Fully Automatic Testing with Functions as Specifications

Pieter Koopman and Rinus Plasmeijer

Institute for Computing and Information Science,
Radboud University Nijmegen, The Netherlands
{pieter,rinus}@cs.ru.nl

Abstract. Although computer systems penetrate all facets of society, the software running those systems may contain many errors. Producing high quality software appears to be difficult and very expensive. Even determining the quality of software is not easy. Testing is by far the most used way to estimate the quality of software. Testing itself is not easy and time consuming.

In order to reduce the costs and increase the quality and speed of testing, testing should be automated itself. An automatical specification based test tool generates test data, executes the associated tests, and makes a fully automatically verdict based on a formal specification. Advantages of this approach are that one specifies properties instead of instances of these properties, test data are derived automatically instead of manually, the tests performed are always up to date with the current specification, and automatic testing is fast and accurate.

We will show that functions in a functional programming language can be used very well to model properties. One branch of the automatic test system GAST handles logical properties relating function arguments and results of a single function call. The other branch of GAST handles specifications of systems with an internal state.

1 Introduction

Testing is still by far the most used method to judge the quality of software. Human testing of software is error-prone, dull, and expensive. Systems are becoming larger and more complex, and hence harder to test. Moreover, the time to market should be reduced, which limits the time for testing. Hence, many approaches to automatic software testing are proposed and used.

Most test systems execute a fixed number of human specified tests. These specified tests are specified either in code, as in JUnit [1], or by a capture and playback tool. We focus on automatic test systems that generate test cases based on a formal specification, e.g. [2]. Instead of specifying a fixed number of fixed values and the expected response, one specifies a relation between input and output that holds for all arguments. Apart from generating the test cases, the test system also executes the tests, and makes a verdict based on the test results.

Advantages of generating test cases from the specification are that a change of specification do not invalidate the test script: it is generated from the updated specification. Increasing the number of tests in order to improve the confidence just requires the change of a parameter.

In [3–6] it is shown that a functional programming language, as used by the test tool Gast, is an excellent carrier for the formal specifications needed. Gast is able to handle two kind of specifications: logical properties about (a combination of) functions, and specifications of reactive systems by Extended State Machines, ESMs. The reactive systems are specified by a, potentially infinite and nondeterministic, state transition function. Those specifications can be partial.

The quality of the test is dependent on the quality of the specification. Obviously, aspects that are not specified cannot be tested by a system that generates tests based on the specification. Experience shows that writing formal specifications is a very useful activity on its own. Most inaccuracies and misconceptions in the requirements are discovered during the construction of the specification.

Nevertheless, experience shows that a significant number of issues raised during testing is caused by inaccuracies in the specification. A better specification increases the speed of testing the product and improves the value of the test results. One can validate the specification to see if it captures the informal requirements correctly. This is a human activity that can be supported by tools. On the other hand one can also verify the consistency of the formal specification. This is usually done by inspections, or by verifying properties of the formal specification using a model checker or theorem prover. Especially for specifications that are heavily data dependent, model checkers have troubles with automatic verification.

In this paper we show that consistency properties of ESM-specifications can also be tested fully automatically using the logical branch of our test tool Gast. Advantages of this lightweight and effective method to improve the quality of specifications are that it do not require the transformation of specifications, errors are reported in terms of the original model, and last-but-not-least it works also very well for models that are strongly dependent on sophisticated data types. The limitation is that the test results are usually weaker than the results of a complete formal verification (if it is possible).

The testing of individual functions is discussed in section 2. The specification and testing of reactive systems in the next section. The testing of specifications in order to verify their quality, is discussed in section 4. Finally, we will discuss related work and draw conclusions.

2 Testing Functions with First Order Logic

The relation between input and output of a single function can be conveniently specified in predicate logic. As an example we consider a function that takes a string and a character as argument and yields a sorted list of the indices for all occurrences of the given character in the string, e.g. $indices(''A\ test'','t')$ should yield the list $[2, 5]$. We can specify the result of this function in at least two ways: we can give a reference implementation (perhaps very inefficient, but obviously correct), or we can state a property about the resulting list on indices.

Specification The declarative specification gives a property of all indices in the sequence yielded by *indices*:

$$s \in \mathit{String},\ c \in \mathit{Char},\ i \in 0..\#s-1 \bullet \mathit{isMember}(i, \mathit{indices}(s,c)) \Leftrightarrow s[i] = c$$

The function *isMember* checks if the element occurs in a list, and $\#s$ is the length of the string s. The characters are numbered from 0 to $\#s - 1$ in $s[i]$ which indicates string subscription.

This property states that an index i is part of the result of $\mathit{indices}(s,c)$ if and only if $s[i] = c$. The universal quantifiers over c and s are often omitted, as is usual in logic.

Every logical property is transformed to a function. The name of the function is used as reference to the property. The function arguments are interpreted as universal quantifiers. GAST provides a complete set of logical operators including $\forall$, $\exists$, $\vee$, $\wedge$, $\neg$, $\Rightarrow$ and $\Leftrightarrow$. The property state above is expressed in CLEAN as:

```
propIndices :: String Char → Property
propIndices s c = p For [0..size s-1]
where p i = isMember i (indices s c) ⟺ (s.[i] == c)
```

The operators **For** and ⟺ in this function definition is provided by the GAST-library. In this situation it is safe to replace ⟺ by the equality ==. For implementation reasons the result of a specifying function is `Property` instead of `Bool`, if the specification contains a test operator like ⟺.

Note that this specification does not specify that the resulting list ought to be sorted, e.a. a function that yields $[5,2]$ as result to $\mathit{indices}("A\ test",'t')$ also obeys this property (and hence passes the test). This is not a fundamental problem, in fact it is a feature: in practice we usually work with partial specifications, rather than complete specifications. Specifying that the indices should be increasing can look like:

```
propIncreasing :: String Char → Bool
propIncreasing s c = increasing (indices s c)

increasing [x,y:r] = x < y && increasing [y:r]
increasing other   = True
```

Reference implementation Using a reference implementation the function *indices* can be specified in CLEAN as:

```
propIndices2 :: String Char → Bool
propIndices2 s c = indices s c == [i \\ i ← [0..size s-1] | s.[i] == c]
```

In this function the reference implementation is stated as the list comprehension `[i\\i ← [0..size s-1] | s.[i] == c]`, this are all numbers `i` between 0 and `size s-1` where element `i` of string `s` is equal to the given character `c`.

Invoking tests The first property can be tested by evaluating the expression `test propIndices`, e.g.:

```
Start = test propIndices
```

This will cause the evaluation of the function `propIndices` for at most 1000 (the standard number of tests) values. In this test the function `indices` is the Implementation Under Test, IUT, all other parts of the language are expected to work correctly.

Partial functions Many functions used in programming are partial functions, they work correctly for a part of the input domain. For example the square root function works only for non-negative real numbers and the factorial function works only for non-negative integer values. From the outside it is not visible that these functions are partial functions, hence the test system is not able to limit the test data automatically to the part of the input type where the function is defined.

Gast provides two ways to cope with partial functions. In logic properties of partial functions usually include an implication operator restricting the actual property to input values that should behave as requested, e.g. $\forall x \in R . x \geq 0 \Rightarrow (\sqrt{x})^2 = x$. In Gast we can directly express this:

```
propSqrt :: Real → Bool
propSqrt x = x ≥ 0 ⟹ let y = sqrt x in y*y == x
```

As an experienced programmer may expect, Gast finds counterexamples very quickly due to rounding errors. A better specification states that the difference $(\sqrt{x})^2$ and x should be less than some small number δ:

```
propSqrt2 :: Real → Property
propSqrt2 x = x ≥0.0 ⟹ let y = sqrt x in y*y - x < delta
where delta = 1E-10
```

The other way to cope with partial functions is to limit the test data explicitly to allowed inputs. For our square root example this can be:

```
propSqrt3 :: Real → Bool
propSqrt3 x = let y = sqrt x in y*y - x < delta
where delta = 1E-10

Start = test (propSqrt3 For [0.0, 0.123456789 .. ])
```

Of course limiting the test data to allowed inputs is more efficient in terms of actual tests done. For the actual testing rejecting test data (and generating them) is a waste of time.

2.1 The test algorithm

The IUT passes the test if the property holds for all generated arguments, i.e. it evaluates to `True`. The property does not hold if a counterexample is found.

