Testing reactive systems with GAST

Pieter Koopman and Rinus Plasmeijer

Nijmegen Institute for Computer and Information Science, The Netherlands
{pieter,rinus}@cs.kun.nl

Abstract. G∀ST is a fully automatic test system. Given a logical property, stated as a function, it is able to generate appropriate test values, to execute the tests, and to evaluate the results of these tests.
Many reactive systems, like automata and protocols, however, are specified by a model rather than in first order logic. There exist tools that are able to test software based on such a model based specification, but these tools lack the ability to verify relations on data values specified in first order logic. In this paper we introduce some extensions of G∀ST to combine the best of both worlds.
The integration of model based testing and testing based on logical properties in a single automatically systems is the contribution of this paper.

1 Introduction

Within the fully automatic test system G∀ST [16] properties over functions and data types are expressed in first order logic. These properties are written as functions in the functional programming language Clean [19]. Based on the types used in these functions, G∀ST automatically and systematically generates test values, it evaluates the property for these values, and it analyses the test results. This avoids the burden to design and evaluate a test suite by hand and makes it easy to repeat the test after changing the program *regression tests*. This automatic and systematic generation of test data is a distinguishing feature of G∀ST that even allows proofs for finite types by exhaustive testing. In [16] we mainly focused on the concepts and the basic implementation of G∀ST.

Its is possible to specify the behaviour of reactive systems, like the famous coffee-vending-machines protocols [5], in logic, as demonstrated by Z [21]. However, these systems are usually specified by a model, instead of by a property in first order logic. G∀ST is not able to test implementations based on models of such interesting systems directly. Many formalisms are used in the literature to make models of reactive systems. We use labelled transition systems (LTS), since they have been shown to be very general and effective for testing [14, 11].

We introduce some extensions that makes G∀ST suited for model based testing. We introduce a general format to specify labelled transition systems directly as a data structure in Clean. The specification of a LTS by a function is shown to be more concise and can handle infinite numbers of labels and states.

To test conformance effectively these specifications are used as a basis for test case generation. These test cases are much more effective for this purpose than the systematic generation of all possible inputs, which in its turn is more effective than random generation of inputs. For each deterministic and finite LTS it becomes possible to prove that the implementation behaves as specified, or to spot an error under the assumption that the implementation is an LTS that does not contain more states than the specification [26].

An advantage of extending G∀ST with the ability to test products specified by a LTS, is that the ability to test data types is preserved and can be combined with the new possibilities. The ability to generate data to test properties involving data types is a weak point of the existing automatic model–based test systems.

Unlike model checkers like SPIN [13], we assume the given specification is correct. In practise however, differences between the specification and the actual implementation appears also to be caused by unintended or forgotten transitions in the specification. This effect of increasing quality and confidence in the specification has also been noticed by other specifications used as a basis for test generation [16], [23].

2 Overview of G∀st

To make this paper self-contained we give an overview of G∀ST. It is an automatic test system embedded in the functional programming language CLEAN.

In G∀ST we use ordinary CLEAN functions to specify properties. As an example we consider the *rotate 13* algorithm, a simple way to encrypt texts. It is used to hide text from casual reading, and rotates the alphabet by half its length, i.e. 13 characters. Characters not in the alphabet are not effected. For example the encryption of `The answer = 42` [1] yields `Gur nafjre = 42`.

A nice property of this encryption method is that its own decryption. In logic this is $\forall c \in \texttt{Char}.\mathit{rot13}(\mathit{rot13}(c)) = c$. In G∀ST this is expressed as:

```
propRot13 :: Char -> Bool
propRot13 c = rot13 (rot13 c) == c
```

Notice that the arguments of the functions that specify the desired property are treated as universal quantified variables.

2.1 Testing and Results

Given an implementation of `rot13`, the property `propRot13` is tested by applying it for a number of characters and check whether it yields **True** for all arguments. This is exactly what the function `test` does: generate arguments of the desired type in a systematic way, evaluate the property for all generated arguments, and investigate whether all test cases are successful. This test is initiated by executing `Start = test propRot13`. To execute this program we need an implementation of the rotation algorithm. We use:

```
rot13 :: Char -> Char
rot13 c | isUpper c = toChar ((toInt(c-'A')+13) rem 26) + 'A'
        | isLower c = toChar ((toInt(c-'a')+13) rem 26) + 'a'
                    = c
```

Executing the test yields: *Proof: success for all arguments after 98 tests.*

Due to the systematic generation of test data, G∀ST can, in this situation, detect that this property holds for all possible well–defined arguments. Hence the result qualifies as a proof rather then just a successful test result. For the type `Char` G∀ST only generates the printable characters, this explains why there are only 98 successful test performed. Below we show how this property is tested for all 256 possible characters if that would be desired.

