Testing with Functions as Specifications

Pieter Koopman

Institute for Computer and Information Sciences,
Radboud University NIjmegen, The Netherlands
pieter@cs.ru.nl

Abstract. In this paper we show that mathematical functions and logical expressions can very well be used as, partial, specifications. Reactive systems can be modelled by powerful extended state transition systems, that can be nondeterministic and can handle parameterized and infinite types for the inputs, outputs and states. These specifications can very concisely and directly be stated in a modern functional programming language. The test tool Gast is able to generate test data based on these specifications, execute the associated tests, and make a verdict fully automatically. Test data can be generated fully automatically, but can also be tailored in various high level ways, if that is desired.
Advantages of this approach are that one specifies properties instead of instances of these properties, test data are automatically derived instead of manually, the tests performed are always up to date with the current specification, and testing is automatic (and hence fast and accurate).

1 Introduction

Testing is still the most used method to judge the quality of software. Human testing of software is error-prone, dull, and expensive. Hence, many approaches to automatic software testing are proposed and used. In most approaches a set of tests is specified by a test engineer. This set of tests can be executed to determine the current quality of the software. The test execution is preferably automatic, in order to make it accurate and easy to repeat.

A limited number of automatic test tools use a specification of the systems as a basis for testing instead of a set of tests that are supposed to verify whether the implementation is conform to the specification [2–4, 7, 9]. The big advantage of this approach is that there is no test engineer needed to design tests for the current specification. Moreover, after a change in the specification there is no review and update of the specified tests needed.

A number of tailor made languages are used for the system specification by these automatic test systems. We show that a modern functional programming language, Clean [8], is very well suited as specification language. Advantages of using an existing high level language are that it is very close to the usual mathematical specifications, no new language has to be constructed and learned, all quality checks (like static type checking) are obtained for free, and a rich collection of libraries is available.

Dagstuhl Seminar Proceedings 04371
Perspectives of Model-Based Testing
http://drops.dagstuhl.de/opus/volltexte/2005/324

In this paper we show how properties of single functions and abstract data types can be expressed in predicate logic and how they can be expressed in test systems. The specification of reactive systems by extended state machines is treated as a second example of this approach.

2 Testing Functions and Abstract Data Types

The relation between input and output of a single function can be conveniently specified in predicate logic. As a very simple example we will consider the square root function. This function has a rational number as argument and produces a rational number. It is tempting to write $\forall r \in \mathbb{R} \,.\, \sqrt{r}^{\,2} = r$ as a general specification of the square root function, but this is incorrect since the square root is a partial function. It is only defined for non-negative arguments. A correct specification for the square root function on rational numbers is:

$$\forall r \in \mathbb{R} \,.\, r \geq 0 \Rightarrow \sqrt{r}^{\,2} = r$$

For an actual implementation of the square root function in a programming language we should also have to cope with rounding errors. In the programming language one works with a type `Real` which is a finite approximation of the rational numbers. Using a finite approximation implies that there can be a some rounding error in each computation. That is the square of the square root of r and r should differ at most some small constant δ.

$$\forall r \in \mathbb{R} \,.\, r \geq 0 \Rightarrow |\sqrt{r}^{\,2} - r| \leq \delta$$

This property can be used to test the implementation of the square root function in a programming language by evaluating $r \geq 0 \Rightarrow |\sqrt{r}^{\,2} - r| \leq \delta$ for a finite number of values: the test data. In a script based test system, like [1], the test engineer specifies the values that should be used in the test, e.g. 0.0, 1.0, 4.0 and $1,000,000$. In the model based test system GAST, the property is stated directly, and the generation of test values is done by the test system.

The property for the suare root function can be expressed directly by a function in CLEAN. The arguments of the function are treated as universally quantified variables by the test system GAST[3]. The test system provides an operator $\Longrightarrow$ that mimics the implication $\Rightarrow$ from logic. The specification becomes[1]:

```
propSqrt :: Real → Property
propSqrt r = r ≥ 0.0 ⟹ abs (s*s -r) ≤ delta
where s     = sqrt r
      delta = 10E-11
```

This property can be tested by executing the CLEAN program:

[1] `propSqrt` is the name of this function. The first line of the function specifies its type: the function `propSqrt` as a real as argument and yields an element of the abstract type *Property*. As usual in modern functional programming languages, we do not write parenthesis around function arguments.

```
Start = test 1000 propSqrt
```

The function `Start` in CLEAN is the equivalent of the function `main` in a C-program: execution starts here. This invokes 1000 tests of the property `propSqrt`. The type of the test data is determined automatically by the arguments of the tested property, here `propSqrt`. Hence the test data are reals. Generation of test data of type real is predefined in GAST: first the predefined border values (0.0, 1.0, and -1.0) are generated followed by a sequence of pseudo random real values. In this paper we will use the terms *sequence* and *list* as synonyms. In this way `r≥0 ⟹ abs(s*s-r)≤delta where s = sqrt r; delta = 10E-11` is tested for 1000 test values of type real for `r`. The implementation of `sqrt` from the library happens to pass this test. Testing with GAST is quick. On a rather old windows machine GAST executes about 30,000 tests of the square root function each second.

