The GUI-fest challenges

In this report we present the solutions to two selected challenges that have been proposed in the 1995 Glasgow GUI-fest. The solutions are defined in the pure, lazy, functional programming language Clean. We have used the currently available version of the Event I/O library version 1.0. The two selected challenges are an object oriented counter viewing program, the 'counter challenge' and a board game, the 'Explode challenge'. One interesting aspect in the way we have approached the counter challenge is to first define a very general framework for browser/editors, inspired by a challenge proposed by Emden Gansner. The counter challenge is then defined as an instance of this general browser/editor framework. The Explode challenge can be worked out as a straightforward Event I/O program. It demonstrates the elegance and expressiveness of functional programming.

1 Introduction

The 1995 Glasgow GUI-fest was held in the week of 24-28 July, during the Glasgow Research Festival. The main topic of interest of the GUI-fest was to investigate the strengths and weaknesses of programming Graphical User Interfaces in functional languages. A number of challenges have been proposed, and eventually it was decided to present solutions to two of these challenges. The first challenge, the 'counter challenge' was created to see how the different languages and I/O models handle encapsulated state in perhaps an object-oriented style. The second challenge is a modest board game, 'Explode', and it was proposed by Rob Noble and Colin Runciman (introduced earlier in Noble and Runciman (1995)).

In this report we present the solutions to these challenges using the lazy, pure, functional programming language Clean (Brus *et al.*, 1987; Nöcker *et al.*, 1991; Plasmeijer and van Eekelen, 1993). The I/O library that has been used in these examples is the current, experimental, Event I/O 1.0 version. This system is a successor of the Event I/O 0.8 distribution version (Achten *et al.*, 1993; Achten and Plasmeijer, 1995), extended with the concept of interactive processes and message passing (Achten and Plasmijer, 1994). In Section 2 we describe the way we solved the counter challenge. Section 3 discusses the approach to the Explode game. Section 4 discusses and concludes. Appendix A contains the full code of the counter challenge, and Appendix B contains the full code of the Explode game. In this report we assume the reader is familiar with functional programming and the Clean Event I/O system.

2 The counter challenge

In this section we show how we have approached the counter challenge. First we give the specification of the challenge. This rather informal specification describes what the program should do seen from a user's point of view. When starting the program, it should present to the user a window as shown in Figure 1. We will call this window a counter window.

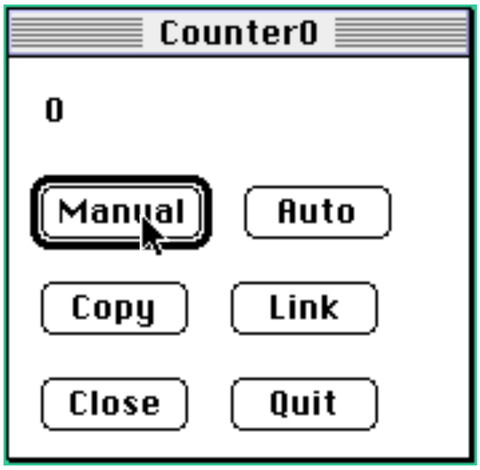

Figure 1 The initial view of the counter program.

The window displays an integer value. There are six buttons, labelled *Manual*, *Auto*, *Copy*, *Link*, *Close*, and *Quit*. Their behaviour, when selected, is as follows:

Manual simply increments the currently displayed value by one.

Auto sets this counter to auto-increment mode in which the count increases periodically until Manual is pressed. The auto-increment mode is undone by pressing Manual.

Link creates a new counter window that gives a new view of the value displayed by the window in which Link has been selected. In general, there can be an arbitrary number of windows, each displaying the same current value. Such a group of windows is called a *link*.

Copy, like Link, also creates a new counter window, except that the new counter window has a completely new initial counter value which is also not shared with any other counter window. So Copy creates a new set of link.

Close closes all windows that are linked to the counter window in which Close has been selected. If it happens to be the last link, then this also terminates the application.

Quit simply terminates the application, and thereby closes all links.

The auto-increment mode applies to all counter windows of the same link. The auto-increment mode is turned off by selecting the Manual button of any of the counter windows of that link. To inform the user which counter windows belong to the same counter value we number the links and display the link number in the name of the counter window.

More generally, the program presents a dynamic number of views of a dynamic number of states. The views are all counter windows, and the states are all counter values and perhaps an auto-increment mode. This lead us to the idea to see if this challenge could be solved as a special kind of browser/editor, a challenge proposed by Emden Gansner. In this challenge, the task was to: “build a simple graph editor for creating, viewing and editing directed acyclic graphs. The program should support at least two different types of concrete representations of a graph. Representations can be either textual or graphical. Changes to the graph should be propagated to all of the views." The directed acyclic graph is called the "model" data, and the representations are called views.

It occurred to us that the counter challenge could be formulated nicely as a special case of the browser/editor. So we first started out to build a general browser/editor framework (Section 2.1), and then apply this framework to create the counter challenge (Section 2.2).

2.1 A general browser/editor framework

The general structure of a browser/editor is some model data, which is a data structure, and a number of views that can write and read the model data. Each view can have local data. Whenever a view changes the model data, the other views should be informed about this change in order to propagate the changes throughout all views. Figure 2 puts this general structure in a scheme.

Because the framework should be general, the model data should allow any type of data structure. So the framework should be polymorphic in terms of the model data. The views are defined as individual interactive processes. The model data is globally accessible to all views. It will be managed by one special interactive process, the control process. All access of the view processes to the model data will be done via the control process. It will take care that manipulation propagation will be done correctly to all view processes. Finally, the framework should be such that view processes, which will be defined by a programmer, do not need to know how the access and manipulation propagation is actually handled. To the view processes and the programmer this information is encapsulated. Figure 3 gives the scheme of our approach.

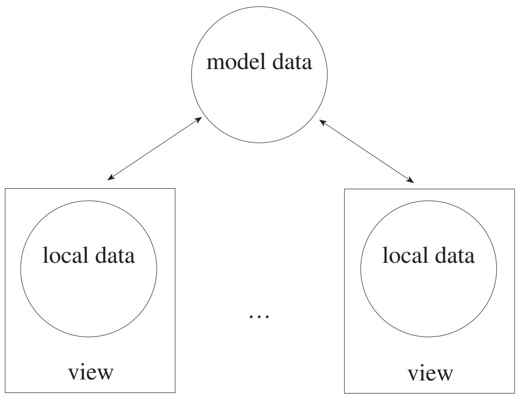

Figure 2 The general structure of the browser/editor.

Below we consider the following main components in sequence: the view processes (Section 2.1.1), the control process (Section 2.1.2), and information encapsulation (Section 2.1.3).

2.1.1 View processes

A view process presents a view on the model data, which is some arbitrary type m. The behaviour of a view process is defined by three functions:

a) The manipulation propagation: what should happen when the model data has been changed.

b) The initial state of the view process: it depends on the current value of the model data.

c) The initial actions of the view process: besides the common initial actions, this function is parameterized with an abstract value without which the view process has no proper access to the model data.

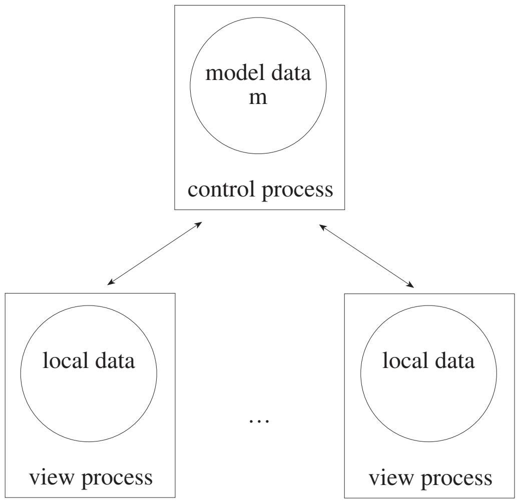

Figure 3 The structure of a general browser/editor using interactive processes.

The definition of these functions is collected in the record type `ViewDef` (Figure 4).

```
::  ViewDef … m
    =  { vRespond   :: RespondF      m (PState …),
         vInitState:: InitStateF … m,
         vInitIO    :: InitIOF     … m
       }
::  RespondF      m ps :== ((m,m), ps) -> ps
::  InitStateF … m     :== m -> …
::  InitIOF    … m     :== (BrowseInfo m) -> (InitIO …)
```

Figure 4 The definition of a view process.

Given the definition of a view, it can be transformed into an interactive process. A view process is an interactive process that has one special receiver device which accepts messages of type (ViewM m), which consists of the single alternative (**RespondTo** (m,m)). Such a message always consists of a pair of the old model data and the new model data. Given such a pair, the view process can respond appropriately. This response is the RespondF function of the view's definition. To accomplish this the receiver is parameterized with the ViewDef structure. Figure 5 gives the definition of the receiver.

```
::  ViewM m = RespondTo (m,m)

viewR::(ViewDef … m) (ViewM m) (PState …) -> (PState …)
viewR viewDef  (RespondTo modelChange) ps
    =   viewDef.vRespond (modelChange, ps)
```

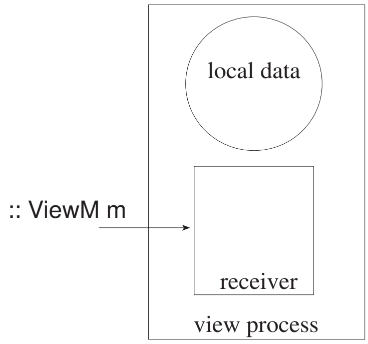

Figure 5 The structure of view processes. Alternative constructors of algebraic data types are printed in **boldface**.