2.2 Evaluating Test Results

The basic rules for evaluating a series of test results are rather simple:

1. As soon as a single counterexample is encountered the property does not hold. The testing process terminates with an appropriate error message.
2. If no counterexamples are found and all possible test values are used, the property is proven for all well–defined arguments. Of course, such a proof is only possible for finite types, and feasible for rather small types.
3. If no counterexamples are found within a certain upperbound of tests, the property passes the test successfully. We gained confidence in its correctness.

2.3 Logical Operators in G∀st

As an additional property we might require that applying `rot13` to any character yields a different character:

```
propRot13b :: Char -> Bool
propRot13b c = rot13 c <> c
```

Executing the this test yields: *Counterexample found after 5 tests: ';'.*

As stated above, only alphabetic characters are changed. Other characters are unaffected by `rot13`. For a more precise formulation of this property we might require that applying `rot13` to a letter yields a different character:

```
propRot13c :: Char -> Property
propRot13c c = isAlpha c ==> rot13 c <> c
```

The operator `==>` mimics the implication operator, $\Rightarrow$, from logic. It has the usual semantics: if the left operand holds, the right-hand operand should be obeyed. For implementation reasons this function yields an element of type `Property` rather than a Boolean. Any Boolean result is transformed to such a `Property` by applying the function `prop`. Semantically the type `Property` is the union of Booleans and functions yielding a Boolean (which are just logical expressions containing a universal quantifier). Evaluating this property by G∀ST yields: *Proof: Success for all not rejected arguments, 52 tests, 46 rejections.*

If the left-hand argument of the operator `==>` yields `False`, the test-value is rejected instead of counted as success. This operator is used to select test values: if the test value is rejected, nothing is known about the property on the right–hand side. It would be misleading to count this as a successful test.

There are several ways for the tester to control the generation of test values. Using the infix operator `For` the property is tested for all values in the list on the right-hand side of the operator. This can, for instance, be used to test `propRot13` for all 256 characters in the standard ASCII set:

```
Start = test (propRot13 For map toChar [0..255])
```

Here G∀ST reports *Passed after 100 tests*. In this situation it is easy to turn this result to a proof. We only have to increase the number of tests allowed.

```
Start = testn 500 (propRot13 For map toChar [0..255])
```

Now G∀ST reports *Proof: success for all arguments after 256 tests*. Other ways to control the generation of test values include the explicit transformation of generated values (e.g. from lists to balanced trees), and the specification of tailor made instances of the generic generation of values [16].

2.4 Automatic Generation of Test Values

Test data generation for predefined types like `Char` is rather easy. G∀STgenerates all possible elements of finite and relatively small types like `Bool` and `Char` as test value. For large types like `Int` and `Real` this is of course not feasible. G∀ST generates by default common border types (like -1, 0 and 1), followed by random values for these types.

The generation of test values for user-defined (recursive) types is interesting. Using Clean's generic programming facilities [12, 3] G∀ST generates instances of these types fully automatically. Test data are generated such that small instances come first and larger values afterwards. Due to the systematic generation duplicates are avoided also in this situation. This implies that G∀ST is able to detect that all instances of a finite type are generated. If a property holds for all these values, it is proven correct. Generation of test values can be customized for any type, if that is desired. See [16] for details.

3 Reactive Systems

A reactive system is an automaton that posseses an internal state and interacts with its environment. In this paper we restrict ourselves to software systems with a single input and output channel. For instance, a communication channel is modelled as a function of type `[Message] -> [Message]`.

For some simple reactive systems we can specify aspects of their behaviour in first order logic. For instance, a system consisting of an unreliable communication channel supervised by an alternating bit protocol is required to yield a list of messages that is identical to the list of messages to be send. In G∀ST this is:

```
propAltBit :: (Int->Bool) (Int->Bool) [Int] -> Bool
propAltBit sError rError input = input == abpSystem sError rError input
```

The function `abpSystem :: (Int->Bool) (Int->Bool) [c] -> [c]` mimics the communication channel. The first two function arguments are used to mimic the introduction of communication errors in the sending and receiving direction of the channel respectively. The last argument, the list `[c]`, is the input of the channel and the result is the output of the alternating bit protocol to the user.

A possible implementation of this protocol is:

```
:: Message c     = M c Bit | A Bit | Error
:: SenderState c = Send Bit | Wait Bit c

sender :: (SenderState c) [c] [Message c] -> [Message c]
sender (Send b) []     as = []
sender (Send b) [c:cs] as = [M c b: sender (Wait b c) cs as]
sender state=:(Wait b c) cs [a:as]
 = case a of
    A d | b==d = sender (Send (~b)) cs as
    _          = [M c b: sender state cs as]

receiver :: Bit [Message c] -> ([Message c],[c])
receiver rState [] = ([],[])
receiver b [m:ms]
 = case m of
    M c d|b==d = ([A   b :as],[c:cs]) where (as,cs) = receiver (~b) ms
    _          = ([A (~b):as],   cs ) where (as,cs) = receiver b ms

channel :: (Int->Bool) [Message c] -> [Message c]
channel error ms = [ if (error n) Error m \\ m <- ms & n <- [1..]]

abpSystem sError rError list = received
where   (acks,received) = receiver firstBit (channel sError messages)
        messages = sender (Send firstBit) list (channel rError acks)
        firstBit = O
```

This implementation passes any test of the property `propAltBit` in G∀ST.