This approach works also for properties with multiple universal quantifiers of arbitrary (user-defined and/or recursive) data types.

This model based testing approach has two main advantages above script based testing. The advantage is that the property tested is reflected here much clearer in the function `propSqrt`, than in instances of this property for specific values, e.g. for CLEANr=1.0 and 4.0 one can state in a script based test `abs(s*s-1.0)≤10E-11 && abs(t*t-4.0)≤10E-11 where s = sqrt 1.0; t = sqrt 4.0`. This makes changes of the required property, for instance by an other rounding error, easier if that would be required. If the maximum rounding error is decreased to 10^{-16} by `delta = 10E-16`, GAST comes promptly with the counterexample 561.628992123927 for the 6^{th} test case. The second advantage of the model based approach to testing is that it is much easier to adjust the number of test cases. In a script based approach one has to state each test that has to be done manually. In the model based approach used in GAST, only the parameter 1000 in the function `Start` has to be changed. If we are satisfied with 10 test values we simply specify `Start = test 10 propSqrt`.

2.1 Testing Abstract Data Types

For properties of functions manipulating abstract data types, ADTs, (like queues, stacks, search trees etc.), the approach introduced above cannot be used directly. The result of the manipulating functions is usually an instance of the ADT, and hence we do not have access to its internal structure. An unattractive way-out is to state properties and test them inside a particular implementation of the ADT. An implementation independent, and hence more appealing, solution is to use the algebraic laws for the ADT as properties to be tested. These algebraic laws are exactly the kind of general properties needed by a model based test system.

As an example we consider an abstract data type for sets of characters[2]. We assume the following manipulation functions for the ADT *Set*.

$$\begin{aligned} \mathit{empty} &: \mathit{Set} \\ \mathit{add} &: \mathit{Char} \times \mathit{Set} \to \mathit{Set} \\ \mathit{remove} &: \mathit{Char} \times \mathit{Set} \to \mathit{Set} \\ \mathit{contains} &: \mathit{Char} \times \mathit{Set} \to \mathit{Bool} \\ \mathit{is\_empty} &: \mathit{Set} \to \mathit{Bool} \end{aligned}$$

The function *empty* creates a new set that does not contain any character. The function *add* makes sure that the given characters occurs in the set. After applying the function remove, the given character will not be part of the set. The predicates *contains* and *is_empty* verify whether the given character is part of the given set, and whether the given set is empty respectively.

Some desirable properties of this type are:

- The empty set is empty: $\mathit{is\_empty}(\mathit{empty})$.
- After adding a character to an empty set, the new set should contain this character: $\forall c : \mathit{Char} \,.\, \mathit{contains}(c, \mathit{add}(c, \mathit{empty}))$.
- In fact the previous property holds for any set, after adding a character to a set, the set should contain that character:
  $\forall c : \mathit{Char} \,.\, \forall s : \mathit{Set} \,.\, \mathit{contains}(c, \mathit{add}(c, s))$.
- After adding and removing a character from an empty set, the obtained set should be empty: $\forall c : \mathit{Char} \,.\, \mathit{is\_empty}(\mathit{remove}(c, (\mathit{add}(c, \mathit{empty}))))$.
- Inserting a character c in a set should not change the previous contents of the set. If a character d was in the set it should remain in the set. That is, if a character was not in the set, it should only be part of the new set if it is the inserted character:

$$\begin{aligned} &\forall c : \mathit{Char} \,.\, \forall s : \mathit{Set} \,.\, \forall d : \mathit{Char}. \\ &\quad (\mathit{contains}(d, s) \vee c = d \Rightarrow \mathit{contains}(d, \mathit{add}(c, s))) \wedge \\ &\quad (\neg(\mathit{contains}(d, s) \wedge c \neq d) \Rightarrow \neg \mathit{contains}(d, \mathit{add}(c, s))) \end{aligned}$$