For a property *prop* with one universal quantified variable, that is a function with one argument, the test algorithm is given by *testLogical*. The function takes the list of all possible test data and the number of test to be done as arguments. If the number of tests to be done is 0, the property passes the test. Otherwise, the property is evaluated for the first test value `t`. If this test succeeds, evaluates to `True`, testing continues with the rest of the test values. Otherwise a counterexample is found, and the test yields fail. The actual implementation shows also the test value that is the counterexample.

```
$testLogical\,([t:ts],n)$ = if $n = 0$
                              then pass
                              else if $prop(t)$
                                   then $testLogical\,(ts, n-1)$
                                   else fail
$testLogical\,([\,],n)$   = proof
```

The list of test values is provided by the operator **For**, or generated by the function `ggen` discussed below. For properties with more than one universal quantified variable the test algorithm tries every combination in a fair order: Instead of combining the first value of the first argument with all possible values for the second argument before looking at the second value for the first argument, the values are combined in an interleaved way. For a 2-argument function f, the system generates two sequences of arguments, call them $[a, b, c, ..]$ and $[u, v, w, ..]$ respectively. The desired order of tests is $f\,a\,u, f\,a\,v, f\,b\,u, f\,a\,w, f\,b\,v, f\,c\,u, ..$ rather than $f\,a\,u, f\,a\,v, f\,a\,w, .., f\,b\,u, f\,b\,v, f\,b\,w, ...$

2.2 Implementation of the test system

The properties to be tested are functions with a variable number of arguments, the universal quantified variables. The result is either a Boolean or an element of the type `Prop` introduced above. We introduce the class `Testable` in order to create a function that eats all kinds of everything as an argument.

```
class Testable a where evaluate :: a RandomStream Admin → [Admin]
```

The random stream is a list of pseudo random numbers used in the selection of test data. The type `Admin` is used to record information of the current test. In theory it would be sufficient to yield a Boolean result indicating if the test was successful or not. In practice we also want to record some information about the tests done. For instance, we do not only want to know that there exists a counterexample, but also the value of the universal quantified variables for this counterexample. This information is stored in the record `Admin`. The list of `admin`'s that is the result of the function `evaluate` contains one record for each test performed. An additional function is used to combine the results of the first N tests.

The instance of `evaluate` for Booleans is very easy. There is only one element in the list of results. If the Boolean is `True` the property holds (`OK`), otherwise a counterexample (`CE`) is found.

```
instance Testable Bool
where
    evaluate b rs admin
     = [{admin & res = if b OK CE, args = reverse result.args}]
```

A function `a→b` as argument of `evaluate` implies that we have to test a logical expression containing a universal quantified variable of type `a`. We can test this expression if `a` is a valid test argument, and we can test things of type `b`. A type `a` is a valid test argument if elements of this type can be transformed to strings by `genShow{|*|}`, and generated by `ggen{|*|}`.

```
class TestArg a | genShow{|*|} , ggen{|*|} a

instance Testable (a→b) | Testable b & TestArg a
where
    evaluate f rs result
    # (rs,rs2) = split rs
    = forAll f (generateAll rs) rs2 result
```

For the implementation of logical operators it is a little inconvenient that the class `Testable` eats almost each function. The type `Property` is merely used to stop the class `Testable` from evaluating a function as a logical expression. In order to be able to continue the evaluation of such an expression we just have to remove the constructor `Prop`.

```
:: Property = Prop (RandomStream Admin → [Admin])

instance Testable Property
where evaluate (Prop p) rs result = p rs result
```

The operator **For** that can be used to supply a list of test data that is to be used instead of the test data generated by GAST is defined as:

```
(For) infixl 0 :: !(x→p) ![x] → Property | Testable p & TestArg x
(For) p list = Prop (forAll p list)
```

The logical implication operator, ⇐, is used for argument selection. If its left-hand argument evaluates to `False`, this test case is rejected: it is neither a success nor a counterexample.

```
class (==>) infixr 1 b :: b p → Property | Testable p

instance ==> Bool
where
    (==>) c p
        | c = Prop (evaluate p)
            = Prop (\rs r = [{r & res = Rej}])
```

A similar instance of ⟹ exists for `Property`.

2.3 Generating test data

Test data are generated by the *generic*[1] [13] function `ggen`. GAST contains instances of the function `ggen` for all basic types. By deriving an instance for a user defined type, all instances of this type are enumerated in a pseudo random order with a very strong small to large bias. The basic idea is that generic programming provides an universal tree representation of arbitrary data types. The test data are obtained by a breadth-first traversal of the tree of all possible instances of the type.

For an individual test it is possible to deviate from the arguments by `ggen`. For instance, if we want to test the function *indices* only for the strings `"Hello world!"` and `"A test"`, we evaluate the expression:

```
test  (propIndices  For ["Hello World!","A test"])
```

Using the given strings in all situations where GAST needs to quantify over strings is obtained by defining an instance of `ggen` for the type `String` like:

```
ggen {|String|}  x y = ["Hello World!","A test"]
```

The arguments `x` and `y` can be used to vary the order of the generated elements.

The generic function `ggen` and the operator **For** yield powerful and flexible generation of test data. The ability to combine logical operators with the concise high level computations of a functional language and the flexible automatic test data generation, makes GAST a very powerful tool for testing functions over complex data types.

Generic test data generation One of the distinguishing features of GAST is that it is able to generate test data in a systematic way [12]. This guarantees that test are never repeated, which is useless in a referential transparent language like CLEAN. For finite data types it is even possible to prove properties using a test system: a properties is proven if it holds for all elements of the finite data type.

The generic function `gen` generates the lazy list of all values of a type by generating all relevant generic representations [13] of the members of that type.

```
generic gen a :: [a]
```

For the type `UINT` there is only one posibility: the constructor `UNIT`.

```
gen {|UNIT|} = [UNIT]
```

For a `PAIR` we combine the lists of values generated by `f` and `g` in all possible ways. We use the library function `diag2` rather than a list-comprehension like `[Pair a b \\ a←f, b←g]` in order to obtain the required fair order. The function `diag2` from the `StdEnv` combines elements for the given lists in a diagonal way. For instance `diag2 ['a'..] [1..]` yields a list that begins like: `[('a',1), ('b',1), ('a',2), ('c',1), ('b',2), ('a',3), ('d',1), ('c',2), ('b',3), ('a',4),...`

[1] To avoid confusion with generic programming in object oriented languages this is also called *polytypical* programming. Generic programming in the OO-spirit is called polymorphic programming in functional programming.

```
gen {|PAIR|}  f g = map (λ(a,b)=PAIR a b) (diag2 f g)
```

For the choice in the type EITHER we use an additional Boolean argument to merge the elements in a nice interleaved way. The definition of the function merge is somewhat tricky in order to avoid that it becomes strict in its list elements.

```
gen {|EITHER|}  f g = merge True f g
where
    merge :: !Bool [a] [b] → [EITHER a b]
    merge left as bs
     | left
        = case as of
             [] = map RIGHT bs
             [a:as] = [LEFT a: merge (not left) as bs]
        = case bs of
             [] = map LEFT as
             [b:bs] = [RIGHT b: merge (not left) as bs]
```

In order to let this merge algorithm terminate for recursive data types we assume that the non recursive case (like Nil for lists, Leaf for trees) is listed first in the type definition. Using some insight knowledge of the generic representation of algebraic data types allow us to make the right initial choice in gen {|EITHER|} . In principle the generic representation contains sufficient information to find the terminating constructor dynamically, but this is more expensive and that does not add any additional power.

Finally we have to provide instances of gen for the basic types of Clean. Some examples are:

```
gen {|Int|}    = [0: [i \\ n←[1..maxint], i←[n,~n]]]
gen {|Bool|}   = [False,True]
gen {|Char|}   = map toChar [32..126] ++ ['\t\n\r']
gen {|String|}  = map toString lists
where
    lists :: [[Char]]
    lists = gen{|*|}
```

The actual algorithm used in Gast is slightly more complicated. It uses a stream of pseudo random numbers to make small perturbations to the order of elements generated. Basically the choice between Left and Right in ggen {|Either|} becomes a pseudo random one instead of strict alternating one.

Pseudo random order of test arguments Many testers believe that for good test results the order of test should be (pseudo) random. There is hardly any evidence that this general claim is correct, but it is easy to satisfy these testers. We want to keep the systematic generation of test values in order to avoid needless repetitions of identical tests and to produce proofs by exhaustive testing. The order of elements in the test suite can be changed a little without effecting the quality of the tests. We still want to have border values near the beginning of the test suite and other values later.

