Interactively Probing Quiescent Properties of Object I/O Applications
– A Feasibility Study –

Peter Achten

Department of Software Technology, Radboud University Nijmegen, Toernooiveld 1, 6525ED Nijmegen, The Netherlands
P.Achten@science.ru.nl

Abstract. In this paper we show how application-users can interactively experiment with Object I/O applications of arbitrary size in order to find its temporary, conditional, or invariant properties. The goal is to gain confidence in the quality of the application, and to seek properties that can be proven correct perhaps using formal proof tools.

1 Introduction

Programming an effective Graphical User Interface (GUI) is a challenging task because of the myriad of details that need to be controlled and managed: the set of possible events, knowledge of the API, general design rules for GUIs, life-cycle maintenance of GUI objects, and so on.

Still, when ignoring this plethora of details, it turns out that the structure of a typical GUI program is basically a *nested while-case loop*. The *while* structure reflects the obligation of a GUI application to poll for *events* until termination; the *case* structure reflects the need to perform case distinction on the events and act according to the needs of the application; this structure is *nested* due to the use of constructs such as modal dialogues and synchronous message passing.

A second characteristic feature of GUI applications is that they use a *structured state*, usually relying on scope rules. This structured state evolves dynamically, as parts of the state are associated with GUI objects. The data itself is in general not very complicated. We call the state *stable* when the application is polling for the next available event, because it can not modify the state in any way until an event is actually been given to it. In testing theory, this state of the application is also known as *quiescence* [15], i.e.: the application can not proceed without further input.

Although the structure of GUI programs is clear, it is hard to reason about GUI applications thoroughly and rigidly. This is caused by the following reasons:

1. The actions that are triggered by the case distinctions operate on the same (parts of the) state structure, thereby interfering with each other. When there are many such actions it is hard to keep track of each of their effects. Even a small application such as *Notepad* on Windows has at least 100 actions.

2. Reasoning about a particular program run boils down to reasoning about a particular *event trace*, an ordered sequence of events. Applications modify the set of admissible events dynamically by techniques such as enabling/disabling, hiding/showing, opening/closing of GUI objects, in order to provide the user with proper feedback on allowed actions on his part. This means that one cannot assume that an event trace is a sequence of random events.
3. The case distinction done by applications is *partial*: a program does not respond to all possible events because that would make even the simplest application unreasonably large. Instead, the underlying system takes standard actions if the application is not interested or chooses to ignore events. As a consequence, one can not rely solely on the code as the specification, but one must also take the behavior of the operating system into account.

When designing the state structure, the programmer usually has some properties in mind that the values of the state structure should satisfy whenever the application is in a stable state. A property is *invariant* for a specific event trace if it holds during all stable states along this trace. Ideally, we would like to *prove* that a property is invariant for every possible event trace because this promotes such a property to an invariant of the application. Unfortunately, for the reasons mentioned above, this is unfeasible.

In this paper we take a pragmatic approach to the problem of establishing (hopefully invariant) properties of GUI applications. We want to encourage GUI programmers to develop as many properties as possible (including false ones!) to any GUI object of an application under construction or one that has been finished long ago by perhaps somebody else. This is known as *run-time assertion checking* [9, 8]. However, for reasons of flexibility, we want to be able to *interactively* add and remove properties. The application, whenever in a stable state, checks all currently added properties and notifies the user whenever a property is violated. In this way, the developer can *probe* the application for properties.

We are going to implement this system for the Object I/O library [2, 4, 1, 5], a comprehensive GUI library that is available for the pure functional programming languages Clean [14] and Haskell [12]. Because of the size of this library, we first conduct this feasibility study to investigate the technical problems that need to be solved. When implemented in Object I/O, the system should allow:

- Properties can be assigned to any GUI object in the application, and even to the whole application. This can be done at compile-time and at run-time.
- Every property that is added is always of the correct type.
- Properties should be storeable on disk.
- There must be no loss of efficiency in case no properties are probed.

This paper is structured as follows. Just as we have done above, we first get rid of all the distracting details of Object I/O programs, and present our technique for *local state transition systems* in general (Section 2). In Section 3 we show that Object I/O is a local state transition system. We then explain the expected issues when adding property probing to Object I/O (Section 4). We discuss related work in Section 5 and conclude in Section 6.

2 Probing Local State Transition Systems

The bare bones structure of Object I/O is that of a *local state transition system* [3], which is basically the same as that of a nested while-case as discussed in the introduction. In this section we reveal this structure (Section 2.1) in order to point out the technical problems that need to be resolved when adding/removing (Section 2.2) and testing (Section 2.3) properties in a type-safe and dynamic way.