We defer the discussion on the creation of view processes until Section 2.1.3.

2.1.2 Control process

The control process manages all access by the view processes, and all manipulation propagation that is required. It is a special interactive process, defined within the framework, and is invisible to the programmer. The control process is a background process, and consists of a receiver only. It is the process that sends (ViewM m) messages to the view processes, in reply to (ControlM m) messages that are sent by the view processes. View processes, identified by a value id of type (ViewId m) can request the control processes to:

a) Change the current model data with a function f (**Action** id f).
b) Create a new view process (**NewView** def).
c) Participate in manipulation propagation (**NewViewId** id).
d) Refrain from manipulation propagation (**CloseView** id).
e) Terminate the control process (**Quit**).

Figure 6 gives the scheme and type definitions.

```
::  ControlM m
    =  Action     (ViewId    m)  (m->m)
    |  NewView    (ViewDef … m)
    |  NewViewId  (ViewId    m)
    |  CloseView  (ViewId    m)
    |  Quit
```

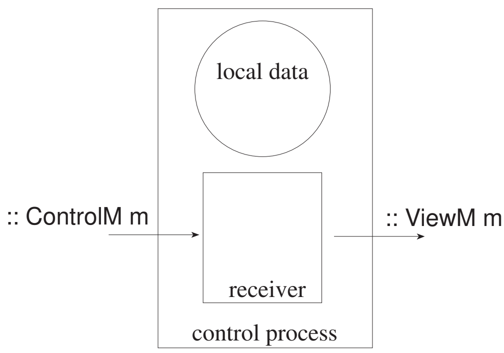

Figure 6 The structure of the control process.

The local state of the control process, the record type (LocalC m), contains of course the model data, but also some further information: the identifications of the view processes that have currently subscribed to be informed about changes of the model data, and the identification of the control process itself.

```
::  LocalC m
    =   { views  :: [ViewId m],
          model  :: m,
          myself :: Maybe (ControlId m)
        }
::  *CState m
:== PState (LocalC m) Int
```

Figure 7 The local and process state of the control process.

Because the control process consists of one receiver only, its behaviour is defined completely by the definition of the receiver function `controlR` of the control process which has type:

```
controlR:: (ControlM m) (CState m) -> (CState m)
```

For every (**Action** sender act) message, compute the new value of the model data, and send to every other view process the pair of the old and new model data values:

```
controlR (Action sender act) ps=:{pLocal=local}
    = {ps & pLocal  ={local & model=newmodel},
            pIOState=broadcast others mess ps.pIOState}
where
    newmodel = act local.model
    others   = filter (not o eqRId sender) local.views
    mess     = RespondTo (local.model,newmodel)
```

For every (**NewView** viewDef) message, create a new view process.

```
controlR (NewView viewDef) ps = openView viewDef ps
```

For every (**NewViewId** id) message, add the view process to the administration of view processes that have subscribed for manipulation propagation.

```
controlR (NewViewId id) ps=:{pLocal=local}
    = {ps & pLocal={local & views=[id:local.views]}}
```

For every (**CloseView** id) message, remove the view process from the administration of view processes that have subscribed for manipulation propagation (so this is basically the reverse operation of the previous alternative).

```
controlR (CloseView id) ps=:{pLocal=local}
    = {ps & pLocal={local & views=filter (not o eqRId id)
                                  local.views}}
```

For every **Quit** message, quit the control process.

```
controlR Quit ps = seqPIO [QuitIO] ps
```

2.1.3 Information encapsulation

In the previous two sections we have defined the view and control processes, and the communication protocol. To get things cooperating properly, the programmer must know in what order messages should be sent. For instance, opening a new view process (by sending **NewView**) should be followed by registering the identification of the new view process (by sending **NewViewId**). These kind of requirements are likely to go wrong. In this section we discuss how we can circumvent these problems and obtain a safe system. We show how we can encapsulate the communication protocol and the view process creation.

First we introduce an abstract data type, `BrowseInfo m`, that will be the sole interface to the programmer to access the model data. The creation of the view processes will guarantee that the bControl field of a BrowseInfo value always is the identification of the receiver of the control process (which is of type RId (ControlM m) because it receives messages of type (ControlM m)). For each view process, the bView field of a BrowseInfo value is always the identification of its receiver that receives the messages of type (ViewM m), and thus is of type RId (ViewM m). The signature of BrowseInfo, containing the view management operations, is given in Figure 8.

```
::  BrowseInfo m
    =   { bView    :: ViewId    m,
          bControl :: ControlId m
        }
::  ViewId    m :== RId (ViewM    m)
::  ControlId m :== RId (ControlM m)

changeModelData::(BrowseInfo m)  (m->m)
                                 (PState .l .p) -> PState .l .p
closeModelData ::(BrowseInfo m)
                                 (PState .l .p) -> PState .l .p
openViewer     ::(BrowseInfo m)  (ViewDef` Void Void m)
                                 (PState .l .p) -> PState .l .p
closeView      ::(BrowseInfo m)  (PState .l .p) -> PState .l .p
```

Figure 8 The signature of the abstract data type BrowseInfo.

As we saw in Section 2.1.1., view processes are defined by values of type ViewDef m. In general, a browser/editor can consist of several initial views. Each view can have its private data, which types can be different. For this reason, the existentially quantified type ViewDef` is introduced which hides the type information of the private data of the initial views (see Figure 9). Furthermore, an initial value for the model data has to be supplied. The only function available to create a browser/editor is the function `openBrowser`. When applied to a browser/editor definition, `openBrowser` creates the control process, and then applies for each view definition the function openView. Recall that openView was also applied by controlR when a view process requested a new view to be created (Section 2.1.2.).

```
::  Browser m
    =   { bViews     :: [ViewDef` Void Void m],
          bModelData :: m
        }
::  ViewDef` E.l E.p m
    =   Hidden (ViewDef l p m)


openBrowser :: (Browser m) *World -> *World
```

```
openBrowser browser world
=   OpenIO [initControlGUI browser]
           (initLocal browser.bModelData,0)
           world

initControlGUI :: (Browser m) (CState m) -> (CState m)
initControlGUI browser ps=:{pLocal=local}
=   seq (map openView browser.bViews) ps1
where
    (me,io1) = OpenReceiver receiver ps.pIOState
    receiver = Receiver [ReceiverFunction controlR]
    ps1      = {ps & pLocal  = {local & myself = Just me},
                     pIOState= io1}
```

Figure 9 The creation of a browser/editor, defined by an initial set of view process definitions.

When a new view is created, the control process evaluates the function openView. Given a view definition, openView spawns a new interactive process. This process, before evaluating the initial actions of the view process, first creates the receiver that will communicate with the control process, sends the identification of the receiver to the control process, and generates the appropriate BrowseInfo value for this view process.

```
openView::(ViewDef` Void Void m) (CState m) -> (CState m)
openView (Hidden viewDef) ps=:{pLocal=local}
=   … NewIO [initView me viewDef]
            (viewDef.vInitState local.model) …
where
    Just me = local.myself

    initView :: (ControlId  m)   (ViewDef .l .p m)
                (PState .l .p) -> (PState .l .p)
    initView controlId viewDef ps
    =   seq initIO {ps & pIOState=io2}
    where
        (id,io1) = OpenReceiver rDef ps.pIOState
        rDef     = Receiver …(viewR viewDef)…
        io2      = ASyncSend controlId (NewViewId id) io1
        initIO   = viewDef.vInitIO { bView   =id,
                                     bControl=controlId }
```

Figure 10 The creation of a new view process.

Finally, we can consider the implementation of the abstract operations of BrowseInfo. These are give in Figure 11. They should be self-explanatory by now.

```
changeModelData::(BrowseInfo m) (m->m)
                                (PState .l .p) -> PState .l .p
changeModelData bInfo f ps
= seqPIO [ASyncSend bInfo.bControl (Action bInfo.bView f)] ps

closeModelData ::(BrowseInfo m) (PState .l .p) -> PState .l .p
closeModelData bInfo ps
= seqPIO [ASyncSend bInfo.bControl Quit] ps

openViewer ::(BrowseInfo m) (ViewDef` Void Void m)
                                (PState .l .p) -> PState .l .p
openViewer bInfo viewDef ps
= seqPIO [ASyncSend bInfo.bControl (NewView viewDef)] ps
```

```
closeView ::(BrowseInfo m)       (PState .l .p) -> PState .l .p
closeView bInfo ps
= seqPIO [ASyncSend bInfo.bControl (CloseView bInfo.bView)] ps
```

Figure 11 The implementation of the abstract operations on BrowseInfo.

2.2 The challenge as a special instance

In this section we show how the counter challenge can be defined as a special case of the browser/editor framework. To do this, we need to settle the model data and a definition of views.

Figure 12 gives the model data type, named Model, which consists of a list of *links*. A link is identified by a number (the synonym type LinkId), and it contains the current count value, and a flag stating whether the link is running in auto-increment mode. The private state of each view process, of type Local, consists of the identification of the link to which it presents a view, the most recent 'up-to-date' value of the model data, and an optional abstract BrowseInfo value.

```
::  Model :== [Link]
::  Link  =   { link  :: LinkId,
                count :: Int,
                auto  :: Bool
              }
::  Local =   { myLink:: LinkId,
                links :: [Link],
                binfo :: Maybe (BrowseInfo Model)
              }
```

Figure 12 The model data type.