  Using a conditional expression we can clarify this example by factoring out the left-hand-sides of the implications:

$$\forall c : \mathit{Char} \,.\, \forall s : \mathit{Set} \,.\, \forall d : \mathit{Char} \,.$$

[2] Readers familiar with polymorphic types, or (functional) programming might wonder why we use a set of characters instead of a polymorphic type for sets, a set that can hold instances of any type. There are two reasons for not using a polymorphic type. First, the version for a set of characters does not require knowledge about polymorphic types and hence will be understandable for a broader public. Second, the version for a specific type is slightly shorter than the polymorphic version. Our approach works flawless for polymorphic types. Testing should always be done for a particular instance of the polymorphic type. Usually Booleans or characters are a good choice.

$$\begin{aligned}&\textbf{if } \mathit{contains}(d, s) \vee c = d\\ &\quad \textbf{then } \mathit{contains}(d, \mathit{add}(c, s)))\\ &\quad \textbf{else } \neg \mathit{contains}(d, \mathit{add}(c, s)))\end{aligned}$$

These properties can be stated directly in Clean:

```
propSet0 :: Bool
propSet0 c = is_empty empty

propSet1 :: Char → Bool
propSet1 c = contains c (add c empty)

propSet2 :: Char Set → Bool
propSet2 c s = contains c (add c s)

propSet3 :: Char → Bool
propSet3 c = is_empty (remove c (add c empty))

propSet4 :: Char Set Char → Property
propSet4 c s d
 = (contains d s || c == d ⟹ contains d (add c s)) ∧
   (∼ (contains d s) && c ≠ d ⟹ not (contains d (add c s)))

propSet4a :: Char Set Char → Bool
propSet4a c s d
    | contains d s || c == d
        = contains d (add c s)
        = not (contains d (add c s))
```

Given an implementation of the ADT *Set*, Gast is able to test each of these properties, e.g. by executing `Start = test 100 propSet4`. Since property 2 and 3 only depend on the finite type of characters, Gast is able to *prove* or falsify these properties by systematically trying all possible characters. A property can be proven by a test system by exhaustive testing of the property. Gast uses a systematic enumeration of the values of a type to achieve a proof whenever it is possible.

2.2 Test Data Generation

For basic types like character, the generation of test data is defined in Gast [5]. Since Gast is a library for Clean, it is easy for the test engineer to change those definitions, if that would be necessary for some particular test purpose.

For user defined data types one should indicate how test data should be generated. There are three possibilities to determine the test data used by Gast.

- Gast contains a generic [11] algorithm for test data generation[5]. For finite types this algorithm yields all instances of the type in pseudo random order. For infinite types, it yields also the instances of the type in a pseudo random

order, but with a very strong small to large bias. The size of a value is determined by the number of constructors it contains. For recursive types, the generated list of test data will be infinitely long. Due to the lazy evaluation in the hosting language, only the test data actually used during testing will be generated. The data generation algorithm prevents duplicates. In order to use this algorithm for user defined type `T`, one has only to state **derive** `ggen T` once in the program.
In order ensure termination GAST will execute at most a given number,N, of tests. If a counterexample is found testing stops immediately. When the number of different test values is less than N, the number of executed test is also smaller than N.

- If for one reason or another, one is not satisfied with the default generic algorithm, or it cannot be applied, one can always define his own test data generation. An example where the default algorithm cannot be used is the generation of binary *search* trees. Ordinary binary trees causes no troubles at all, but the ordering required for search trees cannot be handled by the default algorithm. An other situation where the default algorithm is not always applicable, is for testing ADTs. The test data generation algorithm needs information of the type in order to do its job. This implies that the generation of instances either has to be part of the interface of the ADT, or one has to generate instances using the functions in the interface. For the example of the type *Set* from the previous sections, one can generate instances by inserting all characters form a list of characters in an empty set. With some knowledge of functional programming this is expressed as:
`ggen {|Set|} x y = map (foldr add empty) (ggen {|*|} x y)`.
Instead of this infinite list of *Set*s, we can also specify a finite number of sets that has to be used as test data.
- The two possibilities listed above will create an instance of the generic test data generation for the indicated type. Whenever GAST needs test data of that type, this instance will be used to generate the test data.
Using the *For*-operator, one can specify a list of test data that will be used in a specific test instead of the test data generated by the generic algorithm. Suppose that we want to test the first property of sets stated above only for all lower case characters. This can be done by executing:
`Start = test (propSet1 For ['a'..'z'])`. For a correct implementation the test result will be *proof*, the test will succeed for all listed values.

