property, there is no way to deduce from the specification only some elements \( a \) satisfying this property in the program.
Example 7.1
Let us consider the following specification of natural numbers equipped with multiplication.

```plaintext
spec MULTIPLE =
  types Bool ::= True | False;
  Nat ::= 0 | s(Nat)
  ops  _+_: Nat × Nat → Nat;
        _∗_: Nat × Nat → Nat;
    multiple : Nat × Nat → Bool
  ∀x,y : Nat
    • x + 0 = x
    • x + s(y) = s(x + y)
    • x * 0 = 0
    • x * s(y) = x + (x * y)
    • multiple(x, y) = True ⇔ ∃z : Nat • x = y * z
end
```

To test if 18 is a multiple of 3 in the program, we must be able to find \( z \) such that \( 18 = 3 * z \), which is of course impossible as the program is a black-box.\(^5\) In fact, exhibiting such a value would amount to simply prove the system with respect to the axiom. Therefore, testing a program with respect to such a specification amounts to proving the correctness of this program as established by Theorem 7.2.

Given a \( \Sigma \)-algebra \( A \), we denote by \( Th(A) \) the closed theory of \( A \), that is \( Th(A) = \{ \varphi \mid A \models \varphi, \varphi \text{ closed} \} \). A formula is closed when each occurrence of its variables is in the scope of a quantifier. A theory is closed when each of its formulas is closed.

**Theorem 7.2**
Let \( SP = (\Sigma, Ax) \) be a consistent specification (i.e. \( Alg(SP) \neq \emptyset \)). Let \( K \) be a full subcategory of \( Gen(\Sigma) \). Then, \( SP^* \cap Obs \) is exhaustive for \( K \) if and only if for every \( A \in K \), \( (\Sigma, Ax \cup Th(A)) \) is consistent.

**Proof**

**The if part.** Let \( P \in K \) such that \( P \models SP^* \cap Obs \). Let us show that \( Correct_{Obs}(P, SP) \). As \( SP \cup Th(P) \) is consistent, there exists a \( \Sigma \)-algebra \( A \) such that \( A \models SP \cup Th(P) \). By definition, \( Th(P) \) is a complete theory. In first-order logic, it is well-known that every \( \Sigma \)-algebra \( A \models Th(P) \) is elementarily equivalent to \( P \) on closed formulas. Therefore, for every \( \varphi \in Obs \), \( A \models \varphi \iff P \models \varphi \) whence \( A \equiv_{obs} P \).

Suppose that there exists a \( \Sigma \)-model \( A \in Alg(SP) \) such that \( A \equiv_{obs} P \). Let \( \varphi \in SP^* \cap Obs \). By hypothesis \( A \models \varphi \), then so does \( P \models \varphi \) as well.

**The only if part.** Suppose that \( SP \cup Th(P) \) is not consistent and let us show that \( SP^* \cap Obs \) is not exhaustive. Suppose that \( P \models SP^* \cap Obs \). As \( SP \cup Th(P) \) is inconsistent, the only possibility

---

\(^5\)Note that if the value of \( z \) could be deduced from the specification, this specification would be equivalent to a specification with no existential quantifier.
is that for every $\Sigma$-algebras $A \in Alg(SP)$, there exists $\varphi \in Th(P)$ such that $A \not\models \varphi$. As $P$ is reachable, then this means that for every $\Sigma$-algebra $A \in Alg(SP)$, $A \not\models_{\text{Obs}} P$. We then conclude that $Correct_{\text{Obs}}(P, SP)$ fails. \hfill $\square$

8. RELATED WORK

**Loose semantic and class of tests.** Taking all the observable semantic consequences of $SP$ for properly representing the class of all possible test cases means that all observable properties induced by the specification axioms, and only them, can be tested and that nothing is required on other properties. In particular, it is not required that a correct program has to compute as false any property which cannot be derived from the specification. Some works advocate an opposite position [14, 19, 40]: roughly speaking, in such frameworks, a test case is no more a simple formula, but can be represented as a couple $(\varphi, b)$ composed of a ground formula $\varphi$ and a Boolean value $b$ such that a correct program has to compute the formula $\varphi$ as true (resp. false) if the Boolean value $b$ is true (resp. false). Obviously, from ground instances $\varphi$ of specification axioms, one can derive a positive test case of the form $(\varphi, True)$, while negative test cases of the form $(\varphi, False)$ are new test cases that we do not consider in this paper (except in the presence of negation $\neg$ for quantifier-free first order specifications and general first-order specifications). For example, a possible negative test case could be $(5 + 0 = 7, False)$. To successfully pass this negative test case, a correct program has to compute different values for $5 + 0$ and $7$. All these negative properties are not directly implied by the specification axioms. In general, when adopting both positive and negative test cases, one considers a particular model for defining specification semantics, either the so-called initial model of the specification or the so-called terminal model. Thus, considering both negative and positive test cases is well-adapted for stating by testing techniques whether or not a program under test behaves as a targeted model. Initial semantics often requires to consider executable specifications. For example, axioms of $SP$ can be turned into a term rewriting system verifying that any term can be made equal to a normal form. The use of normal forms has been widely advocated for testing object-oriented programs from algebraic specifications [7, 19, 41]. In a loose approach, in order to obtain such kinds of negative test cases, one has to consider hypotheses on the specification (as the use of constructors or of well structured specifications). In particular, we have introduced in Definition 5.2 the initiality hypothesis that precisely imposes that the program satisfies an equation if and only if it is a semantic consequence of the specification. Thus, somehow, initiality hypothesis encompasses a negative test case.