All view processes are equal except for the link to which they present a view. So the definition of a view process can be suitably defined as a function parameterized with the link identification. As we have discussed in Section 2.1.1., view processes are defined as ViewDef values, by an initial state creation, initial action, and response function.

2.2.1 The initial state

The function `initstate`, which is parameterized with a link identification, determines the initial state of a view process. When applied to the current value of the model data, `initstate` determines whether there are already views open to this link. In that case the model data does not need to be changed. If not, then a new entry to the links field should be added, stating that the initial count value is zero, and that the link is initially running in manual mode. The binfo field is **Nothing**, because the proper BrowseInfo structure has not been created yet.

```
initstate :: LinkId Model -> (Local,Nil)
initstate l model
=   ({ myLink = l,
       links  = if i_exist model [{link=l,count=0,auto=False}:
                                   model],
       binfo  = Nothing},
    Nil)
```

```
where
    (i_exist,_) = selectLink l model
```

2.2.2 The initial actions

The initial actions of a view process are given by the function `initio` (see Appendix A.2). Because the initial actions of all view processes are equal, `initio` is not parameterized with the link identification. The proper BrowseInfo structure is put in the local state of the view process. Each view process has one dialogue window and one timer (which is initially disabled). Their definitions are straightforward. Below we will first discuss the timer function *Incr*, and then the control functions that are associated with the buttons *Manual*, *Auto*, *Copy*, *Link*, *Close*, and *Quit* respectively. (The library function *seq* is defined as *seq* $[f_1 \ldots f_n]\ x = f_n \circ \ldots \circ f_1\ x$. The function *seqPIO* applies seq to the list of IOState transition functions and updates the pIOState field of its process state argument correspondingly. The function *setLocalLinks* replaces the links field of the pLocal field of the process state argument with its first argument.)

The timer, when enabled, should increment the counter value by one. So it updates its local state, takes care of the proper visual feedback, and applies the abstract BrowseInfo operation changeModelData to change the current model data.

```
Incr :: (State .p) -> State .p
Incr ps=:{pLocal={myLink,links,binfo=Just b}}
=   seq [setLocalLinks (newlink links),
         seqPIO [SetWindow windowId
                   [SetTextControl textId (toString newcount)]],
         changeModelData b newlink] ps
where
    (_,me)     = selectLink myLink links
    newcount   = 1+me.count
    newlink ls= map (\l->if (myLink==l.link) {l&count=newcount}
                                             l) ls
```

The manual button should increment the count value by one. So its meaning is very similar to that of the timer. However, it should turn to manual mode if the link is currently running in auto-increment mode. So, to do this, Manual first sets the auto-increment flag to False, and enables the Auto button and disables the timer. Note that these operations have no effect in case the button and the timer are already enabled and disabled respectively. Finally, Manual applies Incr.

```
Manual :: (State .p) -> State .p
Manual ps=:{pLocal={myLink,links,binfo=Just b}}
=   seq [setLocalLinks     (setAuto myLink False links),
         enableAuto,
         changeModelData b (setAuto myLink False),
         Incr] ps

enableAuto ps
=   seqPIO [SetWindow    windowId [EnableControls [autoId]],
            DisableTimer timerId] ps
```

The auto button should turn the link to auto-increment mode. Therefore Auto sets the auto-increment flag to True and enables its timer. Furthermore, the auto button itself is disabled. Note that the auto-increment mode is undone by pressing Manual.

```
Auto :: (State .p) -> State .p
Auto ps=:{pLocal={myLink,links,binfo=Just b}}
```

```
=   seq [setLocalLinks     (setAuto myLink True links),
         seqPIO            [EnableTimer timerId],
         disableAuto,
         changeModelData b (setAuto myLink True)] ps

disableAuto ps
=   seqPIO [SetWindow windowId [DisableControls [autoId]]] ps
```

Copy creates a new set of linked counter windows. It generates a new link identification (newlinknr), and adds a new link element to the model data. Then it applies the abstract BrowseInfo operation openViewer to create the new view process.

```
Copy :: (State .p) -> State .p
Copy ps=:{pLocal={links,binfo=Just b}}
=   seq [changeModelData b (newviews newlinknr),
         openViewer      b (view     newlinknr)] ps
where
    newlinknr = 1+(foldr max 0 (map (\x->x.link) links))
    newviews l model = [{link=l,count=0,auto=False}:model]
```

Link creates a new counter window that gives a new view of the value displayed by the window in which Link has been selected. So it is sufficient to apply the abstract BrowseInfo operation openViewer, parameterized with its link identification, to create the new view process.

```
Link :: (State .p) -> State .p
Link ps=:{pLocal={myLink,binfo=Just b}}
=   openViewer b (view myLink) ps
```

Close closes all windows that are linked to the counter window in which Close has been selected. This is done by removing the entry identified by its linkid from the model data. As we will see in Section 2.2.3, the response function will then close all further linked view processes. If the link happens to be the last link, then Close also terminates the application (the first alternative of Close).

```
Close :: (State .p) -> State .p
Close ps=:{pLocal={links=[_]}} = Quit ps
Close ps=:{pLocal={myLink,binfo=Just b}}
=   seq [changeModelData b (removeLink myLink),
         closeView       b,
         seqPIO [QuitIO]] ps
```

Quit terminates the application. This is done by setting the model data to the empty list which will cause the response functions of the view processes to terminate their processes. It furthermore also terminates the control process by applying the abstract BrowseInfo operation `closeModelData`.

```
Quit :: (State .p) -> State .p
Quit ps=:{pLocal={myLink,binfo=Just b}}
=   seq [changeModelData b (\_->[]),
         closeModelData  b,
         seqPIO [QuitIO]] ps
```

2.2.3 The response function

The response function `respond` is the function that handles the manipulation propagation of a view process whenever the model data has changed. There are three cases:

a) The model data does not contain an entry to the link anymore.
b) The count value of the link has not changed.
c) The count value of the link has changed.

In case *a* the view should be removed from the model data, and the view process terminated. In case *b* and *c*, the running mode of the link should be reflected (enabling or disabling the Auto button) and the new model data should be stored in the local state of the view process. In case *c* the text field of the dialogue window should also display the new counter value.

```
respond :: ((Model,Model),State .p) -> State .p
respond ((_,newmodel),ps=:{pLocal={myLink,links,binfo=Just b}})
|   not i_exist
    =   seqPIO [QuitIO] (closeView b ps)
|   oldcount==newcount
    =   ps1
    =   seqPIO [SetWindow      windowId
               [SetTextControl textId (toString newcount)]] ps1
where
    (i_exist,newme) = selectLink myLink newmodel
    newcount        = newme.count
    (_,oldme)       = selectLink myLink links
    oldcount        = oldme.count
    ps1             = controlChange (setLocalLinks newmodel ps)
    controlChange   = if (autoOn  oldme newme) disableAuto
                     (if (autoOff oldme newme) enableAuto I)
```

3 The Explode game

The specification of the Explode challenge is as follows. The Explode game challenge concerns a program that provides a purely graphical interface to a game of Explode, which is a game for two or more players, in which each player has an inexhaustible supply of stones of their own distinctive colour. The game is played by placing the stones on the vertices of a finite connected graph, according to the rules below (in which we say a vertex is ‘full’ when it holds as many stones as it has incident edges). For this challenge, the program should support a game for a rectangular board, which dimensions can be set before playing.

Initially there are no stones on the graph. Players take it in turns to make a move. Each move increases the number of stones on the graph by one. A player takes a stone from their supply and places it on any vertex v not already containing stones of another colour. If this does not make v full, the move finishes. If it does make v full, then v ‘explodes’: each adjacent vertex is invaded by one of the stones. The colour of any stones already in the invaded vertex turns to that of the invaders. Any of the invaded vertices that is now full also explodes in turn, and so on until the graph is stable. The aim of the game for each player is to make it impossible for their opponents to move. A winning move for player P is either *(a)* one that causes an endless sequence of explosions, or *(b)* one that results in a stable graph in which every vertex contains at least one stone of P’s colour (and hence no other stones).

The Explode challenge can be written as a simple single interactive process application in the Clean I/O library. So the explanation of its code gives a good example of one way to write such an application.

In the Clean Event I/O system, interactive processes are state transition systems. Their state, the process state, is a record consisting of a private state (completely local to the process), a public state (shared between a group of processes), a file system environment (shared between all processes), and the IOState environment (partially local to the process, and containing amongst other things its Graphical User Interface elements). Because the challenge is a single process application, we need to consider the private state only. The private (and public) state of an interactive process represent the 'logical' state the process is in. The crucial component of the logical state of the Explode process is the state of the board. For this reason, this component is defined as an abstract data type (see Appendix B.1). Its implementation is straightforward.

The user interface is defined in a separate module (see Appendix B.2). In the design of the program we need consider the following topics:

a) What is the logical state of the Explode process?
b) How is the board graphically represented to the user?
c) How can infinite sequences of explosions be handled without making the program mute to user interrupts?

These subjects are handled in the following sub sections.

