Annales Univ. Sci. Budapest., Sect. Comp. 31 (2009) 107-146

A SEMANTIC MODEL FOR PROVING PROPERTIES OF CLEAN OBJECT I/O PROGRAMS[1]

M. Tejfel, T. Kozsik and Z. Horváth

(Budapest, Hungary)

Abstract. The functional programming language Clean has an extensive library, the Object I/O library, to build interactive, event-driven applications with graphical user interface. Furthermore, Clean has a dedicated theorem prover, the Sparkle system, to prepare machine-verified proofs for Clean programs. Unfortunately, the Object I/O library uses some features of Clean (uniqueness typing, existential types, non-Clean code) that are currently not supported properly by Sparkle. Moreover, Sparkle is not capable to deal with the size and the complexity of Object I/O within reasonable resource bounds. For these reasons Sparkle cannot handle Object I/O programs directly. However, creating machine-verified proofs for such Clean programs might be accomplished within a model of the Object I/O library, which is already manageable by Sparkle. This paper presents such a model. In this model the properties of the Object I/O library are expressed as axioms. These axioms serve as the building blocks of proofs about interactive, event-driven Clean programs. The paper illustrates this technique with proofs on some simple properties of a small event-driven program.

[1]Supported by GVOP-3.2.2.-2004-07-0005/3.0 ELTE IKKK and OMAA 66öu2 2007 and Ferenc Deák Scholarship.

1. Introduction

Due to referential transparency, reasoning about functional programs can be accomplished with fairly simple mathematical machinery, using, for example, classical logic and induction. This fact is one of the basic advantages of functional programs over imperative ones. Clean [1] is a lazy, pure functional language. It has a dedicated theorem prover, Sparkle [2, 3], which is built into the integrated development environment of Clean. Sparkle supports reasoning about Clean programs almost directly, so it can be used to create machine-verifiable proofs for Clean programs.

The graphical user interface (GUI) is an important part of those programs which have to deal with user interactions. Clean has an extensive library to build GUI applications: the Clean Object I/O library [4]. Unfortunately Sparkle is not capable to handle Object I/O programs directly; the uniqueness type annotations [5] of Clean and existentially quantified types, which are used heavily by Object I/O, are not properly supported in the proof tool. Moreover, some parts of Object I/O are implemented in C and ABC-code (the latter is a platform independent intermediate code in the Clean compilation chain), and contain calls to the operating system – these are also unsupported in Sparkle. Finally, the Object I/O library is so large and complex that Sparkle cannot handle it within reasonable time and memory bounds.

Being able to reason about the properties of interactive programs, however, would be extremely desirable. The solution proposed in this paper is the use of a model of the Object I/O library. The model, which is called SIO (Simplified Object I/O), abstracts away the visualisation aspect of Object I/O. It focuses on the processes that make up an interactive application and on the communication between those processes. User interaction is simulated, without having an actual graphical user interface. Therefore, the implementation of SIO does not contain operating system calls and is written in pure Clean (no C/ABC-code involved). SIO uses neither uniqueness annotations nor existential types and it is simple enough to be efficiently manageable in Sparkle. The public interface of the SIO library provides only a subset of the Object I/O functionality, but it does so using practically the same syntax. The semantics of SIO is defined in accordance with the expected behaviour of Object I/O. These characteristics make it possible to reason about important properties of many Object I/O applications with Sparkle and SIO.

The rest of the paper is structured as follows. First some concepts of the Object I/O library are explained through a GUI application (Section 2). This application will be used as an example in the forthcoming sections. Next the public interface of SIO and its relation to Object I/O are described (Section 3). Then the axiomatic semantics of (a subset of) Object I/O is introduced (Section 4). A proof of a property of the example application is sketched in Section 5. Finally, the discussion of related work and the conclusions are presented (Section 6).

2. Example: an Object I/O Application

This paper uses an implementation of the well-known "Slide game" for illustration purposes. In this game, the player has to slide tiles displaying numbers on a board in order to achieve the winning configuration, which is shown on the screen capture in Figure 1 (representing configurations in row major order, this corresponds to the sequence $\langle 1,2,3,4,5,6,7,8,\_\rangle$). Initially the tiles are shuffled (for the sake of simplicity, assume that the starting configuration is $\langle 8,7,6,5,4,3,2,1,\_\rangle$). There is also a progress monitor, which is drawing a chart during the game. The chart describes the progress of the player: vertical bars on the chart represent distances between the instantaneous and the winning configurations.

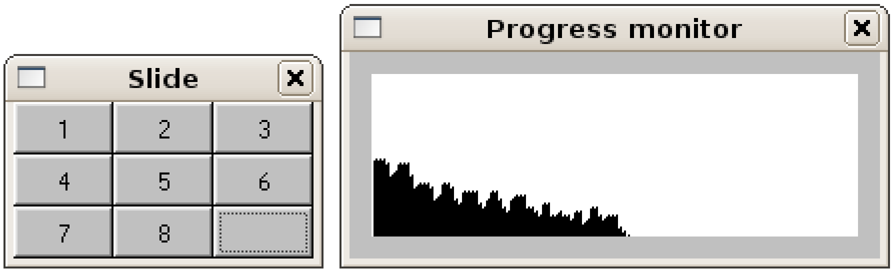

Figure 1. The winning configuration in the *slide game* application

The Object I/O library enables the implementation of an interactive, event-driven application as a collection of communicating state machines. In this particular application there are two state machines (two Object I/O processes), one being responsible for the game logic and the other one for the progress report. Each of the two processes has an associated dialog window. These two dialog windows provide the GUI for the application. The first dialog window is the game board, on which the tiles, as well as the hole, are represented with buttons; a tile neighbouring the hole can be moved (i.e. swapped with the hole) by pressing the corresponding button. The local state of the first process contains a mapping from board positions (buttons) to tile labels. This local state can be changed when the player slides a tile by pressing a button – hence the callback functions associated to the buttons implement the state transitions of the process. The two processes communicate: whenever the local state of the first process is changed, a message is sent to the second process. The reception of such messages triggers state transitions in the second process, which keeps track of the player's progress. The local state of the second process is the history of the values describing the distance between instantaneous and winning configurations. Whenever a message is received, this local state is updated by appending a value computed from the message and the chart is also updated accordingly.

The implementation of the above design is presented in Figures 2 and 3. Note that the program is written using the Object I/O library, as shown by the `import StdIO` statement. The entry point to the application is the definition of the `Start` function, which takes a `world` (the representation of the program environment) as an argument. This is a unique object, which is manipulated by the program, and which user interaction is performed with.

In the `slidegame` module three functions are defined on the global level: `slidegame` (which is the main part of the application), `find` (which is a helper function that returns the first element of a list `ls` satisfying a given property `pred` together with its position in `ls`), and `Start` (which starts the application with the initial configuration $\langle 8,7,6,5,4,3,2,1,\_\rangle$).

The `slidegame` function defines the two processes. First identifiers are created for the components of the GUI, namely for buttons (with `openIds`), and another identifier (with `openRId`) for communication between the two processes. The local state of the first process is represented with a list consisting of pairs. Each pair is a button identifier together with a number to be displayed as the content of the corresponding tile. The hole is represented with the number 0 and displayed as an empty label (see function `toLabel`). Initially the local state of the first process is a list produced from the button identifiers `ids` and from the problem description `problem` with the `zip2` function (the standard `zip2` function takes two lists and produces a list of pairs, each pair contaning the elements at the same position from the input lists). Buttons are created with the `button` function, which takes as arguments the identifier of the button, the identifier used for communicating to the monitor process, the number on the tile and an attribute describing its relative location in the dialog window. The ZF-expression containing the call to `button` draws the relative locations from `locs`, which is an infinite list of `Left` (leftmost in the next row) and `RightToPrev` (right to the previous) values. Each `Left` is followed by `cols-1` pieces of `RightToPrev`, resulting in a `rows` $\times$ `cols` grid layout of buttons. The buttons are then passed to the `game` function.

The `game_proc` and `monitor_proc` variables provide the descriptions of the two processes. The local state of the processes are initialized using `iniState` and `[]`, respectively (given as arguments to the `Process` data constructor). The "`game buttons`" and the "`monitor mId`" functions are used to complete the

```
module slidegame
import StdEnv
import StdIO

Start world = slidegame (3,3) [8,7..0] world

find pred ls = hd [(x,y) \\ x <- ls & y <- [0..] | pred x]

slidegame (rows,cols) problem world
  # (ids,world) = openIds (rows*cols) world
  # (mId,world) = openRId world
  # iniState = zip2 ids problem
  # locs = flatten (repeat [Left:repeatn (cols-1) RightToPrev])
  # buttons = [button id mId n loc \\ (id,n) <- iniState & loc <- locs]
  # game_proc = Process NDI iniState (game buttons) []
  # monitor_proc = Process NDI [] (monitor mId) []
  = startProcesses [game_proc, monitor_proc] world
where

  toLabel n   =   if (n==0) "" (toString n)

  button id mId n loc = ButtonControl (toLabel n) attrs
  with attrs = [ControlId id, ControlFunction (noLS (pressed id mId)),
                ControlWidth (PixelWidth 50), ControlPos (loc,NoOffset)]

  game buttons pst
    # attrs =[WindowItemSpace 0 0, WindowHMargin 0 0, WindowVMargin 0 0]
    = snd (openDialog Void (Dialog "Slide" (ListLS buttons) attrs) pst)

  pressed id mId pst=:{ls=state}
    # empty_data  = find (\(x,y)->y==0) state
    # my_data     = find (\(x,y)->x==id) state
    | neighbours (snd empty_data) (snd my_data)
       = refresh empty_data my_data mId pst
    = pst

  neighbours a b  =  let diff = abs (a - b) in
                     (diff == 1 && a/cols == b/cols) || (diff == cols)

  refresh ((id1,v1),p1) ((id2,v2),p2) mId pst=:{ls=state}
    # pst = appPIO (setControlText id1 (toLabel v2)) pst
    # pst = appPIO (setControlText id2 (toLabel v1)) pst
    # newState = ((updateAt p1 (id1,v2)) o (updateAt p2 (id2,v1))) state
    = snd (asyncSend mId (map snd newState) {pst & ls = newState})
```

Figure 2. The *Slide game* application, Part 1 of 2

```
slidegame (rows,cols) problem world
  ...
where
  ...

  monitor mId pst
    # (dId, pst) = openId pst
    # receiver = Receiver mId (noLS1 (received dId)) []
    # drawing = CustomControl area look [ControlId dId]
    # dialog = Dialog "Monitor" ( receiver :+: drawing ) []
    = snd (openDialog Void dialog pst)
    where

    size = rows * cols
    area = { w = 3*size^2, h = size^2 }
    pensize = 2

    fitness vals = area.h - (sum (map (\(i,v)->abs (i-v)) ps))
        with ps = zip2 [1..] [if (val==0) size val \\ val <- vals]

    received dId vals pst=:{ls=fitnessList}
      = appPIO (setControlLook dId True (True,newLook fs)) {pst & ls=fs}
        with fs = fitnessList ++ [fitness vals]

    look Able {newFrame} = (setPenSize pensize) o (setPenColour Black)
                            o (fill newFrame) o (setPenColour White)

    newLook fs Able _ pict  =  foldr draw_line pict coords
      where coords = zip2 [pensize,2*pensize..] fs
            draw_line = \(x,y) -> drawLine {x=x,y=area.h} {x=x,y=y}
```

Figure 3. The *Slide game* application, Part 2 of 2

initialization of the processes. The processes emerge via the evaluation of the `startProcesses` function: this function starts and initialises the two processes in the `world` environment received as its second argument, lets them interact with the environment, and finally returns the environment in which the processes are terminated. During the evaluation of the "`startProcesses [game_proc, monitor_proc] world`" expression, the two processes go through a sequence of state transitions. The applied state transition functions are defined as part of the process descriptions. The state transitions will eventually be triggered by GUI events.

