POKer, a Process-Interaction Simulator and Controller for use in Collaborative Simulation

Andriy Levytskyy, Eugene J.H. Kerckhoffs

Delft University of Technology
Faculty of Information Technology and Systems
Mediamatica Department
Zuidplantsoen 4, 2628 BZ Delft, The Netherlands
e-mail: a.levytskyy@cs.tudelft.nl

Keywords

Discrete-Event simulation, Process-Interaction world-view, operational semantics, scripting languages, Python.

Abstract

In this paper we present the ideas and implementation of a small process-interaction kernel and cover its evolution from operational semantics of process-oriented simulation languages to the current state of the kernel as (simulation) core of a collaborative environment controller.

1. Introduction

Simulation as problem-solving methodology has been widely adopted in many practical domains. Users are free to select from a number of available general-purpose simulation software [14, 4] or, in case of special user requirements, a custom software system can be developed. The latter scenario has a potential negative side effect: simulation often becomes interpreted and applied by developers in a less formal way. As the result, it becomes more difficult to reason about properties of such simulation systems and prove their correctness.

On the other hand, Simulation and Modelling formal theory provides a reliable framework to support the entire process of modelling and simulation [18]. There exist formal works dedicated to the study of simulation languages and simulators for some major formalisms and world-views [2, 3, 2]. They provide formally proven semantic backgrounds and implementation reference for development of custom simulation systems. One of the widely used basic formalisms is the discrete event formalism, and a subset of it, a process interaction world-view [20], which is of particular interest to us. This world-view gained its popularity due to the easiness of mapping real-world processes into this formalism, and easiness of comprehending the essence of the process by a human.

The above, as well as special application requirements for the simulation system we seek, inspired us to develop our own special-purpose simulation software while trying to preserve its connection to the formal background. In this paper we present the ideas and implementation of a small process-interaction kernel and cover its evolution from operational semantics of process oriented simulation languages to the current state of the kernel as (simulation) core of a collaborative environment controller.

2. View on the environment

2.1. Functional overview

In order to understand better the initial requirements for our custom system, we give a brief functional overview of the intended environment (for more information, please see [9]). It is based on a three layer architecture: (i) front-end layer (high-level front-ends), (ii) middle layer (environment controller), and finally, (iii) back-end layer (distributed heterogeneous hardware and software resources).

The front layer consists of a number of client-side **front-ends**. They provide users with a high level GUI to the system (e.g., web-based interface), enable them to operate in the environment, accomplish background routine jobs necessary for execution of users' activities, and finally, communicate with the middle layer.

Requirements	*Property Description*
Handling external events	Many software systems providing the process interaction world-view do not natively support external events. Scheduling mechanism should be aware of external and internal events, and as a result, properly schedule the involved event notices on the same EL.
Support of both real-time and as-soon-as-possible time regimes	Transparently processing both types of processes (programs) intermixed on the same EL.
Adjustments in synchronization mechanism	Connectivity to environment resources changes operational semantics of standard simulation synchronisation mechanism.
Programming language	Language constructs are needed to support mapping a sequence of user's real-world activities into a process.
Transparent support of different world-views	For users the most natural framework is process interaction. For the simulator, which deals with resource-oriented scenarios, event scheduling is more proper.
Freedom of customization	We need powerful constructs to represent components of our system and freedom to adjust them easily for our needs.
Unrestricted monitoring of the activities in the environment	3rd party systems are often 'black boxes' and it would be impossible or very difficult to make them provide all the time full overview on all that is happening in the environment.
Easy integration with other industry technologies	Web, distributed objects, network programming, inter-process communication, de-facto standard system development languages (C/C++, Java), multi-platform support, are all needed to develop an environment as described in section 2.1.