3.1 The logical state

The logical state of the Explode process is a record type Explode consisting of the current state of the board, the number of players that are currently playing, who's turn it is to play, and a colour table used to identify players visually. The initial value of the Explode record consists of a board of size 5×5, 2 players, starting with player 1, and the basic colours provided by the I/O library (so the maximum number of players is eight).

```
::  Explode     =   { board     :: Board,
                      nrplayers :: Int,
                      turn      :: Int,
                      colours   :: ColourTable    }
::  ColourTable :== [Colour]

initExplode     =   { board     = initBoard,
                      nrplayers = initPlayers,
                      turn      = 1,
                      colours   = initColours     }
where
    initBoard   = justis (newRectBoard initSize)
    initSize    = (initCols,initRows)
    initCols    = 5
    initRows    = 5
    initPlayers = 2
    initColours = [BlackColour,WhiteColour, RedColour,
                   GreenColour,BlueColour,  YellowColour,
                   CyanColour, MagentaColour ]
```

3.2 The user interface of the board

The program presents a window, showing the state of the board, the players, and an indication of who is currently playing. Figure 13 gives a snapshot of the window. This window is opened by the function `OpenExplodeWindow`, given in Figure 14. The window is a dialogue window, so the size of the window is determined completely by the controls it contains. There are two top-level compound controls, both centered below each other. The first compound control contains a text control with the text

“Player:”, and a custom control defined by the function PlayerControl. The second compound control contains a list of custom button controls, defined by the function ExplodeControl. One ExplodeControl corresponds exactly with one field of the board. Each ExplodeControl is therefore parameterized with the element of the board it represents. The window itself is identified by the identification value ExplodeWindowId, and the compound control displaying the board is identified by ExplodeDisplayId. These values are defined globally.

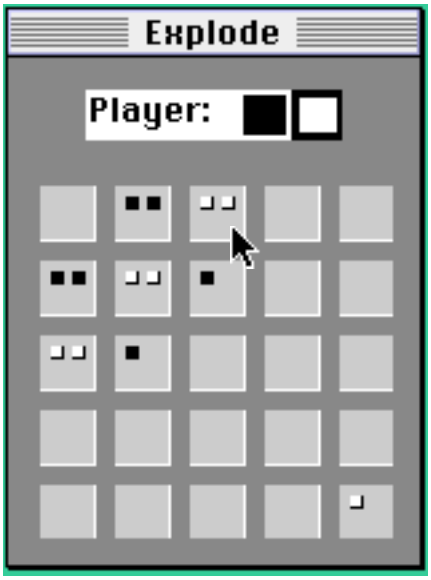

Figure 13 The Explode game running with two players, indicating who’s playing.

```
OpenExplodeWindow :: (State .p) -> State .p
OpenExplodeWindow ps=:{pLocal={board,nrplayers,turn,colours}}
=   seqPIO [OpenWindow explodeDef] ps
where
    explodeDef
    = DialogWindow "Explode"
        [CompoundControl
         [TextControl   "Player:" [],
          PlayerControl nrplayers turn colours]
         (\_ _ -> []) [ControlPos (Center,(0,0))],
         CompoundControl
          [ExplodeControl board colours ExplodeWindowId
                          (col,row)
           \\ row<-[1..initRows],
              col<-[1..initCols]]
          (\_ size -> background [SetPenColour  backColour,
                                  FillRectangle ((0,0),size),
                                  SetPenColour  BlackColour ])
                     [ControlPos (Center,(0,10)),
                      ControlId  ExplodeDisplayId]
        ]
        [WindowId         ExplodeWindowId,
         WindowItemSpace (hmm 2.5,vmm 2.5),
         WindowUpdate    background]
    (initCols,initRows) = dimension board
```

Figure 14 The definition of the Explode window. PlayerControl and ExplodeControl are program defined controls. Observe the use of a list comprehension to summarize the ExplodeControls concisely.

An ExplodeControl is a CustomButtonControl. Its definition is given below. The two most important aspects of its definition are the function *ExplodeLook* that defines the way it looks, and the abstract event handler *placeStone* that should be evaluated in case the control has actually been selected.

```
ExplodeControl :: Board ColourTable Id Position
               -> ControlDef (State .p)
ExplodeControl board colours wid position=:(col,row)
=   CustomButtonControl …
        (ExplodeLook position board colours)
        (if (col>1) controlAtts
                    [ControlPos (Left,(0,0)):controlAtts])
where
    myid        = ExplodeId position (snd (dimension board))
    controlAtts = [ControlId       myid,
                    ControlFunction (placeStone position)]
```

The function ExplodeLook is defined as a global function. The reason for this is that it is now easy to define a function that compares two boards, and performs the visual feedback of the changes between the two boards:

```
showchanges :: Id Board Board (State .p) -> State .p
showchanges wid oldboard newboard ps=:{pLocal={board,colours}}
=   seqPIO [SetWindow wid (map look differences)] ps
where
    differences  = compare oldboard newboard
    look (pos,_) = SetControlLook
                      (ExplodeId   pos (snd (dimension board)))
                      (ExplodeLook pos board colours)
```

The function compare yields a list of positions in which the two argument boards have a different content. ExplodeLook is mapped over this list, parameterized properly with the position that should be updated visually. These updates are process state transition functions. This list of functions can suitably be *seq*uenced over the process state argument of showchanges.

The function placeStone, parameterized with the position of the board element that it represents, places a stone on that position. Its definition is as follows:

```
placeStone :: Position (State .p) -> State .p
placeStone position=:(col,row) ps=:{pLocal={board,turn}}
|   legalMove move board
=   seq [seqPLoc        [\l->{l & board=board1}],
         showchanges     wid board board1,
         seqPIO         [SetWindow        ExplodeWindowId
                        [DisableControls  [ExplodeDisplayId]],
                         EnableTimer      ExplosionsId,
                         DisableMenuItems [NextPlayerId],
                         EnableMenuItems  [HaltId]] ] ps
=   seqPIO [Beep] ps
where
    move   = (turn,position)
    board1 = addStone move board
```

If the move is not legal, then placeStone simply emits a Beep sound and the move had no further effect. If the move is legal then placeStone sets the local state of the Explode process to the new board obtained by adding a stone to the board. It provides the proper visual feedback using the function showchanges discussed above. Then it disables the compound control that contains all ExplodeControls, because other players have to wait to play until a stable situation has been reached. The computation of this stable situation is done by a timer, explained in the next section, and so the timer is enabled. To be able to interrupt this computation the halt command is enabled.

3.3 Interruptable infinite explosions

Placing a stone on a field that thereby becomes full causes an explosion as explained in Section 3. In general it is possible that such an explosion causes a sequence of explosions. This sequence can be infinite. In the Clean Event I/O system, abstract event handlers are evaluated atomically. So, if one would evaluate the full sequence of explosions triggered by a move within one abstract event handler, then the evaluation of this function may possible not terminate and lock the program. One way to circumvent this problem is to have the abstract event handler analyse whether this sequence is infinite. However, also in this case, if the sequence happens to be very long, the program can not be interrupted by the user.

The solution we have taken is to use a timer that evaluates the moves. This timer is opened during the initial actions of the Explode process. Initially, the timer is disabled. In the previous sub section we saw that whenever a legal move has been made, the timer is enabled. In that case the timer's abstract event handler, doAnExplosion, is being evaluated.

```
doAnExplosion :: NrOfIntervals (State .p) -> State .p
doAnExplosion _ ps=:{pLocal={board}}
|   stable  board
=        seqPIO [DisableTimer    ExplosionsId,
                 EnableMenuItems [NextPlayerId]] ps
|   nothing newboard
=   seq [seqPIO [DisableTimer    ExplosionsId,
                 EnableMenuItems [NextPlayerId]],
                 SetWindow       ExplodeWindowId
                [EnableControls [ExplodeDisplayId]]],
         nextplayer] ps
=   seq [seqPLoc     [\l->{l & board=board1}],
         showchanges ExplodeWindowId board board1] ps
where
    newboard = explode board
    board1   = justis newboard
```

For an arbitrary board there are three alternatives `doAnExplosion` has to consider:

a) The board is stable: all fields contain stones of the same player.
b) The board contains no explosive fields.
c) The board contains explosive fields.

Case *a* is a winning situation for the player who made the last move. In this case the timer can be disabled, and the command to select a next player can be enabled. However, because this is a final situation for this game, the ExplodeControls remain disabled. The only sensible choice for the players is either to quit the application or start a new game.

In case *b* the sequence of explosions was finite, and so the timer can also be disabled, and the command to select a next player can be enabled. Because in this case we want to continue playing, the ExplodeControls are also enabled and the turn is given to the next player.

In case *c* there are still fields that can explode. The function explode detonates each of these fields once. The new board is placed in the local state of the Explode process. The visual feedback of the new state of the board is displayed by the function showchanges.

So each time the timer is evaluated it calculates one step in the sequence of explosions that might be caused by a move. Inbetween each step there is a small delay between subsequent timer evaluations to allow the program to respond to other events, such as interrupting the computation of explosions.

4 Discussion and conclusions

In the counter challenge we have first defined a generally applicable framework for a browser/editor on an arbitrary kind of model data. From the functional language ‘goodies’ we have applied are polymorphism, algebraic and abstract data types, and higher-order functions. The ‘goodies’ of the Event I/O system we have applied are the use of interactive processes to improve the module structure, polymorphic message passing, and the dynamic creation of devices, and receivers in particular. The Explode challenge being a single process application, did not depend on these extended features of the Event I/O system. However, it provided a good test-case for the suitability of the new definitions of controls and windows.

References

Achten, P.M., van Groningen J.H.G., and Plasmeijer, M.J. 1993.
High Level Specification of I/O in Functional Languages. In Launchbury, J., Sansom, P. eds., *Proceedings Glasgow Workshop on Functional Programming*, Ayr, Scotland, 6-8 July 1992. Workshops in Computing, Springer-Verlag, Berlin, 1993, pp. 1-17.