The `game` function takes part in the initialization of the first (game) process. Its first argument, `buttons`, is a list of `ButtonControl` values, and its second argument, pst is a so-called process state. Process states have two roles: they contain the local (private) state of a process and they provide access to the global environment of the process. The state transitions of a process may modify the local state and (by interaction and communication) modify the global environment. When startProcesses calls game buttons, it will pass a process state that already contains iniState as the local state. Note that game buttons is the initial state transition of the game process. The game function opens the dialog window representing the game board with certain graphical attributes (e.g. WindowItemSpace). The dialog will contain the buttons passed in the first argument of game.

The button function refers to another set of state transition functions. A "control function" is assigned to each button: (pressed id mId) is executed whenever the button with identifier id is pressed (there is also a noLS function involved in the code, which is responsible for adjusting the type of the expression (pressed id mId); for the sake of brevity, this is not detailed any further here). The state transition function (pressed id mid) receives the current state of the containing process (the game process in this case) and returns the new process state. The third formal argument of the pressed function is an "as-pattern": it reveals that a process state pst is a record, which has a field ls. This ls is the (private) local state of the process, in this case the list containing pairs of button identifiers and tile values. The function looks up the pair describing the hole and the pair describing the button pressed. If the hole "neighbours" the selected tile, the process state is updated by refresh, otherwise the process state is left unchanged. The refresh function sets the label of the affected buttons, updates the local state of the process and sends the second elements of the pairs in the list asynchronously to the monitor.

Figure 3 presents the definition of the monitor function, which is involved in the initialization of the monitor process. This process contains a dialog with a drawing and a receiver. The callback function of the receiver is (received dId). Communication with the progress monitor is possible by sending messages to the receiver using the identifier mId. Whenever a message arrives, the state transition function ((received dId) vals) is called. The first argument to received identifies the drawing, the second argument is filled with the received message, and the third argument is the state of the monitor process. The local state component of the process state is a list of numbers, each number being a fitness value describing a state of the game. When the monitor receives the current configuration from the game process, it computes the fitness value of the configuration and appends it to the local state. Moreover, the drawing is also refreshed by setting its look to newLook fs, where fs is the updated local state. Initially the drawing is a white rectangle, as specified by the look function. In this white rectangle a vertical bar is drawn for each of the fitness values in the local state of the monitor process.

In summary, Object I/O programs are made up of processes, which can communicate by passing messages synchronously or asynchronously. Messages sent synchronously block the sender process until the message is processed by the corresponding receiver process. Processes manage windows and dialogs, which may contain graphical components (e.g. buttons) and communication targets (i.e. receivers). User interaction and communication activate state transition functions that can manipulate the state of the processes, update graphical components and initiate further communication.

3. A model of the Object I/O Library

Sparkle – the dedicated proof assistant for Clean – is not capable to deal with Object I/O applications directly. To enable reasoning about such applications, a model of the Object I/O library is introduced in this paper. The "Simplified Object I/O library", or Sio for short, concentrates on the main logic of Object I/O applications, namely the local states and the dialogues of the processes and the local states and the controls of the dialogues. Sio does not handle the graphical aspects of programs. The GUI of an Object I/O application is replaced with an event-driven state transition system with no graphical representation and no real-world effect. User interaction is simulated: a sequence of "user events" is supplied as a *parameter* to the simulation. The execution model of Object I/O and Sio programs is therefore quite different. On the one hand, an Object I/O program is non-deterministic, its execution depends on how the user acts through the GUI (although this non-determinism is hidden behind the "world" abstraction, apparent from the argument of the `Start` function in Figure 2). On the other hand, the execution of a simulation using the Sio library is determined by the sequence of user events supplied as the parameter to the simulation. However, non-determinism will be reintroduced in the Sio model at the level of the axiomatic semantics, as shown in Section 4.

The idea of this paper is to provide a library with a public interface that is syntactically similar to Object I/O. Programs written using the latter therefore can easily be transformed into ones written using the former. Although the current version of Sio covers only a fragment of the full Object I/O library, it still supports reasonably many GUI applications.

Before reasoning about an Object I/O application, like the one presented in the previous section, one has to slightly modify it, so that it can be compiled with the Sio library. The resulting source code can then be loaded into (a customized version of) Sparkle. Axioms describing the expected behaviour of Object I/O expressed in terms of Sio are provided in Sparkle; these axioms serve as the building blocks of proofs about the analysed application. Section 3.1 gives an overview of the interface of the Sio library and Section 4.2 presents the axioms that capture the semantics.

3.1. The programming interface of Sio

The Sio library offers a subset of the interface of Object I/O. It does so by preserving the syntax as much as possible. Therefore many of the exported definitions of Object I/O are simply repeated in Sio. Examples of such definitions are the algebraic data type `Void` with its data constructor `Void`, the algebraic data type `ControlWidth` with data constructors `PixelWidth`, `TextWidth` and `ContentWidth`, the (synonym) type constructor `IdFun` that is used to construct function types with identical domain and co-domain, the abstract data type `Picture` and the record type `Size`.

```
::Void = Void
::ControlWidth = PixelWidth Int | TextWidth String | ContentWidth String
::IdFun st :== st -> st
::Picture
::Size = {w :: !Int, h :: !Int}
```

In Sparkle all types have to be concrete. However, it is possible to define `Picture` as an abstract data type in Sio (and thus make Sio resemble to Object I/O as much as possible), because during the proving process the implementation of Sio – containing the full definition of `Picture` – will also be loaded into Sparkle. Therefore, `Picture` may remain abstract for programs using Sio, but it will still be a concrete type for Sparkle.

Other definitions of Object I/O appear in Sio without uniqueness type annotations. Such definitions are, for instance, the `IOSt` abstract type constructor, the `PSt` record type constructor, the `noLS` function, the `Ids` type class and the `setPenSize` function.

```
::IOSt a
::PSt  a = { ls :: !a, io :: !IOSt a }
noLS :: (a->b) !(c,a) -> (c,b)
class Ids env where
  openId   ::      !env -> (!Id,      !env)
  openIds  :: !Int !env -> (![Id],    !env)
  openRId  ::      !env -> (!RId m,   !env)
  openRIds :: !Int !env -> (![RId m], !env)
setPenSize:: !Int !Picture -> Picture
```

Some Object I/O definitions are supplied only partially in Sio – like the algebraic data type `WindowAttribute`, which has less data constructors in Sio than in Object I/O. The reason is that `Windows` are not supported in the current version of SIO, only `Dialogs`, therefore the `Window`-specific data constructors were omitted from SIO. Similarly, since the graphical aspects of Object I/O are not relevant for SIO, operations manipulating graphical appearance are often simplified in SIO. For example, `fill` is a (no-op) function in SIO, while it is a component of the type class `Fillables` in Object I/O.

```
:: WindowAttribute st
    = WindowActivate (IdFun st)     | WindowClose (IdFun st)
    | WindowDeactivate (IdFun st)   | WindowHMargin Int Int
    | WindowId Id                   | WindowIndex Int
    | WindowInit (IdFun st)         | WindowInitActive Id
    | WindowItemSpace Int Int       | WindowOuterSize Size
    | WindowPos ItemPos             | WindowViewSize Size
    | WindowVMargin Int Int         | WindowCancel Id
    | WindowOk Id
fill :: a !Picture -> Picture
```

Inside SIO the state of processes and the messages communicated by the processes are stored in data structures. Since different processes have states of different types, and messages sent in a program are usually also of different types, the data structures require special treatment. One possible solution is to use existentially quantified types; however, Sparkle does not support these properly. Hence SIO applies a different approach: types of local process states and messages are encoded in a universal format. The type `UniType` provides a simple encoding scheme, similar to the universal structural representation used by Clean generic programming. Conversion to and from `UniType` is possible via the two components `toUni` and `fromUni` of the type class `Uni`. This type class is already instantiated in SIO for some basic types (e.g. `Int`) and type constructors (e.g. lists) for convenience. Further instances might be necessary to define for types used as messages or local states of processes.

```
::UniType = Unit Int | Pair UniType UniType
class Uni a where
   toUni   :: a -> UniType
   fromUni :: UniType -> a
instance Uni Int
instance Uni [a] | Uni a
```

Some SIO definitions differ from their Object I/O counterpart only in that some type variables are required to support the conversions `fromUni` and `toUni`. For example, function `syncSend`, which can be used to send messages of type `b` synchronously, is typed in SIO in such a way that `b` is required to belong to the type class `Uni`.

```
syncSend  ::  !(RId b) b !(PSt a) -> (!SendReport, !PSt a)  |  Uni b
```

In order to simplify the types appearing in GUI programs, certain definitions are less polymorphic in Sio than their analogues in Object I/O. This is again necessary to facilitate the processing of the GUI programs in Sparkle. As a consequence, compared to Object I/O, some type information (e.g. restrictions on type parameters) are missing from some definitions of the Sio library. However, assuming that the program using Object I/O is type-correct, its counterpart using Sio will be type-safe.

Components of the GUI, such as buttons, labels or edit fields are called *controls* in Object I/O and are represented with different algebraic types. Every control type has a single data constructor, the name of which is the same as that of the type.

```
:: ButtonControl ls pst
     = ButtonControl String [ControlAttribute (ls,pst)]
```

All these control types are instances of the `Controls` type constructor class. Object I/O is open-ended: it is possible to define further control types any time by introducing further instances to this type constructor class. Sio, on the other hand, need not be open-ended, therefore a simpler approach is taken. In Sio all controls have the same type, `Controller`, which is a parametric abstract type. The different controls are created with different functions. These functions are named after the types and their data constructors used in Object I/O.

```
:: Controller ls a
ButtonControl :: String [ControlAttribute (ls,PSt a)] -> [Controller ls a]
               | Uni a & Uni ls
```

This function creates a list containing a single control, one that represents a button.