Table 1. *List of requirements for the simulation system and description of resulting properties.*

The middle layer of the environment consists of one central persistent **environment controller**, which controls and provides access to all the resources distributed in the environment. A user can interactively conduct a sequence of *activities* involving execution of applications on different platforms or trust this job to the controller. In the latter case, the controller uses a special program called *process* (which prescribes the sequence of activities to be accomplished on behalf of the user). A kind of gateway mechanism is used to provide connectivity to the external resources. It interacts with the user via interactive web pages (if additional input parameters are required to run the resource), controls the session with the real resource, and maintains the communication with the web-user. This mechanism conducts functionality essentially related to that of Object Request Broker, and thus has a very high appeal to apply one of the object technology standards. If the user logs out from the system, the connection between the user and his/her program (if any) is closed, but it does not affect the execution of the program in the system. A user state-saving mechanism is used by the server-side connection both to recognise web-users as they log in the environment, and associate them with their respective processes.

The back-end layer consists of a number of shared **environment resources** (applications) that are accessible to the controller and multiple web-users via, for instance, the Common Gateway Interface. The state of the real-world resources is represented within the controller in a set of proxy objects called *resources*. A *library of proxy modules* provides the gateway mechanism of the controller with standard subroutines that "wrap" the real-world resources. Each proxy wrapper is developed for a particular (often legacy) application and stores the interface definition of that server application and knows where to find the implementation of that resource.

Finally, some kind of an inter-process (not to be confused with user processes!) communication is needed to support communication between the software components of this distributed environment.

2.2. Roles of POKer

We have chosen to base the environment's middle layer on a process-oriented simulator (called POKer) because the authors are charmed by the easiness, simplicity and naturality of this approach, which exactly meets our aims (for more information, please see [9]). In the following we list some major tasks (controller and simulator related) that such a middleware has to handle as a part of the intended environment:

- Resource management (resource integrity and synchronisation)
- Interactive and off-line operation (real-time/as-soon-as-possible time management)
- Mapping the history of the user's activities into a form of a process
- Programming a sequence of activities for automated execution
- Handling external requests from multiple users
- User-state integrity
- Object Request Broker functionality, etc…

This list is by no means a complete enumeration of the requirements. In table 1 these rather low-level requirements are summarised for brevity into more general groups and are translated into the desired properties of the simulator's part of the controller. Due to the unique combination of the wanted properties, existing off-shell simulation languages and packages [4, 14] do not satisfy all our requirements. As the result, we were motivated to develop our custom software system, further on referred to as *environment controller* or simply *controller*.

The core of the controller is a run-time system for a small process-interaction simulation language. The former is referred to as Process Oriented Kernel (POKer) or kernel, and the latter as POKer language. In the rest of the paper, we continue our presentation with the basic semantic definitions of the language constructs (section 3). It is followed by the implementation of POKer (section 4). Section 5 covers semantics and syntax aspects of the language. An overview on execution of an example program is given in section 6. Then, in section 7 we briefly discuss Server POKer, an extension to Poker that constitutes the environment controller. Section 8 concludes our presentation with final remarks.

3. Semantic Background

It is crucial to provide users with the right language constructs. These constructs support the proper modelling framework (often referred to as world-view) and allow users easily construct their programs. Execution of these constructs in a program causes the state of the environment (in our case, synthetic environment) to move (⇒) from `state` to `state'`. The state of the environment is represented by three components <EL, R, D> (for more details, please see [10]). EL is an ordered collection of processes sorted out by the values of their next event times. R is a dictionary of resources (proxy objects that represent the real-world resources). D is a helper component that stores process classes (generic user processes) definitions and class instances, and thus forms a declaration space.

We begin the development of the intended process-interaction kernel by identifying the operational semantics of the kernel commands (language constructs). For that purpose, results provided in the works of Birtwistle and Tofts [2] give us sufficient initial background. In the following, we give descriptions of the currently implemented commands in the functional-language like notation adopted from their work.

3.1. decP(classId,classDef)

Informally, it declares a process class by saving the class definition *classDef* under name *classId*.