Together these possibilities yield powerful and flexible generation of test data. For unconstrained data types the instance generated by the generic algorithm is usually appropriate. See for details.

3 Testing Reactive Systems

For testing reactive systems we assume that the system is an extended state machine. From Finite State Machines, FSMs, we inherit the synchronous behaviour of systems [10]. One of our basic assumptions is that the output of the system

is always a sequence of values. Each input yields a, possible 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. Obviously, this does include infinite data types. This does include parameterized data types for input and output. Hence, we have an *extended* state machine rather than a *finite* state machine.
- It is not required that the specification or implementation of the state machine is deterministic.

Since we are aiming at black box testing, stating logical properties about the source state, the input, the target state, and the sequence of responses is no option: the source and target state of the system are unknown.

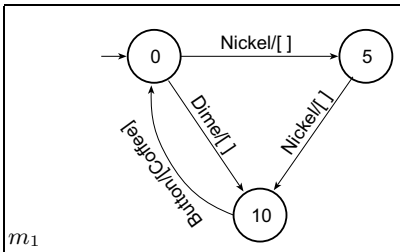
To circumvent this problem we specify an extended state machine that will be used as specification. We require that the observed behaviour of the IUT, Implementation Under Test, is conform to the specified behaviour. The specifying extended state machine is represented by a function, δ_f, that takes the current state and input as arguments and produces a set of tuples containing the new state and associated output sequence: $\mathit{State} \times \mathit{Input} \to \mathbb{P}(\mathit{State} \times [\mathit{Output}])$. This holds for all types *State*, *Input*, and *Output*. Here we used $[X]$ as notation for a sequence of elements of type X, and $\mathbb{P}(X)$ as notation for a set of elements of type X.

The specification function is treated as a partial specification: if for some $s \in \mathit{State}$ and $i \in \mathit{Input}$ the set of specified targets state and output is empty, $\delta_f(s, i) = \emptyset$, the transition is *unspecified*. The specification function itself should be total, i.e. $\mathrm{dom}(\delta_f) = \mathit{State} \times \mathit{Input}$.

As an alternative one can specify the state machine by a relation: $\delta_r \subseteq \mathit{State} \times \mathit{Input} \times \mathit{State} \times [\mathit{Output}]$. Again this holds for all types *State*, *Input*, and *Output*. Here it is obvious that the transition relation can be partial and nondetermistic. The transition function can be derived from this relation by $\delta_f(s, i) = \{\, (t, o) \mid (s, i, t, o) \in \delta_r \,\}$. The transition relation δ_r can be derived straight forwardly from the transition function δ_f. Although δ_f and δ_r are equivalent, we usually prefer to work with the transition function δ_f since it can usually be stated much more compactly than δ_r in GAST.

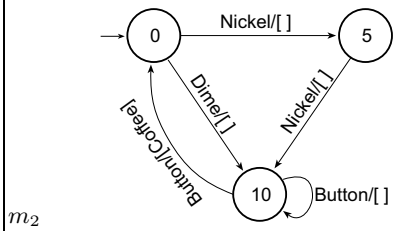
We will often identify a machine with its transition function. Actually this is not quite correct since a complete specification of a machine consists of the transfer function and the initial state.

In the usual descriptions of (finite) state machines found in the literature, the state machine is described by an n-tuple. This tuple contains: the set of states, the set of inputs, the set of outputs, the state transition function and the output function. The role of the sets of possible states, inputs and outputs, is performed here by the types in the transition function. Here we use a single transition function, δ_f, specifying the output and the new state, rather than separate state



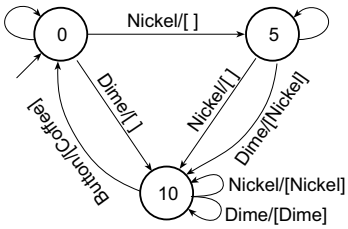
m_1

$$
\begin{aligned}
m_1 &:: State \times IO \to \mathbb{P}(State, [IO]) \\
m_1(S_0, Nickel) &= \{(S_5, [\,])\} \\
m_1(S_0, Dime) &= \{(S_{10}, [\,])\} \\
m_1(S_5, Nickel) &= \{(S_{10}, [\,])\} \\
m_1(S_{10}, Coffee) &= \{(S_0, [Coffee])\} \\
m_1(s, i) &= \emptyset
\end{aligned}
$$