Controls can be composed in Object I/O as well as in Sio. In Object I/O the type constructors `:+:` and `ListLS` – both having data constructors with matching names – serve for this purpose (i.e. given an instance `c` of `Controls`, the type `ListLS c` is also an instance of `Controls`). Obviously, things are much simpler in Sio: two functions, namely `:+:` (which is an infix operator) and `ListLS` create composite controls.

```
(:+:)  :: [Controller ls a] [Controller ls a] -> [Controller ls a]
ListLS :: [[Controller ls a]] -> [Controller ls a]
```

The first concatenates two lists, and the second one flattens a list of lists.

The simpler type for controls results in simpler types for dialog windows. The `Dialog` type constructor and the `Dialog` data constructor of Object I/O are replaced with a simplified abstract type constructor `Dialog_` and a function `Dialog` in Sio. Object I/O allows the programmer to provide "window attributes" for dialogs through the third argument of the `Dialog` data constructor. Window attributes are not used in Sio, but for maximizing analogy to Object I/O, the `Dialog` function in Sio retains this argument.

```
:: Dialog_ ls a
Dialog :: String [Controller ls a] [b] -> Dialog_ ls a  |  Uni ls & Uni a
```

Processes also became simpler in Sio. In Object I/O an existentially quantified type variable in the definition of type `Process` makes the state of the processes private. Since one must be able to access the process states during correctness proofs, Sio introduces the `Process_` abstract type, which does not contain existential quantification and encodes process states with `UniType`. The encoding is performed by the `Process` function, which replaces the corresponding data constructor of Object I/O in a syntax-preserving manner. Similarly to dialogs, the last argument of `Process`, responsible for supplying "process attributes", is not used in Sio.

```
:: Process_
Process :: DocumentInterface a ((PSt a) -> PSt a) [b] -> Process_ | Uni a
openProcesses :: ![Process_] !(PSt a) -> PSt a  |  Uni a
```

In Clean the built-in type `World` serves as the abstraction of the program environment: user interaction, calls to the operating system and the alike are defined as transformations of a unique object of this type. Object I/O is also based on this concept: the slide game is started by giving a `world` to the `Start` function, which returns in turn a modified `world`. Many Object I/O functions, e.g. `startProcesses`, are defined as state transitions on `World`. In Sio the environment of GUI applications is represented differently. Since `World` is a built-in type of Clean, and not a definition in Object I/O, it had to be replaced with another abstract type: `World_`.

```
:: World_
otherworld :: World -> World_
startProcesses :: ![Process_] !World_ -> World_
```

The complete application programming interface of the Sio library is presented in Appendix A.

3.2. Transition from Object I/O to Sio

Thanks to the syntactical similarities between Object I/O and Sio, adaptation of Object I/O programs to Sio is usually simple. If the Object I/O program only uses features available in Sio, the following modifications are sufficient.

1. The `import` statement refering to Object I/O has to be changed to `import Sio`.
2. References to `World` have to be changed to `World_`. Practically this means that the unique world obtained as an argument of `Start`, and possibly modified by non-Object I/O functions, must be turned into `World_` by the `otherworld` function before passing it to the Sio library.
3. Types used as messages or local state of processes should belong to the `Uni` type class. Many types are already instatiated for this type class in Sio, but further instantiations might be necessary.
4. The adaptation becomes harder, if the source code of the GUI application contains explicit type information. As explained in the previous section, the types of elements of the Sio library is often different from those of the corresponding elements in Object I/O. In general, it is not necessary to type expressions and functions explicitly in Clean: type inferencing is used by programmers regularly, especially when uniqueness annotations are to be determined. However, Clean fails to infer the type of some functions (containing polymorphic recursion, higher rank polymorphism or tricky uses of type classes). In such cases types are to be declared by the programmer. Adapting such programs to Sio might require changes in those declarations, as described in the previous section: removing uniqueness annotations, simplifying polymorphic types, replacing different “control” types with `Controller`, etc.

In the case of the slide game application, the first two modifications are sufficient. Hence only the first few lines of the code are changed, as illustrated in Figure 4.

```
module slidegame
import StdEnv
import Sio

Start world = slidegame (3,3) [8,7..0] (otherworld world)
```

Figure 4. Adapting *slide game* to Sio

As explained above, Sio applications operate on `World_`, rather than on `World`. For example, the type of the `Start` rule presented in Figure 4 is the following.

```
Start :: World -> World_
```

Thus the evaluation of the `Start` rule in Sio simply returns a value describing the initial state of the GUI application and provides no ways of user interaction.

In contrast an Object I/O application does not terminate after the computation of the initial state, but goes on with interacting with the user until all processes of the application are closed. In the next section it is shown how to simulate such an execution in Sio.

There is a slight restriction on those Object I/O applications that are intended to be modelled in Sio – a restriction which has not been mentioned so far, and which is related to the callback functions attached to controls. Callback functions are evaluated when the containing control receives an event. The body of the callback function may change the local state of the enclosing dialog and that of the enclosing process. Furthermore, it may modify the global state by changing properties of other controls (setting the label in an EditControl, enabling/disabling a ButtonControl, etc.), and by sending messages to other controls, either synchronously or asynchronously. In Object I/O it is possible that a callback function sends a message to another control (a Receiver) synchronously, and then applies further transformations on the global state. This possibility is restricted in Sio: if a callback function sends a message synchronously, the returned global state can be used to send further messages (both synchronously and asynchronously), but it should be queried or updated *in no other ways*. For example, consider the following imaginary callback function of a Receiver. Assume that the received message is a tuple containing two identifiers, the first identifying a Receiver control and the second an EditControl.

```
badCallback message=:(rId,eId) (dialogState, pst)
  # pst1 = snd (syncSend rId "1st" pst)
  # pst2 = snd (syncSend rId "2nd" pst1)             // still ok
  # pst3 = appPIO (setControlText eId "Done") pst2   // problem
  = (dialogState, pst3)
```

This callback function is completely legal in an Object I/O program, but it is unsupported in Sio: the problem here is that `pst3` was obtained from `pst` by first sending two messages synchronously (this is still all right), and *then* applying `appPIO (setControlText eId "Done")`. This last step modifies the global state of the receiver, since it sets the text of the edit control identifed by `eId`. The axioms in Section 4 describing the semantics of GUI applications are devised in such a way that they cannot be applied on callback functions of such kind – hence these functions are simply considered meaningless. The reason for this limitation (which, observing existing Object I/O applications, seems to be relevant really rarely in practice) will be given in Section 4.1, after having presented more details on Sio.

Finally, a technical note should be made. The current version of Sparkle does not support properties and proofs with type class restrictions, although such support is planned for addition in a future version [6]. Sio, as presented in Section 3.1, was designed to accord with this (hopefully near) future version.

But until support for type classes is added to Sparkle, a workaround is required to enable the construction of machine verifiable proofs. The workaround is to temporarily get rid of class restrictions (such as `Uni`). Type parameters with class restrictions are replaced with concrete types. For example, instead of having process local states with types belonging to the `Uni` type class (and let Sio convert such states into `UniType` and back automatically as required), one should ensure that the process local states are of type `UniType`. The type of process local states are specified with the parameter of the type constructor `PSt`. Following the workaround, this type parameter should be replaced with `UniType`, so `PSt` becomes a type without a parameter.

```
::IOSt
::PSt = { ls :: !UniType, io :: !IOSt }
```

The same happens to other type constructors, e.g. `Controller` and `Dialog_`. Obviously, this removal of polymorphism heavily influences the transition from Object I/O to Sio as well. Since the types of the functions offered by the Sio library are changed by the workaround, the `fromUni` and `toUni` conversion functions must be called explicitly in Sio programs. For example, the `refresh` function in the slide game application is modified as shown in Figure 5. Firstly, the `ls` component of `pst`, viz. the local state of the executing process is converted from `UniType` to the original type `[(Id,Int)]` before it is used to compute `newSt`, which is also of type `[(Id,Int)]`. Secondly, when updating the `ls` field of `pst`, `newSt` is converted into `UniType`. Thirdly, the message sent with `asyncSend` is converted from `[Int]` to `UniType`.

```
refresh ((id1,v1),p1) ((id2,v2),p2) mId pst=:{ls=state}
  # state = fromUni state
  # pst = appPIO (setControlText id1 (toLabel v2)) pst
  # pst = appPIO (setControlText id2 (toLabel v1)) pst
  # newSt = ((updateAt p1 (id1,v2)) o (updateAt p2 (id2,v1))) state
  = snd (asyncSend mId (toUni (map snd newSt)) {pst & ls = toUni newSt})
```

Figure 5. Applying the workaround on the `refresh` function

Section 5 investigates some properties of the slide game application. That section indeed makes use of the presented workaround and the "simplified" Sio library.

3.3. The theorem proving interface of Sio

After the introduction to the syntax, and before the presentation of the semantics of Sio (and that of the modelled Object I/O library), some details about the inner workings of Sio should be given. These details are necessary for the formulation of the axioms of Section 4.2.

A Sio program can be considered as a state transition system. Such a program can be executed with the help of a scheduler, which triggers the state transitions. A state is made up of processes and events to be processed. State transitions result from the processing of events: in each state transition the scheduler selects an event and processes it. There are two kinds of events. "User events" model user interaction and "system events" implement message passing.

The processing of an event starts by the invocation of a "control function". Control functions are callback functions attached to controls (including `Receivers`). Controls are embedded in dialog windows, which are supervised by processes. Both dialogs and processes might have local states. System events (messages) are dispatched to the control functions of receivers, while user events are dispatched to the control functions of other controls (controls that have graphical appearance in Object I/O), e.g. `ButtonControls` and `EditControls`. The control functions receive the local states of the enclosing dialog and process as arguments, and return possibly updated local states. Furthermore, control functions have access to the `IOSt`, which can be used, for example, to send messages to receivers or to get information about other components of the GUI model. Thus `IOSt` represents the "global state" of the application.

When reasoning about a GUI application with Sio, one can analyse the behaviour of the application with respect to a sequence of user events. These user events model user interactions. Each user event corresponds to an action initiated by the user, such as pushing a button or typing some text in an editor field. This mechanism (the scheduler and its parameterization with user events) is made available by the `startUp` function.

```
:: Id = Id Int
startUp :: [(Id,UniType)] World_ -> World_
```

Simulating the execution of an Object I/O program is achieved by the evaluation of the `startUp` function on some arguments. The second argument of this function is the "world containing the Sio program" (i.e. the state transition system), and the first argument is the list of user events. Each event is a pair, where the first component identifies the control that is the target of the user action, and the second component provides additional information, encoded as a value of type `UniType`. For example, in the case of an `EditControl` this second component is the text typed in by the user, and in the case of a `ButtonControl` this second component is ignored. Identifiers carry an integer value and are created through the instances of the `Ids` type class; in the slide game application buttons receive their identifiers through a call to `openIds` on a `world`.