2.1 Local State Transition Systems

The set of types is very similar to those presented in [3], except that here we do include interactive processes (in order to reason about complete programs). A program (`Program`) is a collection of processes (`Process`), each of which encapsulates a state `ps` via an existential quantifier (∃ `ps:`). This state is shared by all of its elements. It effectively models the global data that is accessible by every element. To enforce this, the type (`Proc ps`) is used.

```
:: Program  :== [Process]
:: Process   = ∃ps: Process (Proc ps)
```

Every process (the record type `Proc ps`) has a number of actions that respond to process related events. These are modelled by the list of functions in the field `pcbfs`. Note that the type of an action, ((`Proc ps`) → (`Proc ps`)) provides it with full access to all elements of a program. In particular, the other processes are also an element of a process (`pcontext`). Processes are identified by an `ID`, which is a simple integer. An event `(id,i)::Event` identifies the `i`-th action of the process `id`. This is of course a very simplified form of events.

```
:: Proc ps    = { pstate   :: ps
                , pid      :: ID
                , pcbfs    :: [(Proc ps) → (Proc ps)]
                , pobjs    :: [Object (Proc ps)]
                , pcontext :: [Process] }
:: ID       :== Int
:: Event    :== (ID,Int)
```

Processes have top-level objects (these correspond with menus, windows, and so on), stored in `pobjs`, each of which again encapsulate their piece of local state `ls` and operate on the same state of the program `pst`, which is always (`Proc ps`):

```
:: Object pst = ∃ls: Object ls [Comp ls pst]
```

Top-level objects contain components with access to the shared state (`Proc ps`) and the local state of the top-level object (`ls`). A component (`Comp ls pst`) is either a concrete object (`Obj ls pst`), or it replaces the current local state (`NewLS ls pst`), or it extends the current local state (`AddLS ls pst`).

```
:: Comp ls pst  = Obj   (Obj   ls pst)
                | NewLS (NewLS ls pst)
                | AddLS (AddLS ls pst)
:: NewLS ls pst = ∃new: {newLS :: new, newDef :: [Comp  new     pst]}
:: AddLS ls pst = ∃new: {addLS :: new, addDef :: [Comp (new,ls) pst]}
```

Analogous to processes, concrete objects are identified via an `ID`, have actions, and can contain other objects. An event `(id,i)` identifies the `i`-th action of the concrete object identified by `id`.

```
:: Obj ls pst = { oid   :: ID
                , ocbfs :: [(ls,pst) → (ls,pst)]
                , oobjs :: [Comp ls pst] }
```

With this collection of types we can model scoped state structures. A value `p::Program` represents the complete quiescent state of a program. When an application successfully polls for an event `e = (id,i)`, then the next quiescent state of the program is computed by `(eval e)`.

```
eval :: Event Program → Program
```

We will not discuss its implementation: it is basically a recursive function that searches for a process or concrete object that is identified by `id` and applies the `i`-th action to the current program state. Details can be found in [3].

Example In order to make this discussion more concrete, consider the following small example of a local state transition system that has a few 'bugs':

```
program :: Program
program
  = [Process
       { pstate   = []
       , pid      = 1
       , pcbfs    = []
       , pobjs    = [Object 0
                      (Obj { oid   = 2
                           , ocbfs = [ λ(n,pst=:{pstate=l})
                                       → (n+1,{pst & pstate=[n+1:l]})
                                     , λ(n,pst=:{pstate=l})
                                       → (n-1,{pst & pstate=tl l}) ]
                           , oobjs = [] })]
       , pcontext = [] }]
```

The process maintains and shares a list of integers, `pstate :: [Int]`. The concrete object, identified by `oid = 2`, has two actions: the first adds one element to the list, and the second shortens the list. The object has a local integer state which value should reflect the length of the shared integer list. The second action contains two bugs both caused by unrestricted uses of `tl` in `tl l` and `-` in `n-1`.

2.2 Adding and Removing Properties at Run-Time

A *property* of some data type `st` is a boolean function:

```
:: Prop st :== st → Bool
```

In Section 2.1 we have introduced the elements that we want to probe:

– Complete programs, of type [Process] are probed with (Prop [Process]).
– Processes, of type (Proc ps), with ps the type of the shared state, are probed with (Prop (Proc ps)).
– Concrete objects, of type (Obj ls (Proc ps)), with ls the type of the local shared state of the concrete object. Note that, due to NewLS and AddLS, ls can be a nested tuple composition of several local states. They are probed with (Prop (ls,Proc ps)).

