Testing and Validating the Quality of Specifications

Pieter Koopman, Peter Achten, Rinus Plasmeijer
Radboud University Nijmegen, Institute for Computer and Information Science, The Netherlands
{pieter,p.achten,rinus}@cs.ru.nl

***Abstract*—Model-based testing of state based systems is known to be able to spot non-conformance issues. However, up to half of these issues appear to be errors in the model rather than in the system under test. Errors in the specification at least hamper the prompt delivery of the software, so it is worthwhile to invest in the quality of the specification. Worse, errors in the specification that are also present in the system under test cannot be detected by model-based testing. In this paper we show how very desirable properties of specifications can be checked by systematic automated testing of the *specifications* themselves. We show how useful properties of specifications can be found by generalization of incorrect transitions encountered in simulation of the model.**

I. Introduction

Software systems tend to become larger and more complex. The requirements for the system are changing rapidly, often during the construction of the system. Finally, the time to market should be reduced. These factors can easily hamper the quality of the product. Systematic testing is by far the most used and effective way to determine the quality of a software system. In model-based testing, MBT, the test cases are generated on the fly from a formal specification. This ensures that the test suite is always up-to-date with that specification. More testing for improved confidence in the quality of the software is achieved by just changing a parameter in the test system.

Experience shows that writing a formal specification is an useful activity on its own. Many inaccuracies and misconceptions in the informal or semiformal specification/design of the system are found during the creation of such a formal specification. Testing real world systems also reveals that a significant fraction of the issues found during testing (on average 25%) is due to issues in the formal specification rather than the system under test (sut). Since a specification is a formal artefact similar in nature to the actual software, this is not astonishing. When a specification differs from a sut the issue is spotted during the tests and can be corrected. Nevertheless, the analysis and correction of issues in the specification is time-consuming and hence delays the release of the sut. It is therefore desirable to have high quality specifications before testing the sut is started.

In our model-based test system G∀st [12], [13], we use a transition function in the high level functional programming language Clean [15] as specification. The reasons to use a functional programming language as carrier of our models are:

- Functional languages exclude side-effects. This implies pure and clear semantics.
- Functions appear to be concise models. As shown in this paper they can be easily composed and transformed.
- Clean offers a complete set of high level programming primitives and very useful libraries. The rich type system, particulary the tailor made (recursive) data types, enables us to write very clear and expressive models. Parameterized types are especially useful if we model an extended state machine (a model with an unbounded number of states, inputs or outputs). By using an existing language all these things are obtained for free.
- The static type system of Clean will check that all identifiers that are used in the specification are defined and that they are used in a type correct way.
- The generic programming facilities (see appendix for a short introduction) of Clean enable us to define operations like equality, printing and generation of data elements that are needed in each test once and for all. This implies that the tester can derive these operations automatically for tailor made types in its test rather defining them manually. If the test engineer has specific wishes she can, of course, define her own algorithms instead of using the generic versions.
- Clean is a very efficient language, both as implementation and for the generated code. This implies that the execution of tests is efficient, and that the turn around time after changes in the model or testing parameters is extremely short.

Before we can start testing, the Clean-system checks the specification for matters like type correctness and whether all used identifiers are properly defined. This definitely contributes to the quality of the specifications used for testing. Still there can be many things wrong with a specification. A specification can prescribe unintended behavior in a formally completely correct way. These kind of issues cannot be detected automatically and have to be found by human inspection, perhaps supported by simulation of the execution based on the model. But the specification can also contain errors of a more technical nature which are trackable by a technical approach. For instance, a state based specification of the system can be partial while the specification is assumed to be total, or it can be non-deterministic while it is assumed to be deterministic. Other potential problems are that some transitions violate certain (domain specific) constraints.

In this paper we show that many of these technical issues in the specifications can be found by systematic testing of the specifications themselves. When the properties and invariants

of the specification are stated explicitly, the logical based branch of our test tool is able to test these properties of the specification independent from any implementation, even before an implementation exists. The advantage of this approach over model checking is that we can stay within the formalism, no transformation of models is needed.

In order to make this paper self contained we introduce the specification and testing of reactive systems in section II. In section III we illustrate the kind of specifications used by two examples. They will be used in the rest of this paper as running examples. Section IV shows that some properties can be obtained by a transformation of the specification. Testing of individual functions is discussed in section V. The testing of more general properties of specifications (such as determinism and reachability of states) is discussed in section VI. Verifying domain specific constraints is illustrated with some examples in section VII. Next we show how domain specific properties can be obtained by generalization of errors found by interactively developing an expanded state chart. In section IX we discus related work. Finally, we draw conclusions.

II. Model-Based Testing of Systems with a State

This section handles mathematically the kind of specifications used in our MBT approach. A reactive system has an internal state that can be changed by inputs and is preserved between the inputs. This implies that the reaction on the current input can depend on previous inputs. E.g. the system gets a number as input and the response is the number of inputs seen. A pure function can be specified without a state: the response is completely determined by the arguments. The reactive systems that are discussed here can be nondeterministic. During the tests we look only at the inputs and responses of the reactive system, the internal state is not known. This is called Black Box Testing, BBT.

The reactive system tested is the System Under Test, sut. Since the state of the sut is hidden, stating properties relating input, output and state is not feasible. We specify reactive systems by an extended state machine and require that the observed behavior of the sut conforms to this specification.

An Extended State Machine, ESM, as used by G∀st consists of states with labeled transitions between them. A transition is of the form $s \xrightarrow{i/o} t$, where s, t are states, i is an input which triggers the transition, and o is a, possibly empty, list of outputs. A transition $s \xrightarrow{i/o} t$ is formalized as a tuple (s, i, o, t). The set δ_r contains all allowed transitions in the specification. The transition function is defined by $\delta_f(s, i) = \{(o, t) | (s, i, o, t) \in \delta_r\}$. The type of this function is: $S \times I \rightarrow \mathbb{P}(O \times S)$ where S is the type of states, I is the type of inputs, and O is the type of outputs. We use $\mathbb{P}X$ as notation for a set of elements of type X.

In order to obtain a compact representation it is often more convenient to use functions from output sequences to states as result of δ_f, rather than enumerating all possible tuples. For a single transition we have: $s \xrightarrow{i/o} t \Leftrightarrow \exists f \in \delta_F(s, i) \wedge t \in f(o)$. In this way, a single function can represent arbitrary many output target-state tuples.

A specification is *total* or *complete* if for every $s \in S$ and $i \in I$ there is at least one output and target state defined. Specifications are *partial* if they are not total. Hence, a specification is *partial* if for some state s and input i, $\delta_f(s, i) = \emptyset$. If a specification is *nondeterministic* there are $s \in S$ and $i \in I$ with more than one associated target state. A specification is *deterministic* if for all states and inputs the size of the set of targets contains at most one element: $\# \delta_f(s, i) \leq 1$.