The `startUp` function returns the "world after the application processed the user events". It is also possible that `startUp` diverges: this means that if the user acts as described in the user event list, the GUI application never terminates. The reasons for non-termination might be, among others, an infinite sequence of user events and deadlocked communication.

As an example on the simulation of Object I/O programs, consider the slide game application. Assume that the identifiers returned by `openIds` carry the values 1, 2, ..., 9. Given a `world` of type `World`, the following expression will run the simulation in such a way that tile 1 is moved to the top-left corner.

```
let buttonsToPress = [6,3,2,1,4,7,8,5,2,1,4,7,8,5,2,1,4] in
let userEvents = [(Id i, Unit 0) \\ i <- buttonsToPress] in
startUp userEvents (slidegame (3,3) [8,7..0] (otherworld world))
```

Now assume that a property of the final state of an application has to be proven. It can be formulated in the following way:

$$\forall \mathsf{xs}::[(\mathsf{Id},\mathsf{UniType})].\ \forall \mathsf{world}::\mathsf{World}.\ A(\mathsf{xs}) \rightarrow P\big(\mathsf{startUp}\ \mathsf{xs}\ \pi(\mathsf{world})\big),$$

where A gives the assumptions on user events and the program environment, π is the SIO representation of the application, and P is the property to be proven. In the case of the slide game application of Figure 4 this might become

$$\forall \mathsf{xs}::[(\mathsf{Id},\mathsf{UniType})].\ \forall \mathsf{world}::\mathsf{World}.\ A(\mathsf{xs}) \rightarrow P\big(\mathsf{startUp}\ \mathsf{xs}\ (\mathsf{Start}\ \mathsf{world})\big).$$

The slide game application is deadlock-free, if it terminates for all finite sequences of user actions. This latter is guaranteed if for all well-defined finite lists of user events the result of `startUp` is defined.

$$\forall \mathsf{xs}::[(\mathsf{Id},\mathsf{UniType})].\ \forall \mathsf{world}::\mathsf{World}.\ \mathsf{eval}\ \mathsf{xs} \rightarrow \big(\mathsf{startUp}\ \mathsf{xs}\ (\mathsf{Start}\ \mathsf{world})\big) \neq \bot$$

The `eval` function class is used in Sparkle to formulate the "complete" definedness of nested lazy structures. For example, it is instantiated for `Int`s and for lists of `eval` instances in the standard Clean libraries in the following way.

```
instance eval Int where eval :: !Int -> Bool
                        eval n = True
instance eval [a]  |  eval a   where eval [x:xs] = eval x && eval xs
                                     eval [] = True
```

The instance for `Int` makes use of that the result of a function is only defined if its strict arguments are defined. Therefore `eval` $\perp$ is $\perp$ (i.e. not `True`), and for all `n` of type `Int` such that `n` $\neq \perp$, `eval n` $=$ `True`. The `eval` instance for lists is even more elaborated: `eval xs` $=$ `True`, if `xs` $\neq \perp$, the spine of `xs` is defined and finite, and for every element `x` of `xs`, `eval x` $=$ `True`.

To get `eval` working on lists of user events, further instantiations of the `eval` class are needed: for `Id`, for `UniType` and for pairs of `eval` instances. These instantiations are provided within SIO.

4. Semantics

This section presents an axiomatization of Object I/O. The intention is to capture the expected behaviour of this library by providing the formal semantics of its model, SIO. Reasoning about Object I/O programs will be accomplished with respect to the semantics of SIO and not to the real implementation of Object I/O. The semantics of SIO is described in two parts. The first part describes the kinds of information that is relevant to the model, together with the basic operations to query and update data. The second part – provided as a set of axioms rather than Clean definitions – describes the dynamic characteristics of SIO, namely how the processing of events takes place. The axioms will be written as lemmas in the used proof system, Sparkle. This approach results in a concise and high-level description of the semantics of SIO.

The use of axioms for defining the semantics of SIO rather than the implementation of the SIO scheduler is essential in expressing non-determinism (Section 4.2 explains the reason why non-determinism is required, see the axioms for `selectEv`). Indeed, the scheduler is implemented – hence simulations of Object I/O programs can be executed in SIO –, but this implementation should not be used in proofs. This means that the Reduce tactic of Sparkle should never be applied on the functions that make up the implementation of the scheduler. One means to achieve this is to apply the Opaque tactic of Sparkle on those functions. Another possibility, not yet implemented, is to automatically enforce this obligation by modifying (customizing) Sparkle in such a way that it disallows reducing a given set of functions.

4.1. Information stored in SIO

Proving properties of SIO programs, and even formulating more complex properties than the one about deadlock-freedom shown in Section 3.3, is impossible without understanding how information is stored in SIO. Figure 6 presents the most important type definitions that occur in the implementation of SIO.

```
:: RId mes :== Id

:: Controller ls a
      = ButtonController Id String ((ls,PSt a) -> (ls,PSt a))
      | ReceiverController Id (UniType (ls,PSt a) -> (ls,PSt a))
      | CustomController Id
      | ...

:: Dialog_ ls a = Dialog_ [Controller ls a]
:: Process_ = Process_ UniType ((PSt UniType) -> PSt UniType)

:: SendType = Sync [Id] | Async Id
:: Event = Event Int Id UniType SendType

:: Dialog_Inner = Dialog_Inner UniType [Controller UniType UniType]
:: Process_State = ...
:: Process_Inner = Process_Inner Id Process_State [Dialog_Inner]
:: State = State [Process_Inner] [Event] [Event] Int Id Bool

:: IOSt a :== State
:: World_ :== PSt Void
```

Figure 6. Data structures used inside SIO

The inner state of a SIO application is of type `State`. This state is made up of the processes, the user event queue, the system event queue, the store for fresh identifiers, the identifier of the running process (similarly to controls an identifier is assigned to each process) and a boolean flag (to be explained later). The separation of the user event queue and the system event queue is useful for expressing non-determinism – this will be discussed further in Section 4.2.

For all type variables `a`, the type `IOSt a` is just a synonym for `State`. Furthermore, `World_` is a record of two fields: field `ls` is `Void` and field `io` is `State` again.

When building up a GUI application through the API of SIO, the programmer defines `Dialog_` and `Process_` structures. These are turned into `Dialog_Inner` and `Process_Inner` internally. In the slide game application, for instance, this is achieved through the call to the `startProcesses` function. `Process_Inner` serves as the description of a process, containing the process identifier, the state of the process (not detailed any further in Figure 6) and the dialogs of the process. The local state of the process is part of the process state. Dialogs are stored internally as `Dialog_Inner` structures, which contain the local state of the dialog and its controls.

The different controls are represented by the `Controller` type. A `Controller` only stores the information which is relevant for the SIO model. For example, a button is represented with a `ButtonController` tag, an identifier, a label and a callback function. `Id` and `RId` are used to identify GUI controls and receivers, respectively. No distinction is made internally between these two types.

User and system events are represented with the `Event` type. Each event has an identifier (simply an `Int` value, because it is not accessible through the API of SIO), the identifier of the control to be activated by the event, the content of the event (encoded in `UniType`) and the designation of the `SendType`. An event can be sent synchronously (messages sent with `syncSend` generate synchronous system events) and asynchronously (system events generated by messages sent with `asyncSend` and user events). An asynchronous event stores the identifier of the sender process, and a synchronous event stores both the identifier of the sender process and the identifiers of all the processes that are blocked on that synchronous event. Blocked processes result from chains of `syncSend` calls. When a callback function in a process calls `syncSend`, the process becomes blocked until the sent message is processed by the target receiver. Becoming blocked means that the executing callback function is suspended and the process is unable to consume further events.

As mentioned earlier, there is a restriction in SIO on `syncSend` operations: in a callback function of a control, after the application of `syncSend` the global state of the application can only be changed by further message send operations. This restriction is necessary to avoid a synchronization anomaly in the implementation of SIO. The problem is the following. In Object I/O the execution of a callback is not always an atomic action: if it contains a `syncSend` call, the execution is suspended (the executing process is blocked) until the sent message is processed by the callback function of the target control. During this time other callback functions can be executed. SIO simulates this behaviour to a certain extent: if a callback in a process calls `syncSend`, the scheduler will not dispatch any events to the process until the synchronous message is received and processed by its target callback function. However, internally `syncSend` is not a blocking operation. Each callback is executed as an atomic step in SIO, whether or not it contains a `syncSend` call. Therefore, if a callback invokes `syncSend` and afterwards accesses the global state, the callback will behave differently in Object I/O and in SIO. In Object I/O the accessed global state will be guaranteed to be a state *after* the synchronous message was processed, while in SIO such a guarantee does not exist. The restriction, however, is fairly relaxed. It is allowed in a callback to send messages (both synchronously and asynchronously) after a call to `syncSend`. This is made possible by a trick – and by the boolean flag in the `State` type. When a callback function is executing, a call to `syncSend` sets the flag, which remains set until the end of the callback function. The flag modifies the behaviour of subsequent message send operations: they will not actually send messages, but only store messages in a “blocked event queue” within the current process. When the callback of the target control of the `syncSend` completes, the messages in the blocked event queue can be effectuated. The boolean flag in `State` has another role as well: the operations modifying `State` (other than `syncSend` and `asyncSend`) are paralyzed when the flag is set.

The `startUp` function is responsible to fill the user event queue with events modelling user interaction from its first argument. When this is done, the scheduler is started. The way the scheduler works is further explained now.

4.2. Axioms

The axioms that describe how the processing of events takes place are listed in Appendix C. To give an overview, some of the axioms are explained in detail here. To increase readability, universal quantification of variables is omitted and the premises are separated from the conclusion with a horizontal line.

Let `pst` be of type `State`. The functions `uevsOf` and `sevsOf` return the user and the system event lists of `pst`, respectively.

$$\frac{\mathsf{uevsOf\ pst} = [\,], \quad \mathsf{sevsOf\ pst} = [\,]}{\mathsf{scheduler\ pst} = \mathsf{pst}} \qquad \frac{\mathsf{uevsOf\ pst} \neq [\,] \ \vee\ \mathsf{sevsOf\ pst} \neq [\,]}{\mathsf{scheduler\ pst} = \mathsf{scheduler\ (step\ pst)}}$$

The first two axioms describe the `scheduler` function, which takes the program state as argument, and returns a possibly updated program state. If there are no events, the scheduler terminates, otherwise it performs `step` and then goes on.

Function `allblocked` returns true if and only if for all events in the user and system event queues, the target process of the event (the process supervising the dialog that contains the target control of the event) is blocked at a synchronous communication.

$$\frac{\mathsf{allblocked\ pst.io}}{\mathsf{step\ pst} = \mathsf{pst}} \qquad \frac{\neg\mathsf{allblocked\ pst.io}}{\mathsf{step\ pst} = \mathsf{processEv\ (selectEv\ pst)\ pst}}$$

These two axioms specify the `step` function. If there is an event that can be processed (the target process of the event is not blocked), the “first” such event is selected (by `selectEv`) and processed (by `processEv`). If no events can be processed, `step` simply returns its argument. Compare this with the axioms of `scheduler`: if there are unprocessed events, but no event is processable, the scheduler never returns; this corresponds to deadlock.