\[\text{The other possibility is there exists } \varphi \in Ax \text{ such that } P \not\models \varphi. \text{ But in this case, } P \not\models_{\text{Obs}} SP^* \cap \text{Obs } \text{which is impossible since we supposed the contrary.}\]
However, if we consider abstract or incomplete specifications, then it is important not to consider negative tests. Let us take an example to illustrate this claim:

```plaintext
spec Nat =
    sort Nat;
    ops 0 :→ Nat;
        ∞ :→ Nat;
        s : Nat → Nat;
    _+_ : Nat × Nat → Nat;
    ∀x,y : Nat:
        • x + 0 = x
        • x + ∞ = ∞
        • x + s(y) = s(x + y)
end
```

In the above specification, there can be several possible ways to interpret the constant ∞: as a natural number playing the role of a bound whose precise value is not specified, for example, or as a constant outside the set of natural numbers abstractly representing the cardinality of all natural numbers. Thus, this specification is abstract since the designer has several choices to implement ∞.

To conclude, loose and initial semantics have always coexisted for algebraic specifications [25]. The initial semantics focuses on a unique model and is preferred at the very late stages of the design process, and for specifying usual built-in types provided with programming languages (as Booleans, natural numbers...). The loose semantics considers the class of all models satisfying axioms and is preferred at the first stages, when design choices are still to be done, and for high-level abstract data types (as sets or height-balanced binary search trees for example).

**Exhaustiveness.** Similar exhaustiveness results as the ones given in Section 4 have also been obtained by Chen et al. [6, 7] and Zhu [14].

However, the result presented in Theorem 4.3 is different from the one of Zhu [14] (Theorem 4.4) due to the loose semantics followed here. The reason is the following. Although SP is complete, this does not imply that all models in Alg(SP) are elementary equivalent (we accept that two terms in TΩ are equal in a model). Hence, by using our notations, consider the observational equivalence ∼obs defined by Zhu [14] as follows:

\[
t ∼obs t' ⇔ (∀c : s ∈ µCtx, SP |= c[t] = c[t'])
\]

It verifies that if P is successful then ∼obs ⊆ ∼P. However, we cannot conclude that ∼obs = ∼P (that is P∗ ∩ Obs = SP∗ ∩ Obs) except if P implements the initial model of Alg(SP). In the latter case, the model associated to P is by construction the terminal model of Zhu [14], and is considered as correct in both approaches. On the contrary, any other correct program, in our sense, only verifies the inclusion ∼obs ⊆ ∼P. This means, adequately with an algebraic loose approach, that a correct program may implement more equalities than strictly required by the specification. To illustrate this, let us consider the following specification, which specifies a counter that rings every second
tick.

\[ \text{spec COUNTER = } \]
\[ \begin{align*}
\text{type } & \text{Bool ::= True | False;} \\
\text{sort } & \text{Count;} \\
\text{ops } & \text{reset :\to Count;} \\
\text{} & \text{tick : Count \to Count;} \\
\text{} & \text{ring : Count \to Bool;} \\
\forall c : \text{Count; } \\
\text{} & \cdot \text{ring(reset)} = \text{True} \\
\text{} & \cdot \text{ring(tick(reset))} = \text{False} \\
\text{} & \cdot \text{ring(tick(tick(c))))} = \text{ring(c)} \\
\text{} & \cdot \text{tick(c)} = \text{reset} \Rightarrow \text{True} = \text{False}
\end{align*} \]
\[ \text{end} \]

The last axiom requires that the counter never resets, since it prevents any \( \text{tick}(c) \) to be equal to the \( \text{reset} \) constant. The loose semantics admit as models, the initial one (ensuring that \( \text{tick}^n(\text{reset}) \) is not equal to \( \text{reset} \) for any \( n \)), and some others such that the one for which \( \text{tick}^n(\text{reset}) \) is equal to \( \text{reset} \). This last model is such that \( \text{True} \) is equal to \( \text{False} \), while the first one makes different \( \text{True} \) and \( \text{False} \).

Zhu looks for the terminal model by testing all equalities \( t = t' \), and inequalities \( t \neq t' \), up to observability, that hold in the terminal model. In particular, it must be checked that \( \text{True} \) does not equal to \( \text{False} \) in the program. Here, there is at least one model where \( \text{True} \) equals to \( \text{False} \) (for example the trivial model where each sort is reduced to only one value) and one model where \( \text{True} \) and \( \text{False} \) are different (for example the initial model), therefore such (inequality) test cases do not make sense.

### 9. CONCLUSION

In this paper, we studied conditions on specifications and programs to ensure that the set of semantic consequences of the specification of a program is an exhaustive test set for this program. The existence of an exhaustive test set is an essential property in a testing framework because it prevents from rejecting a correct program or dually to accept an incorrect program. We studied conditions for exhaustiveness in different algebraic formalisms (equational, conditional, quantifier free and general first-order formulas) provided with a loose semantics and in the presence of non-observable sorts. We show in particular that the easiest way to get an exhaustive test set is to consider test cases of the same shape as the axioms of the specification. Otherwise, some rather strong hypotheses have to be imposed, in particular on the program under test, which may not be easy to verify.

**Acknowledgements.** Special thanks for Paolo Ballarini and Thomas Bellet for their careful reading of this text and for their linguistic corrections.

**REFERENCES**