The actual implementation of Gast use a generic function `ggen` rather than `gen`. The difference is that `ggen` has a pseudo random perturbation of the elements in the list of test values. In the resulted list is looks like elements are moved back and forth a bit in a pseudo random order. The actual algorithm used by `ggen` is a bit more cleaver, only the values really needed in the list are generated. See [12] for details about this algorithm. The only real difference is in the instance for `EITHER`. In order to avoid a complicated analysis of the generic representation of a data type we assume that the nonrecursive constructors in a type, like `Nil` and `Tip`, are defined first. The argument `n` is there for technical reasons, it controls the change between choosing between `LEFT` and `RIGHT` in the instance `ggen {|EITHER|}` . If `merge` goes deeper in the recursion, the change of selecting `LEFT` increases. This cause termination for any shape of the type that has the nonrecursive constructors defined first. The other additional argument is a list of pseudo random numbers controlling the choice between `LEFT` and `RIGHT`.

```
ggen {|EITHER|}  f g n rnd
        ♯ (r1,rnd) = split rnd
          (r2,rnd) = split rnd
        = merge n rnd (f n r1) (g (n+1) r2)
where
    merge :: Int RandomStream [a] [b] → [EITHER a b]
    merge n [i:r] as bs
     | (i rem n) ≠ 0
        = case as of
            [] = map RIGHT bs
            [a:as] = [LEFT a: merge n r as bs]
        = case bs of
            [] = map LEFT as
            [b:bs] = [RIGHT b: merge n r as bs]
```

Also the generation of elements of basic types is pseudo random. For instance for integers we make sure that the common border values occur very soon and we continue with the list of pseudo random numbers `rnd`.

```
ggen {|Int|}  n rnd = randomize [0,1,-1,maxint,minint] rnd 5 id
```

Having the border values in front of the test set guarantees that the error in a property like

```
propAbs :: Int → Bool
propAbs n = abs n ≥ 0
```

is found quickly. The result of executing `Start = test propAbs` is: *Counterexample 1 found after 1 tests: -2147483648(minint).* The exact number of tests needed is dependent of the seed used in the generation of pseudo random numbers. Any experienced computer scientist knows that the inverse of `minint` in 2-complement notation is `minint` itself. This implies that the absolute value of any integer but `minint` is positive.

2.4 Controlling input values

Gast provides a rich palette of tools to control the actual values used as test argument. In this section we will give an overview and some examples of their use. This should make it easier to chose the most appropriate approach for a particular test.

Generic generation should be the default choice for generating values of a data type. The algorithm outlined above generates a list of values from small to large with some pseudo random perturbation of the order. Especially for recursive and parameterized data types this is very useful.

Consider the type `Tree` as an example.

```
:: Tree a = Empty | Node (Tree a) a (Tree a)
```

For this tree we define a function `swap` that recursively swaps the left and right subtree in a given tree.

```
swap :: (Tree a) → Tree a
swap Empty = Empty
swap (Node l n r) = Node (swap r) n (swap l)
```

A desirable property for each tree is that swapping it twice yields the original tree: `twice swap t === t`. In order to prevent that we have to define equality for trees, we use the generic equality ===. The desired instance is derived by writing **derive** `gEq Tree`. Also the generation of trees, `ggen`, and the printing of trees, `genShow`, is derived.

In order to execute the test CLEANneeds a fixed type for the trees to be generated: the overloading in `propSwap` should be solved. We should chose a type of trees to be generated and tested. Usually it is a good strategy to use a finite and small type, like `Char` or `Bool`, in the actual tests. Swapping trees with the same structure, but different constants in de nodes does not yield more information. Here we have chosen to test trees of Booleans by giving `propSwap` the type `(Tree Bool) → Bool` rather than the more general `(Tree a) → Bool | < a`.

```
derive ggen Tree
derive gEq Tree
derive genShow Tree

propSwap :: (Tree Bool) → Bool
propSwap t = twice swap t === t

Start = test propSwap
```

As you might expect, this property passes any number of tests.

User defined generation of elements of a type can be very useful if only a small fraction of the domain has to be used in the tests. The generic function `ggen` yields a list of all elements to be used as test value: the *test suite*.

Suppose that for one reason of another we want to test only with trees that have only empty left subtrees. Instead of deriving the generation of trees, we can define:

```
ggen {|Tree|} elems n r = l
where l = [Empty: [ Node Empty e t \\ (e,t) ← diag2 (elems n r) l]]
```

In order to make a pseudo random perturbation of the order of these elements we can use the function `randomize` from the GAST-library. Apart from the list of values to be randomized, the function randomize has a list of pseudo random numbers as argument, a number controlling the amount of perturbation, and a function that given a list of pseudo random integers yields the rest of the numbers to be generated. We can randomize our trees with empty left subtrees by:

```
ggen {|Tree|} elems n r = randomize l r 2 (λ_→[])
where l = [Empty: [ Node Empty e t \\ (e,t) ← diag2 (elems n r) l]]
```

Usually user defined instance of `ggen` are not used for recursive types, but for small sets of constants. The function `holidays` produces the number of days off per year given an age between 18 and 65. Suppose that we want to test whether the number of days per year is between 20 and 30. A straight forward definition of this property seems:

```
propHolidays a = 20 ≤ h && h ≤ 30 where h = holidays a
```

Testing this property with `Start = test propHolidays` shows immediately strange answers for the default arguments like 0, `maxint` and `minint`. A more accurate property is:

```
propHolidays1 a = 18 ≤ a && a ≤ 65 ⟹ 20 ≤ h && h ≤ 30 where h = holidays a
```

But the test results of `Start = test propHolydays1` are disappointing: none of the first 5000 generated integers appears to be a valid age. This implies that nothing is tested about `holidays`. This is not strange if we consider that only 48 of the 2^{32} integers are valid test arguments. Changing the general generation of integers is clearly undesirable.

There are at least three ways to tackle this problem. First we can transform an arbitrary integer to a valid age in an ad-hoc way:

```
propHolidays2 a = propHolidays1 (18+abs (a rem (65-18+1)))
```

This works, but has as disadvantages that it is rather ad-hoc, that it does more tests than needed (testing the same age multiple times), and does not give maximal information (a proof by exhaustive testing is possible for this small number of arguments.

The second way to solve the testing problem does not suffer from these disadvantages. We specify the arguments that should be used in the actual test by the **For** combinator: `Start = test (propHolydays1 For [18..65])`. This yields the proof we had hoped.

When we want to test more properties with this age the third way to solve the problem becomes more appropriate. We can define a special type for age and define the instances by hand instead of deriving them:

```
:: Age = Age Int
ggen {|Age|}  n r = map Age [18..65]
derive genShow Age
```

This has as additional advantage that the values get tagged by their type. This can be very handy for properties over many variables over similar types. The property can be stated as:

```
propHolidays3 (Age a) = 20≤h && h≤30 where h = holidays a
```

As you night expect testing this property also yields a proof.

For properties over many variables proofs of properties over ages might become too much work. In those situation it is worthwhile to randomize the test values with the function `randomize` from the GAST library. According to good testing practise we take care of testing the boundary values 18 and 65 quickly:

```
ggen {|Age|}  n r = map Age ([18,65] ++ randomize [19..64] r 46 λ_→[])
```

The choice between a property with the operator **For** and a special type with a dedicated instance of `ggen` depends on the taste of the test architect and the expected use. When the values are needed at several places we do encourage the use of a special type. Data types involved in this kind of properties are usual not recursive, hence writing a handcrafted instance of `ggen` is usually pretty easy.

Abstract and restricted data types require their own approach in testing. Suppose that we want to test a particular implementation of a binary search tree. The interface of this type (given in a `.dcl`-file) is:

```
:: SearchTree a

empty :: (SearchTree a)
addElement :: a (SearchTree a) → SearchTree a | < a
isElement :: a (SearchTree a) → Bool | < a
```

How can we test this abstract data type, ADT? Here we cannot derive instances of `SearchTree` for two reasons. **(1)** For the user of this library the search trees are an ADT. This also implies that the generic system has no knowledge about the internal structure of the ADT. Hence, it is impossible to derive an instance of `ggen` for search trees. **(2)** Since search trees are restricted data types, (all elements smaller that the value of a node should be in the left subtree, ..) deriving instance of such a restricted data type do not produce only correct instances: the generic system only knows the types and nothing about the additional restrictions. In addition what property can we specify over an instance of an ADT with unknown content and structure?

Nevertheless, we can test this ADT. The key step is to fill a tree with a list of known elements. A simple property is that a search tree should contain a value if and only if it is inserted in the tree:

```
propSearchTree :: Char [Char] → Property
propSearchTree c list = isMember c list ⟺ isElement c (toSearchTree list)
```

```
toSearchTree list = foldr addElement empty list
```

For a proper implementation of `SearchTree` this property passes any number of tests, but it does not guarantee that the ADT is really constructed using search trees.