Semantics:

```
decP (classId, classDef)
⇒  if classId ∈ D then error      else
    let EL' = current::EL         in
    let D' = D[classId/classDef]  in
        main(EL', R, D')
```

3.2. newP(id, classId, dt)

Informally, it creates an instance named *id* of an existing class *classId*, and schedules this instance as process *dt* time units later of the current simulation time. Also, a copy of the process is saved to the declaration space (D).

Semantics:

```
newP (id, classId, dt)
⇒  if id ∈ D              then error else
    if classId ∉ D        then error else
    if dt < 0             then error else
    let classDef = LOOKUP classId D        in
    let en = (evt+dt,id,classDef,[],data)  in
    let EL' = current::(ENTER en EL)       in
    let D' = D[id/en]                      in
        main(EL', R, D')
```

3.3. hold(dt)

Informally, it schedules the next event of the *current* process by adding *dt* to its '*Next Event Time'* attribute *evt* and places the *current* process on the EL according to the new event time. Alternatively, in a case *dt* is *None*, this command updates the process instance in D and triggers a gateway mechanism to the server-object that is associated with the resource owned by the *current* process.

Semantics:

```
hold(dt)
⇒ if  dt == None
     then
       let D' = D[C/current]                in
         gateway(Attr,data)
         main(EL,R,D')
     else
       if dt < 0  then error else
       let en = (evt+dt,C,Body,Attr,data)  in
       let EL'= ENTER en EL                in
         main(EL',R,D)
```

3.4. newR(id)

Informally, it creates a new resource object under name *id* and initialises it according to the resource capacity. This object is saved to the dictionary of resources (R) and its name to D.

Semantics:

```
newR (id, Cap)
⇒  if id ∈ D     then     error    else
   if Cap < 0    then     error    else
   let EL' = current::EL             in
   let R' = R[id/([],Cap,true)]      in
   let D' = D ++ id                  in
       main(EL', R', D')
```

3.5. getR(id)

Informally, it requests a resource under name *id*. If the resource is free, then it grants ownership to the calling process, and decreases its capacity value by 1. Otherwise, it delays the process.

Semantics:

```
getR (id)
⇒ if id ∈ Attr  then error else
   case LOOKUP id R of
    ([],Cap,true)
    ⇒ let Attr' = Attr ++ id                 in
      let EL'=(evt,C,Body,Attr',data)::EL    in
      let Cap' = Cap - 1                     in
      if  Cap' <= 0
        then let fl = false      in
        else let fl = true       in
      let R'= R[id/([],Cap',fl)]             in
        main(EL', R', D)

   | (Q,Cap,false)
    ⇒ let Q' = Q @ [current]                 in
        let R' = R[id/(Q',Cap,false)]        in
          main(EL, R', D)

   | anything else ⇒ error
```

3.6. putR(id)

Informally, it releases resource *id*. If the resource's waiting queue is empty, then its capacity value is increased by 1. Otherwise, the first delayed process is released by scheduling it at the same time as the *current* process and placing it on the EL after the *current* process.

Semantics:

```
putR (id)
⇒ if id ∉ Attr  then     error    else
   let Attr' = Attr - id                  in
   let EL'= (evt,C, Body,Attr',data)::EL  in
     case LOOKUP id R     of
      ([],0,false) or ([],Cap,true)
       ⇒ let R' = R[id/([],Cap+1,true)]  in
           main(EL', R', D)

   | ( (t1,p1,B1,A1,d1)::Q,  Cap,false)
     ⇒ let R' = R[id/(Q,Cap,false)]      in
       let A1' = A1 ++ id                in
       let en = (evt,p1,B1,A1',d1)       in
       let EL'' = ENTER en EL'           in
        main(EL'', R', D)

   | anything else ⇒ error
```

3.7. close()

Informally, it removes the containing process instance from D. If such instance does not exist in D, then a warning is issued.