Although this works fine, the properties that can be specified in this way are limited. For instance, it is unclear how the behaviour of the sender of the alternating bit protocol is specified in this formalism. Usually labelled transition systems are used to specify this kind of behaviour.

3.1 Labelled Transition Systems

A labelled transition system (*LTS*) description is defined in terms of a set of states and labelled transitions between these states. To have a clear separation between input and output labels we deviate from the usual definition of a LTS by using different types labels. Moreover, we allow on input to generate a list of outputs. By introducing additional intermediate states, such a LTS can be transformed to a traditional LTS.

Given Q a non-empty countable set of states, I a non-empty countable set of input symbols, and O a non-empty countable set of output symbols, we have a transition relation $T \subseteq Q \times I \times \langle O \rangle \times Q$. Given some $q_0 \in Q$ a labelled transition system is give by the tuple (Q, I, O, T, q_0).

For the moment we restrict ourselves to deterministic systems: the output and new state are uniquely determined by the current state and the input. In fact we have a Mealy finite state machine [18]. That is $(q_1, i_1, o_1, q_2) \in T$ and $(q_3, i_2, o_2, q_4) \in T$ we have $(q_1 = q_3 \wedge i_1 = i_2) \Rightarrow (q_2 = q_4 \wedge o_1 = o_2)$. Due to this restriction we do not have spontaneous transitions (transitions not triggered by an input) yet. One often writes $(q_1, i, o, q_2) \in T$ as:

$$q_1 \xrightarrow{i/o} q_2$$

Where other model checkers and test systems often use some tailor made language (like Promela the language used within SPIN [13] and TorX [23]) to describe the labelled transition systems that serves as specification, we prefer a specification in Clean. This has two advantages. First, we can use the full power of a functional programming language to write the specification or to write functions that generate the desired specification. Second, there is no need for an additional language.

Instead of explicit sets of states, Q, and labels, I and O, we employ the type system of Clean to enforce the correct use of states and labels. A straightforward realisation of a LTS consists of a record containing a list of transitions and an initial state. In Clean this is:

```
:: Transition state input output :== (state,input,[output],state)
:: LTS state input output
    =   { trans     :: [Transition state input output]
        , initial   :: state
        }
```

The use of type-parameters for the sets of states and labels involved gives us maximum flexibility. We can even use various different types of transition systems in the same context if desired.

Usually the LTS is a partial function, so we have to decide what to do when an input is received in a state that is not covered by the LTS. Like most model checkers we choose to ignore the input: the state does not change and the output is empty. This is known as *implicit completion* of the model.

3.2 Example: a Conference Protocol

The conference protocol described here is a well known case study in many model specifications and testers [25]. The conference protocol is used to describe the behaviour of a conference protocol entity (CPE). The conference protocol allows a fixed number of entities to chat in various conferences. In order to chat the user is able to issue the following commands to the CPE:

Join nickname conference The user joins the named conference under the given nickname. A user participates in at most one conference at any time.
Datarequest messages All users in the conference receive this message.
Leave The user leaves the current conference.

There is a network through which the CPEs communicate. The interface from a CPE to the network is via a User Datagram Protocol (UDP). The CPE sends Protocol Data Units (PDUs) to the network. The network delivers these PDUs to the indicated CPE and adds the identification of the sender. There are no assumptions on the order of arriving of the messages, nor on the reliability of the connection. A CPE can receive the following inputs from the network:

DataPDUin cpe message This CPE receives a messages from cpe.

AnswerPDUin cpe nickname conference The indicated cpe wants to join the named conference under the given nickname.

JoinPDUin cpe nickname conference Request to join the named conference from the indicated cpe under the supplied nickname.

LeavePDUin cpe The indicated cpe leaves the current conference.

To accomplish its task a CPE can send the following output messages:

JoinPDUout cpe nickname conference Send a request to the named cpe to join the named conference. The network transforms this message to an AnswerPDUin input where the cpe of destination is replace by the sender.

AnswerPDUout cpe nickname conference Conformation that cpe wants to participate in the conference.

DataPDUout cpe message Send the given message to the indicated cpe.

LeavePDUout cpe indicate to *cpe* that this users leaves the conference.

Data nickname message Show a received message to the user.

After the definition of appropriate data types to hold CPE identifiers, messages, nicknames and conferences, these messages are transformed directly to the corresponding algebraic data types. The state of an CPE is either Idle, or it participates in a Conference. The list of tuples consisting of a CPEid and a Nickname records which other CPEs participate in this conference and their nicknames. This list is sorted and each CPE occurs at most once.

```
:: CPEstate = Idle | Conf ConferenceID Nickname [(CPEid,Nickname)]
```

The number of states is finite if the conference-ids, nicknames and CPEids id finite. If you consider the use of parameterised states in a LTS as cheating, you can define a mapping to and from the possible states to a sufficiently large algebraic type.