Function `isMember` is from the standard library of Clean: it decides whether an item is a member of a list. Function `blocked` returns true when the target process of the given event is blocked. The `pos` function returns the index of an item in a list; the result is the length of the list if the given item is not in the list. Function `target` returns the identifier of the target control of the event: the control that should react to the event. Finally, `sender` returns the identifier of the process that sent the message which was the origin of the event.

$$\frac{\neg\mathsf{allbocked\ pst.io},\quad \mathsf{selectEv\ pst} = \mathsf{e}}{\mathsf{isMember\ e\ (uevsOf\ pst)}\ \vee\ \mathsf{isMember\ e\ (sevsOf\ pst)}}$$

$$\frac{\neg\mathsf{allbocked\ pst.io},\quad \mathsf{selectEv\ pst} = \mathsf{e}}{\neg\mathsf{blocked\ e\ pst.io}}$$

$$\frac{\mathsf{pos\ e_1\ (uevsOf\ pst)}\ <\ \mathsf{pos\ e_2\ (uevsOf\ pst)},\quad \neg\mathsf{blocked\ e_1\ pst.io}}{\mathsf{selectEv\ pst} \neq \mathsf{e_2}}$$

$$\frac{\begin{array}{c}\mathsf{pos\ e_1\ (sevsOf\ pst)}\ <\ \mathsf{pos\ e_2\ (sevsOf\ pst)},\quad \neg\mathsf{blocked\ e_1\ pst.io}\\ \mathsf{target\ e_1} = \mathsf{target\ e_2},\ \ \mathsf{sender\ e_1} = \mathsf{sender\ e_2}\end{array}}{\mathsf{selectEv\ pst} \neq \mathsf{e_2}}$$

The four axioms above describe how the next event to process is selected. The first axiom says that `selectEv` returns an event from one of the event queues, and the second axiom adds that the target process of the event is not blocked. The third axiom says that in the case of the user events, processable events are served in the order of their occurrence in the queue. The fourth axiom says that two events from the same process to the same control will be processed in the order of generation. No scheduling restriction is postulated in the case of system events coming from different processes or going to different controls, and in the case of a race between a system event and a user event. This approach introduces non-determinism in the axioms. Correctness proofs should not rely on the order in which unrelated system events are processed. The axioms are devised in such a way that a property of a program can only be derived if the property holds for arbitrary scheduling. Moreover, whether a user event or a system event is processed at a given instance of time may depend on the speed with which the user interacts with the program. The non-determinism in the axioms provides the necessary abstraction to discard exact timing issues when constructing proofs.

Given an identifier, `controlId` decides whether it identifies a control. Moreover, `remove` removes (the first occurrence of) an item from a list.

$$\frac{\mathsf{pst.io} = \mathsf{State\ ps\ uevs\ sevs\ is\ cp\ fl},\quad \neg\mathsf{controlId\ (target\ e)\ ps}}{\mathsf{processEv\ e\ pst} = \{\mathsf{pst}\ \&\ \mathsf{io} = \mathsf{State\ ps\ (remove\ e\ uevs)\ (remove\ e\ sevs)\ is\ cp\ fl}\}}$$

This axiom can be used to get rid of invalid events: if the target of an event is not an identifier of a control, `processEv` simply removes the event from the event queues. Obviously, the event will be in one of the event queues; `remove` will remove it from that event queue, and it will leave the other queue unchanged.

The `procLS` and `diaLS` functions return the local state of the process and that of the dialog containing a given control. Function `control` returns the `Controller` structure corresponding to a control identifier in a program state. The `isButton` function decides whether a control is a button and $\mathtt{callback}_{\mathsf{b}}$ returns the control function associated to a button control. Finally, `updState` updates the program state.

```
updState :: Event UniType (PSt UniType) -> State
```

The equality

```
updState e lsd {ls = lsp, io = State ps uevs sevs is cp flag}
                              =
           State ps' uevs' sevs' is cp False
```

means that

- in `ps'` the local state of the dialog and the process that contains the control identified by "`target e`" is `lsd` and `lsp`, respectively, otherwise `ps'` is the same as `ps`, and
- the event is removed from the event queues: "`uevs' = remove e uevs`" and "`sevs' = remove e sevs`".

$$\frac{\begin{array}{c}\mathsf{async\ e}, \quad \mathsf{t} = \mathsf{target\ e}, \quad \mathsf{ls_p} = \mathsf{procLS\ t\ pst}, \quad \mathsf{ls_d} = \mathsf{diaLS\ t\ pst} \\ \mathsf{ctrl} = \mathsf{control\ t\ pst}, \quad \mathsf{isButton\ ctrl}, \quad \mathsf{fun} = \mathsf{callback_b\ ctrl} \\ (\mathsf{ls'_d}, \mathsf{pst'}) = \mathsf{fun}\ (\mathsf{ls_d}, \{\mathsf{ls} = \mathsf{ls_p}, \mathsf{io} = \mathsf{setCurrPr\ t\ pst.io}\}) \\ \mathsf{s} = \mathsf{updState\ e\ ls'_d\ pst'}\end{array}}{\mathsf{processEv\ e\ pst} = \{\mathsf{pst}\ \&\ \mathsf{io} = \mathsf{s}\}}$$

The last presented axiom describes how to process a user event that corresponds to pushing a button on the GUI. The button is looked up, the local state of the containing dialog and process is determined, and its control function is called. The control function takes the local state of the dialog ($\mathtt{ls}_{\mathsf{d}}$), the local state of the process ($\mathtt{ls}_{\mathsf{p}}$) and the state of the complete program (`pst.io`) as arguments. It returns updated local states, and – e.g. if some messages are sent or other controls are modified – an updated program state. The final step is to refresh the local state of the containing process and dialog in the program state and to remove the processed event from the event queue. This is accomplished with the `updState` function.

5. Reasoning with Sio and Sparkle

The way to use the Sparkle proof tool and the Sio library together will be illustrated now with two examples; the first one is more general, while the second one is specific to the slide game application. The first example shows that meaningless user input has no effect. User interaction is simulated in Sio by calling `startUp` with a list of user events. Since it is possible to pass malformed user events (e.g. events with invalid target) to this function, care must be taken to ignore such events. This approach is analogous to the way `syncSend` and `asyncSend` in Object I/O handle messages with non-existing targets. The rule that captures this situation is the ninth axiom of Section 4.2. Note that in general it is impossible to tell whether an event has a valid target in advance, by simply looking at the parameters passed to `startUp`. The reason is that the process structure need to be static neither in Object I/O, nor in Sio: the GUI program might create processes (and dialogs with controls) dynamically, e.g. as a response to a former event. Therefore the validity of the target identifier of an event depends on when the event is processed. The property that is formulated here is that if the user of `slidegame` does not do anything meaningful, then the program does not do anything either. If all the user events have invalid target with respect to the initial process structure, then the process structure will not change, hence all the events will be invalid (and will be ignored) when they are processed.

The `controlId` function can be used to decide whether a given identifier indeed identifies a control in a process structure. The predicate `all` is from the standard library; it can be used to universally quantify a predicate over the elements of a list. The predicate describing that all events in a list of events have invalid targets is the following.

```
wrongAll :: ![Event] ![Process_Inner] -> Bool
wrongAll events processes  =  all wrong events
  where  wrong event  =  not (controlId (target event) processes)
```

Now the desired property is written as follows.

$$\begin{array}{l}\forall \mathsf{processes} :: [\mathsf{Process\_Inner}] \quad \forall \mathsf{uevs} :: [\mathsf{Event}] \\ \forall \mathsf{idStore} :: \mathsf{Int} \quad \forall \mathsf{currProc} :: \mathsf{Id} \quad \forall \mathsf{flag} :: \mathsf{Bool} \\ \mathsf{wrongAll\ uevs\ processes} \rightarrow \\ \mathsf{scheduler}\ \{\mathsf{PSt}|\mathsf{ls} = \mathsf{toUni\ Void},\ \mathsf{io} = \mathsf{State\ processes\ uevs\ []\ idStore\ currProc\ flag}\} \\ \qquad = \{\mathsf{PSt}|\mathsf{ls} = \mathsf{toUni\ Void},\ \mathsf{io} = \mathsf{State\ processes\ []\ []\ idStore\ currProc\ flag}\}\end{array}$$

This property can be proved by induction – either on `uevs` or on the length of `uevs`. Both approaches have advantages: the former is a bit simpler, and the latter is more general. If induction on `uevs` is used, one has to show that the scheduler (i.e. `selectEv`) will always select the first event in the user event queue (note that events having wrong target are never blocked and never generate system events). If induction on the length of `uevs` is used, one has to prove that removing an element from a finite list reduces the length of the list. The latter approach is more general in the sense that it can also be used when some events in the user event queue might be blocked (which is not the case in this example), and hence the events are processed in an order different from how they show up in the queue. In this paper a proof using induction on the length of the user event queue is sketched. For this reason the following lemma is introduced and proved by induction on `n`.

$$\begin{aligned}
&\forall \mathsf{n} :: \mathsf{Int} \quad \forall \mathsf{processes} :: [\mathsf{Process\_Inner}] \quad \forall \mathsf{uevs} :: [\mathsf{Event}] \\
&\forall \mathsf{idStore} :: \mathsf{Int} \quad \forall \mathsf{currProc} :: \mathsf{Id} \quad \forall \mathsf{flag} :: \mathsf{Bool} \\
&\mathsf{length\ uevs} = \mathsf{n} \rightarrow \\
&\mathsf{wrongAll\ uevs\ processes} \rightarrow \\
&\mathsf{scheduler}\ \{\mathsf{PSt}|\mathsf{ls} = \mathsf{toUni\ Void},\ \mathsf{io} = \mathsf{State\ processes\ uevs\ []\ idStore\ currProc\ flag}\} \\
&\qquad = \{\mathsf{PSt}|\mathsf{ls} = \mathsf{toUni\ Void},\ \mathsf{io} = \mathsf{State\ processes\ []\ []\ idStore\ currProc\ flag}\}
\end{aligned}$$

Since $\forall \mathsf{uevs}\ \forall \mathsf{processes}\ (\mathsf{wrongAll\ uevs\ processes} \rightarrow \mathsf{length\ uevs} \geq 0)$, the cases $\mathsf{n} = \bot$ and $\mathsf{n} < 0$ are easy to prove. The case $\mathsf{n} = 0$ means that $\mathsf{uevs} = []$, so the first axiom of Section 4.2 applies. The last case of the induction is the inductive case. The second, the fourth, the fifth and the ninth axioms are needed to dissolve this case.