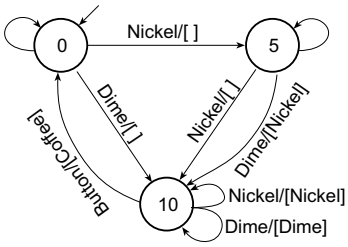
m_2

$$
\begin{aligned}
m_2 &:: State \times IO \to \mathbb{P}(State, [IO]) \\
m_2(S_0, Nickel) &= \{(S_5, [\,])\} \\
m_2(S_0, Dime) &= \{(S_{10}, [\,])\} \\
m_2(S_5, Nickel) &= \{(S_{10}, [\,])\} \\
m_2(S_{10}, Coffee) &= \{(S_0, [Coffee]), (S_{10}, [\,])\} \\
m_2(s, i) &= \emptyset
\end{aligned}
$$



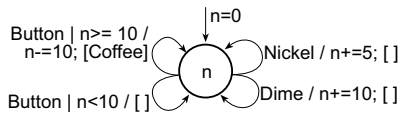
m_3

$$
\begin{aligned}
m_3 &:: State \times IO \to \mathbb{P}(State, [IO]) \\
m_3(S_0, Nickel) &= \{(S_5, [\,])\} \\
m_3(S_0, Dime) &= \{(S_{10}, [\,])\} \\
m_3(S_5, Nickel) &= \{(S_{10}, [\,])\} \\
m_3(S_{10}, Coffee) &= \{(S_0, [Coffee])\} \\
m_3(s, i) &= \{(s, [\,])\}
\end{aligned}
$$



m_4

$$
\begin{aligned}
m_4 &:: State \times IO \to \mathbb{P}(State, [IO]) \\
m_4(S_0, Nickel) &= \{(S_5, [\,])\} \\
m_4(S_0, Dime) &= \{(S_{10}, [\,])\} \\
m_4(S_5, Nickel) &= \{(S_{10}, [\,])\} \\
m_4(S_5, Dime) &= \{(S_{10}, [Nickel])\} \\
m_4(S_{10}, Coffee) &= \{(S_0, [Coffee])\} \\
m_4(S_{10}, Nickel) &= \{(S_0, [Nickel])\} \\
m_4(S_{10}, Dime) &= \{(S_0, [Dime])\} \\
m_4(s, i) &= \{(s, [\,])\}
\end{aligned}
$$



m_5

$$
\begin{aligned}
m_5 &:: \mathbb{Z} \times IO \to \mathbb{P}(\mathbb{Z}, [IO]) \\
m_5(n, Nickel) &= \{(n + 5, [\,])\} \\
m_5(n, Dime) &= \{(n + 10, [\,])\} \\
m_5(n, Coffee) &= \textbf{if } n \geq 10 \\
&\qquad \textbf{then } \{(n - 10, [Coffee])\} \\
&\qquad \textbf{else } \{(n, [\,])\}
\end{aligned}
$$

Fig. 1. Some coffee vending machines.

transition function and output functions, to link the state transition and the associated output for nondeterministic state machines.

As illustration we show some state machines modelling coffee vending machines in figure 1. The global specification of these machines is that it can deliver coffee after insertion of coins with a value of 10 cent, and pressing the 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. The state of the first four 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 that can be defined as:

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

In the definition of the transition function m_i of these machines we will follow usual rules of functional programming. Names of specific values, called constructors, always start with an uppercase character. Names starting with a lowercase character are variables that match any value. The textual order of function alternatives indicates their priority, the first alternative that matches the actual arguments should be applied.

Note the difference between the last alternatives of the transition function. The alternative $m(s, i) = \emptyset$ indicates that noting is defined for all states, s, and inputs, i, that are not covered by the alternatives above this alternative. The alternative $m(s, i) = \{(s, [\,])\}$ indicates that for any state, s, and input, i, not covered by the preceding alternatives the state, s, should not change and the output is the the empty list, $[\,]$, i.e. there is no output at all.

We will discus each of the machines briefly:

m_1 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 *Coffee* in state S_0 is undefined.

m_2 This machine is very similar to m1, but on input *Coffee* in state S_{10}, it can either produce coffee and go to state S_0, or produce nothing and stay in state S_{10}. Hence, this machine is nondeterministic.

m_3 The unlabelled transitions mean: *on any other input, stay in this state and produce the empty output* ($[\,]$). This is a total specification, since a transition is defined for all possible states and inputs: $\forall s \in \mathit{State} \,.\, \forall i \in \mathit{IO} \,.\, m_3(s, i) \neq \emptyset$.

m_4 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.

m_5 This machine uses a single integer as state to record the amount of money inserted. It does not return money if the amount of money exceeds 10, but remembers the total amount of money. For example: after inserting two dimes, it will be able to produce two coffee. Note that this machine has infinitely many states. Only states where n is a non-negative multiple of 5 can be reached.