The specification for a given CPE is generated by the function in figure 1. It is sufficient to grasp the idea of the specification, don't bother about details. The occurring nicknames, conference-ids, and messages are modelled by simple algebraic datatypes. The lists of members of these types used are generated by the systematic generation functions of G∀ST. For instance:

```
:: ConferenceID = Conference1 | Conference2

ConferenceIDs   :: [ConferenceID]
ConferenceIDs   =: generateAll aStream
```

All possible conferences occurring as state for a given CPE are generated by the function `Conferences :: CPEid -> [CPEstate]`. Due to the restrictions impossed on the list of participants, it should be ordered and each partner occurs at most once, it is not possible to use generic generation for the conferences.

This generation function for the LTS remains correct if we add, change, or remove members in any of the types involved. Hence, it is more powerful and convenient to use than the definitions of the labelled transition systems used in most existing model–based test systems. For instance, TorX uses a specification of the LTS in Promela. In the Promela specification at [25] the number of

```
CPElts :: CPEid -> LTS CPEstate CPEin CPEout
CPElts myId
 = { initial = Idle
   , trans
      = [   (Idle, Join nn confId
            ,[JoinPDUout cpe nn confId \\ cpe <- CPEids | cpe<>myId]
            , Conf confId nn [])
        \\  nn          <- Nicknames
        ,   confId      <- ConferenceIDs
        ] ++
        [   (conf, JoinPDUin cpe nn2 id
            ,[AnswerPDUout cpe nn id],Conf id nn (mkset (cpe,nn2) mem))
        \\  conf=:(Conf id nn mem)  <- Conferences myId
        ,   cpe         <- CPEids
        ,   nn2         <- Nicknames
        |   cpe <> myId && not (isMember cpe (map fst mem))
        ] ++
        [   (conf, AnswerPDUin cpe nn2 id
            ,[],Conf id nn (mkset (cpe,nn2) mem))
        \\  conf=:(Conf id nn mem)  <- Conferences myId
        ,   cpe         <- CPEids
        ,   nn2         <- Nicknames
        |   cpe<>myId && not (isMember cpe (map fst mem))
        ] ++
        [   (conf, Leave, [LeavePDUout cpe \\ (cpe,_) <- mem], Idle)
        \\  conf=:(Conf id nn mem)  <- Conferences myId
        ] ++
        [   (conf, LeavePDUin cpe, [], Conf c nn [t\\t<-mem|fst t<>cpe])
        \\  conf=:(Conf c nn mem) <- Conferences myId
        ,   (cpe,_)     <- mem
        ] ++
        [   (conf, DataPDUin cpe mes,[Data nn2 mes], conf)
        \\  conf=:(Conf id nn mem)  <- Conferences myId
        ,   mes         <- Messages
        ,   (cpe,nn2)   <- mem
        ] ++ // to compensate loss of AnswerPDU
        [   (conf, DataPDUin cpe mes,[JoinPDUout cpe nn id], conf)
        \\  conf=:(Conf id nn mem)  <- Conferences myId
        ,   mes         <- Messages
        ,   cpe         <- CPEids
        |   cpe <> myId && not (isMember cpe (map fst mem))
        ] ++
        [   (conf, Datareq mes,[DataPDUout cpe mes\\(cpe,_) <- mem],conf)
        \\  conf=:(Conf id nn mem)  <- Conferences myId
        ,   mes         <- Messages
        |   not (isEmpty mem)
        ]
   }
```

Fig. 1. The specification of a CPE by the data structure LTS

partners is hardwired into the specification. Moreover, this specification is very concise if we compare it to all other specifications collected at [25]. The difference in size between this specification and the others is at least a factor 2.

3.3 Executing a Deterministic LTS

To use a given LTS as basis for testing, we must to be able to execute it. That is, given a LTS, a current state and an input we need to be able to determine the associated output and new state. Since the LTS is currently deterministic, we have in fact a finite state machine, FSM.

Often we prefer to be able to give a sequence of inputs and obtain a list of associated outputs rather than giving a single input. This is achieved by the following function to execute a deterministic LTS.

```
runFSM :: (LTS s i o) [i] -> [[o]] | == s & == i
runFSM {trans,initial} inputs = run initial inputs
where
    run state [] = []
    run state [i:r]
     = case [(o,t) \\ (s,j,o,t) <- trans | s==state && i==j] of
        []      = [[]:run state r]  // undefined: ignore input
        [(o,t)] = [o :run t r]
        _       = abort "This LTS is not deterministic!"
```

3.4 The Implementation Under Test

We perform a *black box test* of the Implementation Under Test (IUT): we can only observe the output of the system given an input. To show clearly that a single input produces a sequence of outputs and a system in a new state we use the type:

```
:: IUT input output = IUT (input -> ([output],IUT input output))
```

It is often convenient to transform this to a function that converts a sequence of inputs to the associates outputs. This is done by:

```
runIUT :: (IUT i o) [i] -> [[o]]
runIUT iut [] = []
runIUT (IUT f) [a:r] = [o:runIUT iut r] where (o,iut) = f a
```

3.5 Testing the Conference Protocol

After the introduction of a representation for model based specifications and the tools to execute the specification and the IUT, we are ready to formulate properties to be tested automatically by G∀ST.