In order to test any of these elements at run-time with an appropriate property one needs to provide a property of the *correct type*. Unfortunately, only the type of complete programs is immediately accessible; the types of the scoped state of processes and concrete objects can not be retrieved, even though we, as program developer, are well aware of their concrete types. The deliberate existential quantification has rendered it impossible for us to check properties afterwards using a solution within the *static* type system.

We need to resort to *dynamic* type checking if we are to solve this issue. For several years now, Clean has dynamic types [13, 16]. There are basically two ways to use dynamic types for our problem:

1. Do not use existential types to hide the types of the states but use dynamics. In that case, checking for type equality is straightforward.
2. Use existential types to hide the types of the states, but do the type equality match inside the object's scope where the types are known.

We do not choose alternative 1 because it violates one of the main design decisions of our approach, viz. to use static types wherever possible. Instead, we show that alternative 2 can be used within the framework.

First we wrap properties in a dynamic, and give such a property a name:

```
:: UserProp      = { name :: PropName
                   , prop :: PropDynamic }
:: PropName    :== String    // A sensible name
:: PropDynamic :== Dynamic   // A (Prop st) function
```

We need to make a few modifications to the data types that we have introduced in Section 2.1. We are going to use an association list to store for each object its properties. The key value is (Just id) with id::ID of processes and concrete objects, and Nothing for complete programs. The association list is stored globally in the Program type, which now becomes a record.

```
:: Program    = { procs :: [Process]     // As before
                , props :: [Property] } // The property list
:: Property :== (Maybe ID,[UserProp])    // For each element, all its properties
```

The second change that we need to make is related with the dynamic type system. We are going to match the type of a property encapsulated in a PropDynamic with the state in scope of concrete objects and processes. This is done by a *type dependent* function [13]. A type dependent function can match a dynamic type with a static type, provided the static type belongs to the TC type class. We need

to impose this restriction to the type variables of `Proc` and `Obj`. Because Clean does not support type class restrictions on data type definitions, we do this with an explicit dictionary (`DictTC`) which amounts to the same thing:

```
:: DictTC  a   = { unpack :: Dynamic → (Bool,Prop a) }

:: Proc    ps  = { ..., pdict :: DictTC (Proc ps) }
:: Obj  ls pst = { ..., odict :: DictTC (ls,pst)  }
```

The `unpack` member is a function that returns the content of its dynamic argument if it correctly contains a property of the right type. It is easy to define a type dependent function that creates a dictionary of the desired type:

```
dictTC :: DictTC a | TC a
dictTC = { unpack = λdx → case dx of
                            (x :: Prop a^) = (True, x)
                            _              = (False,⊥) }
```

We can now proceed by defining the function `addProperty` that associates a property with an element:

```
addProperty :: (Maybe ID) UserProp Program → (Bool,Program)
```

The task of (`addProperty mid prop prog`) is to extend the `prog.props` list with an entry for (`mid`,`prop`) either by extending an existing entry or creating a new one. The function fails (returns `False`) if the type of the property does not match. The key problem of this function is the check for type equality. Let us assume that this function, `propertyTypeMatches`, exists. Then the definition of `addProperty` is straightforward:

```
addProperty :: (Maybe ID) UserProp Program → (Bool,Program)
addProperty mid p=:{prop} program=:{props}
  | propertyTypeMatches mid prop program
    = case span (λ(mid‘,_) = mid≠mid‘) props of
        (otherProps,[])                       // no properties yet
          = (True,{program & props=otherProps++[(mid,[p])]})
        (otherProps,[(_,ps):otherProps2])     // extend properties
          = (True,{program & props=otherProps++[(mid,ps++[p]):otherProps2]})
  | otherwise
    = (False,program)
```

Note that `addProperty` maintains the order of already present properties, so that properties are tested in the same order all the time.

The function application (`propertyTypeMatches mid p program`) must decide whether the indicated object operates on the type as given by `p`. If `mid == Nothing`, then it must be a program property, and hence the dynamic content should be matched with type (`Prop [Process]`):

```
propertyTypeMatches :: (Maybe ID) PropDynamic Program → Bool
propertyTypeMatches Nothing dp _ = case dp of
                                     (_ :: Prop [Process]) = True
                                     otherwise             = False
```

If (mid == (Just id)), then it must either correspond with a process or with a concrete object.

```
propertyTypeMatches (Just id) dp {procs}
  = any (processMatches id dp) procs
```