Semantics:

```
close()
⇒ let EL' = current::EL   in
  let D' = D - C          in
    main(EL', R, D')
```

Note: `hold()` is intended to be used as the last command in a process to clean up the definition space and to do some other last-minute activities.

3.8. main()

Informally, it continuously executes the processes from the EL, according to their event times. This execution is based on the *next-event approach*: the next event to take place is always the execution of the first command in the body of the current process (current event notice on the EL).

Semantics:

lines

```
   main(EL,R,D)
   ⇒ case (EL,R,D) of
        ([],R,D)
         ⇒ sleep()
05
      | ((evt,C,[],Attr,data)::EL,R,D)
         ⇒ if Attr ≠ [] then error else
             main(EL,R,D)

10    | ((evt,C,(b::Body),Attr,data)::EL,R,D)
         ⇒ let timeout = evt - wallclock()    in
           if  timeout > 0
             then
               let Body'=b::Body              in
15             let en=(evt,C,Body',Attr,data)in
               let EL' = en::EL               in
                 sleep(timeout)
                 main(EL',R,D)
             else
20             case b of
                  decP(classId, classDef)
                | newP(id, classId, dt)
                | hold(dt)
                | newR(id, Cap)
25              | getR(id)
                | putR(id)
                | close()
```

Interpretation:

(*line 03*) If the EL is empty then the kernel goes into a sleep mode.

(*06*) Otherwise, if the body of the first process on the EL is empty, this process is removed.

(*10*) Otherwise, the kernel executes the first command from the body of the first process on EL. It first defines the time left before the next event. For this purpose, `main()` contrasts the current time (`wallclock`) to the simulation time (`time`) of the imminent process (*11*). If the latter is still ahead, the kernel goes into a sleep mode (*14-17*). Otherwise, the kernel processes that event notice (*21-27*).

Note: In sleep mode, the kernel waits until a timeout occurs (if the *timeout* argument is present and not `None`) or until it is notified about any new processes on the EL. Once awakened or timed out, it continues execution.

4. Implementation

4.1. Functional vs. Imperative language

The chosen semantic background of process interaction simulation languages was conveyed in notations adopted from modern functional languages, and therefore it would have been most easy to do the implementation also in one of the pure functional languages. Moreover, an important feature of *pure* functional languages (functional languages with no side-effects) is that they allow developing software, which is more amenable to formal methods and easier to reason about. Using a pure functional language one can make assertions about programs and prove these assertions to be correct. It is possible to do the same for traditional, imperative programs – but just much harder – via "*exhaustive*" testing [19], which may be reassuring but it can never be convincing.

Although the pure functional languages allow to prove correctness of the program, they lack some features that make it possible to write more efficient software, and proof-of-correctness can still be prone to human errors. As the result, our choice for prototyping and development of the simulation language (as well as the proposed environment) goes to an imperative language: the scripting programming language *Python* (see section 4.2). In order to isolate side-effects of the program's parts from being visible to the rest of the program we are 'encapsulating' those parts from changing the global state of the program in a way similar to [15], and ensure that the parts of our imperative code have the same input-output behaviour as their functional versions.

Programming and reasoning about run-time systems of discrete-event simulation languages is difficult, because their operation involves dynamically changing scenarios. The approach described above and a good understanding of operational semantics can facilitate modular prototyping of the system in development. Given the interactivity of scripting languages, debugging and testing smaller modular parts of code for correctness can be even further facilitated.

4.2. About Python

Python [11] is a modern language, which combines the usability of traditional scripting languages (such as Tcl, Scheme, and Perl) and the power of advanced programming tools typically found in systems development languages (such as C/C++, Java). It is an interpreted, object-oriented, high-level programming language with clear syntax. Its high-level built-in data structures, combined with dynamic typing and dynamic binding, and easiness of integration into C/C++ systems (and visa-versa), make it very attractive for *rapid application development*, as well as for use as a scripting or glue language to connect existing components together.