The proof of this property consists of about 1000 proof steps. It takes about eight seconds to verify it in Sparkle using a 1600 MHz Pentium M machine. The Sparkle `sec` file containing the machine-verifiable proof is available at [7].

The second example is related to the slide game application. The property analyzed here is, roughly, that there is always a hole on the board, a button without a number printed on it (actually, a stronger statement also holds, i.e. there is always *exactly one* hole on the board). More precisely, it is shown that the list that forms the local state of the *game* process contains a pair with zero as the second component (remember that the hole is represented with a zero in the local state). Clearly, this property will hold during the whole game, but here it is proven merely that the property holds after a single user action. Based on experience from this proof, one could construct an inductive proof (similarly to the previous example) for the more general property, but that is not covered in this paper.

Again, a predicate is defined that queries the data structures exposed by SIO for theorem proving. Given an argument of type `PSt a` for some `a`, the predicate takes the local state of the first process, converts it from `UniType` to `[(Id,Int)]` (the explicitly typed `typedFromUni` provides type information necessary for Sparkle), and searches for the value 0 among the second components of the list elements. For simplicity, the definition of the `localState` function is omitted.

```
containsZero :: (PSt a) -> Bool
containsZero {io=State processes _ _ _ _ _} =
  isMember 0 (map snd (typedFromUni (localState (hd processes))))
  where  typedFromUni :: UniType -> [(Id,Int)]
         typedFromUni val = fromUni val
         localState (Process_Inner _ proc_state _) = ...
```

As mentioned in Section 3.2, properties with class restrictions are not yet supported in Sparkle, therefore a workaround is applied in order to avoid typing problems during proof construction. According to this workaround, `PSt a` is turned into a non-parametric type `PSt` by replacing the type parameter with `UniType`.

The property to prove can be written in the following way.

$$\forall \mathsf{e} :: \mathsf{Event} \quad \forall \mathsf{world} :: \mathsf{World} \quad \Big(\mathsf{eval\ e} \rightarrow \mathsf{containsZero}\ \big(\mathsf{startUp}\ [\mathsf{e}]\ (\mathsf{Start\ world})\big)\Big)$$

For proving this property, many of the Sio axioms are required, although those related to synchronously sent messages and blocked processes are not needed. Note, however, that processing the user event might trigger an `asyncSend` (if the target of `e` is a button neighbouring the hole), which will add an event to the system event queue. The scheduler will process this system event as well.

The complete proof of the above property is about 2300 steps (the proof is available at [7]). At first sight it might be surprising that Sparkle requires 300 MB of heap space to verify the proof, and that this verification takes more than an hour on a 1600 MHz Pentium M machine. It is very tedious to construct proofs if some of the proof steps take tens of minutes. For this reason some technical tricks might be useful to employ. One trick is to introduce an abbreviation whenever a large expression springs up during the proof, and another trick is to use many "section files" containing incomplete proofs. The approach Sparkle follows is to store one or more complete proofs in each section file, and to use them as lemmas for proving properties in other section files. Assume, for example, that the section file A proves a property used in another section file B. The problem with this approach is that when the property is applied as a lemma in B, Sparkle loads (and verifies the proofs in) A. If the verification of A requires lots of memory and lots of time, working on B becomes rather cumbersome. The proving process could be made much more efficient if Sparkle provided more support for modularizing proofs – similarly to how the compilation of different compilation units is separated from each other and from the linking of library units in many programming languages. To alleviate this problem, the following guidelines were invented while assembling the proof for the property of the second example.

- If a large expression springs up during the proof (e.g. a `PSt` object containing the Sio model of the Object I/O application with all the `Controllers` and callback functions), then copy-and-paste this expression into a Clean source file, turn it into a function definition, and use this abbreviation instead of the large expression (this can be achieved with the Rewrite tactic).
- If the incomplete proof(s) in a section file requires too much resources (time and memory) to work with, create new section files, move the unproven subgoals into them and work with these section files.
- Do not refer from one section file to another. Do not apply properties proven in one section file as lemmas for proofs in another section files.
- Create copies of section files in which you can check that a property proved in one section file can be used as a lemma in a proof in another section file. Do this for pairs, do not combine them transitively.

This second example reveals that the ability to describe the behaviour of an Object I/O program in temporal logic (rather than the classical logical approach used in this paper) would highly facilitate reasoning. The property that there is always a hole on the game board is best to describe as an "always true" property of the slide game application. Sparkle-T [8, 9], which extends Sparkle with support for temporal logic, could be used for this purpose – its integration with Sio is planned as future work.

6. Conclusion

This paper presented a model of the Clean Object I/O library. Based on this model it is feasible to reason about interactive Clean programs with Sparkle, the dedicated theorem prover for Clean. The model – Sio – has a programming interface supporting a part of the Object I/O functionality. Many programs written using the Object I/O library can be transformed into the model. In some cases the transformation is almost trivial, but it becomes harder if types of functions are explicitly declared in the code. Sio models the semantics of (a part of) Object I/O by exposing information on the modeled GUI application to the prover and by providing axioms (described as lemmas in the proof tool) that can be used for the proofs.

Considering related work, a framework for reasoning about file I/O in Clean and Haskell programs is described in [10, 11, 12]. The semantics of file operations is defined in an explicit model of external world-state. Proofs are performed regarding the observable effect of real-world functional I/O. The Sparkle proof-assistant was used for machine-verifying these proofs. Sparkle does not support

Clean programs with I/O, so the proofs are meta-proofs. Paper [13] introduces a model for interactive programs. That approach is more general than the one presented here. The model discussed in this paper focuses on Object I/O applications, hence the transformation of real Clean programs into this model is expected to be simpler.

In Sparkle classical logic can be used to describe properties of programs. In the case of interactive applications, however, further useful properties can be expressed with temporal logic. Sparkle-T [8, 9], an extension to Sparkle supports some temporal logical operators. This machinery could make reasoning about communicating Object I/O processes much simpler, therefore in the future it will be investigated how to combine the power of Sparkle-T with the Sio model.

Acknowledgement

We would like to express our gratitude to the reviewers for the precious feedback that helped improve the paper significantly. We are also obliged to the Clean team at the Radboud University of Nijmegen, especially to Marko van Eekelen, Maarten de Mol, Peter Achten and Rinus Plasmeijer for their conceptual support and valuable technical assistance.

References

[1] **Plasmeijer R. and van Eekelen M.,** Concurrent Clean Version 2.0 Language report, `http://clean.cs.ru.nl/download/Clean20/doc/CleanRep2.0.pdf`, 2001.

[2] **de Mol M., van Eekelen M.C.J.D. and Plasmeijer M.J.,** Theorem proving for functional programmers, *IFL,* eds. T.Arts and M.Mohnen, M., LNCS **2312**, Springer Verlag, 2001, 55–71.

[3] **de Mol M., van Eekelen M. and Plasmeijer R.,** Proving properties of lazy functional programs with Sparkle, *Central-European Functional Programming School: Second Summer School, CEFP 2007, Cluj-Napoca, Romania, June 23-30, 2007. Revised Selected Lectures*, Springer Verlag, 2008,

41-86.
`http:dx.doi.org/10.1007/978-3-540-88059-2_2`

[4] **Achten P. and Plasmeijer M.J.,** Interactive functional objects in Clean, *Implementation of functional languages,* eds. C.Clack, K.Hammond and A.J.T.Davie, LNCS **1467**, Springer Verlag, 1997, 304–321.

[5] **Barendsen E. and Smetsers S.,** Uniqueness typing for functional languages with graph rewriting semantics, *Mathematical Structures in Computer Science*, **6** (1996), 579–612.

[6] **van Kesteren R., van Eekelen M.C.J.D. and de Mol M.,** Proof support for generic type classes, *Trends in Functional Programming, Vol. 5.,* ed. H.W.Loidl, Intellect, 2004, 1–16.

[7] **Kozsik T. and Tejfel M.,** Correctness of distributed functional programs, `http://aszt.inf.elte.hu/~fun_ver/index.html.en#ToC12`, 2007.

[8] **Tejfel M., Horváth Z. and Kozsik T.,** Temporal properties of Clean programs proven in Sparkle-T, *CEPF*, ed. Z.Horváth, LNCS **4164**, Springer Verlag, 2005, 168–190.

[9] **Tejfel M., Horváth Z. and Kozsik T.,** Extending the Sparkle Core language with object abstraction, *Acta Cybern.*, **17** (2006), 419-445.

[10] **Dowse M., Strong G. and Butterfield A.,** Proving make correct: I/O proofs in Haskell and Clean, eds. R.Pena, and T.Arts, *Implementation of Functional Languages, 14th International Workshop, IFL 2002, Madrid, Spain, September 16-18, 2002. Revised Selected Papers*, LNCS **2670**, Springer Verlag, 2003, 68–83.

[11] **Dowse M., Butterfield A. and van Eekelen M.C.J.D.,** Reasoning about deterministic concurrent functional I/O, *IFL,* eds. C.Grelck, F.Huch, G.Michaelson and P.W.Trinder, LNCS **3474**, Springer Verlag, 2004, 177–194.

[12] **Dowse M. and Butterfield A.,** Modelling deterministic concurrent I/O, *International Conference on Functional Programming (ICFP),* eds. J.H.Reppy and J.L.Lawall, ACM, 2006, 148–159.

[13] **Achten P., van Eekelen M. and Plasmeijer R.,** Towards a unified semantic model for interactive applications using arrows and generic editors, *Proceedings of the Seventh Symposium on Trends in Functional Programming, TFP 2006, Nottingham, UK, 19-21 April 2006,* ed. H.Nillsson, 279–292.