A process matches if its pid::ID matches id *and* the dynamic property dp matches the dictionary pdict or any of its component matches:

```
processMatches :: ID PropDynamic Process → Bool
processMatches id dp (Process proc) = procMatches id dp proc
where
  procMatches :: ID PropDynamic (Proc ps) → Bool | TC ps
  procMatches id dp { pid,pobjs,pdict }
    = id==pid && fst (pdict.unpack dp) || any (objectMatches id dp) pobjs
```

The search for the proper concrete object is handled recursively:

```
objectMatches :: ID PropDynamic (Object (Proc ps)) → Bool | TC ps
objectMatches id dp (Object _ cs) = any (compMatches id dp) cs
where
  compMatches :: ID PropDynamic (Comp ls (Proc ps)) → Bool | TC ls
  compMatches id dp (Obj {oid,odict,oobjs})
    = id==oid && fst (odict.unpack dp) || any (compMatches id dp) oobjs
  compMatches id dp (NewLS {newDef})    = any (compMatches id dp) newDef
  compMatches id dp (AddLS {addDef})    = any (compMatches id dp) addDef
```

Finally, it is convenient to have a version of addProperty that aborts in case the property type does not match the indicated object's state:

```
add :: (Maybe ID) UserProp Program → Program
add mid prop program
    # (ok,program) = addProperty mid prop program
    | ok           = program
    | otherwise    = abort ("Could not add "+++prop.name)
```

Given the global storage of properties in a Program, it is trivial to add a function that removes properties:

```
delProperty :: PropName Program → Program
delProperty name program=:{props}
  = {program & props=[(mid,[p \\ p←ps | p.name≠name]) \\ (mid,ps)←props]}
```

Example We continue our example on page 4 by extending it with properties. The only change of definition of program is the extension with two record fields pdict=dictTC and odict=dictTC at the appropriate places, as well as an empty properties list (props=[]).

We introduce a property for each kind of element, viz. program, process, and concrete object. They are:

```
singleProp                  // Program property
 = { name="singleProcess"
```

```
   , prop=dynamic (λprocs → length procs==1) :: Prop [Process]            }
sortedProp                 // Process property
 = { name="sortedProp"
   , prop=dynamic (λ{pstate=l} → l==reverse $ sort l) :: Prop (Proc [Int])}
lengthProp                 // Object property
 = { name="lengthProp"
   , prop=dynamic (λ(n,{pstate=l}) → n==length l) :: ∀a:Prop (Int,Proc [a])}
```

`singleProp` states that there is one single process at every stable state; `sortedProp` says that the integer list of the process is in reverse order; `lengthProp` defines that the integer value of the concrete object correctly keeps track of the length of the list of its parent object. Note that this function is polymorphic in the element type of the list state.

2.3 Testing Properties at Quiescence

In the previous section we have explained how properties of programs, processes, and concrete objects can be added and removed at run-time. In this section we show how these properties can be tested when the application is in a stable state, is quiescent. The function `reportProperties` evaluates all current properties of its program argument and collects the results in a report:

```
reportProperties :: Program → PropertiesReport
```

The report assigns, for a certain run, a *verdict* for each tested property. A verdict is a simple boolean, which is true iff the property holds.

```
:: PropertiesReport = { run     :: Int
                      , reports :: [PropertyReport] }
:: PropertyReport :== (Maybe ID,[(UserProp,Verdict)])
:: Verdict        :== Bool
```

To keep track of the runs, the program type is extended with a run-count that is incremented by `eval`:

```
:: Program = { ..., run :: Int }
```

The purpose of (`reportProperties program`) is to test every property in the `props` field of `program`. Recall that properties are boolean functions on the particular state of the object with which they are associated. This means that `reportProperties` must construct the appropriate state of each object at which the property function can be applied. The function can then compute the verdict simply by application of the property to the constructed state.

The top-level of this function is easily defined:

```
reportProperties :: Program → PropertiesReport
reportProperties program=:{run,props}
  = foldl (programProperties program) {run=run,reports=[]} props
```

The function application (`programProperties program pr prop`) needs to test a program property in case `prop` = (`Nothing`,`props`). We know that `props` contains (`Prop [Process]`) property functions because `addProperty` is type-safe.

```
programProperties :: Program PropertiesReport Property → PropertiesReport
programProperties program=:{procs} pr=:{reports} (Nothing,props)
  = {pr & reports=reports++[(Nothing,[ (p,programProperty p.prop procs)
                                     \\ p<-props
                                     ]
                              )]}
where programProperty :: PropDynamic → Prop Program
      programProperty (f :: Prop [Process]) = f
```