Note that the last machine in figure 1 is a state machine rather than a finite state machine. The state is an integer and hence there are infinitely many possible values for the state. In a straightforward implementation of this machine in Gast one will use the type `Int` to represent the state. This implies that there are only 2^{32} possible values for the state. If this number of states is considered as to small, one should use the type `BigNum` to represent the state, this type has not an upperbound on the size of the numbers.

3.1 Implementations under test

As stated above, 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. We assume that the IUT is an object with a method *apply*. This method takes an input as argument and produces a sequence of outputs. The type of this method is: $apply : Input \rightarrow [Output]$.

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 usually cannot prevent that inputs are applied. It is perfectly acceptable if some inputs in specific states bring the implementation in an error state.

In the examples above, a coffee vending machine cannot prevent that a user presses a button or inserts a coin in any state. This implies that m_1 and m_2 cannot be a correct implementation. In these machines, for instance the effect of applying the input *Coffee*, pressing the coffee button, in state S_0 is undefined. The machines m_3, m_4 and m_5 are input enabled, and hence can be used as IUT.

The implementation can be in any programming language, it is only required that Gast can provide an input to the IUT and observe the associated output.

3.2 Conformance

The conformance between an implementation and a specification is very similar to the *ioco* relation [9] for labelled transition systems, LTSs.

Informally conformance between a specification *spec* and an implementation *imp* is defined as:

> An output of the IUT to some input i is said to be conform to the specification in state s, if the specification defines something for this state and input, $\delta_f(s, i) \neq \emptyset$, and the observed output is allowed by the specification: $imp.apply(i) \in \{o \in [Output] \mid \exists t \in State \,.\, (t, o) \in \delta_f(s, i)\}$.

For situations where the specification does not specify anything, $spec(s, i) = \emptyset$, any behaviour of the IUT is allowed. These transitions are excluded from the tests by Gast, since the specification cannot be used to determine the conformance after such a transition.

Since we do not know the state of the IUT (the assumption is that it is a black box), we can only use the notion of conformance introduced above in the start

state. Only for the first transition the source state of the IUT is known: the IUT is in its initial state. To extend the notion of conformance to other states we introduce the notion of *trace*. A trace is a sequence of input output pairs, where there is a single state that connects the input output pairs. Some traces for m_1 above are: $[\,]$, $[(\mathit{Nickel}, [\,])]$, $[(\mathit{Nickel}, [\,]), (\mathit{Nickel}, [\,])]$, $[(\mathit{Nickel}, [\,]), (\mathit{Nickel}, [\,]), (\mathit{Coffee}, [\mathit{Coffee}])]$, $[(\mathit{Nickel}, [\,]), (\mathit{Nickel}, [\,]), (\mathit{Coffee}, [\mathit{Coffee}]), (\mathit{Nickel}, [\,])]$, $[(\mathit{Dime}, [\,])]$, and $[(\mathit{Dime}, [\,]), (\mathit{Coffee}, [\mathit{Coffee}])]$. Some sequences of input-output pairs that are *not* traces of m_1 are: $[(\mathit{Nickel}, [\,]), (\mathit{Dime}, [\,])]$, and $[(\mathit{Dime}, [\mathit{Coffee}])]$.

The notion of conformance introduced above is extended to any state reachable by a trace of the *specification*.

In the examples in figure 1 one can always determine the target state of the specification if the source state and the input/output pair is known. In general this is not necessary: nondeterminism is not restricted to observable nondeterminism. If for some source state s and input i in a transition function δ_f we have $(t_1, o_1) \in \delta_f(s, i)$ and $(t_2, o_2) \in \delta_f(s, i)$ and $o_1 = o_2$, it is *not* required that $t_1 = t_2$. In terms of the State-chart: there can exist several transitions from a given state that have identical input/output labels but a different target state. Machine m_2 has observable nondeterminism: in state S_{10} there are two transitions possible for the input *Coffee*. When the observed output is empty the machine stays in state S_{10}, with the output $[\mathit{Coffee}]$ the target state is S_0.