A trace σ is a sequence of inputs and associated outputs from the given state. A trace is defined inductively: the empty trace connects a state to itself: $s \stackrel{\epsilon}{\Longrightarrow} s$. We can combine a trace $s \stackrel{\sigma}{\Longrightarrow} t$ and a transition $t \xrightarrow{i/o} u$, to the trace $s \stackrel{\sigma;i/o}{\Longrightarrow} u$. An *input trace* contains only the input elements of a trace.

We define $s \xrightarrow{i/o} \equiv \exists t. s \xrightarrow{i/o} t$ and $s \stackrel{\sigma}{\Longrightarrow} \equiv \exists t. s \stackrel{\sigma}{\Longrightarrow} t$. All traces from state s are defined as: $traces(s) \equiv \{\sigma | s \stackrel{\sigma}{\Longrightarrow}\}$.

The inputs allowed in some state are given by $init(s) \equiv \{i | \exists o. s \xrightarrow{i/o}\}$. The states after applying trace σ in state s are given by $s \,\mathsf{after}\, \sigma \equiv \{t | s \stackrel{\sigma}{\Longrightarrow} t\}$. We overload $traces$, $init$, and after for sets of states instead of a single state by taking the union of the notion for the members of the set. When the transition function, δ_f, to be used is not clear from the context, we add it as subscript.

A. Conformance

This section defines the conformance relation between the specification and the sut. It is not required that the sut and the specification have exactly the same behavior. On parts that are not covered by the specification, any behavior of the sut is allowed. On the other hand, it is not necessary that the sut shows all allowed traces for nondeterministic specifications.

The sut is modeled as a black box transition system. One can observe its traces, but not its state. The sut, and its specification need not have identical input output behavior in all situations to say that the sut conforms to the specification.

Conformance of a sut to a specification spec is defined as:

$$\begin{aligned} \mathsf{sut}\ \mathit{conf}\ \mathsf{spec} \equiv\ & \forall \sigma \in traces_{\mathsf{spec}}(s_0), \\ & \forall i \in init(s_0\ \mathsf{after}_{\mathsf{spec}}\ \sigma),\ \forall o \in O \\ & (t_0\ \mathsf{after}_{\mathsf{sut}}\ \sigma) \xrightarrow{i/o} \Rightarrow\ (s_0\ \mathsf{after}_{\mathsf{spec}}\ \sigma) \xrightarrow{i/o} \end{aligned}$$

Here s_0 is the initial state of spec, and t_0 the initial state of sut. Intuitively the conformance relation reads: if the specification allows input i after trace σ, then the observed output of the sut should be allowed by the specification. The initial state t_0 of the sut is generally not known. The sut is assumed to be in state t_0 when we switch it on and when we reset it during testing.

The specification spec can be partial: nothing is specified about the behavior for some state and input combinations. The conformance relation allows any behavior of the sut if nothing is specified. Since everything is allowed, it makes no sense to test this. The sut cannot refuse inputs: in every state the sut should accept any input.

B. Testing Conformance

The conformance relation covers all possible traces. However, most extended state machines can show an infinite number of traces and each of these traces can have an unbounded length. Hence, it is generally impossible to determine conformance by investigating all traces completely. A test system approximates conformance by checking a finite number of finite traces. G∀st tests an initial part (by default 1000 steps) of a finite number (by default 100) of traces. The test system records that set of last states of the traces, rather than the complete traces. The actual trace is only recorded to give information about a conformance problem if it is detected. G∀st is able to generate inputs, also for complex user defined recursive data types, fully automatically. However, the user can guide this generation completely if that would be desired. The details are not relevant for this paper. See [13], [11].

C. Representation of Specifications in G∀st

As shown above two types of specification functions are used: $S \times I \rightarrow \mathbb{P}(O \times S)$ and $S \times I \rightarrow \mathbb{P}(O \rightarrow \mathbb{P}S)$. These specification functions are represented in G∀st by functions in the functional programming language Clean. The type of all specifications is given by the type synonym `Spec`. We use type variables (`s`, `i`, and `o`) to abstract from concrete types for state (S), input (I) and output (O). This guarantees that the test tool G∀st is able to test specifications of any type for state, input and output. The main test functions only impose some restrictions on these types. These restrictions guarantee for instance that these types can be printed (in order to generate traces) and that there is an equivalence relation to compare output elements. The algebraic data type `Trans` captures the fact that a specification yields a set of tuples or functions when it is given the current state and input. In the Clean representation of the transition functions we use lists of outputs, `[o]`, rather than single output elements, `o`, for two good reasons. First, this gives a compact and convenient notation for no output. Without the list we either had to extend each output type with an element indicating that there is no output, or we had to use a type like `Maybe o` which lifts the domain `o` with a `Nothing` value in case of no output, and `(Just o)` values otherwise. Second, in contrast to the `Maybe` type, the lists allow sequences of outputs which is often handy.

```
:: Spec s i o :== s → i → [Trans o s]
:: Trans o s =  Pt [o] s | Ft ([o]→[s])
```

We give some examples in section III below.

III. Examples

In this section we introduce two examples. These examples illustrate the kind of specifications used by G∀st and will be used in this paper to illustrate how these specifications can be transformed to give the desired properties and we show how properties of these specifications can be tested.

The first example consists of some variants of a vending machine that occurs in many papers about MBT. The second example is a larger and more realistic specification that models a system giving information about telephone numbers.

A. Example 1: Vending Machines

Figure 1 shows two extended state machines modeling vending machines. The global specification of these vending machines is that it can deliver tea or coffee after insertion of coins with a sufficient value, and pressing the correct button.

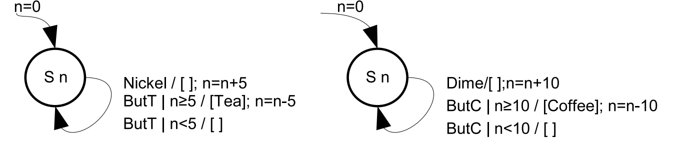

Fig. 1. The vending machine on the left delivers tea, the right one coffee.