In order to test whether the trees are really subtrees we add an instance of `toList` to the interface of search trees that yields the values in the tree according to an inorder tree walk:

```
instance toList SearchTree
where toList l = inorder l []
      where inorder Empty c = c
            inorder (Node l n r) c = inorder l [n:inorder r c]
```

For a proper search tree this list of values should be sorted. We can test this with a property like:

```
propIsSearchTree :: ([Char] → Bool)
propIsSearchTree = increasing o toList o toSearchTree
```

Using `increasing` from section 2. Our implementation of search trees passes any number of tests initiated by `Start = test propIsSearchTree`. Together with the results of testing `propSearchTree` this greatly increases our confidence in the correctness of the implementation of `SearchTree`.

This shows that it is also possible to test abstract and restricted data types with Gast. In general the ability to derive the generation of instances of arbitrary types from the generic algorithm is a great pleasure. In this section we have shown that there can be reasons to define a list of test values explicitly and how this can be done.

3 Testing Reactive Systems

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 numbers as input and the response is the number of inputs seen. 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 Implementation Under Test, IUT. Since the state of the IUT is hidden, stating properties relating input, output and state is not feasible. To circumvent this problem we specify reactive systems by an extended state machine and require that the observed behavior of the IUT conforms to this specification.

From Finite State Machines, FSMs, we inherit the synchronous behavior of systems. Each input yields a, possibly empty, sequence of outputs. After producing this sequence of outputs the system becomes quiescent; it waits for a new input. Among other advantages this yields a convenient notion of no output: the empty sequence. We extend the FSM model in several directions:

- The state, input and output can be of any (recursive) data type. This includes also infinite data types and parameterized data types. Hence, we have a state machine rather than a *finite* state machine. A machine specification having parameterized data types is also known as an *extended* state machine.
- It is not required that the specification or implementation of the state machine is deterministic.

3.1 Extended State Machines

An Extended State Machine, ESM, as used by GAST consists of states with labelled 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, t, o). A relation based specification δ_r is a set of these tuples: $\delta_r \subseteq S \times I \times S \times [O]$. Where S is the type of states, I is the type of inputs, and O is the type of outputs. We use $[O]$ in the transitions to indicate a sequence of elements of type O. It is not required that all these types are different. Specifications can be *partial*: not for every $s \in S$ and $i \in I$ there must be a tuple in δ_r specifying the new state and the output. A specification is *total* if it is not partial. If a specification is *nondeterministic* there are $s \in S$ and $i \in I$ with more than one tuple in δ_r.

In practice it is usually more convenient to have a specifying function instead of a transition relation. The transition function takes the current state and input as argument and produces the set of all specified tuples of target state and output sequence. The transition function is defined by $\delta_f(s,i) = \{(t,o) | (s,i,t,o) \in \delta_r\}$. The type of this function is: $State \times Input \rightarrow \mathbb{P}(State \times [Output])$. Here we used $\mathbb{P}(X)$ as notation for a set of elements of type X. Transition $s \xrightarrow{i/o} t$ is equivalent to $(t,o) \in \delta_f(s,i)$. A specification is *partial* if for some state s and input i, $\delta_f(s,i) = \emptyset$. A specification is *deterministic* if for all states and inputs the size of the set of targets contains at most one element: $\# \, \delta_f(s,i) \leq 1$.

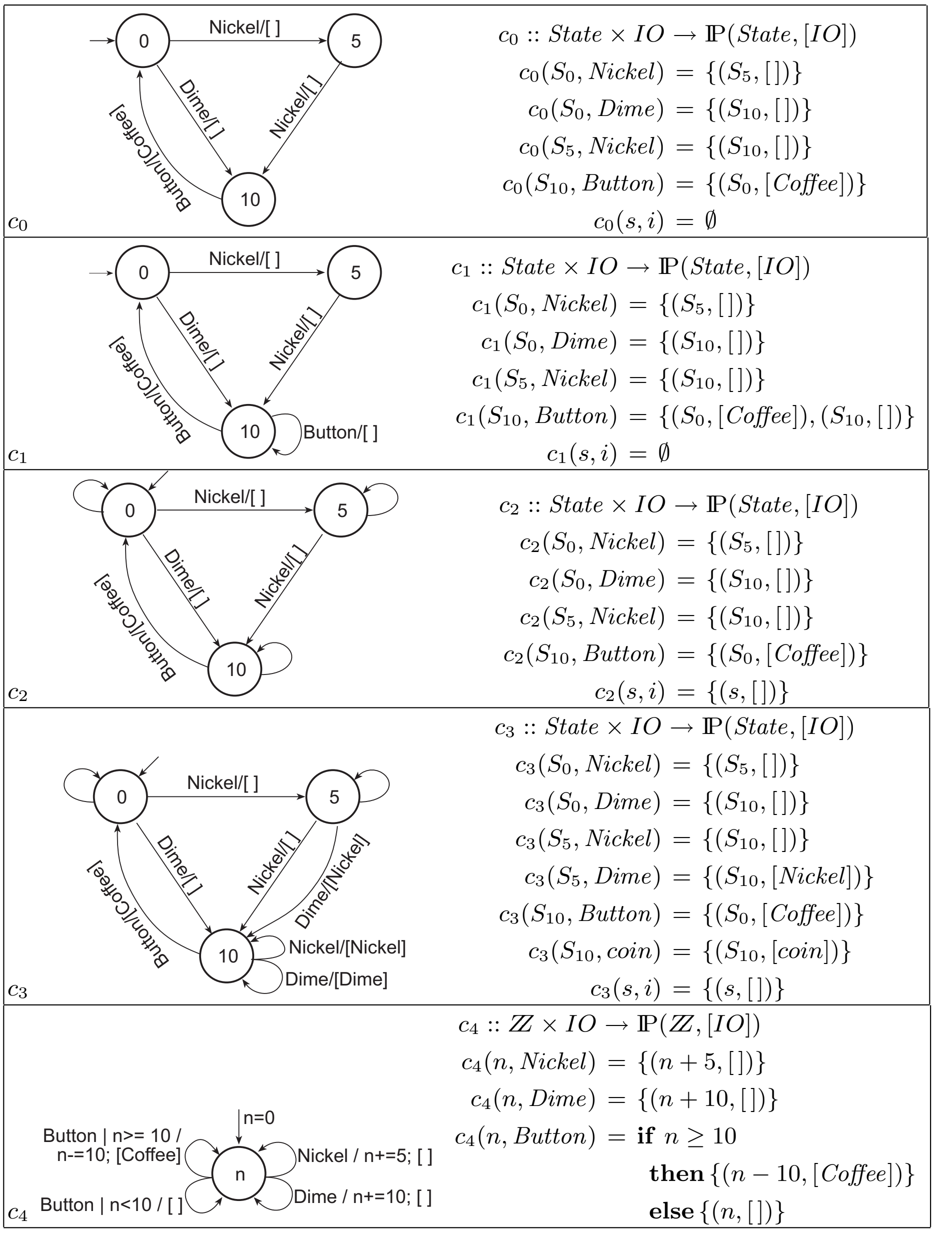
A trace σ is a sequence of inputs and associated outputs from the given state. A trace is defined inductively: the empty trace connects a state to itself: $s \overset{\epsilon}{\Longrightarrow} s$. We can combine a trace $s \overset{\sigma}{\Longrightarrow} t$ and a transition $t \xrightarrow{i/o} u$, to the trace $s \overset{\sigma;i/o}{\Longrightarrow} u$. 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 $s \overset{\sigma}{\Longrightarrow} \; \equiv \exists t. s \overset{\sigma}{\Longrightarrow} t$. All traces from a given state are defined as: $traces(s) \equiv \{\sigma | s \overset{\sigma}{\Longrightarrow}\}$.

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 $s \, \mathsf{after} \, \sigma \equiv \{t | s \overset{\sigma}{\Longrightarrow} t\}$. 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 will add it as subscript.

We will often identify a machine with its transition function. However, a complete description also determines the initial state s_0.