In case `prop = (Just id,props)` then the correct state context needs to be built for a process or a concrete object. First consider testing a process property. If a process is found with a `pid::ID` that matches `id` then we know that the list of properties `props` contains functions of type `(Prop (Proc ps))`, with `ps` the current value of its state. We can safely unpack every such property using the dictionary of the process (`pdict`) and apply it to the process. Note that we need to filter itself from the list of all processes in the `pcontext` field before doing so.

```
programProperties program=:{procs} pr (Just id,props)
  = foldl processProperties pr procs
where
  processProperties :: PropertiesReport Process → PropertiesReport
  processProperties pr (Process proc) = procProperties pr proc
  where
    procProperties :: PropertiesReport (Proc ps) → PropertiesReport
    procProperties pr=:{reports} proc=:{pid,pobjs,pdict}
      | id==pid
        = {pr & reports=reports++[(Just id,[ (p,snd (pdict.unpack p.prop) pst)
                                           \\ p<-props
                                           ])]}
      | otherwise
        = foldl (objectProperties pst) pr pobjs
    where
      pst = {proc & pcontext=[p \\ p=:(Process {pid})<-procs | pid≠id]}
```

Testing concrete object properties proceeds in an analogous way, except that now a local state structure needs to be built to which the properties can be applied. This is a recursive definition that follows the nested structure of the objects and their state scopes, which makes it a bit verbose.

```
objectProperties :: (Proc ps) PropertiesReport (Object (Proc ps))
                          → PropertiesReport
objectProperties pst pr (Object ls cs) = foldl (compProperties ls pst) pr cs
where
  compProperties :: ls (Proc ps) PropertiesReport (Comp ls (Proc ps))
                            → PropertiesReport
  compProperties ls pst pr=:{reports} (Obj {oid,odict,oobjs})
    | id==oid
      = {pr & reports=reports++[(Just id,[ (p,snd (odict.unpack p.prop) (ls,pst))
                                         \\ p<-props
                                         ])]}
    | otherwise
```

```
    = foldl (compProperties ls pst) pr oobjs
  compProperties _ pst pr (NewLS {newLS,newDef})
    = foldl (compProperties newLS pst) pr newDef
  compProperties ls pst pr (AddLS {addLS,addDef})
    = foldl (compProperties (addLS,ls) pst) pr addDef
```

Given the `reportProperties` function, it is straightforward to define extended `eval` functions that produce property report(s) and next program(s):

```
step :: Event Program → (PropertiesReport,Program)
step event program = (reportProperties program,eval event program)

steps :: [Event] Program → ([PropertiesReport],Program)
steps es program = seqList (map step es) program
```

Finally, for decent output, we define an instance of the `toString` function for a `PropertiesReport` that displays the number of tested properties, and names those that have failed in a particular run.

Example We continue our example on page 7. The `program` definition is extended with `run=0` field. We add the program, process, and object properties as discussed above, after which we want to inspect the property reports of all quiescent states that result from a specific scenario:

```
Start
  ♯ program      = add Nothing  singleProp program
  ♯ program      = add (Just 1) sortedProp program
  ♯ program      = add (Just 2) lengthProp program
  ♯ (rs,program) = steps scenario program
  = (map toString rs,program)
```

The scenario first picks the first action of concrete object 2, then takes the second action twice, and ends with the first action:

```
scenario = [(2,1),(2,2),(2,2),(2,1)]
```

This scenario is sufficient to reveal the first 'bug' in the program:

```
Step 0: tested 3 properties. 0 failing properties.
Step 1: tested 3 properties. 0 failing properties.
Step 2: tested 3 properties. 0 failing properties.
tl of []
```

This means that everything runs properly until just before the second invocation of the second action (step 2), but after doing that action apparently the tail of an empty list is taken. This bug is easily fixed by replacing `tl` with `tl‘`:

```
tl‘ :: [a] → [a]
tl‘ xs = if (isEmpty xs) xs (tl xs)
```

Running the program through the same scenario reveals the second 'bug':

```
Step 0: tested 3 properties. 0 failing properties.
Step 1: tested 3 properties. 0 failing properties.
Step 2: tested 3 properties. 0 failing properties.
Step 3: tested 3 properties. 1 failing property.
        lengthProp
```

Property `lengthProp` is violated immediately after the second invocation of the second action (detected by step 3). This is caused by the local integer that decreases below 0 and therefore incorrectly reflects the length of the integer list. We fix this bug by replacing `n-1` with `n.-.1`, defined as:

```
(.-.) infixl 6 :: !Int !Int → Int
(.-.) m n = max 0 (m-n)
```

Running the scenario again renders the properties invariant with respect to this event trace.