An input is either a nickel, a dime, or pressing the tea or coffee button. The output is either tea or coffee. The state of the machine is recorded in the algebraic data type $State$, which records the amount of money inserted as an integer.

```
:: In    = Nickel | Dime | ButC | ButT
:: Out   = Coffee | Tea
:: State = S Int
```

The specification of both vending machines in G∀st is:

```
specC :: State In → [Trans Out State]
specC (S n) Dime   = [Pt [] (S (n+10))]   // insert dime
specC (S n) ButC                          // on pushing the coffee button:
    | n≥10 = [Pt [Coffee] (S (n-10))]     // produce coffee if sufficient balance
           = [Pt [] (S n)]                // do nothing if balance is insufficient
specC s i = []                            // otherwise no transition defined

specT :: State In → [Trans Out State]
specT (S n) Nickel = [Pt [] (S (n+5))]
specT (S n) ButT
    | n≥5  = [Pt [Tea] (S (n-5))]
           = [Pt [] (S n)]
specT s i = []
```

The last line (e.g. `specC s i = []`) of both specifications indicates that nothing is specified for other combinations of state and input. If nothing is specified for a combination of state and input any behavior is allowed, see section II-A. The existence of the alternative `[Pt [] (S n)]` for `(S n) ButC` that is chosen on insufficient balance is important. Without that alternative *anything* is allowed according to the conformance relation, including the delivery of the required product. By design both specifications are partial, e.g. `specC` does not specify what ought to happen on the inputs `Nickel` and `ButT`.

B. Example 2: Qui-Donc

This example is adapted from [16]. *Qui-Donc* (French for *who there?*) is a service of France Telecom that finds the person associated to a telephone number. We model the behavior of the Qui-Donc system by an ESM.

Initially the system is in the Wait state. When the user dials the number of this service, the user receives a Welcome message and the system goes to the Stars state. If the user does nothing for six seconds the system gets a TimeOut event. After the third TimeOut the system returns to the Wait state with NotAllowed message. After an other Timeout the system repeats the Welcome message. After entering a Star the system is in the Digits state and produces an Enter message. In this state the user can enter a maximum of 10 digits and a Hash (#). If the user waits too long the system receives a TimeOut and repeats the Enter message. After three time outs the call is terminated with a Bye message. If the user enters an emergency number

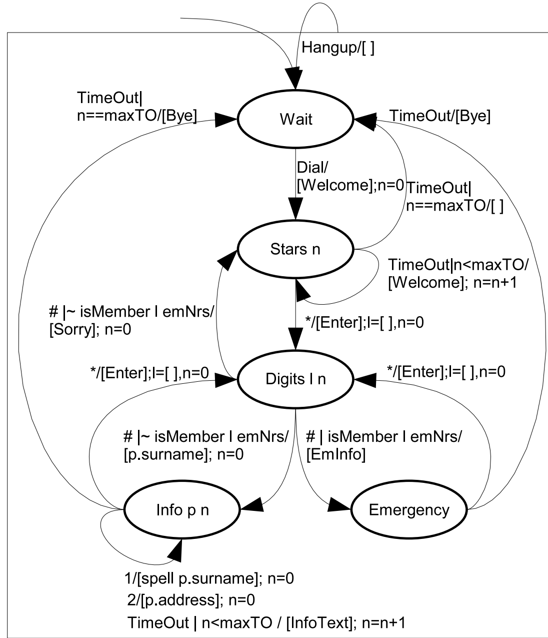

Fig. 2. State diagram of the Qui-Donc system.

terminated with a Hash the system gives an explanation and waits for a Star. After the Star the user can do a new search. For a known 10-digit number the system gives information about the owner of that number. For other numbers the system gives an Error message and waits for a new number. In any state the user can Hangup, and the Qui-Donc system returns to its Wait state. The ESM depicted in figure 2 is specified with G∀st in figure 3. A telephone `Number` is a type synonym for a list of integers. The timeout count, `TOcount` is an integer. The specification uses parameterized algebraic data types for the state of the system, its input and output:

```
:: Number  :== [Int]
:: TOcount :== Int

:: QDstate = Wait | Stars TOcount | Digits Number TOcount
           | Info (Maybe Person) TOcount | Emergency
:: QDin    = Dial | Star | Hash | Digit Int | TimeOut | Hangup
:: QDout   = Welcome | NotAllowed | Enter | Error | EmInfo | Sorry
           | QSstring String | Help | Addr Number | Bye | InfoText
```

This specification for the Qui-Donc system is nondeterministic. In state `Digits ds 10` two transitions are possible for input `Hash`. The result `Ft anyName` accepts any input. It models the situation that the number occurs in the database of the real Qui-Donc system but is not known by the specification, while `Pt [Sorry] (Digits [] 0)` models the situation that the given number does not occur in the database. The specification allows both possibilities since it does not know the contents of the database.

Compared with the FSM and EFSM specification in [16] our specification captures the behavior of the complete Qui-Donc system, while Utting's specification only captures the proposed tests. Since we use extended state machines rather than a FSM, our specification is more compact. For instance the FSM states Star1, Star2 and Star3 are all represented by our state `Star n`. The state `Digits Number TOcount` holds 10^{10} different phone numbers, each with 4 timeout count values.

```
QDspec :: QDstate QDin → [Trans QDout QDstate]
QDspec Wait          Dial      = [Pt [Welcome] (Stars 0)]
QDspec (Stars n)     Star      = [Pt [Enter] (Digits [] 0)]
QDspec (Stars n)     TimeOut
    | n < maxTO                = [Pt [Welcome] (Stars (n+1))]
                               = [Pt [NotAllowed] Wait]
QDspec (Digits [] n) TimeOut
    | n < maxTO                = [Pt [Enter] (Digits [] (n+1))]
                               = [Pt [Bye] Wait]
QDspec (Digits ds n) (Digit d)
    | 0≤d && d≤9 && length ds < maxDigits
                               = [Pt [] (Digits (ds ++ [d]) 0)]
QDspec (Digits ds n) Hash
    | isMember ds emergencyNrs = [Pt [EmInfo] Emergency]
    | length ds == maxDigits
        = case findPers persons ds of
            Just p  = [Pt [Str (p.surname)] (Info (Just p) 0)]
            Nothing = [Pt [Sorry] (Digits [] 0), Ft anyName]
        = [Pt [Error] (Digits [] 0)]
where anyName [Str s]          = [Info Anybody 0]
      anyName other            = []
QDspec Emergency  Star         = [Pt [Enter] (Digits [] 0)]
QDspec Emergency  TimeOut      = [Pt [Bye] Wait]
QDspec (Info p n) TimeOut
    | n<maxTO                  = [Pt [InfoText] (Info p (n+1))]
                               = [Pt [Bye] Wait]
QDspec (Info p n)        Star      = [Pt [Enter] (Digits [] 0)]
QDspec (Info Anybody  n) (Digit 1) = [Ft λlist → [Info Anybody 0]]
QDspec (Info (Just p) n) (Digit 1)
    = [P (spell (p.surname)) (Info (Just p) 0)]
QDspec (Info Anybody  n) (Digit 2) = [Ft λstr → [Info Anybody 0]]
QDspec (Info (Just p) n) (Digit 2) = [Pt [Str p.address] (Info (Just p) 0)]
QDspec state             Hangup    = [Pt [] Wait]
QDspec state             input     = []  // otherwise no transition defined
```

Fig. 3. Model of the Qui-Donc system for G∀st.