Examples As illustration we show some state machines modelling coffee vending machines in figure 1. In section 4 we will test some properties of these specifications. The global specification of these coffee vending machines is that it can deliver coffee after insertion of coins with a value of 10 cent, and pressing the



$$\begin{aligned} c_0 &:: \mathit{State} \times \mathit{IO} \rightarrow \mathbb{P}(\mathit{State}, [\mathit{IO}]) \\ c_0(S_0, \mathit{Nickel}) &= \{(S_5, [\,])\} \\ c_0(S_0, \mathit{Dime}) &= \{(S_{10}, [\,])\} \\ c_0(S_5, \mathit{Nickel}) &= \{(S_{10}, [\,])\} \\ c_0(S_{10}, \mathit{Button}) &= \{(S_0, [\mathit{Coffee}])\} \\ c_0(s, i) &= \emptyset \end{aligned}$$

$$\begin{aligned} c_1 &:: \mathit{State} \times \mathit{IO} \rightarrow \mathbb{P}(\mathit{State}, [\mathit{IO}]) \\ c_1(S_0, \mathit{Nickel}) &= \{(S_5, [\,])\} \\ c_1(S_0, \mathit{Dime}) &= \{(S_{10}, [\,])\} \\ c_1(S_5, \mathit{Nickel}) &= \{(S_{10}, [\,])\} \\ c_1(S_{10}, \mathit{Button}) &= \{(S_0, [\mathit{Coffee}]), (S_{10}, [\,])\} \\ c_1(s, i) &= \emptyset \end{aligned}$$

$$\begin{aligned} c_2 &:: \mathit{State} \times \mathit{IO} \rightarrow \mathbb{P}(\mathit{State}, [\mathit{IO}]) \\ c_2(S_0, \mathit{Nickel}) &= \{(S_5, [\,])\} \\ c_2(S_0, \mathit{Dime}) &= \{(S_{10}, [\,])\} \\ c_2(S_5, \mathit{Nickel}) &= \{(S_{10}, [\,])\} \\ c_2(S_{10}, \mathit{Button}) &= \{(S_0, [\mathit{Coffee}])\} \\ c_2(s, i) &= \{(s, [\,])\} \end{aligned}$$

$$\begin{aligned} c_3 &:: \mathit{State} \times \mathit{IO} \rightarrow \mathbb{P}(\mathit{State}, [\mathit{IO}]) \\ c_3(S_0, \mathit{Nickel}) &= \{(S_5, [\,])\} \\ c_3(S_0, \mathit{Dime}) &= \{(S_{10}, [\,])\} \\ c_3(S_5, \mathit{Nickel}) &= \{(S_{10}, [\,])\} \\ c_3(S_5, \mathit{Dime}) &= \{(S_{10}, [\mathit{Nickel}])\} \\ c_3(S_{10}, \mathit{Button}) &= \{(S_0, [\mathit{Coffee}])\} \\ c_3(S_{10}, \mathit{coin}) &= \{(S_{10}, [\mathit{coin}])\} \\ c_3(s, i) &= \{(s, [\,])\} \end{aligned}$$

$$\begin{aligned} c_4 &:: \mathbb{Z} \times \mathit{IO} \rightarrow \mathbb{P}(\mathbb{Z}, [\mathit{IO}]) \\ c_4(n, \mathit{Nickel}) &= \{(n+5, [\,])\} \\ c_4(n, \mathit{Dime}) &= \{(n+10, [\,])\} \\ c_4(n, \mathit{Button}) &= \textbf{if } n \geq 10 \\ &\qquad \textbf{then } \{(n-10, [\mathit{Coffee}])\} \\ &\qquad \textbf{else } \{(n, [\,])\} \end{aligned}$$

Fig. 1. Some coffee vending machines.

coffee button. An input is either a nickel, a 5-cent coin, a dime, a 10-cent coin, or pressing the coffee button. The output is either the return of a coin, or coffee. For simplicity we will use the same type *IO* for input and output, we take care that the *Coffee* is never an input and *Button* is never an output. The state of the first three machines is the algebraic data type *State*, it just records the amount of money inserted. The last machine uses a number as state. The types *State* and *IO* are enumeration types defined as:

$$\begin{aligned} \mathit{State} &= S_0 \mid S_5 \mid S_{10} \\ \mathit{IO} &= \mathit{Nickel} \mid \mathit{Dime} \mid \mathit{Coffee} \mid \mathit{Button} \end{aligned}$$

We will discus each of the machines briefly:

c_0 This is the simplest partial specification meeting the informal requirements. After inserting two nickels, or one dime, and pressing the coffee button, the machine produces coffee. Note that this is a partial specification, for instance the effect of the input *Button* in state S_0 is undefined.

c_1 This is a partial specification meeting the informal requirements. After inserting two nickels, or one dime, and pressing the coffee button, the machine can produce coffee. This machine is very similar to c_0, but nondeterministic. On input *Button* in state S_{10}, it can either produce coffee and go to S_0, or do nothing. This is a partial specification, for instance the effect of *Button* in S_0 is undefined.

c_2 The unlabelled transitions are applicable on any other input and produce the empty output. They make the specification total. All these transitions are represented by the last function alternative.

c_3 This is also a total specification. It states that coins should be returned if the value of the inserted money becomes higher than 10 cents.

c_4 This machine uses a single integer as state. It stores the total amount of money inserted and produces coffee while there is enough money. There are infinitely many states. Only non-negative multiples of 5 can be reached.

Some traces of c_2 are: $[\,]$, $[(\mathit{Nickel}, [\,])]$, $[(\mathit{Nickel}, [\,]), (\mathit{Nickel}, [\,])]$, $[(\mathit{Nickel}, [\,]), (\mathit{Nickel}, [\,]), (\mathit{Button}, [\mathit{Coffee}])]$, and $[(\mathit{Dime}, [\,]), (\mathit{Button}, [\mathit{Coffee}])]$. Sequences of input-output pairs that are *not* traces of c_2 are: $[(\mathit{Nickel}, [\,]), (\mathit{Dime}, [\,])]$, and $[(\mathit{Dime}, [\mathit{Coffee}])]$.

3.2 Representation of specification functions in Gast

In order to test machines in Gast, the specifying function is expressed in the functional programming language Clean [7]. The resulting set of pairs is represented by a list of pairs. The Clean compiler will check the specification on matters like type correctness and proper use of identifiers.

As example we show the representation of c_1 in Clean. The enumeration types used as well as the transition function can be mapped directly to Clean.

```
:: State = S0 | S5 | S10
:: IO = Nickel | Dime | Coffee | Button

c1 :: State IO → [(State,[IO])]
c1 S0  Nickel = [(S5  ,[])]
c1 S0  Dime   = [(S10,[])]
c1 S5  Nickel = [(S10,[]) ]
c1 S10 Button = [(S0  ,[Coffee ]), (S10,[])]
c1 s   i      = []
```

Function arguments starting with a capital are constants that must be matched to the actual arguments in order to make this alternative applicable. Lowercase arguments match any actual argument. Alternatives are tried in textual order, the first one that matches is applied.

Using higher order functions, specifications can be manipulated. As a very simple example we list the function `enableInput`, that enables input in any state. This function takes a machine specification `m` as argument, and yields an input enabled version of `m`. If no transition is specified for some state and input, it adds the transition to the same state with an empty output sequence. Since this is a polymorphic function, it will work for any specification using arbitrary types for state `s`, input `i` and output `o`.

```
enableInput :: (s i → [(s,[o])]) → s i → [(s,[o])]
enableInput m = m‘
where m‘ s i = case m s i of
                 [] = [(s,[])]
                 r  = r
```

Applying this function to c_0 yields a specification that is equivalent to c_2. Applying it to c_2, c_3, or c_4 does not change these specifications. Note that `enableInput c1` is not equivalent to c_2, the first specification still contains the transition $S_{10} \xrightarrow{Button/[]} S_{10}$, which is not present in c_2.

3.3 Implementations under test

The assumption is that also the implementation under test is an extended state machine. Since the IUT is a black box, its state is invisible. Even if the IUT is nondeterministic, it will choose exactly one transition on each input.

In contrast to the specification, the implementation should be *input enabled*: the result of any input in any reachable state should be specified. In terms of the transition function this is $\forall s \in State. \forall i \in Input. \delta_f(s, i) \neq \emptyset$. The motivation for this requirement is that an IUT cannot prevent that inputs are applied. It is perfectly accaptable if some inputs in specific states brings the implementation in an error state. For our tests it is sufficient if the IUT accepts each input that is allowed by the specification. The broader input enabledness requirements prevents complicated analysis or runtime problems.

In the examples above, a coffee vending machine cannot prevent that a user presses the button or inserts a coin in any state. This implies that c_1 cannot be a correct implementation. or instance the effect of applying the input *Button*, pressing the coffee button, in state S_0 is undefined. The machines c_2, c_3 and c_4 are input enabled, and hence can be used as IUT.

The implementation can be in any programming language or even in hardware, for testing it is only required that GAST can provide an input to the IUT and observe the associated output.