3 Object I/O is a Local State Transition Systems

In this section we explain the relation between the local state transition systems of the previous section with Object I/O. We do this in an informal manner, by means of an Object I/O program that is equivalent to the one shown in Section 2. A precise account has appeared earlier in [3].

As in the local state transition example, the program consists of a single interactive process. Instead of manipulating an integer list, this program manipulates an `Id` list (second argument of `Process`):

```
Start :: *World → *World
Start world
  = startProcesses [Process MDI [] initGUI [ProcessClose closeProcess]] world
```

In Object I/O the state of an interactive process is given by the record type (`:: PSt ps = {ls :: ps, io :: IOSt ps}`), with `ps` the state as discussed in the previous section. `io` is a combination of the fields `pobjs` and `pcontext`. It is an abstract data type by which the programmer must access all GUI elements and the external world. In this example, we have a (`PSt [Id]`) process state.

The `Id` values are used to identify windows that are opened and closed dynamically by the two actions `open` and `close`. These actions are the callback functions of two menu items, labelled "Open" and "Close" respectively. Their parent object is the top-level menu object that corresponds with the `Object` of the local state transition system, and indeed, it encapsulates an integer state (first argument of `openMenu`).

```
initGUI :: (PSt [Id]) → PSt [Id]
initGUI pst = snd $ openMenu 0 mDef pst
where
  mDef :: Menu (:+: MenuItem MenuItem) Int (PSt [Id])
  mDef    = Menu "&File"
              (   MenuItem "&Open"  [MenuFunction open ]
              :+: MenuItem "&Close" [MenuFunction close]
```

```
            )    []

open :: (Int,PSt [Id]) → (Int,PSt [Id])
open (n,pst=:{ls=l})
  # (wid,pst) = openId pst
  # wDef      = Window ("Window "+++toString (n+1)) NilLS [WindowId wid]
  # pst       = snd $ openWindow ⊥ wDef pst
  = (n+1,{pst & ls=[wid:l]})

close :: (Int,PSt [Id]) → (Int,PSt [Id])
close (n,pst=:{ls=[wid:l]}) = (n-1,closeWindow wid {pst & ls=l})
close (n,pst)               = (n,pst)
```

Recall that in the local state transition system there are three kinds of elements that can be probed:

Programs probed with (`Prop [Process]`). Object I/O has a data type similar to `[Process]` (viz. `Context`) but this is an internal data type and should not be accessed by the programmer. There are no retrieval operations defined on this data type, so basically, the programmer can not define program properties as in the previous section. (Note that this suggests that the API of Object I/O might be lacking functionality here, so it is worthwhile to see what useful access functions can be added.)

Processes probed with (`Prop (Proc ps)`). Object I/O processes are probed by (`Prop (PSt ps)`); in the example by (`Prop (PSt [Id])`).

Objects probed with (`Prop (ls,Proc ps)`). The GUI objects in the program are `mDef`, its two `MenuItem` elements, and the dynamically created windows (`wDef`). The menu and its items share an integer local state, so they are probed by (`Prop (Int,PSt [Id])`). The window has no significant local state (`⊥ :: ∀a:a`), so it is probed by (`∀a:Prop (a,PSt [Id])`).

4 Issues of Probing Object I/O Programs

In the previous section we have shown in a very informal way how Object I/O relates to local state transition systems. In this section we discuss the major issues that are likely to be occur when Object I/O applications are dynamically probed. First, dynamically adding/removing properties to an Object I/O program, requires *identification* of the GUI elements at run-time (Section 4.1). Second, an important issue when keeping track of properties is that they should be *free of side-effects* (Section 4.2). Finally, which state paradigm should be used: *explicit* or *implicit* state passing (Section 4.3).

4.1 Run-Time Identification of GUI elements

From the account in Section 2.2 we know that it is sufficient to retrieve the `Id` value of an element in order to associate a property with it using `addProperty`. The example in Section 3 shows that these identification values are known only

at run-time. When probing GUI elements dynamically, the user needs to identify them. For this purpose we include a *GUI browser* for each application with which the user can select a GUI element, and thereby its identification value. This browser can be defined in Object I/O using the API inspection functions, and a tree list control to present the hierarchical structure of the GUI. Fig. 1 gives screenshots of this browser for the example program at several stable states.