IV. Transformation of Specifications

In our approach specifications are functions. Functions are first class citizens in the functional programming language Clean. This enables the transformation of specifications (by transforming the functions representing them). These transformations are used to give specifications some specific desirable properties, like being total. As an example we show a function that joins two specifications. Using the appropriate instance of the operator `+` it is even possible to add specifications with an expression like `s+t`. The specification obtained by `s+t` contains the transitions in `s` as well as the transitions in `t`.

```
instance + (a→b) | + b where (+) f g = λx.f x + g x
instance + [a] where (+) l m = l ++ m
```

Using this operator we can define a vending machine `specV` that combines the behavior of `specT` and `specC`, or a vending machine `specC2` that allows the user to pay coffee with two nickels as:

```
specV :: Spec State In Out
specV = specT + specC

specC2 = specC + nickel

nickel (S n) Nickel = [Pt [] (S (n+5))]
nickel s     i      = []
```

The machine modeled by `specV` is able to produce coffee as well as tea. Note that `specV` also allows the user to pay coffee with two nickels and is able to produce two teas for a dime. Even if the user restricts herself to tea, the behavior of `specV` is different (allows paying with dimes) from `specT`. The combined system `specV` has the same type of states (`State`) as its components `specT` and `specC`, this is exactly what we want. In the traditional composition of FSMs [17], the new state is usually the product of the old states, here this would be `(State, State)`

In a similar way one can define the difference of specifications, that is all behavior of one model that is not shown by

another model.

As defined above, a specification is total if it contains at least one transition for each state and input. The default way to make a specification total is to add a transition with an empty output sequence that preserves the state if nothing else is defined. Using a higher order function it is easy to define a function that makes any given specification total:

```
mkTotal :: (Spec s i o) → Spec s i o
mkTotal spec = λs i.case spec s i of
                    [] = [Pt [] s]
                    t  = t
```

It might be tempting to define `mkTotal spec = spec + λs i.[Pt [] s]`, but that adds the do-nothing transition to each and every state. In a similar way we can make a specification deterministic by selecting only the first state if there is more than one target state. In [11] we use this technique to transform specifications of thin client web-applications without browser navigation to specifications that cover browser navigation.

Apart from transforming specifications in order to give it desirable properties such as totality and determinism, it is also possible to *test* whether a specification possesses these properties. We will use the possibilities of G∀st to test logical properties. Testing logical properties is briefly revisited in section V. In section VI we use these techniques to verify properties of specifications.

V. Testing Logical Properties

Not all desirable properties of specifications can be guaranteed by a transformation of the specification. Nevertheless, we often want to know whether a specification possesses a property (e.g. like being deterministic). We will show that automated systematic testing can reveal whether a specification possesses a property and give examples for several important properties. In this section we review automated systematic testing introduced in [12].

Apart from state machine based testing, G∀st is also able to test logical properties like $\forall x : \mathbb{N}. x + 1 > x$. In G∀st such a property is represented by a function. The function arguments represent its universally quantified variables. For our example we use the type `Int` to represent $\mathbb{N}$. This implies that the predicate has type `Int → Bool`. The body of the function representing this property is just `p x = x+1 > x`.

A universally quantified property is tested by evaluating the corresponding function for a large number of arguments. The function `test` in G∀st initiates testing. A simplified, but correctly executing, version of this function is:

```
test :: p → Bool | holds p
test p = and (take MaxTests (holds p))

class holds a :: a → [Bool]
instance holds Bool where holds b = [b]

instance holds (a→b) | gen{|*|} a & holds b // Forall
where holds p = diagonal [holds (p a) \\ a←gen{|*|} ]

:: For a b = FOR infix 0 (a→b) [a]        // the infix for operator
instance holds (For a b) | holds b         // for a given test suite
where holds (For p t) = diagonal [holds (p a) \\ a←t]
```

The text in the type signatures after | are context restrictions. Identifiers followed by {|∗|} refer to instances of generic functions, see appendix, for the given type. Identifiers without such suffix refer to 'ordinary' overloaded functions. An expression like `[f a b\\a←l, b←m|p a b]` is a *list-comprehension*, it computes the list of all values `f a b` for all values `a` coming from list `l` and `b` from list `m` that satisfy the predicate `p a b`. There can be any positive number of generators (of the form `a←l`) and the predicate is optional.

The function `test` takes `MaxTests` elements from the list of booleans produced by `holds p`. The function `and` yields true if all these booleans are true and false otherwise. Here we need only three instances of the class `holds`. The instance for booleans just yields the list containing that boolean. The instance for a function corresponds to a universally quantified property. The list of test values is generated by `gen{|*|}`, see appendix. The final instance uses the given test suite instead of the one generated by `gen{|*|}`. The property is applied to each of these test values, `p a`. We apply `holds` on this result; either a boolean value, or another universally quantified argument. The function diagonal takes a fair mix of values from multiple generators for properties with more than one generator, like $\forall x. \forall y. x + y = y + x$.

By default the number of tests, `MaxTest`, to be done is 1000. The actual test algorithm gives more information than just true or false (Fail). The result is Proof if the property holds for all elements in the generated test suite (proof by exhaustive testing). The value Pass indicates that the property holds for the first `MaxTest` values, there are untested values in the test suite. If the result is Fail, G∀st also indicates the number of tests done and the test values that cause the counterexample found. Apart from these universal quantified properties G∀st contains a library with all operators from first order logic. As example, to test the property `p` for all integers from 1 to 100 we execute:

```
Start = test (p For [1..100])
```

The result is `Proof: success for all arguments after 100 tests`. When we want to test the property for an unbounded number of odd values we evaluate `test (p For [1,3..])`. The result is `Passed after 1000 tests`. If we have no strong opinion about the test values to be used we can leave it to the generic algorithm of G∀st to generate the test values by executing `Start = test p`. Now the result is `Counterexample 1 found after 5 tests: 2147483647`. The property `p` holds in mathematics, but not for the finite precision approximation of integers used in computers. The counterexample found is maxint in 32-bit integers. Because values such as 0, 1 and maxint often cause problems in properties, these test values are generated quickly by G∀st.

VI. Testing the Quality of Specifications

In this section we show how automated systematic testing as introduced in the previous section can be applied to models of state machines to determine whether relevant properties known from the literature holds for these specifications. This analysis of specifications can reveal errors in the specification, or increase the confidence in its correctness and consistency.