3.4 Conformance

Intuitively an IUT is conform to a specification if the observed transitions are part of the specification, or the specification does not specify anything for this state and input: $\delta_f(s, i) = \emptyset$. Formally, conformance of the iut to the specification spec is defined as:

$$\mathsf{iut}\ \mathit{conf}\ \mathsf{spec} \equiv \forall \sigma \in \mathit{traces}_{\mathsf{spec}}(s_0).\forall i \in \mathit{init}(s_0\ \mathsf{after}_{\mathsf{spec}}\ \sigma)\forall o \in [O].$$
$$(t_0\ \mathsf{after}_{\mathsf{iut}}\ \sigma) \xrightarrow{i/o}\ \Rightarrow\ (s_0\ \mathsf{after}_{\mathsf{spec}}\ \sigma) \xrightarrow{i/o}$$

If the specification allows input i after trace σ, the observed output of the IUT should be allowed by the specification.

This notion of conformance is very similar to the *ioco* relation of [8] for Labelled Transition Systems, LTSs. In an LTS each input and output is modelled by a separate transition. In our approach an input and all induced outputs up to *quiescence* are modelled by a single transition with a sequence of outputs. The conformance relation for (timed) EFSMs in [9] is similar, our systems have a sequences of output. If there is no other information we use quiescence to determine the end of the sequence of outputs of the IUT. In [9] there is no notion of quiescence, and the EFSMs have only a single output.

Examples Since both the specification and the IUT are given as an ESM, an ESM can be used as specification *and* as IUT. We have seen that c_1 in figure 1 cannot be a correct implementation since it is not input enabled. The machines c_2, c_3 and c_4 are correct implementations of c_1. Although the response [ ] to the input *Button* in S_{10} will never occur. According to the conformance relation this is not necessary. It is sufficient that the behavior for some specified input after a trace is allowed by the specification. Note that c_3 and c_4 have behavior that is not covered by the specification. This is allowed according to the conformance relation because nothing is specified for that combination of state and input.

Machine c_3 is not a correct implementation of c_2. After the inputs [*Dime*, *Dime*] the machine c_2 only allows the output [ ], while c_3 produces [*Dime*]. The same input trace shows that c_2 is not a correct implementation of c_3.

Although c_4 behaves correctly to c_2 as specification for the input sequence [*Dime*, *Dime*], it is not a correct implementation. This is shown for instance by the input sequence [*Dime*, *Dime*, *Button*, *Button*]. For this input c_4 produces a second cup of coffee, while c_2 only allows an empty output.

Finally, c_4 is not a correct implementation of c_3, nor is c_3 an implementation of c_4. This is shown for instance by the input sequence [*Dime*, *Dime*].

Testing Conformance The testing algorithm takes a sequence of inputs as argument. The specification and implementation start in their initial state. As long as the specification specifies transitions for the current state and input, $spec(s,i) \neq \emptyset$, the next input is applied to the IUT. If the response is conform to the specified behavior, testing continues with the next input element and the new state of the specification, otherwise an error is found. If nothing is specified for the current state and input, testing of this input sequence is terminated. The associated test result is pass. If the end of the input sequence is reached the implementation has been successfully tested with this input sequence.

The function *testConformance* takes the sequence of inputs, the observed trace and the number of steps to go as argument and produces a test verdict.

$$
\begin{array}{ll}
testConformance\,([i:is],\sigma,n) = & \textbf{if } n \neq 0 \wedge i \in init(s_0\ \mathsf{after}_{\mathsf{spec}}\ \sigma) \\
 & \textbf{then let } o = iut.apply(i) \textbf{ in} \\
 & \qquad \textbf{if } (s_0\ \mathsf{after}_{\mathsf{spec}}\ \sigma) \xrightarrow{i/o} \\
 & \qquad\ \ \textbf{then } testConformance\,(is,\sigma;i/o,n-1) \\
 & \qquad\ \ \textbf{else } \mathsf{fail} \\
 & \textbf{else } \mathsf{pass} \\
\quad testConformance\,([\,],\sigma,n) \ = & \mathsf{proof}
\end{array}
$$

The first condition checks if there are still inputs to be tested, $n \neq 0$, and if the specification states something for the next input after the observed trace, $i \in init(s_0\ \mathsf{after}_{\mathsf{spec}}\ \sigma)$. The innermost condition verifies whether the observed transition is allowed by the specification in the current state. For a more efficient implementation we keep track of the states allowed after the current trace. For deterministic specifications there is at most one state allowed at any moment. If the transition is allowed, testing continues with the rest of the inputs. An IUT passes the test of an input sequence, if the sequence becomes empty. During one test run, GAST can test several input sequences. The IUT and the specification are reset before each new input sequence.

Test Data Generation In order to test conformance, GAST needs a collection of input sequences. GAST has several algorithms for input generation, e.g.:

- Systematic generation of sequences based on the input type by the same algorithm that is used for logical properties.
- Sequences that cover all transitions in a *finite* state machine. Under the assumption that the IUT has more states than the specification, this can prove the conformance of the IUT [10].
- Pseudo random walk through the transitions of a specification. This generates long test sequences that can penetrate deep in the state space. It appears to be very effective for machines with a large or infinite number of transitions
- User defined sequences for specific purposes.

Due to the lazy evaluation of CLEAN, only those inputs are generated than are actually needed by the test algorithm. This allows us to work with potentially infinite lists of inputs. This is known as *on-the-fly* generation of test data.

The machines given in figure 1 are so small that each of these algorithms indicated the errors very soon if the implementation is incorrect. The FSM-based algorithm can be used to prove that c_3 is correct implementations of c_2, it cannot be used for c_4 as specification since it has infinitely many states.

In section 4 we will use the part of GAST introduced in the next section to test properties of these ESM-specifications.

4 Quality of Specifications

Apart from the number of tests, the quality of testing is determined by the quality of the properties stated. Obviously, aspects of a system that are not specified cannot be tested. The CLEAN-compiler used by GAST checks many aspects, like proper use of identifiers and type-correctness, of the specification before it can be used. Semantical errors cannot be catched by the compiler. Incorrect specifications can cause strange test results. If the specification and the IUT contain the same error, it will pass unnoticed. In practice many issues spotted during testing are due to incorrect specifications.

In an incremental software process this is not a serious problem. The specification and the implementation are improved at the same time. Testing shows the differences in behavior of the implementation and the specification. In this way the quality of the specification and the implementation increases. This approach is only feasible when testing is fast and automatic, GAST was found to be very useful [5].

For software processes that creates the software in one go, like the waterfall-model or the V-model, incorrect specifications can seriously delay the delivery of the system. It is desirable to verify and improve the quality of specifications before they are used to test the actual implementation.

4.1 Testing Specifications

In every situation it is desirable to check properties of the specifications used. This can be done by a model checker, like FDR or SPIN, but also by testing. Formal verification by a model checker requires a transformation of the model to a suitable input language, like CSP or Promela. Testing can be done with the given specifications and appears to be fast and effective. For specifications of small finite systems[2], a number of properties cannot only be tested, but the property can even be proven correct or falsified by a counterexample. The specifications of reactive systems introduced in section 3 are ordinary functions in CLEAN, hence they can be tested like any function as shown in section 2. In this section we will show how, general or domain specific, properties of ESM specifications can be tested.

[2] A system is finite if the number of states, inputs and outputs is finite.

General properties like determinism and completeness can be checked for any ESM specification. A specification is deterministic is for every state and input there is at most one transition defined:

$$\forall\, s \in S, i \in I \bullet \#\, \delta_f(s,i) \leq 1$$

To make this property applicable to any specification in GAST we parameterize it with the machine specification `m`:

```
propDeterministic :: (Spec s i o) s i → Bool
propDeterministic m s i = length (m s i) ≤ 1
```

Where `Spec s i o` is an abbreviation for `s i → [(s,[o])]`. Specification `c1` introduced above can be tested by evaluating `test (propDeterministic c1)`. GAST spots the counterexample for state S_{10} and input *Button* in a split second. Due to the limited number of states and inputs GAST will prove that specifications `c2` and `c3` are deterministic. For `c4` there are infinitely many states, so a proof by exhaustive testing is not possible. Testing yields `pass`.

In the same spirit we can test whether a specification is total:

$$\forall\, s \in S, i \in I \bullet \delta_f(s,i) \neq \emptyset$$

This can also be specified directly in GAST:

```
propTotal :: (Spec s i o) s i → Bool
propTotal m s i = not (isEmpty (m s i))
```

As we might expect, GAST find counterexamples for `c1`, proves the property for `c2` and `c3` and machine `c4` passes any number of tests. These general properties can be applied to any ESM-specification and are part of the library GAST.