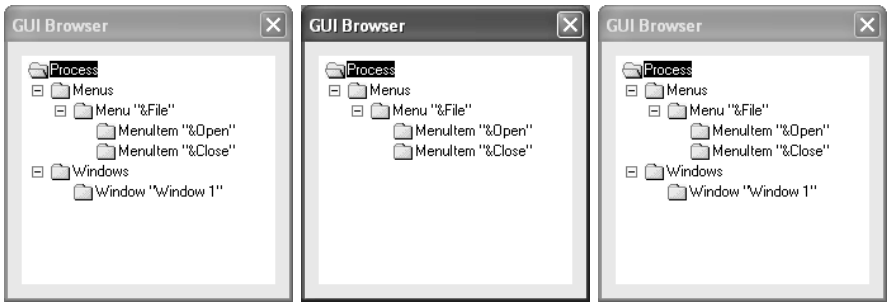

Fig. 1. The GUI browser after: [(2,1)], [(2,1),(2,2),(2,2)], and [(2,1),(2,2),(2,2),(2,1)].

Clean dynamics can be stored on disk, so the user can browse the file system in search of interesting properties, or create them using the Clean IDE and store them on disk. This gives the two arguments of the `addProperty` function, which should allow us to associate a stored property with a given GUI element.

4.2 Properties Should Have No Side-Effects

A fundamental problem with systems that allow some kind of runtime checking of program properties is the *side-effect problem* [8]. It states that the properties should, for obvious reasons, have no side-effects on the programs. In local state transition system terminology, this means that a property should not change the state that it inspects. In Section 2.2 this was effortlessly realized by defining a property over a state `st` as the simple function type `st → Bool`. We would like to adopt this simple scheme to Object I/O, but unfortunately this is not possible. The main reason is that the types in Object I/O have been designed to allow the programmer to use *unique* state, i.e. state that can be destructively updated [6]. This requires the 'container types' to be at least as unique as their content, which limits their flexibility. As a consequence, we are forced to use the following property type:

```
:: Prop st :== st → (Bool,st)
```

For instance, a concrete object property function with local state `ls` and process state (`PSt ps`) has type ((`ls`,`PSt ps`) → (`Bool`,(`ls`,`PSt ps`))).

How can we make sure that this property function has no side-effect? In Object I/O we identify the functions that are read-only, and those that have side-effects. Allow only read-only functions on the (`IOSt ps`) value of a (`PSt ps`). This is easy to do and can even be *enforced* by providing a dedicated subset of modules that have this property. What is left is a *proof obligation* that the non-`IOSt` states do not change in the property function.

A disadvantage of systems with proof obligations is that they require good support of proof tools such as Sparkle [10]. If they are not, formal proofs might become a mere formality.

4.3 The Influence Of The State Paradigm

Object I/O, as discussed so far in this paper, uses an *explicit* state passing paradigm. One advantage of this paradigm is that each object carries in its type full information about which state it manipulates, so we can quickly check if a property that is to be associated with an object actually matches the type of the state.

However, it is also possible to use an *implicit* state passing paradigm using `MVar`s [11]. This has been discussed in [1]. The `GUI` monad is then a regular `IOSt` state monad which uses `MVar`s to hold the logical state. Advantages of this approach are the simpler types of Object I/O GUI elements, and the ability to have more complex state structures without loss of control over access. The major disadvantage that was raised against its use is its less declarative nature because programmers need to explicitly *take* and *put* values from these variables.

Surprisingly, this paper identifies another disadvantage of implicit state passing: in contrast with explicit state passing, the type of an object no longer contains information on the state that is manipulated by the object. This means that our approach of unifying the type of a dynamic property with the state of an object no longer works. Instead, one needs to associate properties over `MVar`s that happen to be manipulated by objects. Identification and matching of `MVar`s against property types can be done in a similar way as object identification in Sections 4.1 and 2.2, but is more complicated because all `MVar`s must be retraceable for identification purposes, and a property of type ($\mathtt{st}_1 \ldots \mathtt{st}_n \rightarrow$ `GUI Bool`) must be matched against (`MVar` $\mathtt{st}_1$)$\ldots$(`MVar` $\mathtt{st}_n$).

5 Related Work

The point of the work described in this paper is clearly that of run-time checking of properties that the programmer (or interrogator) thinks to be true of the application. This area is also known as *assertion checking*, and has been integrated in the object-oriented languages Eiffel [9] and JML– Java Modelling Language [8]. We have in common with the Eiffel approach that we want to use *executable* properties. With the JML approach we share the reuse of the host language and libraries in order to encourage programmers to probe their applications. The main differences are: because of the side-effect problem JML can't handle I/O

methods, which is clearly a must in our case; we do not annotate source code for our properties, but rather probe the application at run-time using dynamically associated properties; this requires properties to be persistent, which is achieved using Clean dynamic technology; we do not intend these properties to be subject to formal verification as in JML.