In the next subsection we define the generation of states and inputs used in the tests. Since an ESM used as specification can have an infinite number of states, it is convenient to define

equivalence classes. The behavior of the specification in only one state in such an equivalency class needs to be tested. In subsequent subsections testing of a number of well known properties is discussed.

A. State Space Equivalence Classes

For many specifications the state space or the number of possible input values is unbounded. Tailor made definitions of the generation of instances for these types makes the tests more effective and efficient. For instance we can consider only the states with multiples of 5 cents for the vending machine and obtain a sensible maximum amount of 200 cents in the machine by:

```
gen {|State|} n r = [S i \\ i←[0,5..200]]
```

Similarly, in the tests of the Qui-Donc specification we use only states with digits that are on track to one of the few known numbers, the emergency number, or the invalid number that contains ten zeros. This is by no means a restriction of G∀st, it is used as an illustration. The inputs for the vending machines can just be derived by `gen`. For the inputs of Qui-Donc we make sure that only the numbers `[0..9]` are used as argument for `Digit`.

A key idea to test many properties effectively is to define equivalence of states and inputs for the tests. In the tests we try to take at least one representee of each (important) equivalence class. This approach is inspired by abstract interpretation in proof systems [4] and collapsing of states in model checkers like Spin [6] and Uppaal [2].

```
class equiv a :: a a → Bool
```

For example, in vending machines all states that represent more than 200 cents of inserted money are considered to be equivalent. These states represent the situation that relatively much money has been inserted.

```
instance equiv State where equiv (S n) (S m) = n > 200 && m > 200 || n == m
```

We will use this equivalency of states in Sect. VI-E and Sect. VI-G. In the rest of this section we discuss a number of properties of state machines known from literature and how these properties can be tested by G∀st.

B. Testing whether a Specification is Total

As defined above, a specification is total if a transition is defined for each combination of input and output. We can easily test this by requiring that the number of transitions should not be empty:

```
isTotal :: (Spec s i o) s i → Bool | gen {|*|} s & gen {|*|} i
isTotal spec s i = ~(isEmpty (spec s i))
```

We test the coffee vending machine for this property by evaluating `test (isTotal specC)`. The test result is `Counterexample 1 found after 4 tests: (S 0) ButT`. When we make the specification complete by `mkTotal` as defined above we test `isTotal (mkTotal specC)`. Testing yields `Proof: success for all arguments after 164 tests`. Since we use `mkTotal` this is hardly a surprise, it is more a check of the specification transformer `mkTotal`. Testing whether the combined vending machine is total by evaluating `test (isTotal specV)` gives an identical proof result, which is less evident. The proofs are only possible due to the finite number of states generated. When generation of states would have been derived the result of the last test is `Pass`.

C. Testing whether a Specification is Deterministic

A specification is deterministic if there is at most one output and target state defined for each combination of state and input. By design G∀st can handle nondeterministic specifications. Even deterministic systems can be specified by nondeterministic specifications due to lack of information in the specification. The Qui-Donc system is supposed to be deterministic, but its specification is nondeterministic because we do not model the database containing the number information. This implies that the specification has to allow the situation that the number is known as well that it is unknown to the system.

Nevertheless, it can be important to know whether a specification is deterministic or not. It is tempting to test this by requiring that the length of the yielded list of transitions, `[Trans o s]`, is at most one. However that is too simplistic for two reasons. First, a yielded transition can be a function, `Ft f`. Such a function `f` is supposed to accept any output as argument. Hence it cannot be deterministic. The second problem by testing the length of the list of resulting transitions is that it can contain the same pair of output and target state (`Pt o t`) twice, for instance by composing specifications as in `specV+specT`. So, a good test for checking whether a specification is deterministic verifies that all pairs in the list of transitions are identical and that this list does not contain functions:

```
isDeterministic :: (Spec s i o) s i → Bool | gEq {|*|} o & gEq {|*|} s
isDeterministic spec s i = unique (spec s i) Nothing
where
    unique []       pair     = True
    unique [Ft f:r] pair     = False
    unique [p]      Nothing  = unique r (Just p)
    unique [p:r]    (Just q) = p === q && unique r (Just p)
```

Using this property G∀st proves that `specT`, `specC`, `specV`, and `specT+specV` are deterministic. Even testing the combination of `specC2` and `specT` for determinism yields proof, although the specification contains the transition for a nickel twice. The Qui-Donc specification is not deterministic, the first counterexample found by G∀st is `Counterexample 1 found after 182 tests: (Info Nothing 1) (Digit 1)`.

D. Testing whether a Specification is Consistent

Transitions of the form `Ft f` deserve some special attention. The function `f` has type `[o]→[s]`, that is given an output (list of element of type `o`) it yields the list of allowed target states of type `s`. For a successful conformance test the list of target states is supposed to be nonempty. It is tempting to require that the list of target states of all functions of this kind is nonempty, but that is too restrictive. It should be allowed to have a specification of the form `spec s i = [Ft f,Ft g]`, where the functions `f` and `g` together cover all possible outputs. The results of `f` and `g` for individual arguments can be empty. Moreover, returning an empty list is a convenient way to indicate a nonconformance of the sut.

So, making it impossible to return an empty list is too restrictive, but it is interesting to know if a given specification yields an empty list of target states for some state, input and output. This can be tested by the property `isConsistent`.

```
isConsistent :: (Spec s i o) [[o]] s i → Property | TestArg o
isConsistent spec outputs s i
```

```
 = case [f \\ Ft f ← spec s i] of
     [] = prop True
     fs = defined (foldl1 (+) fs) For outputs
where defined f o = ~(isEmpty (f o))
```

Testing the vending machines yields pass. One can argue that the tests containing multiple products as output are not very useful. We can test only single product output by evaluating `test (isConsistent specT [[Coffee],[Tea]])`. G∀st proves this property in 164 tests. Since these specifications do not contain functions, these test results are not very exciting. A tea vending machine containing a function is:

```
specT2 :: State In → [Trans Out State]                              1
specT2 (S n) Nickel = [Pt [] (S (n+5))]                              2
specT2 (S n) ButT   = [Ft λo.if (n≥5 && o == [Tea]) [S (n−5)] []]     3
specT2 s     i      = []                                             4
```

Testing this specification for consistency by evaluating `test (isConsistent specT2)` finds a counterexample for the arguments `(S 20) ButT []` after 4 tests. This test result indicates that the transition on line 3 yields an empty set of target states starting from state `(S 20)` on input `ButT` and the empty output. The specification can be corrected by replacing the empty list (`[]`) on line 3 by `[S n]`: for this state and input the state should be unchanged on an empty output.

Testing the Qui-Donc specification for consistency yields counterexamples for outputs that are not a single string. We restrict the tested outputs to some known names by letting G∀st evaluate `test (isConsistent QDspec [[Str "Koopman"], [Str "Plasmeijer"], [Str "Achten"]])`. The result of this test is `Proof` after 644 tests.