Achten, P.M. and Plasmeijer, M.J. 1994.
A Framework for Deterministically Interleaved Interactive Programs in the Functional Programming Language Clean. In Bakker, E. ed. *Proceedings Computing Science in the Netherlands, CSN'94*, Jaarbeurs Utrecht, The Netherlands, November 21-22, Stichting Mathematisch Centrum, Amsterdam, 1994, pp. 30-41.

Achten, P.M. and Plasmeijer, M.J. 1995.
The ins and outs of Clean I/O. In *Journal of Functional Programming* **5**(1) - January 1995, Cambridge University Press, pp. 81-110.

Brus, T., Eekelen, M.C.J.D. van, Leer, M.O. van, and Plasmeijer, M.J. 1987.
Clean: A Language for Functional Graph Rewriting. In Kahn. G. ed. *Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, USA, LNCS **274**, Springer-Verlag, pp. 364-384.

Noble, R. and Runciman, C. 1995.
Gadgets: Lazy Functional Components for Graphical User Interfaces. To appear in *Proceedings of Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, The Netherlands, 19-23 September, 1995, LNCS **??**, Springer-Verlag, pp. ??

Nöcker, E.G.J.M.H., Smetsers, J.E.W., Eekelen, M.C.J.D. van, and Plasmeijer, M.J. 1991.
Concurrent Clean. In Aarts, E.H.L., Leeuwen, J. van, Rem, M., eds., *Proceedings of Parallel Architectures and Languages Europe*, June, Eindhoven, The Netherlands. LNCS **506**, Springer-Verlag, pp. 202-219.

Plasmeijer, M.J. and van Eekelen, M.C.J.D. 1993.
Functional Programming and Parallel Graph Rewriting. Addison-Wesley Publishing Company 1993.

Appendices

A Counter challenge

A.1 The browser/editor framework

```
definition module browser


import StdEventIO, StdReceiver


::  Browser m
    =   { bViews     :: [ViewDef` Void Void m],
          bModelData :: m
        }
::  ViewDef`    E.l E.p m
    =   Hidden (ViewDef l p m)
::  ViewDef    l p m
    =   { vRespond   :: RespondF       m (PState l p),
          vInitState :: InitStateF l p m,
          vInitIO    :: InitIOF    l p m
        }
::  RespondF       m ps :== ((m,m),ps)      -> ps
::  InitStateF l p m    :== m               -> (l,p)
::  InitIOF    l p m    :== (BrowseInfo m) -> InitIO l p
```

openBrowser creates a browsing program, consisting of one control process, and an arbitrary number of view processes.

```
openBrowser    ::(Browser m) *World -> *World
```

BrowseInfo and its operations are needed for viewers to change the model data, create new viewers, and close their administration.

```
::  BrowseInfo m

changeModelData::(BrowseInfo m)  (m->m)  (PState .l .p) -> PState .l .p
closeModelData ::(BrowseInfo m)          (PState .l .p) -> PState .l .p
openViewer     ::(BrowseInfo m)  (ViewDef` Void Void m)
                                         (PState .l .p) -> PState .l .p
closeView      ::(BrowseInfo m)          (PState .l .p) -> PState .l .p
```

changeModelData	applies the action to the model data (asynchronously). The pair of old model data and new model data is then applied to all vRespond functions of the other view processes.
closeModelData	closes the control process. This action makes sense only at termination of the whole program.
openViewer	spawns a new view process that participates in all viewing actions of the program.

closeView — removes the view process from the program administration. As a result the view process is excluded from further updates of the model data. Note that the view process is not terminated by this function! This is the responsibility of the process itself. Note also that view processes can be terminated without applying closeView.

```
implementation module browser
```

```
import StdBool, StdList, StdFunc
import StdEventIO, StdReceiver
```

The exported types and functions of browsers.

```
::  Browser m
    =   { bViews     :: [ViewDef` Void Void m],
          bModelData :: m
        }
::  ViewDef` E.l E.p m
    =   Hidden (ViewDef l p m)
::  ViewDef    l p m
    =   { vRespond   :: RespondF       m (PState l p),
          vInitState :: InitStateF l p m,
          vInitIO    :: InitIOF    l p m
        }
::  RespondF       m ps :== ((m,m),ps)      -> ps
::  InitStateF l p m    :== m               -> (l,p)
::  InitIOF    l p m    :== (BrowseInfo m) -> InitIO l p
```

openBrowser creates a browsing program, consisting of one control process, and an arbitrary number of view processes.

```
openBrowser :: (Browser m) *World -> *World
openBrowser browser world
    =   OpenIO { ioDefInit =[initControlGUI browser],
                 ioDefAbout="Browse"            }
               ( initLocal browser.bModelData, 0)
               world
```

BrowseInfo and its operations are needed for viewers to change the model data, create new viewers, and close their administration.

```
::  BrowseInfo m
    =   { bView    :: ViewId m,
          bControl :: ControlId m
        }

changeModelData :: (BrowseInfo m) (m->m) (PState .l .p) -> PState .l .p
changeModelData bInfo action ps
=   seqPIO [ASyncSend bInfo.bControl (Action bInfo.bView action)] ps

closeModelData :: (BrowseInfo m) (PState .l .p) -> PState .l .p
closeModelData bInfo ps
=   seqPIO [ASyncSend bInfo.bControl Quit] ps

openViewer :: (BrowseInfo m) (ViewDef` Void Void m) (PState .l .p) -> PState .l .p
openViewer bInfo viewDef ps
=   seqPIO [ASyncSend bInfo.bControl (NewView viewDef)] ps

closeView :: (BrowseInfo m) (PState .l .p) -> PState .l .p
closeView bInfo ps
=   seqPIO [ASyncSend bInfo.bControl (CloseView bInfo.bView)] ps
```

The types and functions of a view process:

```
::  ViewM  m =   RespondTo (m,m)
::  ViewId m :== RId (ViewM m)
```

The receiver viewer function:

```
viewR :: (ViewDef .l .p m) (ViewM m) (PState .l .p) -> PState .l .p
viewR viewDef (RespondTo modelchange) ps = viewDef.vRespond (modelchange, ps)
```

The types and functions of the control process:

```
::  ControlM  m =   Action    (ViewId m) (m->m)
                  | NewView              (ViewDef` Void Void m)
                  | NewViewId (ViewId m)
                  | CloseView (ViewId m)
                  | Quit
::  ControlId m :== RId (ControlM m)

::  LocalC    m =   { views  :: [ViewId m],
                      model  :: m,
                      myself :: Maybe (ControlId m)
                    }
::  Maybe     x =   Nothing | Just x
::  *CState   m :== PState (LocalC m) Int
```

The initial local state:

```
initLocal :: m -> LocalC m
initLocal initModelData = { views  = [],
                            model  = initModelData,
                            myself = Nothing }
```

The initial actions:

```
initControlGUI :: (Browser m) (CState m) -> CState m
initControlGUI browser ps=:{pLocal={model,views,myself}}
=   seq (map openView browser.bViews) ps1
where
    (me,io1) = OpenReceiver rDef ps.pIOState
    rDef     = Receiver [ReceiverFunction controlR]
    ps1      = { ps & pLocal={ myself   = Just me,
                               model    = model,
                               views    = views  },
                      pIOState = io1
               }
```

The receiver control function:

```
controlR :: (ControlM m) (CState m) -> CState m
controlR (Action sender act) ps=:{pLocal}
=   {ps & pLocal  = {pLocal & model=newModel},
          pIOState= broadcast others (RespondTo (model,newModel)) ps.pIOState}
where
    newModel = act pLocal.model
    others   = filter (not o eqRId sender) pLocal.views
controlR (NewView viewDef) ps
=   openView viewDef ps
controlR (NewViewId sender) ps=:{pLocal}
```

```
=   {ps & pLocal = {pLocal & views=[sender:pLocal.views]}}
controlR (CloseView sender) ps=:{pLocal}
=   {ps & pLocal = {pLocal & views=filter (not o eqRId sender) pLocal.views}}
controlR Quit ps
=   seqPIO [QuitIO] ps
```

Creating a new view process:

```
openView :: (ViewDef` Void Void m) (CState m) -> CState m
openView (Hidden viewDef) ps=:{pLocal={model,myself=Just me}}
=   seqPIO [NewIO ioDef initState] ps
where
    ioDef     = { ioDefInit = [initview me viewDef],
                  ioDefAbout= "Browse"              }
    initState = viewDef.vInitState model

    initview::(ControlId m) (ViewDef .l .p m) (PState .l .p) -> PState .l .p
    initview controlId viewDef ps
    =   seq initIO {ps & pIOState=io2}
    where
        (viewId,io1) = OpenReceiver rDef ps.pIOState
        rDef         = Receiver [ReceiverFunction (viewR viewDef)]
        io2          = ASyncSend controlId (NewViewId viewId) io1
        browseInfo   = { bView   = viewId,
                         bControl= controlId }
        initIO       = viewDef.vInitIO browseInfo
```

Asynchronous broadcast:

```
broadcast :: [RId m] m (IOState .l .p) -> IOState .l .p
broadcast [id:ids] message ioState
=   broadcast ids message (ASyncSend id message ioState)
broadcast _ _ ioState = ioState
```