M. Tejfel, T. Kozsik and Z. Horváth
Dept. of Programming Languages and Compilers
Eötvös Loránd University
Pázmány Péter sétány 1/C,
H-1117 Budapest, Hungary
`{matej,kto,hz}@inf.elte.hu`

A. The application programming interface of Sio

```
definition module sio
import StdEnv
import StdMaybe

::UniType = Unit Int | Pair UniType UniType

class Uni a where
     fromUni:: UniType -> a
     toUni:: a -> UniType
instance Uni Int
instance Uni Char
instance Uni String
instance Uni Void
instance Uni Id
instance Uni (a, b) | Uni a & Uni b
instance Uni (a,b,c) | Uni a & Uni b & Uni c
instance Uni (a,b,c,d) | Uni a & Uni b & Uni c & Uni d
instance Uni [a] | Uni a
instance Uni (Maybe a) | Uni a

:: WindowAttribute st  =  WindowActivate    (IdFun st)
                       |  WindowClose       (IdFun st)
                       |  WindowDeactivate  (IdFun st)
                       |  WindowHMargin     Int Int
                       |  WindowId          Id
                       |  WindowIndex       Int
                       |  WindowInit        (IdFun st)
                       |  WindowInitActive  Id
                       |  WindowItemSpace   Int Int
                       |  WindowOuterSize   Size
                       |  WindowPos         ItemPos
                       |  WindowViewSize    Size
                       |  WindowVMargin     Int Int
                       |  WindowCancel      Id
                       |  WindowOk          Id

:: ControlAttribute st
        =       ControlActivate     (st -> st)
        |       ControlDeactivate   (st -> st)
        |       ControlFunction     (st -> st)
        |       ControlHide
        |       ControlId           Id
        |       ControlMinimumSize  Size
        |       ControlPos          ItemPos
        |       ControlResize       (Size -> Size -> Size -> Size)
        |       ControlSelectState  SelectState
        |       ControlTip          String
        |       ControlWidth        ControlWidth

:: ControlWidth        =  PixelWidth        Int
                       |  TextWidth         String
                       |  ContentWidth      String

:: Colour  =   RGB RGBColour
           |   Black          | White
           |   DarkGrey       | Grey      | LightGrey
           |   Red            | Green     | Blue
           |   Cyan           | Magenta   | Yellow

:: RGBColour = { r:: !Int, g:: !Int, b :: !Int}

:: ItemPos :== ( ItemLoc, ItemOffset)

:: ItemLoc  =  Fix
            |  LeftTop                  |  RightTop
            |  LeftBottom               |  RightBottom
            |  Left         |  Center   |  Right
            |  LeftOf Int               |  RightTo Int
            |  Above Int                |  Below Int
            |  LeftOfPrev               |  RightToPrev
            |  AbovePrev                |  BelowPrev

:: ItemOffset  =  NoOffset
               |  OffsetVector Vector2
               |  OffsetFun  Int ((Rectangle,Point2) -> Vector2)
instance zero ItemOffset

:: Controller ls a
(:+:)  :: [Controller ls a] [Controller ls a] -> [Controller ls a]
ListLS :: [[Controller ls a]] -> [Controller ls a]

EditControl  ::  String ControlWidth Int
                 [ControlAttribute (ls,PSt a)]
             ->  [Controller ls a] | Uni a & Uni ls
TextControl  ::  String
                 [ControlAttribute (ls,PSt a)]
             ->  [Controller ls a] | Uni a & Uni ls
ButtonControl::  String
```

```
                 [ControlAttribute (ls,PSt a)]
             ->  [Controller ls a] | Uni a & Uni ls
Receiver     ::  (RId m)
                 (b (ls,PSt a) -> (ls,PSt a))
                 [ControlAttribute (ls,PSt a)]
             ->  [Controller ls a] | Uni a & Uni ls
CustomControl::  Size look
                 [ControlAttribute (ls,PSt a)]
             ->  [Controller ls a] | Uni a & Uni ls

getWindow      :: !Id !(IOSt l) -> (!Maybe WState, !IOSt l)
setControlLook :: !Id !Bool (Bool,look) !(IOSt l) -> IOSt l
getControlText :: !Id !WState ->  (Bool,Maybe String)
setControlText :: !Id !String !(IOSt l) -> IOSt l

:: Id = Id Int
instance == Id

:: RId m

:: State
:: IOSt l
:: PSt  l  = { ls:: !l, io :: !IOSt l}
:: WState

:: Process_
:: Dialog_ ls a
:: World_

my_ :: World -> World_

class Ids env where
   openId   :: !env -> (!Id, !env)
   openIds  :: !Int !env -> (![Id], !env)
   openRId  :: !env -> (!RId m, !env)
   openRIds :: !Int !env -> (![RId m], !env)

instance Ids World_
instance Ids (IOSt l)
instance Ids (PSt  l)

accPIO :: !((IOSt a) -> (b,IOSt a)) !(PSt a) -> (!b, !PSt a)
appPIO :: !((IOSt a) -> (IOSt a)) !(PSt a) -> (PSt a)

:: Vector2      =   { vx :: !Int, vy :: !Int}
:: Size         =   { w  :: !Int, h  :: !Int}
:: Point2       =   { x  :: !Int, y  :: !Int}
:: Rectangle    =   { corner1 :: !Point2, corner2 :: !Point2}
:: ViewDomain   :== Rectangle
:: ViewFrame    :== Rectangle
:: UpdateArea   :== [ViewFrame]
:: UpdateState  =   { oldFrame :: !ViewFrame
                    , newFrame :: !ViewFrame
                    , updArea  :: !UpdateArea
                    }

:: SelectState  =   Able | Unable

hmm :: !Real -> Int

:: Void = Void
:: IdFun st :== st -> st

noLS  :: (.a->.b) !(.c,.a) -> (.c,.b)
noLS1 :: (.x->.a->.b) .x !(.c,.a) -> (.c,.b)

syncSend  :: !(RId b) b !(PSt a) -> (!SendReport, !PSt a) | Uni b
asyncSend :: !(RId b) b !(PSt a) -> (!SendReport, !PSt a) | Uni b

:: SendReport  =  SendOk
               |  SendUnknownReceiver
               |  SendUnableReceiver
               |  SendDeadlock
               |  OtherSendReport !String

:: Picture
setPenColour:: !Colour !Picture -> Picture
setPenSize:: !Int !Picture -> Picture
drawLine:: !Point2 !Point2 !Picture -> Picture
fill:: a !Picture -> Picture

openProcesses:: ![Process_] !(PSt a) -> PSt a | Uni a
startProcesses:: ![Process_] !World_ -> World_

openDialog   :: ls !(Dialog_ ls a) !(PSt a)  -> (!ErrorReport, !PSt a)
              | Uni a & Uni ls
openReceiver :: ls ![Controller ls a] !(PSt a) -> (!ErrorReport, !PSt a)
              | Uni a & Uni ls

:: ErrorReport  =  NoError
                |  ErrorViolateDI
                |  ErrorIdsInUse
                |  ErrorUnknownObject
                |  ErrorNotifierOpen
                |  OtherError !String

:: DocumentInterface  =  NDI  |  SDI  |  MDI

Process :: DocumentInterface a ((PSt a) -> PSt a) [b] -> Process_ | Uni a
Dialog  :: String [Controller ls a] [b] -> Dialog_ ls a

startIO ::  DocumentInterface a ((PSt a) -> PSt a) [c] World_ ->  World_
         |  Uni a
```

B. Manipulation of data stored in Sio

message :: !Event → UniType – The message in an event.

target :: !Event → Id – The identifier of the target controller of an event.

sender :: !Event → Id – The identifier of the sender process of an event.

async :: !Event → Bool – Whether the event is asynchronous.

sync :: !Event → Bool – Whether the event is synchronous.

controlId :: !Id ![Process_Inner] → Bool – Whether the given identifier identifies a control.

procLS :: !Id !(PSt a) → UniType – The local state of the process containing the controller identified by the given identifier.

diaLS :: !Id !(PSt a) → UniType – The local state of the dialog containing the controller identified by the given identifier.

getTxt :: !Id !State → (Bool, Maybe String) – The label of the controller identified by the given identifier.

setTxt :: !Id !String !State → State – Set the label of a controller identified by the given identifier.

control :: !Id !(PSt a) → Controller UniType UniType – The controller identified by the given identifier.

isButton :: !(Controller ls a) → Bool – Whether the controller is a ButtonController.

isEditor :: !(Controller ls a) → Bool – Whether the controller is an EditorController.

isReceiver :: !(Controller ls a) → Bool – Whether the controller is a ReceiverController.

$\mathsf{callback_b}$ :: !(Controller Is a) → ((Is, PSt a) → (Is, PSt a)) – The callback function of a ButtonController.

$\mathsf{callback_r}$ :: !(Controller Is a) → (UniType (Is, PSt a) → (Is, PSt a)) – The callback function of a ReceiverController.

currProc :: !(PSt a) → Id – The identifier of the currently running process.

uevsOf :: !(PSt a) → [Event] – The user event queue of `pst.io` for a given `pst`.

sevsOf :: !(PSt a) → [Event] – The system event queue of `pst.io` for a given `pst`.

blockedpr :: !Id !(PSt a) → Bool – Whether the process identified by the given identifier is blocked at a synchronous communication.

blocked :: !Event !State → Bool – Whether the target process of the given event is blocked at a synchronous communication.

allblocked :: !State → Bool – Whether blocked holds for all events in the user and system event queues.

syncBlocked :: !State → Bool – The "blocked flag" of the current state.

procSt :: !Id !(PSt a) → (Id, Process_State) – The identifier and the state of the process with the given identifier (the first argument is returned).

syncBlock :: !(Id, Process_State) !(PSt a) → (PSt a) – Set the state of the process to blocked (with an empty blocked event queue), and set the "blocked flag" of the current state to True.

addBlockSync :: !Id !UniType !(Id, Process_State) !(PSt a) → (PSt a)
Add a synchronous message event to the blocked event queue of the process.

addBlockAsync :: !Id !UniType !(Id, Process_State) !(PSt a) → (PSt a)
Add an asynchronous message event to the blocked event queue of the process.

addSevSync :: !Id !UniType !(Id, Process_State) !(PSt a) → (PSt a)
Add a synchronous message event to the system event queue.

addSevAsync :: !Id !UniType !(Id, Process_State) !(PSt a) → (PSt a)
Add an asynchronous message event to the system event queue.

wakeUp :: ![Id] (PSt a) → (PSt a) – Wake up the blocked processes identified by the given list of identifiers, and create system events from the events in the blocked event queues of the processes waken up.

updState :: !Event !UniType !(PSt UniType) → State – Update the local state of the process and that of the dialog containing the target controller of the given event, and set the "blocked flag" of the state to False.

setCurrPr :: !Id !State → State – Set the "current process" argument of the state to the process containing the controller identified by the given identifier.

pos :: !a ![a] → Int | Eq a – Return the (zero-based) index of an item in a list, or the length of the list if the item is not in the list.

remove :: !a ![a] → [a] | Eq a – Remove (the first occurrence of) an item from a list. Similar to the standard library function `removeMember`.