E. Testing the Reachability of States

The reachability of specific states can reveal important information about a specification. The vending machines will not work as desired if they cannot reach a state where an item can be the output. Similar Qui-Donc will not work properly if the state `Info` cannot be reached. Reachability of states is a typical property that is usually verified by model-checkers.

The function `reachable` yields the list of all states that are reachable in `n` steps according to specification `spec` starting from `states`. This function does a breadth first search of the state space of the specification. Since many specifications have a huge state space (e.g. above 10^{10} for Qui-Donc), we provide some help to limit the number of states. The argument `eq` defines equality on states to remove similar states as introduced at the start of this section. The function `input` defines the inputs to be used at the given state. Similarly, the function `out` generates the outputs for a transition of the form `Ft f` in the given specification.

```
reachable :: Int (Spec s i o) [s] (s s→Bool) (s→[i]) (s i→[[o]])
             (s→Bool) → [s]
reachable n spec states eq input out pred = states ++ r n states []
where
  r 0 states seen = []
  r n states seen = new ++ r (n−1) new (new ++ seen)
  where new = rmEquiv eq [t \\ s ← states, i ← input s, p ← spec s i
                              , t ← targets p s i out
                              | ~(member eq seen t) && pred t]
```

We use auxiliary functions `member::(x x→Bool) [x] x → Bool` and `rmEquiv::(x x→Bool) [x] → [x]`. Both functions are parameterized by an equivalence relation of elements. The function `member` checks if the given element occurs in the given list. The function `rmEquiv` removes elements from the given list that are equivalent to an element previously occurring in that list.

For the Qui-Donc specification we have lists `listQDin` of all interesting inputs, and `listQDstate` of the interesting states. To determine reachabilty in Qui-Donc we try all inputs in each state by using the anonymous function `(λs.listQDin)` as the argument `input` of `reachable`. We use only states that are interesting by the predicate `member equiv listQDstate`. Using this we can verify that the state `Emergency` (corresponding to the number 112) is not reachable in 5 steps, but is reachable in 6 steps.

F. Testing whether a Specification is Initially Connected

A state machine is initially connected if each state can be reached from the initial state. Testing the reachabilty for all interesting states can be done by the function `reachable` from the previous subsection. It is more informative to know which states are unreachable than just the information that a specification is initially connected or not. The unreachable states from the given list of `allstates` are delivered by:

```
unreached :: (Spec s i o) [s] (s s→Bool) (s→[i]) (s i→[[o]])
             (s→Bool) [s] → [s]
unreached spec states eq input out pred allstates
 = [ x \\ x ← allstates | ~(member eq reachedStates x)]
where reachedStates = reachable 100000 spec states eq input out pred
```

Testing initially connectedness is now just checking whether the list of unreached states is empty. All specifications in this paper appear to be initially connected for the states considered. It is clear that `specC` cannot reach states like `S 5` and `S 15`, since it accepts only dimes. This is found promptly by G∀st.

G. Testing whether a Specification is Strongly Connected

A state machine is strongly connected if every state can be reached from every other state. This is a rather strong restriction on state machines that is sometimes unwanted. If we know already that a state machine is initially connected it is strongly connected if the initial state is reachable from every other state.

```
connected :: (Spec s i o) s (s s→Bool) (s→[i]) (s i→[[o]]) (s→Bool) [s]
             → Bool
connected spec initState eq input out pred allstates
 = and [member eq (reachable 10000 spec [s] eq input out pred) initState
       \\ s←allstates]
```

All state machines treated in this paper appear to pass the test for strong connectedness if we restrict ourselves to interesting states that are equivalent.

VII. Domain Specific Properties

The properties handled in the previous section are problem independent in the sense that we can determine for each state machine whether these properties hold. With some effort one can also determine problem specific properties and test whether these properties hold for the specification at hand. Testing the specification for such a property increases the confidence in its correctness. We illustrate this with some simple examples.

We require that our vending machines are fair: they should not lose money. This implies that the amount of money `n` in state `S n` is a fair representation of the difference between the amount of money inserted and the value of all products

obtained. Instead of checking this property for many traces, we check it for all transitions. If it holds for all transitions and the initial state `S 0`, it will hold forever by induction. First we define a class `value` and the appropriate instances. The key instances cover the input `In` and output `Out` of the specification.

```
instance value State where value (S n) = n
instance value In    where value i = case i of Nickel = 5; Dime = 10; _ = 0
instance value Out   where value o = case o of Coffee = 10; Tea = 5
```

Some additional instances are needed, e.g. to compute the value of a list of elements. Now the key property is easily defined. For all inputs and outputs of a given specification the value of the input and the current state has to be equal to the value of each transition (i.e. output plus target state) that is defined by the specification. This is expressed by:

```
propFair :: (Spec State In Out) State In → Property
propFair spec s i = p For spec s i where p t = value s + value i == value t
```

Testing with this property shows that `mkTotal` is not as innocent as it looks. If we test `mkTotal specC`, the first counterexample `(S 5) Nickel (Pt [] S 5)` is found after 5 tests: the user inserts money but the state remains unchanged. The specification `mkTotal specC` is unfair!

In the same way we define a property that states that the amount of money in all reachable target states is non-negative when we start with a non-negative amount of money.

```
propNoDebit :: (Spec State In Out) State In → Property
propNoDebit spec s=:(S n) i = n≥0 ⟹((λ(Pt o (S m)).m≥0) For spec s i)
```

For the states generated as described above, G∀st proves this property. However, if we leave the generation of states to G∀st by stating **derive** `gen State`, G∀st finds the counterexample `2147483647 Dime (Pt [] S −2147483639)` after 6 tests. The problem is caused by integer overflow in the counter of the state.

VIII. Finding domain specific properties

Above we have shown that it is possible to find issues in a state based specification by testing logical properties of such a specification. Some of these properties are universal and can be used for (almost) any specification, these properties are typically looked up in textbooks. This leaves us with the problem of finding domain specific logical properties. In our experience an effective way to obtain such domain specific properties for specifications is by simulating or inspection of the specification. When we manually find an incorrect transition, we can often find a domain specific logical property of the specification by generalization of the behavior we require instead of the erroneous transition.

The esmViz tool [10] can simulate and visualize the models used by G∀st for MBT. Using this tool we can interactively generate an expanded state diagram of the specified extended state machine. In a diagram of an *extended* state machine all states that differ only in a parameter are mapped to the same node in the graph, e.g. the states `S n` in figure 1 and the states `Digits l n` in figure 2. In an expanded state diagram states that differ in only a parameter are drawn as separate nodes, e.g. the states `On 20` and `On 30` in figure 4.