A.2 The counter as a special case

```
module counter


import StdEnv, StdWindow, StdControl, StdPicture, StdTimer
import browser


::  Model         :== [Link]
::  Link          =   { link   :: LinkId,
                        count  :: Int,
                        auto   :: Bool
                      }
::  Local         =   { myLink :: LinkId,
                        links  :: [Link],
                        binfo  :: Maybe (BrowseInfo Model)
                      }
::  LinkId        :== Int
::  Maybe  x      =   Nothing | Just x
::  Nil           =   Nil
::  *State public :== PState Local public


Start :: *World -> *World
Start world = openBrowser browser world
where
    browser    = { bViews    =[view initLinknr],
                   bModelData=[{link=initLinknr,count=0,auto=False}]}
    view l     = Hidden { vRespond  = respond,
```

```
                     vInitState= initstate l,
                     vInitIO   = initio     }
initLinknr = 0

initstate :: LinkId Model -> (Local,Nil)
initstate l model
=    ({ myLink = l,
        links  = if i_exist model [{link=l,count=0,auto=False}:model],
        binfo  = Nothing},
      Nil)
where
    (i_exist,_) = selectLink l model

initio :: (BrowseInfo Model) -> InitIO Local .p
initio b = [initIO` b]
where
    initIO` b ps=:{pLocal={myLink,links}}
    =   seqPIO [OpenWindow wDef,OpenTimer tDef]
        {ps & pLocal={ps.pLocal & binfo=Just b}}
    where
        tDef= Timer TicksPerSecond [TimerId       timerId,
                                    TimerSelect   Unable,
                                    TimerFunction (\_ ps->Incr ps)]
        wDef= DialogWindow ("Counter"+++toString myLink) manualId
              [TextControl (toString me.count) [ControlId          textId,
                                                ControlSize        (150,20)  ],
               ButtonControl "Manual"          [ControlFunction    Manual,
                                                ControlId          manualId,
                                                                   left],
               ButtonControl "Auto"            [ControlFunction    Auto,
                                                ControlId          autoId,
                                                ControlSelectState autoAble  ],
               ButtonControl "Copy"            [ControlFunction    Copy,
                                                                   left      ],
               ButtonControl "Link"            [ControlFunction    Link      ],
               ButtonControl "Close"           [ControlFunction    Close,
                                                                   left      ],
               ButtonControl "Quit"            [ControlFunction    Quit      ]
              ]
              [WindowId windowId,
               WindowOk manualId
              ]
        left     = ControlPos (Left,(0,0))
        manualId = 3
        (_,me)   = selectLink myLink links
        autoAble = if me.auto Unable Able

        Incr :: (State .p) -> State .p
        Incr ps=:{pLocal={myLink,links,binfo=Just b}}
        = seq [setLocalLinks (newlink links),
               seqPIO [SetWindow      windowId
                       [SetTextControl textId (toString newcount)]],
               changeModelData b newlink] ps
        where
            (_,me)    = selectLink myLink links
            newcount  = 1+me.count
            newlink ls= map (\l->if (myLink==l.link) {l & count=newcount} l) ls

        Manual :: (State .p) -> State .p
        Manual ps=:{pLocal={myLink,links,binfo=Just b}}
        = seq [setLocalLinks     (setAuto myLink False links),
               enableAuto,
               changeModelData b (setAuto myLink False),
               Incr] ps

        Auto :: (State .p) -> State .p
        Auto ps=:{pLocal={myLink,links,binfo=Just b}}
        = seq [setLocalLinks (setAuto myLink True links),
               seqPIO        [EnableTimer timerId],
```

```
                  disableAuto,
                  changeModelData b (setAuto myLink True)] ps

           Copy :: (State .p) -> State .p
           Copy ps=:{pLocal={links,binfo=Just b}}
           = seq [changeModelData b (newviews newlinknr),
                  openViewer      b (view newlinknr)] ps
           where
               newlinknr = 1+(foldr (\x y->max x y) 0 (map (\x->x.link) links))
               newviews l model = [{link=l,count=0,auto=False}:model]

           Link :: (State .p) -> State .p
           Link ps=:{pLocal={myLink,binfo=Just b}} = openViewer b (view myLink) ps

           Close :: (State .p) -> State .p
           Close ps=:{pLocal={links=[_]}} = Quit ps
           Close ps=:{pLocal={myLink,binfo=Just b}}
           = seq [changeModelData b (removeLink myLink),
                  closeView       b,
                  seqPIO          [QuitIO]] ps

           Quit :: (State .p) -> State .p
           Quit ps=:{pLocal={myLink,binfo=Just b}}
           = seq [changeModelData b (\_->[]),
                  closeModelData  b,
                  seqPIO          [QuitIO]] ps

    timerId           = 1
    windowId          = 1
    [textId,autoId:_] = [1..]

    respond :: ((Model,Model), State .p) -> State .p
    respond ((_,newmodel),ps=:{pLocal={myLink,links,binfo=Just b}})
    |   not i_exist
    =   seqPIO [QuitIO] (closeView b ps)
    |   oldcount==newcount
    =   ps1
    =   seqPIO [SetWindow windowId
               [SetTextControl textId (toString newcount)]] ps1
    where
        (i_exist,newme)    = selectLink myLink newmodel
        newcount           = newme.count
        (_,oldme)          = selectLink myLink links
        oldcount           = oldme.count
        ps1                = controlChange (setLocalLinks newmodel ps)
        controlChange      = if (autoOn  oldme newme) disableAuto
                              (if (autoOff oldme newme) enableAuto I)

    enableAuto  ps = seqPIO [SetWindow    windowId [EnableControls  [autoId]],
                             DisableTimer timerId] ps
    disableAuto ps = seq    [SetWindow    windowId [DisableControls [autoId]] ps


selectLink :: LinkId [Link] -> (Bool,Link)
selectLink i [l:ls]
|   i == l.link = (True,l)
                = selectLink i ls
selectLink i [] = (False,{link=0,count=0,auto=False})

removeLink :: LinkId [Link] -> [Link]
removeLink i [l:ls]
|   i == l.link = ls
                = [l:removeLink i ls]
removeLink i [] = []

setAuto :: LinkId Bool [Link] -> [Link]
setAuto i a [l:ls]
|   i == l.link = [{l & auto=a}:ls]
                = [l:setAuto i a ls]
```

```
setAuto i a []  = []

autoOn  :: Link Link -> Bool
autoOn  old new = not old.auto && new.auto

autoOff :: Link Link -> Bool
autoOff old new = old.auto && not new.auto


setLocalLinks :: [Link] (State .p) -> State .p
setLocalLinks ls ps = {ps & pLocal={ps.pLocal & links=ls}}
```

B The Explode game

B.1 The board data structure

definition module board

```
from StdIOCommon import Size


::  Board
::  Player   :== Int
::  Position :== (Int,Int)              // (row,col) of board.
                                        // Rows range from left to right  (1-width ),
                                        // Cols range from top  to bottom (1-height).
::  Move     :== (Player, Position)
::  Content  :== Maybe (Player, Int)    // If stones>0 then (Just (player,stones)) otherwise Nothing.
::  Maybe x  =   Just x | Nothing
```

convenient operations on Maybe:

```
just    :: (Maybe x) -> Bool      // just (Just _)       = True;   just Nothing        = False
nothing :: (Maybe x) -> Bool      // nothing Nothing  = True;   nothing (Just _)    = False
justis  :: (Maybe x) -> x         // justis (Just x)     = x;      justis Nothing      = undef
```

creation of boards:

```
newRectBoard:: Size -> Maybe Board   // Create a new rectangular, empty board of size (w,h).
                                     // If (w<3 || h<3) then (newRectBoard @1) = Nothing.
```

checks on boards:

```
stable      :: Board      -> Bool    // All positions are filled with stone of same player
legalMove   :: Move Board -> Bool    // A move (player,pos) is legal if:
                                     //         pos is in range &&
                                     //         content pos @2=Nothing ||
                                     //         content pos @2=Just (player,_)
```

generating new boards:

```
explode :: Board -> Maybe Board      // Explode a position if available.
                                     // If nothing exploded then (explode @1) = Nothing.
addStone:: Move Board -> Board       // Place the stone on the given position.
                                     // If (not (legalMove @1 @2)) then (addStone @1 @2)=@2.
```

access on boards:

```
dimension   ::Board          -> Size        // The size of the board.
maxboardload::Board          -> Int         // The maximum of all maximal possible stone loads.
content     ::Position Board -> Content     // The content of @2 at @1.
                                            // If @1 out of range then (content @1 @2) = Nothing.
compare     ::Board Board    -> [(Position, Content)]
                                            // The differences between @1 and @2.
                                            // If dimension @1<>dimension @2: result is [].
```

implementation module board

```
import StdEnv,StdList
from   StdIOCommon import Size

::  Board    :== [[Content]]               // Board = [column], column = [row].
::  Player   :== Int
::  Position :== (Int,Int)                 // (row,col) of board.
                                           // Rows range from left to right  (1-width ),
                                           // Cols range from top  to bottom (1-height).
::  Move     :== (Player, Position)
::  Content  :== Maybe (Player, Int)       // If stones>0 then (Just (player,stones)) else Nothing.
::  Maybe x  =   Just x | Nothing
```

convenient operations on Maybe:

```
just    :: (Maybe x) -> Bool
just    (Just _)     = True
just    _            = False

nothing :: (Maybe x) -> Bool
nothing Nothing      = True
nothing _            = False

justis  :: (Maybe x) -> x
justis  (Just x)     = x
```