Note that all properties in this section are tested at a specification, and not at an implementation of that model. These tests can be done before an implementation exists.

Domain specific properties As example of a domain specific property we require that coffee machines "do not lose money": the value of a state and the input should be equal to the value of the target state and the associated output for any transition specified. In logic this property reads:

$$\forall\, s \in S, i \in I, (t,o) \in \delta_f(s,i) \bullet value(s) + value(i) = value(t) + value(o)$$

This can be directly transformed to GAST. First, we construct a class *value* that yields the value of states, inputs and outputs. The value of a state is the amount of money inserted, the value of a coin is its value, the value of *Coffee* is 10, and the value of *Button* is 0. In order to test various machines easily, we make the specification to check an argument, `m`, of the property.

```
propFair m s i = p For m s i
where p (t,o) = value s + value i == value t + value o
```

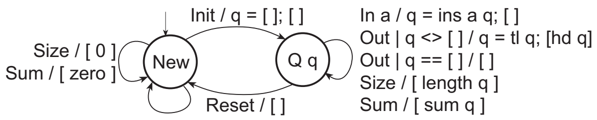


Fig. 2. The priority queue as state chart.

GAST proves this property for c_1, and c_3. The property does not hold for c_2, one of the counterexamples found by the test system is inserting a *Dime* in state S_5. The property holds also for c_4. Since this machine has infinitely many states, the property passes the test, but a proof for all states is impossible. The states can be limited to multiples of 5 between 0 and 100 by evaluating `test (propFair m5 For [0,5..100])`. Now the property is proven by GAST.

This property shows also that making a specification input enabled by adding transitions to the same state without output, as done by `enableInput`, is not an harmless operation. Property `propFair` does not hold for the input enabled version of machine `c1`, `enableInput c1`.

Finally, we can require that the value of a target state is nonnegative if the value of the source state is nonnegative: $s \xrightarrow{i/o} t \bullet value(s) \geq 0 \Rightarrow value(t) \geq 0$.

```
propNonNeg m s i = p For m s i
where p (t,o) = value s ≥ 0 ⟹ value t ≥ 0
```

This property is proven for the states `[0,5..100]` of `m5`, but testing it for all states yields counterexamples due to integer overflow. GAST proves this property for the other specifications.

4.2 A priority queue

As a more sophisticated example we show a priority queue that always dequeues the smallest elements first. It is only able to enqueue and dequeue elements after the input *Init*. The input *Reset* brings it back to the state *New*. This system is specified by the state chart in figure 2, or the function *QSpec* in figure 3.

Using overloading, the function *QSpec* is defined to be very general, it works for any type for which the operators `<`, `+`, and a `zero` are defined. This implies that we can put for instance integers, doubles, or characters in such a priority queue.

Testing general properties of the specification The quality of this specification can be investigated by testing some of its desired properties. The specification passes any number of tests for being deterministic. When testing it for being total, GAST almost immediately spots a counterexample for state Q [ ] and input *Init*. The fact that the specification is not total implies that not all behavior can be tested by GAST: according to the conformance relation, any behavior is allowed if the specification does not specify a transition for a given state and input.

$$\begin{aligned}
\mathit{QSpec} :: \ (\mathit{Qstate}\ a) \times (\mathit{Qin}\ a) &\rightarrow [(\mathit{Qstate}\ a, [\mathit{Qout}\ a])] \mid <, +, \mathit{zero}\ a\\
\mathit{QSpec}\,(\mathit{New}, \mathit{Init}) &= [(Q\ [\,], [\,])]\\
\mathit{QSpec}\,(\mathit{New}, \mathit{Size}) &= [(\mathit{New}, [\mathit{Int}\ 0])]\\
\mathit{QSpec}\,(\mathit{New}, \mathit{Sum}) &= [(\mathit{New}, [\mathit{El}\ \mathit{zero}])]\\
\mathit{QSpec}\,(\mathit{New}, \mathit{any}) &= [(\mathit{New}, [\,])]\\
\mathit{QSpec}\,(Q\ q, \mathit{In}\ a) &= [(Q(\mathit{ins}\ a\ q), [\,])]\\
\mathit{QSpec}\,(Q\ [a : q], \mathit{Out}) &= [(Qq, [\mathit{El}\ a])]\\
\mathit{QSpec}\,(Q\ q, \mathit{Size}) &= [(Qq, [\mathit{Int}\ (\mathit{length}\ q)])]\\
\mathit{QSpec}\,(Q\ q, \mathit{Sum}) &= [(Qq, [\mathit{El}\ (\mathit{sum}\ q)])]\\
\mathit{QSpec}\,(\mathit{state}, \mathit{Reset}) &= [(\mathit{New}, [\,])]\\
\mathit{QSpec}\,(\mathit{state}, \mathit{Out}) &= [(\mathit{state}, [\,])]\\
\mathit{QSpec}\,(\mathit{state}, i) &= [\,]
\end{aligned}$$

The additional function *ins* inserts an element at the appropriate place in an ordered list. It can be defined as:

$$\begin{aligned}
\mathit{ins}\,(a, [\,]) &= [a]\\
\mathit{ins}\,(a, [b : x]) \mid a < b &= [a, b : x]\\
&= [b : \mathit{ins}\ a\ x]
\end{aligned}$$

Fig. 3. The priority queue as function.

Testing specific properties of specifications For this specification we can also state and test some specific properties like for all states s_1 that are reached after applying the input sequence $[\mathit{In}\ c]$ for any c starting in any state s, the size of the queue should be one bigger than it was before. The size of the queue is determined by applying the input *Size*.

$$\begin{aligned}
&\forall s \in (\mathit{Qstate\ Char}), \forall c \in \mathit{Char}, \forall (s_0, [\mathit{Int}\ n]) \in \mathit{QSpec}(s, \mathit{Size}),\\
&\forall s_1 \in (s\ \mathsf{after}_{\mathit{QSpec}}\ [\mathit{In}\ c]), \forall (s_2, [\mathit{Int}\ m]) \in \mathit{QSpec}(s_1, \mathit{Size}) \cdot m = n + 1
\end{aligned}$$

We used list comprehensions in Clean to mimic set notation. The operator `after` implements the after operation introduced in section 3.1.

```
propQsize :: (Qstate Char) Char → Bool
propQsize s c = and [ m == n+1  \\ (s0,[Int n])  ← QSpec s Size
                                 ,  s1 ← ([s] after QSpec) [In c]
                                 ,  (s2,[Int m]) ← QSpec s1 Size ]
```

After 6 tests, Gast tells us that this property does not hold for the state *New* and input $'d'$, in fact it does not hold for any input in state *New*.

Since the priority queue works for any type with operator `<` and `+`, and a `zero`, we can choose the type used in testing. If it works for one type, it will work for every type, provided that the operators are implemented correctly. Usually, a small type gives the best test results. For that reason we will use characters here, although a small special type like `:: T = A | B | C` is even more effective.

In the same way we can test whether the sum of the elements increases if we insert a positive element. Even if we rule out the problems with the state *New*, Gast finds counterexamples caused by overflow. This indicates that the specification does not handle the limitations of finite representations of elements.

$$\forall s \in (Qstate\ Char), \forall c \in Char, \forall (s_0, [El\ v]) \in QSpec(s, Sum),$$
$$\forall s_1 \in (s\ \mathsf{after}_{QSpec}\ [In\ c]), \forall (s_2, [El\ w]) \in QSpec(s_1, Size) \cdot w > v$$

```
propQ4a s c = c > zero ⟹ and [ w>v \\ (s0,[El v]) ← QSpec s Sum
                                     , s1 ← ([s] after QSpec) [In c]
                                     , (s2,[El w]) ← QSpec s1 Sum ]
```

Since we have access to the states of the specification, we can use its internals in our tests. For instance, we can test whether the elements in the queue are ordered such that head of the list is smaller that or equal to any element in that list. With this test we verify the distinguishing feature of a *priority* queue. This property is not enforced by the type system, but by the manipulations allowed in *QSpec*. We test this for every state reached by applying any input sequence in the initial state *New*.

$$\forall i \in (Qin\ Char), \forall (Q\ q) \in (New\ \mathsf{after}_{QSpec}\ i), \forall e \in q \cdot hd(q) \leq e$$

For Gast this can be expressed as:

```
propPriority :: [Qin Char] → Bool
propPriority i = and [ hd q ≤ e \\ Q q ← ([New] after QSpec) i, e ← q ]
```

This property passes any number of tests, Gast tests 100,000 different input sequences in 40 seconds.