We demonstrate this with the specification of a vending machine that should be able to produce tea for 10 cents, coffee for 20 cents and chocolate for 30 cents. The user can insert coins up to a value of 40 cents. A specification containing almost as many nasty errors as we could slip in is:

```
:: State   = Off | On Int
:: Input   = SwitchOn | SwitchOff | Coin Int | But Product
:: Product = Coffee | Tea | Chocolate
:: Output  = Cup Product | Return Int

vSpec :: State Input → [Trans Output State]
vSpec Off SwitchOn  = [Pt [] (On 0)]
vSpec s   SwitchOff = [Pt [] Off]
vSpec (On s) (Coin c)
 | s<Max    = [Pt [] (On (s+c))]
            = [Pt [] (On s)]
vSpec (On s) (But coffee) | s≥20  = [Pt [Cup Coffee] (On (s−20))]
vSpec (On s) (But Tea)    | s≥10  = [Pt [Cup Coffee] (On (s−10))]
vSpec (On s) (But p)              = [Pt [] (On s)]
vSpec state input = []

Max = 40
```

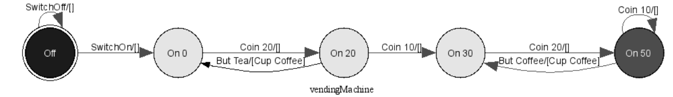

Fig. 4. Expanded state diagram of `vSpec` as generated by esmViz.

An expanded state diagram of this machine is depicted in figure 4. The transitions shown in this diagram correspond to inputs chosen by the user of the tool esmViz. Hence, the expanded state transition diagrams are often incomplete. This has the advantage that they are concise and the user can focus better on specific behavior. Based on a specification like `vSpec` the tool presents the inputs allowed in the current states, and draws an expanded state diagram of the transitions chosen by the user. If we look carefully at the transitions in this diagram we see some issues.

1) Wrong product: In figure 4 the delivered product in transition $\text{On 20} \xrightarrow{\text{But Tea}/[\text{Cup Coffee}]} \text{On 0}$ is not the required product. This is caused by the variable `coffee` in the function alternative `vSpec (On s) (But coffee)`. Here the constructor `Coffee` (starting with an upper case) was needed. As a general property we state that the delivered product should always be equal to the required product:

```
pProduct :: State Product → Property
pProduct s p = checkProd For vSpec s (But p)
where checkProd (Pt [Cup q] t) = p == q
      checkProd _               = True
```

Testing this reveals many issues containing `Tea` or `Chocolate` as required product and states with a values `On n` with `n≥20`.

2) All products: Given this erroneous transition, we might wonder if the specified machine is able to produce tea. More general: we expect that for all products `p` in the type `Product` the machine is able to produce a cup of that product for the input `But p`.

```
pAllProducts :: Product → Property
pAllProducts p
 = Exists λs. ~(isEmpty [p \\ (Pt [Cup q] t) ← vSpec s (But p)|p == q])
```

In this property we use the keyword `Exists` to indicate an existentially quantified property (∃), rather than an universally quantified property (∀). G∀st produces the counter examples `Tea` and `Coffee`. The test suite is generated using the generic algorithm by **derive** `gen Product`, hence it is known to contain all possible products.

3) Losing money: Note that the machine loses the value of the input coin in the transition On 50 $\xrightarrow{\text{Coin 10/[]}}$ On 50. As a general version of this issue we state that all transitions must be fair as introduced in section VII. Also for this property G∀st finds 12 issues. They correspond to **1)** the transition for vSpec (On s) (Coin c) where $s \geq$ Max (the unfair transition observed). **2)** the second class of issues is caused by switching the machine off in a state On s with s>0. **3)** also the production of coffee instead of tea is caught by this property since the value of coffee, 20, is unequal to the amount of the state change (10).

4) Illegal states: From the presence of a state On 50 in this diagram we conclude that it is possible to reach a state with too much money in the machine. The general property states that for all reachable states, the value of the target state after a transition is less or equal to Max. We can easily generate only allowed states by:

```
gen {|State|} = [Off: [On v \\ v ← [0,5..Max]]]
```

Exactly the same approach is used to verify that the amount of money in the machine is never negative as demonstrated in the previous section.

A. An improved specification

An improved version of this specification reads:

```
vSpec2 :: !State !Input → [Trans Output State]
vSpec2 Off SwitchOn  = [Pt [] (On 0)]
vSpec2 s   SwitchOff = [Pt [Return (value s)] Off]
vSpec2 (On s) (Coin c)
        | s+c≤Max    = [Pt [] (On (s+c))]
                     = [Pt [Return c] (On s)]
vSpec2 (On s) (But p)
        | s≥value p  = [Pt [Cup p] (On (s-value p))]
                     = [Pt [] (On s)]
vSpec2 state input   = []
```

Within a split second G∀st *proves* that all listed properties hold for this specification even if we enlarge Max to 4000.

We have successfully used the same approach for the Qui-Donc system. This system has more than 10^{10} states. The expanded state diagram becomes completely unreadable if we try to draw a significant fraction of these states. Fortunately it is not needed to draw large amounts of states. To our rescue we noted that all interesting traces contain about 20 transitions or less. For these experiments it appeared useful, but not essential, to use 5-digit internal phone numbers rather than the 10-digit numbers used in the full specification. The esmViz tool contains several operations to prune the obtained diagram in order to keep it small and clear.

IX. Related work

It is widely recognized that the quality of specifications in software engineering is important and not self-evident. Using MBT to determine the quality of models however is rare. Back in the 80's there was some initial work [8], [9], but after that people seem to rely on inspection by humans (as part of the pretest quality assurance) or model checkers like Uppaal [2]. Using a proof system requires a transformation of the G∀st-model to a format understood by the model checker. Usually the model must be simplified to enable the model checker to prove the specified properties. Our approach has as advantage that the same model can be used for MBT of a sut and as subject of testing properties. For finite cases G∀st is able to produce a proof by exhaustive testing. If the search space is too large for exhaustive testing, our test system can still increase the confidence by doing many useful tests.

An ESM used as specification can be depicted as UML state chart. This is fine to get an overview, but it is cumbersome to make such a state chart a complete specification. None of the UML tools offers an expressive power similar to Clean. Hence the state chart has to be restricted, or entered as free text. Moreover, specifications in UML cannot be changed and composed like the models in G∀st. ProB [14] is able to verify properties like consistency and refinement of specifications in the language B by automatic testing. In our approach it is much easier to add domain specific properties, and our specifications can directly be used in simulations and to test the sut.

The Quickcheck [3] tool is only able to test logical properties, not the state based systems used here. Also Alloy [7] handles logical properties.