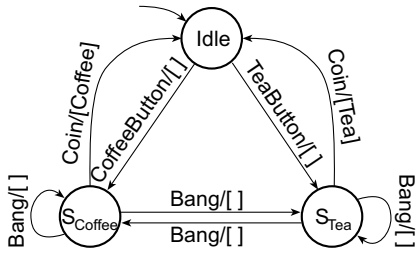
A famous example of a real nondeterministic machine is a vending machine that can change its state after a *Bang*. The usual behaviour is that the user select the wanted product and then inserts a *Coin*. If there is a *Bang* between those inputs, the vending machine is allowed to give any desired product when a *Coin* is inserted. Since the output for all transitions with a *Bang* as input is identical, the empty output, it is impossible to determine the current state after a transition labelled $\mathit{Bang}/[\,]$ form state S_{Coffee} or S_{Tea}.

The types used in the specification of this machine are:

$$\begin{aligned} \mathit{VendingState} &= \mathit{Idle} \mid S_{\mathit{Coffee}} \mid S_{\mathit{Tea}} \\ \mathit{Input} &= \mathit{CoffeeButton} \mid \mathit{TeaButton} \mid \mathit{Coin} \mid \mathit{Bang} \\ \mathit{Output} &= \mathit{Coffee} \mid \mathit{Tea} \end{aligned}$$

In figure 2 the specification of this nondeterministic machine is given in the same format as figure 1.

Testing for 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, $\mathit{spec}(s, i) \neq \emptyset$, the next input is applied to the IUT. If the response is conform to the specified behaviour, 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.



$$
\begin{aligned}
spec :: VendingState \times Input &\rightarrow \mathbb{P}(State, [Output]) \\
spec(Idle, CoffeeButton) &= \{(S_{Coffee}, [\,])\} \\
spec(Idle, TeaButton) &= \{(S_{Tea}, [\,])\} \\
spec(S_{Tea}, Bang) &= \{(S_{Tea}, [\,]), (S_{Coffee}, [\,])\} \\
spec(S_{Coffee}, Bang) &= \{(S_{Tea}, [\,]), (S_{Coffee}, [\,])\} \\
spec(S_{Tea}, Coin) &= \{(Idle, [Tea])\} \\
spec(S_{Coffee}, Bang) &= \{(Idle, [Coffee])\} \\
spec(s, i) &= \emptyset
\end{aligned}
$$

Fig. 2. A nondeterministic vending machine.

We have seen that m_1 and m_2 in figure 1 cannot be implementations since they are not input enabled. The machines m_3, m_4 and m_5 are correct implementations of m_1. Any trace produced by m_1 is also a trace of the other machines. Note that the implementations do have behaviour that is not covered by the specification.

The machines m_3, m_4 and m_5 are also correct implementations of m_2, although the response [ ] to the input *Coffee* in S_{10} will never occur. According to the conformance relation this is not necessary. It is sufficient that the behaviour for some specified input after a trace is allowed by the specification.

Machine m_4 is not a correct implementation of m_3. After the inputs [*Dime*, *Dime*] the machine m_3 only allows the output [ ], while m_4 produces [*Dime*]. The same input sequence shows that m_3 is not a correct implementation of m_4.

Although m_5 behaves correct to m_3 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*, *Coffee*, *Coffee*]. For this input m_5 produces a second cup of coffee, while m_3 only allows an empty output.

Finally, m_5 is not a correct implementation of m_4, nor is m_4 an implementation of m_5. This is shown for instance by the input sequence [*Dime*, *Dime*].

For the vending machine of figure 2 the implementation that always change state on a *Bang* as well as the implementation that never change state on a *Bang* are correct implementations. Since this is a partial specification also a machine that produces *Coffee* on a *Bang* in the *Idle* state can be a correct implementation.

For the states S_{Coffee} and S_{Tea} however, the response on the input *Bang* is defined. This implies that any implementation that produces *Coffee* on a *Bang* in one of these states is not conform the specification.

Representation of specification functions in Gast In order to test machines in Gast, the specifying function is expressed in the functional programming language Clean. The resulting set of pairs is represented by a list of pairs [4]. The Clean compiler will check the specification on matters like type correctness and proper use of identifiers.

As example we show the representation of m_2 in Clean. First we define the enumeration types `State` and `IO`. The function `m2` is the representation of the machine m_2 from figure 1.

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

m2 :: State IO → [(State,[IO])]
m2 S0  Nickel = [(S5  ,[])]
m2 S0  Dime   = [(S10,[])]
m2 S5  Nickel = [(S10,[]) ]
m2 S10 Coffee = [(S0  ,[Coffee ]), (S10,[])]
m2 s   i      = []
```

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 `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, input and output.