C. Axioms

$$\frac{\mathsf{uevsOf\ pst} = [],\quad \mathsf{sevsOf\ pst} = []}{\mathsf{scheduler\ pst} = \mathsf{pst}} \qquad \frac{\mathsf{uevsOf\ pst} \neq [] \ \vee\ \mathsf{sevsOf\ pst} \neq []}{\mathsf{scheduler\ pst} = \mathsf{scheduler\ (step\ pst)}}$$

$$\frac{\mathsf{allblocked\ pst.io}}{\mathsf{step\ pst} = \mathsf{pst}} \qquad \frac{\neg\mathsf{allblocked\ pst.io}}{\mathsf{step\ pst} = \mathsf{processEv\ (selectEv\ pst)\ pst}}$$

$$\frac{\neg\mathsf{allbocked\ pst.io},\quad \mathsf{selectEv\ pst} = \mathsf{e}}{\mathsf{isMember\ e\ (uevsOf\ pst)} \ \vee\ \mathsf{isMember\ e\ (sevsOf\ pst)}}$$

$$\frac{\neg\mathsf{allbocked\ pst.io},\quad \mathsf{selectEv\ pst} = \mathsf{e}}{\neg\mathsf{blocked\ e\ pst.io}}$$

$$\frac{\mathsf{pos\ e_1\ (uevsOf\ pst)} < \mathsf{pos\ e_2\ (uevsOf\ pst)},\quad \neg\mathsf{blocked\ e_1\ pst.io}}{\mathsf{selectEv\ pst} \neq \mathsf{e_2}}$$

$$\frac{\begin{array}{c}\mathsf{pos\ e_1\ (sevsOf\ pst)} < \mathsf{pos\ e_2\ (sevsOf\ pst)},\quad \neg\mathsf{blocked\ e_1\ pst.io}\\ \mathsf{target\ e_1} = \mathsf{target\ e_2},\quad \mathsf{sender\ e_1} = \mathsf{sender\ e_2}\end{array}}{\mathsf{selectEv\ pst} \neq \mathsf{e_2}}$$

$$\frac{\mathsf{pst.io} = \mathsf{State\ ps\ uevs\ sevs\ is\ cp\ fl},\quad \neg\mathsf{controlId\ (target\ e)\ ps}}{\mathsf{processEv\ e\ pst} = \{\mathsf{pst}\ \&\ \mathsf{io} = \mathsf{State\ ps\ (remove\ e\ uevs)\ (remove\ e\ sevs)\ is\ cp\ fl}\}}$$

$$\frac{\begin{array}{c}\mathsf{async\ e},\quad \mathsf{t} = \mathsf{target\ e},\quad \mathsf{ls_p} = \mathsf{procLS\ t\ pst},\quad \mathsf{ls_d} = \mathsf{diaLS\ t\ pst}\\ \mathsf{ctrl} = \mathsf{control\ t\ pst},\quad \mathsf{isButton\ ctrl},\quad \mathsf{fun} = \mathsf{callback_b\ ctrl}\\ (\mathsf{ls'_d}, \mathsf{pst'}) = \mathsf{fun}\ (\mathsf{ls_d}, \{\mathsf{ls} = \mathsf{ls_p}, \mathsf{io} = \mathsf{setCurrPr\ t\ pst.io}\})\\ \mathsf{s} = \mathsf{updState\ e\ ls'_d\ pst'}\end{array}}{\mathsf{processEv\ e\ pst} = \{\mathsf{pst}\ \&\ \mathsf{io} = \mathsf{s}\}}$$

$$\frac{\begin{array}{c}\mathsf{async\ e},\quad \mathsf{t} = \mathsf{target\ e},\quad \mathsf{ls_p} = \mathsf{procLS\ t\ pst},\quad \mathsf{ls_d} = \mathsf{diaLS\ t\ pst}\\ \mathsf{ctrl} = \mathsf{control\ t\ pst},\quad \mathsf{isReceiver\ ctrl},\quad \mathsf{fun} = \mathsf{callback_r\ ctrl}\\ (\mathsf{ls'_d}, \mathsf{pst'}) = \mathsf{fun\ (message\ e)}\ (\mathsf{ls_d}, \{\mathsf{ls} = \mathsf{ls_p}, \mathsf{io} = \mathsf{setCurrPr\ t\ pst.io}\})\\ \mathsf{s} = \mathsf{updState\ e\ ls'_d\ pst'}\end{array}}{\mathsf{processEv\ e\ pst} = \{\mathsf{pst}\ \&\ \mathsf{io} = \mathsf{s}\}}$$

$$\frac{\begin{array}{c}\mathsf{async\ e},\quad \mathsf{t} = \mathsf{target\ e},\\ \mathsf{ctrl} = \mathsf{control\ t\ pst},\quad \mathsf{isEditor\ ctrl}\\ \mathsf{s} = \mathsf{setControlText\ t\ (fromUni\ (message\ e))\ pst.io}\end{array}}{\mathsf{processEv\ e\ pst} = \{\mathsf{pst}\ \&\ \mathsf{io} = \mathsf{s}\}}$$

$$\frac{\begin{array}{c}\mathsf{sync\ e},\quad \mathsf{t} = \mathsf{target\ e},\quad \mathsf{ls_p} = \mathsf{procLS\ t\ pst},\quad \mathsf{ls_d} = \mathsf{diaLS\ t\ pst}\\ \mathsf{ctrl} = \mathsf{control\ t\ pst},\quad \mathsf{isReceiver\ ctrl},\quad \mathsf{fun} = \mathsf{callback_r\ ctrl}\\ (\mathsf{ls'_d}, \mathsf{pst'}) = \mathsf{fun\ (message\ e)}\ (\mathsf{ls_d}, \{\mathsf{ls} = \mathsf{ls_p}, \mathsf{io} = \mathsf{setCurrPr\ t\ pst.io}\})\\ \mathsf{s} = \mathsf{updState\ e\ ls'_d\ pst'},\quad \mathsf{blocked\ e\ s}\end{array}}{\mathsf{processEv\ e\ pst} = \{\mathsf{pst}\ \&\ \mathsf{io} = \mathsf{s}\}}$$

$$\frac{\begin{array}{c}\mathsf{sync\ e},\quad \mathsf{t} = \mathsf{target\ e},\quad \mathsf{ls_p} = \mathsf{procLS\ t\ pst},\quad \mathsf{ls_d} = \mathsf{diaLS\ t\ pst}\\ \mathsf{ctrl} = \mathsf{control\ t\ pst},\quad \mathsf{isReceiver\ ctrl},\quad \mathsf{fun} = \mathsf{callback_r\ ctrl}\\ (\mathsf{ls'_d}, \mathsf{pst'}) = \mathsf{fun\ (message\ e)}\ (\mathsf{ls_d}, \{\mathsf{ls} = \mathsf{ls_p}, \mathsf{io} = \mathsf{setCurrPr\ t\ pst.io}\})\\ \mathsf{s} = \mathsf{updState\ e\ ls'_d\ pst'},\quad \neg\mathsf{blocked\ e\ s}\end{array}}{\mathsf{processEv\ e\ pst} = \ \mathsf{wakeUp\ (blockedlist\ e)}\ \{\mathsf{pst}\ \&\ \mathsf{io} = \mathsf{s}\}}$$

$$\frac{\mathsf{sync\ e},\quad \mathsf{t} = \mathsf{target\ e},\quad \mathsf{ctrl} = \mathsf{control\ t\ pst},\quad \mathsf{isButton\ ctrl}}{\mathsf{processEv\ e\ pst} = \bot}$$

$$\frac{\mathsf{sync\ e},\quad \mathsf{t} = \mathsf{target\ e},\quad \mathsf{ctrl} = \mathsf{control\ t\ pst},\quad \mathsf{isEditor\ ctrl}}{\mathsf{processEv\ e\ pst} = \bot}$$

$$\frac{\begin{array}{c}\mathsf{proc} = \mathsf{currProc\ pst},\quad \mathsf{ps} = \mathsf{procSt\ proc\ pst},\quad \mathsf{blockedpr\ proc\ pst}\\ \mathsf{pst'} = \mathsf{addBlockSync\ targetId\ msg\ ps\ pst}\end{array}}{\mathsf{snd(syncSend\ targetId\ msg\ pst)} = \mathsf{pst'}}$$

$$\frac{\begin{array}{c}\mathsf{proc} = \mathsf{currProc\ pst},\quad \mathsf{ps} = \mathsf{procSt\ proc\ pst},\quad \mathsf{blockedpr\ proc\ pst}\\ \mathsf{pst'} = \mathsf{addBlockAsync\ targetId\ msg\ ps\ pst}\end{array}}{\mathsf{snd(asyncSend\ targetId\ msg\ pst)} = \mathsf{pst'}}$$

$$\frac{\begin{array}{c}\mathsf{proc} = \mathsf{currProc\ pst},\quad \mathsf{parentPR\ targetId\ pst} \neq \mathsf{proc}\\ \mathsf{ps} = \mathsf{procSt\ proc\ pst},\quad \neg\mathsf{blockedpr\ proc\ pst}\\ \mathsf{pst'} = \mathsf{addSevSync\ targetId\ msg\ ps\ pst},\quad \mathsf{pst''} = \mathsf{syncBlock\ ps\ pst'}\end{array}}{\mathsf{snd(syncSend\ targetId\ msg\ pst)} = \mathsf{pst''}}$$

$$\frac{\begin{array}{c}\mathsf{proc} = \mathsf{currProc\ pst}, \quad \mathsf{ps} = \mathsf{procSt\ proc\ pst}, \quad \neg\mathsf{blockedpr\ proc\ pst}\\ \mathsf{pst}' = \mathsf{addSevAsync\ targetId\ msg\ ps\ pst},\end{array}}{\mathsf{snd}(\mathsf{asyncSend\ targetId\ msg\ pst}) = \mathsf{pst}'}$$

$$\frac{\begin{array}{c}\mathsf{proc} = \mathsf{currProc\ pst}, \quad \mathsf{parentPR\ targetId\ pst} = \mathsf{proc}, \quad \mathsf{ps} = \mathsf{procSt\ proc\ pst},\\ \neg\mathsf{blockedpr\ proc\ pst}, \quad \mathsf{ls_p} = \mathsf{procLS\ targetId\ pst}, \quad \mathsf{ls_d} = \mathsf{diaLS\ targetId\ pst}\\ \mathsf{ctrl} = \mathsf{control\ targetId\ pst}, \quad \mathsf{isReceiver\ ctrl}, \quad \mathsf{fun} = \mathsf{callback_r\ ctrl}\\ (\mathsf{ls}'_\mathsf{d}, \mathsf{pst}') = \mathsf{fun\ msg}\ (\mathsf{ls_d}, \{\mathsf{ls} = \mathsf{ls_p}, \mathsf{io} = \mathsf{setCurrPr\ targetId\ pst.io}\})\\ \mathsf{s} = \mathsf{updState\ proc}\ \mathsf{ls}'_\mathsf{d}\ \mathsf{pst}'\end{array}}{\mathsf{snd}(\mathsf{syncSend\ targetId\ msg\ pst}) = \{\mathsf{pst}\ \&\ \mathsf{io} = \mathsf{s}\}}$$

$$\frac{\mathsf{syncBlocked\ iost}}{\mathsf{setControlText\ targetId\ str\ iost} = \perp}$$

$$\frac{\neg\mathsf{syncBlocked\ iost}}{\mathsf{setControlText\ targetId\ str\ iost} = \mathsf{setTxt\ targetId\ str\ iost}}$$

$$\frac{\mathsf{syncBlocked\ iost}}{\mathsf{getWindow\ targetId\ iost} = \perp}$$

$$\frac{\neg\mathsf{syncBlocked\ iost}, \quad (\mathsf{Just\ wst}, \mathsf{iost}) = \mathsf{getWindow\ targetId\ iost}}{\mathsf{getControlText\ targetId\ wst} = \mathsf{getTxt\ targetId\ iost}}$$