For instance, a desirable property for any implementation of the conference protocol, `cpeImpl :: CPEid -> IUT CPEin CPEout`, is that its outputs is equal to the outputs obtained by execution of the specification:

```
propCPE :: [CPEin] CPEid -> Bool
propCPE input id = runFSM (CPElts id) input == runIUT (cpeImpl id) input
```

This is a standard property for G∀ST. Hence, it is tested like any other property in G∀ST by executing `Start = test propCPE`. The first argument is the list of inputs. The second argument is a CPE–identifications. These test data are generated like any other test data in G∀ST.

This model–based property can be combined with an ordinary logical property. If we have a logical predicate `properState :: CPEstate -> Bool` to check the sanity of states (cpe's are ordered and not duplicated), we can combine thes properties to:

```
propCPE1 :: [CPEin] CPEid -> Property
propCPE1 input id = propCPE input id /\ (properState For (Conferences id))
```

When we are convinced that the protocol handles all CPEs equal, we can also limit the test to a single CPE-id. For `CPE1` the last property becomes:

```
propCPE2 :: [CPEin] -> Bool
propCPE2 input = propCPE1 input CPE1
```

Testing these properties reveals some discrepancies between the initial versions of the specification and the implementation. The differences concerned the handling of unusual inputs, like receiving a `DataPDUin` from a cpe that is not member of the conference. This lead us to improvements of the implementation as well as the specification. Afterwards G∀ST reports that these properties passes the tests.

When the implementation passes some significant number of tests it is tempting to believe that the implementation conforms to the specification. However, analysis of the generated inputs showed that only a few conferences were established during the tests. Although the inputs are generated systematically, only a small fraction of the generated inputs correspond to actually entering a conference and sending messages. Typically, only one single data transfer within a conference is established in the first 100 tests that are generated.

Tests with systematically generated inputs appear to be very valuable to verify that the specification and the implementation ignore the same inputs, even if the sequence of messages is completely meaningless. This only tests that the IUT shows the specified behaviour: *robustness testing*.

3.6 Implementations with other Types

The type of the IUT used above suits our tests very well. However, not each implementation to be tested has such a type. A custom alternative type for the implementation is `cpeImpl2 :: CPEid [CPEin] -> [[CPEout]]`. Even when the IUT produces a single stream of output tokens, `cpeImpl3 :: CPEid [CPEin] -> [CPEout]`, rather than a sequence of output per event we can still test these implementations in G∀ST by adapting the property slightly:

```
propCPE‘ :: [CPEin] CPEid -> Bool
propCPE‘ input id = runFSM (CPElts id) input == cpeImpl2 id input

propCPE‘‘ :: [CPEin] CPEid -> Bool
propCPE‘‘ input id = flatten (runFSM (CPElts id) input) == cpeImpl3 id input
```

For `propCPE‘‘` only have lost the ability to check whether a particular output token is generated as response to the correct input. A particular element of the output might be generated too late or too early.

4 Test Data Generation from a LTS

To check the correct behaviour for meaningful sequences of messages, *conformance*, we can use the LTS as a source of information for input sequences. For instance, each meaningful sequence of inputs with an input corresponding to a transition from the initial state. We can use the existing knowledge of testing a FSM [26, 17]. An input sequence is usually called a path in the world of FSM–testing. If one assumes that the IUT is also deterministic, we do not learn anything new from executing a path which is a prefix of another tested path. If we furthermore assume that the IUT does not have more states than the specification, it is useless to test the same transition twice. Both assumptions are standard in FSM testing. We use this knowledge to construct a finite amount of longer and meaningful inputs. This implies that we are now able to prove things by exhaustive testing, instead of just executing successful tests. We discuss some test generation algorithms.

A1 From each state in the specification we only test the transitions from that state. To terminate each input sequence we randomly choose to end the path here, or to use one of the possible transitions at each point. This is basically the algorithm for test-data generation used by TorX.

A2 Since it is useless to test the same transition twice, we terminate a path when there is no untested transition from the current state instead of randomly.

A3 The paths generated by the previous algorithm do not verify the final state at the end of the path. Since the IUT is a black box we cannot check this final state directly. This algorithm checks the final state by performing additional transitions: we allow each transition twice in a path.