```
enableInput :: (s i → [(s,[o])]) → (s i → [(s,[o])])
enableInput m
    = λ s i = case m s i of
                [] = [(s,[])]
                l  = l
```

Applying this function to `m2` yields a specification that is equivalent to m_3.

3.3 Test Data Generation

The conformance checking algorithm discussed above works fine, but requires that a list of inputs is given as argument[3]. This leaves us with the problem of generating proper input sequences. There are several algorithms predefined in Gast. Since Gast is an open library, it easy to add user defined input sequence generation, if that would be desired. We discuss some implemented input generation algorithms briefly.

[3] Due to the lazy evaluation, only the inputs actually needed by the test algorithm are generated. This allows us to work with potentially infinite lists of inputs.

– The simplest way to generate input sequences is by using the generic generation of data of type [Input], as discussed in section 2.2. There are two drawbacks for this simple approach. First, the algorithm does not look at the specification, it might generate many sequences that are truncated pretty soon for a partial specification. Second, the strong bias to short sequences in the beginning of the input generation causes that it takes a while before testing reaches states that require many steps from the initial state.
– A more sophisticated algorithm looks at the inputs allowed in the current state of the specification and chooses one of them in a pseudo random order. This avoids inputs that are undefined by the specification and makes long input sequences that penetrates far in the state space from the beginning. Instead of the whole specification, on can also use a partial specification as a *model* of special behaviour to be tested.
– The previous algorithm can be improved for nondeterministic machines by looking at the response of the IUT. Instead of generating a sequence of inputs that can be accepted by the IUT, this algorithm looks at the response of the IUT in order to determine the transition chosen, and hence the current state. For deterministic systems this is not necessary, but for nondeterministic systems this really helps to keep the state of the specification, and hence the input generation, and the state of the IUT synchronized. This is known as *on the fly* testing in the literature [9].
– For finite state machines one can do still better. From FSM testing literature it is known that: if we assume that the implementation has no more states than the implementation and we test each transition in the specification (on output and target state), we *prove* the correctness of the implementation. GAST is able to generate a finite number of finite sequences of inputs that covers each transition and verifies the final state by a given identifying sequence.

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 last algorithm can be used to prove that m_3 and m_4 are correct implementations of m_1. The last algorithm can not be used for m_5 since it has infinitely many states.

4 Discussion

In this paper we have shown that functions can be used very well as specifications, and that these specifications are a very suited basis for automatic testing. The functional programming language CLEAN is a very good carrier of the functions as specifications. Using generics and lazy evaluation, test data can be generated in a very convenient and flexible way. Using higher order functions, specifications can be manipulated very easily.

Specification based testing has three advantages over test script based testing:

– One specifies the property that has to be obeyed by the implementation, instead of some instances of this property, i.e. a higher level of abstraction.

– The test instances are derived automatically instead of manually.
– After changes in the specification, the automatically generated test cases are always consistent with the current specification. For manually derived test cases, one has to check their validity after each change in the specification.

The proposed test method for reactive systems is successfully applied in several cases. For instance, a commercial controller of an embedded system written in C++ was tested based on a specification by a finite state machine of about 300 states. In another application a Smart Card Applet in JAVA was tested with a specification based on parameterized inputs, outputs and state [6]. Here the specifying state machine and the applet were developed incrementally and synchronously. Automatic testing based on the current specification appears to be very convenient and effectively to check the conformance between the specification and the implementation. In both situations Gast spotted conformance issues much faster and more accurately than manual tests could have done.

5 More Information

More information can be obtained from:

– The web-site of Gast: `www.cs.ru.nl/~pieter` or the web-site of Clean: `www.cs.ru.nl/~clean`.
– Sending an email to `pieter@cs.ru.nl`.

An old version of this library is incorporated in version 2.1 of Clean.

Acknowledgements

We thank Rinus Plasmeijer and Jan Tretmans for useful feedback that helped to improve this paper.

References

1. See `www.junit.org`. Similar tools exists for many programming languages.
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. P. Koopman and R. Plasmeijer. Generic Generation of Elements of Types Draft proceedings TFP'05, see `www.cs.ru.nl/~marko/tfp05`, 2005.
6. 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`.
7. 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`.

8. R. Plasmeijer, M van Eekelen. Clean language report version 2.1. `www.cs.ru.nl/~clean`.
9. 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.
10. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines – a survey. *Proc. IEEE*, 84(8):1090–1126, 1996.
11. A. Alimarine and R. Plasmeijer. A Generic Programming Extension for Clean. In: Arts, Th., Mohnen M.: IFL 2001, LNCS 2312, pp 168–185, 2002.