X. Conclusions

MBT of state based systems often reveals issues in the system under test as well as in the model used as basis for testing. This is not very strange: the software and its specification are similar formal artifacts, and it is known that humans do make errors in creating them. High level languages and analysis, like static type systems, reduce the number of errors, but do not eliminate them completely. If the tested system and the model do not contain the same error an issue is found during the test. This implies that not all errors in the specification pass unnoticed. But, late detection of errors in the specification can delay the software process. Even if the errors in a specification are found, it hampers progress. So, it is worthwhile to spend additional effort in improving and verifying the quality of specifications. As a rule of thumb test managers say that 40% of the issues found in automatic testing (executing scripts) correspond to errors in the sut. In our Model-Based testing experience with G∀st this is on average 75% which is already an important improvement. Using the techniques introduced here we are able to improve our models significantly. This should increase the fraction of issues that indicate errors in the sut, but we have not yet enough experience to give figures.

In this paper we have shown how we can give specification properties like being total by specification transformations. More important, we show how the logical branch of the test system G∀st can be used to express desirable properties of specifications for state based systems. Using some examples we demonstrated that issues in such a model can indeed be found by testing. Manual verification is still needed to detect domain specific issues. If we generalize these issues to constraints that should hold, G∀st is able to spot similar issues quickly and accurately.

Our approach heavily builds on modeling specifications as functions in a functional programming language. The advantages of this approach are clear semantics, concise specifications, the language compiler checks many aspects of the

specifications. In this paper we have shown that it is possible to check additional properties of specifications, domain specific as well as more general properties, within the same framework. There is empirical evidence that people without any back ground in functional programming are able to write design required properties, implement them in the test tool and find issues within two weeks.

APPENDIX

GENERIC PROGRAMMING

Generic programming [5], [1] enables us to write algorithms once and then use these algorithms for any type. This technique builds on a uniform representation of types within the language and compiler generated transformation to and from that representation. The minimal set of types needed for the generic representation of any type is:

```
:: UNIT        = UNIT                  // any constructor
:: PAIR   x y = PAIR x y              // glue types x and y together
:: EITHER x y = LEFT x | RIGHT y      // a choice between types x and y
```

The real implementation of generics in Clean uses some additional constructs to represent information about types and constructors. These are not essential to understand generic programming. As example we consider a user defined polymorphic data type `List` defined as:

```
:: List x = Nil | Cons x (List x)
```

The generic system generates a consistent representation of the constructors `Nil` and `Cons` using the generic types defined above. The generic list, `Listg`, is just the choice between these constructors (`Nilg` and `Consg`).

```
Nilg       = LEFT UNIT
Consg a x = RIGHT (PAIR a x)
:: Listg x = EITHER UNIT (PAIR x (List x))
```

The ubiquitous generic programming example is equality. If we define equality for the generic types in the obvious way, Clean can derive equality for lists by first transforming the lists to their generic representation and then compare these generic representations. A more interesting example is the generation of lists of the inhabitants of a type. We start out by defining the generic type and instances for the basic generic types defined above.

```
generic gen a :: [a]

gen {|UNIT|}              = [UNIT]
gen {|PAIR|}    xs  ys = map (λ(x,y).PAIR x y) (diag2 xs ys)
gen {|EITHER|}  xs  ys = fuse True xs ys
fuse True   [x:xs] ys = [LEFT  x:fuse False xs ys]
fuse False xs [y:ys] = [RIGHT y:fuse True  xs ys]
fuse b      [] ys      = map RIGHT ys
fuse b      xs []      = map LEFT  xs
```

The only possible element of type `UNIT` is the constructor `UNIT`. Hence the generic generator `gen` generates a singleton list containing only this constructor. For a `PAIR` we need to combine elements from two lists. The generic system provides these lists as the arguments `xs` and `ys`. Using the library function `diag2` we combine these lists in a breadth-first way. For the type `EITHER` we take elements from both provided lists in turn. We use a boolean to indicate the turn. After these definitions we can *derive* the generation of elements of type `List` as:

```
derive gen List
```

For basic types like `Bool` we have to specify the possible values.

```
gen {|Bool|} = [False, True]
```

After these preparations the Clean system is able to generate Lists of Booleans. These `(List Bool)` values are used by G∀st to test properties over these lists. The initial fragment of this list is: `[Nil, Cons False Nil, Cons True Nil, Cons False (Cons False Nil),...`

REFERENCES

[1] A. Alimarine and R. Plasmijer. A generic programming extension for clean. In T. Arts and M. Mohnen, editors, *Proceedings of the 14th International Workshop on Implementation of Functional Languages, IFL 2001*, pages 257–278, Stockholm, Sweden, Sept. 2001. Ericsson Computer Science Laboratory.

[2] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[3] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, pages 268–279. ACM Press, 2000.

[4] M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers - Sparkle: A functional theorem prover. In T. Arts and M. Mohnen, editors, *The 13th International Workshop on Implementation of Functional Languages, IFL 2001, Selected Papers*, volume 2312 of *LNCS*, pages 55–72, Stockholm, Sweden, 2002. Springer.

[5] R. Hinze. A new approach to generic functional programming. In *The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. Boston, Massachusetts, January 2000.

[6] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[7] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2006.

[8] C. Jard and G. v. Bochmann. An approach to testing specifications. *SIGSOFT Softw. Eng. Notes*, 8(4):53–59, 1983.

[9] R. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, 11(1):32–43, 1985.

[10] P. Koopman, P. Achten, and R. Plasmeijer. On the Validation of Specifications used in Model-Based Testing. In O. Chitil, editor, *Proceedings Implementation and Application of Functional Languages, 19th International Symposium, IFL 2007*, volume Technical Report of *No. 12-07*, pages 230–231, Freiburg, Germany, September 27-29 2007. Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK.

[11] P. Koopman, P. Achten, and R. Plasmeijer. Model-Based Testing of Thin-Client Web Applications and Navigation Input. In *Proceedings of the Tenth International Symposium on Practical Aspects of Declarative Languages PADL 08*, San Francisco, USA, January 7-8 2008.

[12] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In R. Peña and T. Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 84–100. Springer, 2003.

[13] P. Koopman and R. Plasmeijer. Testing reactive systems with GAST. In S. Gilmore, editor, *Trends in Functional Programming 4*, pages 111–129, 2004.

[14] M. Leuschel and M. Butler. ProB: An Automated Analysis Toolset for the B Method. *Journal Software Tools for Technology Transfer*, 2008.

[15] R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.ru.nl/∼clean/.

[16] M. Utting and B. Legeard. *Practical Model-Based Testing*. Morgan Kaufmann, 2006.

[17] W. T. W. P. Wagner F., Schmuki R. *Modeling Software with Finite State Machines: A Practical Approach.* CRC Press, 2006.