Usually, properties of specifications are verified with a model checker. Due to the data dependencies used in the properties it is in this case not simple to verify the shown properties with a model checker. Moreover, in order to use a model checker the given model specification has to be translated to the world of the model checker, and produces results in terms of its own model. Here the model specification in Clean is used by Gast to test its properties. Only the desired logical property has to be stated in Clean.

The successful tests shown here do not indicate that testing of specifications has the same power as a fully fledged model checker. Due to the sophisticated logic used in state-of-the-art model checkers, they have their own significant contribution. Nevertheless, testing specifications is an elegant, powerful and lightweight alternative approach to verify properties of specifications.

Testing implementations In order to verify the testing quality of Gast we made a correct implementation of this queue and ten mutants containing common (programming) errors like: an ordinary queue instead of a priority queue, a stack instead of a priority queue, a queue of at most 25 elements, various errors in the function *ins* for duplicated elements, return to the state *New* when the queue becomes empty by an *Out*, and an implicit *Init* on an input *In a* when the system is in the state *New*.

We tested with the standard generic generation of test data for a queue of characters. Gast generates, tests and evaluates about $50,000$ individual inputs per second on an average PC. It found errors in all mutants. Errors were always spotted within 0.5 seconds, usually much faster. This depends on the order of inputs and hence on the seed used for the generation of pseudo random numbers.

5 Related work

Many automatic test tools are developed in order to speedup testing and make (regression) more accurate. Most of these test tools are script based and execute a predefined sequence of actions. One of the best known examples is JUNIT [1] for JAVA-programs, it has been ported to a very large number of other programming languages.

Model based tools like Gast are more powerful since they generate the test data themselves. In this way it is possible to increase the quality of the test by generating more test data, instead by manually specifying more tests. Moreover, generating test data based on the current version of the specification guarantees that the tests done are always conform the current version of the specification.

For the testing of logical properties the test tool Quickcheck [6], is clearly the closest related tool. Distinguishing futures of Gast are the systematic generation of test data (instead of programmer controlled pseudo random generation), and hence the ability to proof properties. Also the set of logical operators in Gast is richer.

For the testing of reactive systems a number of model based test systems is available. The dominant approach is based on labelled transition systems and Torx [8] can be regarded as the godfather of tools using this approach. None of these tools uses a functional programming language to express the state transition function. We have shown that a fpl yields a very concise way to specify reactive systems.

Much work has been done to verify and improve the quality of specifications. Most notably is the work to prove properties of the specifications with proof tools or model checkers. In fact a number of the properties shown can be verified with Clean's own proof system called Sparkle[14]. This would require a significant user guidance that looses its value after any tiny change of the specification.

Model checkers, like [15, 16], are usually geared to verify properties about the communication between processes, they have troubles with data intensive systems like the priority queue used in this paper. Model checkers require a translation of our specifications. Moreover, these systems require user guidance, and hence specific skills of the user. Testing properties provides a valuable and effective alternative within the framework used to write the specification.

The tools BLAST [17] and MOPS [18] verify properties of C-programs. Both model checkers are FSM based. They differ in model, programming language, purpose and techniques used from our approach.

There are a few reports on specification testing in the literature, e.g. [19–22], usually based on Z, [11], VDM, or B specifications. It focuses on animation of specifications to validate the specification by humans, on the evaluation of given input-output combinations, or on the question if there are instances of the specified transitions. A number of significant consistency aspects is checked by the strong type system of the CLEAN-compiler. To the best of our knowledge, this is the first report on testing specifications fully automatically in a systematic way, by generating test values, executing the tests, and generating a test verdict.

6 Discussion

In specification based testing one states general properties instead of instances of these properties for specific arguments. Advantages of this approach are the higher level of abstraction, and the automatic generation of test data from the specification. Manual generation of test data is dull and error-prone. Moreover, the validity of generated test data has to be checked after every change of the underlaying specification.

Using functional programming languages as notation for specifications appears to be very effective. The obtained specifications are very concise, and well suited to be handled by a test system like GAST.

Having the ability to test logical properties and reactive systems united in a single tool, allows us to verify the quality of specifications by automatic testing. This is a unique property of GAST. The properties tested are consistency rules for the specification of reactive systems. The same specification is used to test implementation and properties of the specification itself. Testing consistency properties of ESMs is possible since GAST combines the ability to automatically test logic properties and reactive systems. The tested properties can be general properties of specifications as well as domain specific. Testing specifications increases the quality and confidence in those specifications, and hence the quality of tests of implementations done with these specifications. Testing these properties is a lightweight and effective alternative for model checking. It works also in data intensive situations that are often hard for model checkers. An other advantage is that no other formalisms, translations and tools are needed. The examples in this paper show that many "obvious" properties of specifications are falsified by testing.

By verifying logical properties through testing, an ESM-specification can be made consistent. Another important quality attribute of an ESM-specification is to validate that it correctly states what the user requires. This validation cannot be done by fully automatic testing, but requires human guidance. Due to the executable nature of our specifications, they are also very suited for validation by simulation. We will address this in another paper.

References

1. See `www.junit.org`
2. Bernot, G., Gaudel, M. C., and Marre, B. Software testing based on formal specifications: a theory and a tool, Software Engineering Journal, Nov. 1991, pp387–405.
3. P. Koopman, A. Alimarine, J. Tretmans and R. Plasmeijer. Gast: Generic Automated Software Testing. In R. Peña, *IFL 2002*, LNCS 2670, pp 84–100, 2002.
4. P. Koopman and R. Plasmeijer. Testing reactive systems with Gast. In S. Gilmore, *Trends in Functional Programming 4*, pp 111–129, 2004.
5. A. van Weelden et al: On-the-Fly Formal Testing of a Smart Card Applet. SEC 2005. Or technical report NIII-R0403, at `www.cs.ru.nl/research/reports`.
6. K. Claessen, J. Hughes. QuickCheck: A lightweight Tool for Random Testing of Haskell Programs. ICFP, ACM, pp 268–279, 2000. See also `www.cs.chalmers.se/~rjmh/QuickCheck`.
7. R. Plasmeijer, M van Eekelen. Clean language report version 2.1. `www.cs.ru.nl/~clean`.
8. J. Tretmans. Testing Concurrent Systems: A Formal Approach. In J. Baeten and S. Mauw, editors, *CONCUR'99 – 10^{th}*, LNCS 1664, pp 46–65, 1999.
9. M. Núñez, I. Roderíguez. Encoding PARM into (Timed) EFSMs. FORTE 2002, LNCS 2529, pp 1–16, 2002.
10. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines – a survey. *Proc. IEEE*, 84(8):1090–1126, 1996.
11. ISO/IEC 13568:2002 standard. See also `vl.zuser.org`.
12. P. Koopman and R. Plasmeijer. Generic Generation of Elements of Types. In *Sixth Symposium on Trends in Functional Programming (TFP2005)*, Tallin, Estonia, Sep 23-24 2005.
13. A. Alimarine and R. Plasmeijer. A Generic Programming Extension for Clean. In: Arts, Th., Mohnen M.: IFL 2001, LNCS 2312, pp 168–185, 2002.
14. M. de Mol, M. van Eekelen, R. Plasmeijer. Theorem Proving for Functional Programmers. - SPARKLE: A Functional Theorem Prover. In: Arts, Th., Mohnen M.: IFL 2001, LNCS 2312, pp 55–71, 2002.
15. G. Holzmann. The SPIN Model Checker. ISBN 0-321-22862-6, 2004.
16. G. Behrmann, A. David, K. Larsen. A Tutorial on Uppaal LNCS 3185, 2004.
17. D. Beyer, A. Chlipala, T. Henzinger, R. Jhala, R. Majumdar. The Blast query language for software verification. LNCS 3148, pp 2–18, 2004
18. H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. Proceedings 11th Annual NDSS, San Diego, CA, February 2004
19. Kazmierczak P. Dart, L. Stirling, M. Winikoff: Kazmierczak Dart, Stirling, Winikoff: Verifying requirements through mathematical modelling and animation. *Int. J. Softw. Eng. Know. Eng.*, 10(2), pp 251–273, 2000.
20. R. Kemmerer Testing formal specifications to detect design errors. *IEEE Tr. on Soft. Eng.*, 11(1), pp 32–43, 1985.
21. S. Liu Verifying consistency and validity of formal specifications by testing in J. Wing et al: FM'99, LNCS 1708, pp 896–914, 1999.
22. T. Miller and P. Strooper A framework and tool support for the systematic testing of model-based specifications *ACM Tr. Soft. Eng. and Meth.*, pp 409–439, 2003.