A4 In this algorithm we use a function of type `state -> [input]`, to determine the inputs used to test the final state. Ideally, we use a *unique input output sequence*, UIO [2], or a *distinguishing sequence*, DS, to identify the final state. If these are unknown or do not exist, we can use a short sequence of inputs.

Finding the shortest set of paths that achieve the goals of $A3$ and $A4$ is yet another variant of the travelling salesman problem. We use a simple algorithm that chooses the first transition available. An input sequence is terminated when we cannot extend it without taking a transition too often. Until all transitions are used enough we extend a prefix of one of the used inputs with transitions that still need to be done.

In the table 1 we list the number of states, the number of transitions in the LTS, and the generated number of input sequences according to algorithms above for various numbers of CPE's, nicknames, conferences and messages. By its nature $A1$ always generate infinitely many paths. For a particular test we choose some of these paths. This table shows that the number of input sequences generated by $A2$ is rather big, even for specifications of modest size. It is too large for a quick and complete automatic test.

Algorithm $A4$ produces less paths and is more accurate, but requires known paths to verify the final state. For testing the conference protocol we used:

# CPEs	nick names	confer ences	mes sages	# states	trans itions	paths generated A1	A2	A3	A4
1	1	1	1	2	2	∞	1	1	1
2	1	1	1	3	9	∞	118	4	3
3	1	1	1	5	28	∞	>10,000	11	6
2	2	1	1	7	30	∞	>10,000	14	8
2	1	2	1	5	18	∞	27,848	7	6
2	1	1	2	3	12	∞	7,827	4	3
2	2	2	2	13	80	∞	>10,000	26	16
3	3	3	3	145	2070	∞	>10,000	567	282

Table 1. Number of paths generated for various size of types.

```
CPEtestSeq :: CPEstate -> [CPEin]
CPEtestSeq state = [ Datareq mess, Join nn confId ]
where mess = hd Messages; nn = hd Nicknames; confId = hd ConferenceIDs
```

Algorithms $A3$ is used, when an appropriate test sequence for final states is not at hand. It usually gives good results.

It is important to realize that these tests only check if the IUT behaves as specified by the LTS, this is known as *conformance testing*. Testing with the generated input sequences does not show whether the IUT shows any unspecified behaviour. For this purpose we need exhaustive tests of all inputs in all states. The default generation algorithm of G∀ST provides a useful approximation.

5 Functional and Nondeterministic Specifications

The `LTS` type straightforwardly represents labelled transition systems. However, it suffers from the following drawbacks:

1. It allows nondeterminism, but a thorough examination of the data structure is necessary to see whether the specification is deterministic or not.
2. It is limited to a finite number of transitions. This makes it even impossible to specify a system that echoes a given integer or string.
3. It is impossible to use typical functional language features, like guards and pattern matching, in the specification.

All these problems are solved by using functions of type

```
:: Spec state input output :== state -> input -> [(state,[output])]
```

as specification. Just like above, we use implicit completion when we use this specification: inputs for states not specified do not change the state and produce no output. Consider the following system that returns the absolute value of negative integers. This small definition covers the transition for all integers.

```
absoluteValue state n
    | n<0   = [(state,[~n])]
            = []
```

In many test circumstances it is required that the function is total. This is achieved by adding a default yielding an empty list of transitions. To compare the new specification with the specification by a data structure in figure 1 we list the

```
cpeSpec myId Idle (Join nn conf)
 = [(Conf conf nn [],[JoinPDUout cpe nn conf \\ cpe<-CPEids| cpe<>myId])]
cpeSpec myId state=:(Conf conf nn mem) input
 # memberCPEs = map fst mem
 = case input of
    Datareq mes = [(state,[DataPDUout cpe mes \\ (cpe,_) <- mem])]
    Leave       = [(Idle,[LeavePDUout cpe \\ (cpe,_) <- mem])]
    DataPDUin id mes
      | isMember id memberCPEs
        = [(state,[Data nn mes\\(cpe,nn) <- mem | cpe == id])]
      | id<>myId
        = [(state,[JoinPDUout id nn conf])] // handle lost join
     AnswerPDUin id nn2 conf2
       | conf == conf2 && not (isMember id [myId: memberCPEs])
         = [(Conf conf nn (mkset (id,nn2) mem),[])]
     JoinPDUin id nn2 conf2
       | conf == conf2 && not (isMember id [myId: memberCPEs])
        = [(Conf conf nn (mkset (id,nn2) mem),[AnswerPDUout id nn conf])]
    LeavePDUin id = [(Conf conf nn [t \\ t=:(m,_) <- mem | m <> id],[])]
    _ = []          // to make the specification total
 cpeSpec _ _ _ = [] // to make the specification total
```

Fig. 2. The specification of a CPE by a function.

specification of the conference protocol by a function in figure 2. This is clearly more compact than the previous version. Since all lists yielded have at most length one, it is obvious that this specification is deterministic. In contrast to the specification by a data structure, listed in section 3.2, this version also works if we we use types like `Int` for cpe–ids and `String` for messages and nicknames. This makes this kind of specifications really more powerful.