Probing application properties dynamically has the same flavor as using a *tracing/debugging tool* such as Freja, Hat, and Hood [7] or those used in more conventional programming languages such as C. With such tools one inspects the run-time values of an application whereas we focus on relations between run-time values expressed as properties.

Another area that is related to our work is that of *testing* [15] because in both areas it is the *application* itself that is subject to probing and we can give verdicts only for specific event traces, which in practice will not exhaust the possible event trace search space. At this moment the theory and practice of testing of GUI applications is starting to grow. Our project is a first step to investigate what can be done in this area.

6 Conclusions and Future Work

In this paper we have shown how systems that are based on local state transition systems can be probed at run-time for their stable state based properties. These properties can be added and removed at any stable state of the application. There are no limitations to the size of the application. We have shown what needs to be done additionally for one particular instance of local state transition systems, the Object I/O library. This provides us with a directly usable means to probe GUI applications of arbitrary size.

There are many directions of research to take based on this framework. Among others these are: adding good property management functionality to the framework; extend it with value-inspection and back-tracing in case a property is found to be invalid; explore the formal verification potential of our approach.

Acknowledgements

The author would like to thank Marko van Eekelen for his comments.

References

1. P. Achten and S. Peyton Jones. Porting the Clean Object I/O library to Haskell. In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on the Implementation of Functional Languages, IFL'00, Selected Papers*, volume 2011 of *LNCS*, pages 194–213. Aachen, Germany, Springer, Sept. 2001.
2. P. Achten and R. Plasmeijer. Interactive Functional Objects in Clean. In C. Clack, K. Hammond, and T. Davie, editors, *Proc. of the 9th International Workshop on the Implementation of Functional Languages, IFL 1997, Selected Papers*, volume 1467 of *LNCS*, pages 304–321. St.Andrews, UK, Springer, Sept. 1998.

3. P. Achten and R. Plasmeijer. The implementation of interactive local state transition systems in Clean. In P. Koopman and C. Clack, editors, *Proceedings of the 11th International workshop on the Implementation of Functional Languages, IFL'99*, number LNCS 1868, pages 115–130. Springer-Verlag, Sept. 2000.
4. P. Achten and M. Wierich. A Tutorial to the Clean Object I/O Library - Version 1.2. Technical Report CSI-R0003, Computing Science Institute, Faculty of Mathematics and Informatics, University of Nijmegen, The Netherlands, Feb. 2000. 294 pages.
5. K. Angelov. ObjectIO for Haskell. Description and Sources at www.haskell.org/ObjectIO/, Applications at /free.top.bg/ka2_mail/, 2003.
6. E. Barendsen and S. Smeters. Uniqueness typing for functional languages with graph rewriting semantics. In *Mathematical Structures in Computer Science*, volume 6, pages 579–612, 1996.
7. O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood – A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on the Implementation of Functional Languages, IFL'00, Selected Papers*, volume 2011 of *LNCS*, pages 176–193. Aachen, Germany, Springer, Sept. 2001.
8. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the Design of JML Accomodates Both Runtime Assertion Checking and Formal Verification. In *Formal Methods for Components and Objects*, volume 2852 of *LNCS*, pages 262–284. Springer Verlag, 2003. Also available as Technical Report TR 03-04a, Department of Computer Science, 226 Atanasoff Hall, Iowa State University, Ames, Iowa, USA.
9. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
10. M. Mol, M. Eekelen, and R. Plasmeijer. Theorem proving for functional programmers - Sparkle: A functional theorem prover. In T. Arts and M. Mohnen, editors, *The 13th International Workshop on Implementation of Functional Languages, IFL 2001, Selected Papers*, volume 2312 of *LNCS*, pages 55–72, Stockholm, Sweden, 2002. Springer.
11. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308, St.Petersburg Beach, Florida, 1996. ACM.
12. S. Peyton Jones and Hughes J. et al. *Report on the programming language Haskell 98*. University of Yale, 1999. http://www.haskell.org/definition/.
13. M. Pil. Dynamic types and type dependent functions. In D. Hammond and Clack, editors, *Implementation of Functional Languages (IFL '98)*, LNCS, pages 169–185. Springer Verlag, 1999.
14. R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.kun.nl/∼clean/contents/contents.html.
15. J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 1996.
16. M. Vervoort and R. Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In R. Peña and T. Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 101–117. Springer, Sept. 2003.