convenient operations on where (corner,interior,side,board) a position is:

```
corner    :: Size Position -> Bool
corner    (w,h) (x,y) = x==1&&y==1 || x==1&&y==h || x==w&&y==1 || x==w&&y==h

interior  :: Size Position -> Bool
interior  (w,h) (x,y) = between 2 (w-1) x && between 2 (h-1) y

edge      :: Size Position -> Bool
edge      size pos    = not (interior size pos || corner size pos)

inboard   :: Size Position -> Bool
inboard   (w,h) (x,y) = between 1 w x && between 1 h y

between   :: Int Int Int -> Bool
between   min max x   = min<=x && x<=max

maxstones :: Size Position -> Int
maxstones size pos
|   corner   size pos = 2
|   interior size pos = 4
                      = 3
```

creation of boards:

```
/*  newRectBoard:
        Create a new rectangular, empty board of size (w,h).
```

```
        If (w<3 || h<3) then (newRectBoard @1) = Nothing.
*/
newRectBoard :: Size -> Maybe Board
newRectBoard (w,h)
|   w<3 || h<3    = Nothing
                  = Just (repeatn w (repeatn h emptyContent))
where
    emptyContent = Nothing
```

checks on boards:

```
/*  stable:
        All positions are filled with stone of same player
*/
stable :: Board -> Bool
stable board
|   nothing c = False
              = all (\row->all (posstable p) row) board
where
    c         = content (1,1) board
    (p,_)     = justis c

    posstable p (Just (p`,_)) = p==p`
    posstable p _             = False


/*  explosive:
        All elements that can explode.
*/
explosive :: Board -> [Position]
explosive board = flatten (map (explosivecol (dimension board)) (zip2 [1..] board))
where
    explosivecol :: Size (Int,[Content]) -> [Position]
    explosivecol size (colnr,col) = explosivefields size (colnr,1) col

    explosivefields :: Size Position [Content] -> [Position]
    explosivefields size position=:(col,row) [c:cs]
    |   canexplode size position c = [position:positions]
                                   = positions
    where
        positions                  = explosivefields size (col,row+1) cs

        canexplode size pos (Just (_,stones)) = stones==maxstones size pos
        canexplode  _ _ _                     = False
    explosivefields _ _ _          = []



/*  legalMove:
        The move (player,pos) is legal if:
        pos is in range &&
        content pos @2 = Nothing || content pos @2 = Just (player,_)
*/
legalMove :: Move Board -> Bool
legalMove (player,pos) board
|   inboard size pos    = nothing c || player==p
                        = False
where
    size  = dimension board
    c     = content pos board
    (p,_) = justis c
```

generating new boards:

```
/*  explode:
        Explode all positions.
        If (explosive @1=[]) then (explode @1) = Nothing.
```

```
*/
explode :: Board -> Maybe Board
explode board
|   expls==[]  = Nothing
               = Just (seq (clear++fill) board)
where
    expls      = explosive board
    (player,_) = justis (content (hd expls) board)
    clear      = map (\pos->changeContent pos (\_->Nothing)) expls
    fill       = flatten (map (\(col,row)->[addStone (player,(col-1,row  )),
                                           addStone (player,(col,  row-1)),
                                           addStone (player,(col,  row+1)),
                                           addStone (player,(col+1,row))]) expls)


/*  addStone:
        Place the stone of the player on the given position, and set the
        position to be owned by this player.
        If position is not in range then (addStone @1 @2) = @2.
*/
addStone :: Move Board -> Board
addStone (player,pos) board
    = changeContent pos (addStone` (maxstones size pos) player) board
where
    size = dimension board

    addStone` :: Int Player Content -> Content
    addStone` max player Nothing           = Just (player,1)
    addStone` max player (Just (_,stones)) = Just (player,min (stones+1) max)


/*  changeContent:
        If position is not in range then changeContent @1 @2 @3 = @3.
*/
changeContent :: Position (Content->Content) Board -> Board
changeContent pos=:(col,row) f board
|   inboard size pos          = cbefore++[rbefore++[f c]++rafter]++cafter
                              = board
where
    size                      = dimension board
    (cbefore,[column:cafter]) = split (col-1) board
    (rbefore,[c     :rafter]) = split (row-1) column

    split :: Int [x] -> ([x],[x])
    split 0 xs        = ([],xs)
    split i [x:xs]    = ([x:xs1],xs2)
    where
        (xs1,xs2)    = split (i-1) xs
```

access on boards:

```
/*  dimension:
        The size of the board.
*/
dimension :: Board -> Size
dimension board=:[col:_] = (#board,#col)


/*  maxboardload:
        The maximum of all maximal possible stone loads.
*/
maxboardload :: Board -> Int
maxboardload _ = 4


/*  content:
```

```
        The content of @2 at @1.
        If @1 out of range then (content @1 @2) = Nothing.
*/
content :: Position Board -> Content
content pos=:(col,row) board
|   inboard (dimension board) pos = board!(col-1)!(row-1)
                                  = Nothing


/*  compare:
        The differences between @1 and @2.
        If dimension @1 <> dimension @2 then compare @1 @2 = [].
*/
compare :: Board Board -> [(Position, Content)]
compare board1 board2
|   x1==x2 && y1==y2 = comparecols 1 (zip2 board1 board2)
                     = []
where
    (x1,y1)          = dimension board1
    (x2,y2)          = dimension board2

    comparecols :: Int [([Content],[Content])] -> [(Position, Content)]
    comparecols col [zipcol:zipcols]
    = comparepos (col,1) zipcol++comparecols (col+1) zipcols
    where
        comparepos :: Position ([Content],[Content]) -> [(Position, Content)]
        comparepos position=:(col,row) ([c1:cs1],[c2:cs2])
        |   c1==c2     = comparepos (col,row+1) (cs1,cs2)
                       = [(position,c2):comparepos (col,row+1) (cs1,cs2)]
        comparepos _ _ = []
    comparecols _ [] = []

instance == (Maybe x) | Eq x
(==) :: !(Maybe x) !(Maybe x) -> Bool | Eq x
(==) Nothing    Nothing   = True
(==) (Just x1)  (Just x2) = x1==x2
(==) _          _         = False
```

B.2 The user interface

```
module Explode


import StdEnv
import StdEventIO, StdControl, StdWindow, StdMenu, StdTimer
import StdPicture, StdFont, StdIOState, StdSystem
import board
```

The major type definitions of the Explode game:

```
::  Explode       =  { board      :: Board,        // The board to play on
                       nrplayers  :: Int,          // The nr. of players (2-#colours)
                       turn       :: Int,          // Who is playing
                       colours    :: ColourTable   // The available colours (#colours>=2)
                     }
::  ColourTable   :== [Colour]
::  *State public :== PState Explode public
::  Nil           =  Nil
```