The test sequence generation algorithms in section 4 operate on data structures. To uses these algorithms we need to generate transitions from the specification by a function. All transitions from a given state are produced by:

```
generateTrans :: (Spec s i o) s [i] -> [Transition s i o]
generateTrans spec s inputs = [(s,i,o,s2)\\i<-inputs, (s2,o)<-spec s i]
```

To obtain the entire transition relation, we just have to construct these transitions for every reachable state. For finite types we can use generic generation for the list of inputs to be tested. For infinite and extremely large types, like `Int`, the tester has to supply a list of inputs to be used.

6 Testing Nondeterministic Systems

Consider a simple vending machine spcified by the nondeterministic LTS:

$$Final_T \xleftarrow{Coin/[Tea]} S_{tea} \xleftarrow{Button/[]} Idle \xrightarrow{Button/[]} S_{coffee} \xrightarrow{Coin/[Coffee]} Final_C$$

If the button is pressed the machine decides to produce either tea or coffee, but nothing happens until a coin is inserted. A better vending machine returns to

Idle after producing coffee or tea. From the input/ouput one cannot decide in which state the machine is after pressing the button. It is also impossible to bring this machine in state S_{tea} by supplying inputs, it is always possible for the machine to take the other branch. This machine is specified in G∀ST as:

```
vendingSpec Idle    Button = [(Stea,[]),(Scoffee,[])]
vendingSpec Stea    Coin   = [(FinalT,[Tea])]
vendingSpec Scoffee Coin   = [(FinalC,[Coffee])]
vendingSpec state   input  = []
```

To cope with this situation we use the **ioco**–test [23, 24, 4]. The name **ioco** stands for *input/output conformance*. The basic idea is that the observed output corresponding to some input must be allowed by the specification. To deal with nondeterminism we maintain a list of possible current states, instead of a single current state. This is implemented by:

```
testIOCO :: (Spec s i o) [s] (IUT i o) [[i]] -> [String] | == o
testIOCO spec states iut paths = test 1 paths
where
  test n []         = ["All tests successful"]
  test n [p:paths] = [toString n: ioco iut states p (test (n+1) paths)]
  ioco iut       []     path     cont = ["Error!"]
  ioco iut       states []       cont = ["OK\n":cont]
  ioco (IUT iut) states [i:path] cont = ioco iut2 states2 path cont
  where (iutout,iut2) = iut i
        states2 = [t\\s<-states, (t,specout)<-spec s i| specout==iutout]
```

This test does not require that the system is really nondeterministic. It can, for instance, be used to test the conference protocol where the input is generated by one of the algorithms discussed above. Paths can be generated by the algorithms $A1..A4$, introduced in section 4.

A more sophisticated **ioco**–test algorithm might generate the input on basis of the observed behaviour. This *on the fly* testing [9] remains future work.

7 Related Work

The closest related test system for logical properties is QuickCheck [7, 8]. The discriminating difference between QuickCheck and G∀ST is the systematic test data generation in G∀ST. Test data generation in QuickCheck is based on a class, the user has to supply an instance for each new type, and random data generation. In G∀ST the test data generation for a new type comes for free since it is based on generics [3, 12]. Moreover, the generation of test data is systematic from small to large without duplicates. When a property holds for all values in a type, it is proven.

The extension of G∀ST introduced in this paper makes it a model–based test system [6] like TorX [23, 22], Autolink [20, 15], TGV [10], and UIO Test [9]. Basically these systems generate inputs for the system to be tested based on the LTS–specification. Currently these systems have difficulties with conditions on values. In G∀ST however, such conditions can easily be expressed in first order logic. We are aware of a number of running projects to extend model–based test systems with capabilities to handle restrictions on types. No results have been reported yet. The model based specifications in CLEAN appear to be clearer, shorter and more general than the example specifications collected at [25].

8 Conclusion

By representing a labelled transition system as a data type and enabling the execution of such a LTS, we are able to test systems specified by a LTS in G∀ST. This is a significant improvement since many interesting systems are specified by a model instead of a property in first order logic.

Moreover, such a LTS is used as a basis for test data generation. These input sequences test that the system behaves correct for inputs that are part of the specification. The default data generation of G∀ST is used to verify that the system does not show undesired behaviour for other inputs.

The use of functions instead of a data type to specify a LTS has two significant advantages. The specification becomes even more concise and it is able to handle infinite data types for labels and states.

The model based testing is well integrated with the automatic testing of logical properties. This makes G∀ST with this extension stronger than existing model–based testers. These systems are known to be weak at testing data types. There are several projects running to extend model–based test systems with the ability to generate data values, no results have been reported yet.

Acknowledgement

We thank Peter Achten, Marko van Eekelen, Jan Tretmans, Rene de Vries, Arjen van Weelden and Ronny Wichers Schreur for their contributions to this paper.