Starting the Explode game:

```
Start :: *World -> *World
Start world
=   OpenIO {ioDefInit=initIO,ioDefAbout="Explode"} (initExplode,Nil) world
where
```

```
initExplode     = { board     = initBoard,
                    nrplayers = initPlayers,
                    turn      = 1,
                    colours   = initColours    }
where
    initBoard   = justis (newRectBoard initSize)
    initSize    = (initCols,initRows)
    initCols    = 5
    initRows    = 5
    initPlayers = 2
    initColours = [BlackColour,WhiteColour, RedColour, GreenColour,
                   BlueColour, YellowColour,CyanColour,MagentaColour ]

initIO  = [OpenExplodeWindow, seqPIO [OpenMenu 0 menuDef, OpenTimer timerDef]]

menuDef = Menu "Explode"
              [MenuItem "New..." [MenuFunction    new,
                                  MenuShortKey    'n'           ],
               MenuItem "Next"   [MenuId          NextPlayerId,
                                  MenuFunction    nextplayer    ],
               MenuItem "Halt"   [MenuId          HaltId,
                                  MenuShortKey    '.',
                                  MenuFunction    halt,
                                  MenuSelectState Unable        ],
               MenuItem "Quit"   [MenuShortKey    'q',
                                  MenuFunction    (seqPIO [QuitIO])]]
          []
where
    halt :: (State .p) -> State .p
    halt ps = seqPIO [DisableTimer ExplosionsId, DisableMenuItems [HaltId]] ps

    new :: (State .p) -> State .p
    new ps=:{pLocal={board,nrplayers,colours}}
        = seqPIO [DisableTimer ExplosionsId, OpenWindow newgameDef] ps
    where
        newgameDef
        = DialogWindow "New Game"
            [EditControl   (toString nrplayers) 100 1  [ControlId nrid],
             TextControl   ("nr.of players (2.."+++(toString maxplayers)+++")")
                                                   [],
             EditControl   (toString w) 100 1 [ControlId wid,  left],
             TextControl   "width (3..)"      [],
             EditControl   (toString h) 100 1 [ControlId hid,  left],
             TextControl   "height (3..)"     [],
             ButtonControl "Cancel"           [ControlFunction
                                                        (cancel NewGameId),
                                                           left],
             ButtonControl "Ok"               [ControlId       okid,
                                               ControlFunction ok  ]]
            [WindowId NewGameId,
             WindowOk okid     ]
        left                  = ControlPos (Left,(0,0))
        (w,h)                 = dimension board
        maxplayers            = #colours
        [okid,nrid,wid,hid:_] = [1..]

        cancel :: Id (State .p) -> State .p
        cancel wid ps = seqPIO [CloseWindow wid] ps

        ok :: (State .p) -> State .p
        ok ps
        |   players<2 || players> maxplayers
        =   notice ("Nr.of players between 2 and "+++toString maxplayers) ps1
        |   width<3   || height<3
        =   notice ("Width and height should be larger than 3") ps1
        =   seq [cancel  NewGameId,
                 seqPLoc [newlocal players (width,height)],
                 NewExplodeWindow] ps1
        where
```

```
                (_,info,io) = GetWindow NewGameId ps.pIOState
                ps1         = {ps & pIOState=io}
                players     = toInt (snd (GetEditTextControl nrid info))
                width       = toInt (snd (GetEditTextControl wid  info))
                height      = toInt (snd (GetEditTextControl hid  info))

                newlocal players size local
                            = {local & board    = justis (newRectBoard size),
                                       nrplayers= players,
                                       turn     = 1}

                notice text ps
                =   OpenModalWindow noticeDef ps
                where
                    noticeDef
                    = DialogWindow "Incorrect input"
                      [TextControl   text [],
                       ButtonControl "Aha!" [ControlId       confirmid,
                                             ControlFunction (cancel NoticeId)]]
                      [WindowId NoticeId,
                       WindowOk confirmid]
                    where
                        confirmid = 1


    timerDef = Timer 0 [TimerId       ExplosionsId,
                        TimerSelect   Unable,
                        TimerFunction doAnExplosion]
    where
        doAnExplosion :: NrOfIntervals (State .p) -> State .p
        doAnExplosion _ ps=:{pLocal={board}}
        |   stable  board    =         seqPIO [DisableTimer    ExplosionsId,
                                               EnableMenuItems [NextPlayerId]] ps
        |   nothing newboard = seq    [seqPIO [DisableTimer    ExplosionsId,
                                               EnableMenuItems [NextPlayerId],
                                               SetWindow       ExplodeWindowId
                                              [EnableControls [ExplodeDisplayId]]],
                                       nextplayer] ps
                             = seq    [seqPLoc [\l->{l & board=board1}],
                                       showchanges ExplodeWindowId board board1] ps
        where
            newboard         = explode board
            board1           = justis newboard



/*  OpenExplodeWindow:
        The explode window in which the game is played.
*/
OpenExplodeWindow :: (State .p) -> State .p
OpenExplodeWindow ps=:{pLocal={board,nrplayers,turn,colours}}
=   seqPIO [OpenWindow explodeDef] ps
where
    explodeDef = DialogWindow "Explode"
                 [CompoundControl [TextControl   "Player:" [],
                                   PlayerControl nrplayers turn colours]
                                  (\_ _ -> []) [ControlPos (Center,(0,0))],
                  CompoundControl [
                     ExplodeControl board colours ExplodeWindowId (col,row)
                     \\ row<-[1..initRows],
                        col<-[1..initCols]]
                     (\_ size -> background [SetPenColour  backColour,
                                             FillRectangle ((0,0),size),
                                             SetPenColour  BlackColour ])
                     [ControlPos (Center,(0,10)),
                      ControlId  ExplodeDisplayId]
                 ]
                 [WindowId        ExplodeWindowId,
                  WindowItemSpace (hmm 2.5,vmm 2.5),
```

```
                 WindowUpdate    background]
     (initCols,initRows) = dimension board

    background upd ps   = seqPIO [DrawInWindow ExplodeWindowId drawfs] ps
    where
        drawfs     = [  SetPenColour backColour
                     :  map FillRectangle upd    ]
                     ++
                     [  SetPenColour BlackColour ]
        backColour = RGB 0.5 0.5 0.5


/*  NewExplodeWindow:
        Close the current explode window and open a new one, depending on the new
        local state.
*/
NewExplodeWindow :: (State .p) -> State .p
NewExplodeWindow ps
=   seq [seqPIO [CloseWindow ExplodeWindowId],OpenExplodeWindow] ps


/*  ExplodeControl:
        The explode control, in control of one field of the explode game:
*/
ExplodeControl :: Board ColourTable Id Position -> ControlDef (State .p)
ExplodeControl board colours wid position=:(col,row)
=   CustomButtonControl (buttonW+1,buttonH+1)
                        (ExplodeLook position board colours)
            (if (col>1) controlAtts
                       [ControlPos (Left,(0,0)):controlAtts])
where
    myid               = ExplodeId position (snd (dimension board))
    controlAtts        = [ControlId       myid,
                          ControlFunction (placeStone position)    ]
    (buttonW, buttonH) = ExplodeSize board

    placeStone :: Position (State .p) -> State .p
    placeStone position=:(col,row) ps=:{pLocal={board,turn}}
    |   legalMove move board = seq [seqPLoc         [\l->{l & board=board1}],
                                    showchanges     wid board board1,
                                    seqPIO [SetWindow        ExplodeWindowId
                                           [DisableControls  [ExplodeDisplayId]],
                                            EnableTimer      ExplosionsId,
                                            DisableMenuItems [NextPlayerId],
                                            EnableMenuItems  [HaltId]] ] ps
                             = seqPIO [Beep] ps
    where
        move   = (turn,position)
        board1 = addStone move board

StoneSize :== (4,4)

ExplodeSize :: Board -> Size
ExplodeSize board
=   (w*space,h*space)
where
    maxload   = maxboardload board
    d_real    = sqrt (toReal maxload)
    d_int     = toInt d_real
    cols_rows = if ((d_real*d_real)<(toReal maxload)) (d_int+1) d_int
    space     = 2*cols_rows+1
    (w,h)     = StoneSize

ExplodeLook :: Position Board ColourTable SelectState Size -> [DrawFunction]
ExplodeLook position board colours _ (buttonW,buttonH)
    =   [ SetPenColour  (RGB 0.8 0.8 0.8),
          FillRectangle explodebox,
          DrawRectangle explodebox    ]
        ++
```

```
          drawload
        ++
        [ SetPenColour  BlackColour   ]
where
    c                = content position board
    has_stones       = just c
    colour           = if has_stones (colours!(player-1)) WhiteColour
    (player,stones)  = justis c
    drawload         = if has_stones
                           (take stones (map (drawstone colour) stone_rects))
                           []
    explodebox       = ((0,0),(buttonW,buttonH))
    (w,   h)         = StoneSize
    (col, row)       = ((buttonW-w)/w, (buttonH-h)/h)
    stone_rects      = [((x*w,y*h),((x+1)*w,(y+1)*h))
                       \\ y<-[1,3..row],x<-[1,3..col]]
    drawstone col rect=:(lt,(r,b)) picture
        =   FillRectangle rect              (
            SetPenColour  col               (
            FillRectangle (lt,(r+1,b+1))    (
            SetPenColour  BlackColour picture)))


/*  PlayerControl:
        The player control, showing who's playing:
*/
PlayerControl :: Int ColourTable -> ControlDef (State .p)
PlayerControl nrplayers turn colours
=   CustomControl (pcW*nrplayers,pcH)
                  (PlayerLook nrplayers turn colours) (ListCS [])
                  [ControlSelectState Unable,
                   ControlId          myid  ]
where
    (pcW,pcH) = PlayerSize

PlayerLook :: Int Int ColourTable SelectState Size -> [DrawFunction]
PlayerLook nrplayers turn colours _ (w,h)
    =   [EraseRectangle ((0,0),(pcW*nrplayers,pcH))] ++ drawcolours ++ drawturn
where
    turnW       = 2

    drawcolours = map drawcolour (zip2 [0..nrplayers-1] (take nrplayers colours))
    drawcolour (i,col) picture
        = seq [SetPenColour  col,
               FillRectangle colourbox,
               SetPenColour  BlackColour,
               DrawRectangle colourbox ] picture
    where
        colourbox = ((i*pcW+turnW,turnW),((i+1)*pcW-turnW,pcH-turnW-1))

    drawturn    = [SetPenSize    (turnW,turnW),
                   SetPenColour  BlackColour,
                   DrawRectangle (((turn-1)*pcW,0),(turn*pcW,pcH-(turnW-1))),
                   SetPenSize    (1,1)]
    (pcW,pcH)   = PlayerSize

PlayerSize :: Size
PlayerSize = (20,20)


showchanges :: Id Board Board (State .p) -> State .p
showchanges wid oldboard newboard ps=:{pLocal={board,colours}}
    =   seqPIO [SetWindow wid (map look (compare oldboard newboard))] ps
where
    look (pos,_) = SetControlLook (ExplodeId   pos (snd (dimension board)))
                                  (ExplodeLook pos board colours)

nextplayer :: (State .p) -> State .p
nextplayer ps=:{pLocal={nrplayers,turn,colours}}
```

```
=   seq [seqPLoc [\l->{l & turn=turn`}],
         seqPIO  [SetWindow ExplodeWindowId [SetControlLook PlayerId look]]] ps
where
    turn`= 1+(turn mod nrplayers)
    look = PlayerLook nrplayers turn` colours
```

global Ids:

```
ExplodeWindowId  = 1          // The Id of the explode window
PlayerId         = 1          // The Id of the PlayerControl
ExplodeDisplayId = 2          // The Id of the explode display (a compound)

ExplodeId        :: Position Int -> Id
ExplodeId        (col,row) nrRows = col*10^(nrdigits nrRows)+row

NewGameId        = 2          // The Id of the New Game dialogue
NoticeId         = 3          // The Id of the notice dialogue

ExplosionsId     = 1          // The Id of the timer

NextPlayerId     = 1          // The Id of the Next command
HaltId           = 2          // The Id of the Halt command
```

convenience operations:

```
nrdigits :: Int -> Int
nrdigits x = if (x<0) (1+(nrdigits` (0-x))) (nrdigits` x)
where
    nrdigits` x = if (x<10) 1 (1+(nrdigits` (x/10)))
```