References

1. Douglas Adams. *The Hitch Hiker's Guide to the Galaxy*, ISBN 345391802, 1979.
2. A. Aho, A. Dahbura, D. Lee, and M. Uyari *An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours* In Protocol Specification, Testing and Verification VIII, volume 8, 1998.
3. A. Alimarine, R. Plasmeijer. *A Generic Programming Extension for Clean.* IFL2001, LNCS 2312, pp.168–185, 2001.
4. M. van der Bijl, A. Rensink, and J. Tretmans *Component Based Testing with IOCO*, CTIT Technical Report TRCTIT0334, University of Twente, 2003.
5. A. Belinfante, J. Feenstra, R. Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, L. Heerink *Formal Test Automation: A Simple Experiment*, in *Int. Workshop on Testing of Communicating Systems 12* pp 179-196, 1999.
6. E. Brinksma, J. Tretmans *Testing Transition Systems: An Annotated Bibliography*", in *Modeling and Verification of Parallel Processes–4th Summer School MOVEP 2000* LNCS 2067, pp 186-195, 2001.
7. K. Claessen, J. Hughes. *QuickCheck: A lightweight Tool for Random Testing of Hasskell Programs.* International Conference on Functional Programming, ACM, pp 268–279, 2000. See also `www.cs.chalmers.se/~rjmh/QuickCheck`.

8. K. Claessen, J. Hughes. *Testing Monadic Code with QuickCheck*, Proceedings of the ACM SIGPLAN workshop on Haskell 2002, Pittsburgh, pp 65–77, 2002.
9. J. Fernandez, C. Jard, T. Jéron, C. Viho *Using On-the-Fly Verification Techniques for the generation of test suites*, LNCS 1102, 1996.
10. J. Fernandez, C. Jard, T. Jeron, C. Viho. *Using on the fly verification techniques for the generation of test suites*, Conference on Computer Aided Verification, 1996.
11. L. Heerink, J. Feenstra, and J. Tretmans *Formal Test Automation: The Conference Protocol with PHACT* In H. Ural et al *Testing of Communicating Systems - Procs. of TestCom 2000*, Kluwer, pp 211–220, 2000.
12. R. Hinze, *Polytypic values possess polykinded types*, Fifth International Conference on Mathematics of Program Construction, LNCS 1837, pp 2–27, 2000.
13. Gerard J. Holzmann *SPIN Model Checker, The: Primer and Reference Manual* Addison Wesley, isbn 0-321-22862-6, 2003.
14. N. Goga *Comparing TorX, Autolink, TGV and UIO Test Algorithms*, SDL 2001: Meeting UML: 10th International SDL Forum Copenhagen, Denmark, June 27-29, 2001, Proceedings. LNCS 2078, pp 379–402, 2001.
15. A. Kerbrat, T. Jéron, R. Groz *Automated Test Generation from SDL Specifications*, in *The Next Millennium–Proceedings of the* 9^{th} *SDL Forum*, pp 135–152, 1999.
16. Pieter Koopman, Artem Alimarine, Jan Tretmans and Rinus Plasmeijer: *Gast: Generic Automated Software Testing*, in Ricardo Peña: *IFL 2002, Implementation of Functional Programming Languages*, LNCS 2670, pp 84–100, 2002.
17. D. Lee, and M. Yannakakis, M *Principles and Methods for Testing Finite State Machines– A Survey*, The Proceedings of the IEEE, 84(8), pp 1090-1123, 1996.
18. George Mealy *A method for synthesizing sequential circuits*, Bell System Technical Journal, 34(5):1045–1079, 1955
19. Rinus Plasmeijer and Marko van Eekelen: *Concurrent Clean Language Report (version 2.0)*, 2002. `www.cs.kun.nl/∼clean`.
20. M. Schmitt, A. Ek, J. Grabowski, D. Hogrefe, and B. Kock *Autolink – putting SDL–based test generation into practise* Proceedings of the 11^{th} International Workshop on Testing Communication Systems, pp 227–243, Kluwer Academic, 1998.
21. Mike Spivey *The Z Notation: A Reference Manual,* 2^{nd} *ed*, Prentice Hall, 1992.
22. J. Tretmans, E. Brinksma *Côte de Resyste – Automated model–based Testing*, in *Progress 2002 –* 3^{rd} *Workshop on Embedded Systems*, pp 246–255, 2002.
23. J. Tretmans *Test generation with inputs, outputs and repetive quiscence.* Software–Concepts and Tools, 17(3):103–120, 1996.
24. J. Tretmans *Testing concurrent systems: A fromal approach.* In J. Baeten and S. Mauw *Concur'99* LNCS 1664, pp 46–65, 1999.
25. Formal specifications of the conference protocol in various languages. `http://fmt.cs.utwente.nl/ConfCase/v1.00/specifications/specs.html`
26. H. Ural, *Formal methods for test sequence generation*, Computer Communications Journal, **15**(5), pp 311–325, 1992