The Python interpreter, extensive standard and 3rd party libraries (CGI, Sockets, interfaces to distributed object architectures, etc.) are available in source or binary form without charge for all major platforms, and can be freely distributed [13]. Moreover, Python's de-facto standard GUI package *Tkinter* [8] is based on *Tcl/Tk*, a graphical user interface toolkit that makes it possible to create powerful GUIs [12]. Among other Python's GUIs, Tkinter is the most commonly used one, and is almost the only one that is portable between Unix, Mac and Windows.

The combination of the above-mentioned aspects and features makes Python extremely suitable for prototyping in software development projects.

4.3. POKer

Due to the Python's mature built-in object types, such as lists, dictionaries, tuples and the availability of powerful operations to process them, the implementation process was quite straightforward. Moreover, there was no need (at least at this stage) to extend Python's built-in object types (e.g., by embedding them into 'wrap' classes) in order to implement the operational semantics of the language constructs and components (EL, R, D). The following is a summary of some aspects intrinsic to the given implementation.

Tuples are written as series of arbitrary objects in parentheses (any further occurrences of such syntax in the code imply tuples). Because tuples are immutable, they provide some integrity and therefore are well suited to implement POKer

pDemos	*POKer*	*Remarks*
`decP(classId, classDef)`	`(decP,(classId, classDef))`	
`newP(id, classId, dt)`	`(newP,(id, classId, dt))`	
`hold(dt)`	`(hold,(dt,))`	`Additional argument value: None`
`newR(id)`	`(newR,(id, Cap))`	`Capacity is added`
`getR(id)`	`(getR,(id,))`	
`putR(id)`	`(putR,(id,))`	
`close()`	`(close,())`	

Table 2. *pDemos and POKer commands.*

objects with defined structure (e.g., processes, resources, etc.). Another feature is that argument lists in Python are also constructed as tuples.

Sets (dictionaries) are written as series of `key:value` pairs, separated by commas and enclosed in curly brackets. Set processing can be efficiently implemented with the built-in Python operations.

Lists are written as a series of objects, separated by commas and enclosed in square brackets. Python's list object has already sufficient built-in operations, such as slicing, concatenation, etc.

POKer's commands were implemented as modular referentially transparent subroutines that exhibit explicit input-output behaviour and are based on the respective operational semantics definitions. As the result, it is much easier to verify their correctness by interactively testing those subroutines in the command line of the Python interpreter. The effect of such interactive execution on the state of the system is immediately obvious and can be easily compared to the state, determined according to the operational semantics.

The `main` routine is framed so that it executes the first command of the current (first) process on the EL by calling the respective subroutine. This is accomplished with the Python built-in function `apply()`, by passing the tuple of the subroutine as argument along with the tuple of the subroutine's arguments:

example 1: `apply(`*subroutine*`, (`*arg1, arg2, ...*`))`

This approach allows executing commands in a generic fashion, without knowing their names and arguments ahead of time.

All command subroutines and `main()` are packed into a separate module called `poker`. Besides the aforementioned subroutines, this module also contains three more objects:

`current()`
: this class hides some supporting code (related to such functionality as handling input/output channels, messages, etc.) that is irrelevant to the implementation of the language constructs. Users are free to provide their own implementation of `current`.

`enter()`
: this helper function assists entering event notices on the `EL`

`gateway()`
: this subroutine provides connectivity to the environment resources. It looks-up the proper wrapper from the library of proxy modules and executes it in a separate thread or process (in the OS sense), otherwise the overhead associated with that execution would affect the integrity of the kernel's timing mechanism. The above-mentioned library consists of wrappers developed for particular resources inherent to the particular environment at hands. Therefore, users have to provide their own libraries of wrappers for applications that are available in the environment at hand.

5. Syntax and semantics

The definition of a particular language consists of both syntax (how the various symbols of the language may be combined) and semantics (the meaning of the language constructs). Section 3 has already provided a sufficient coverage of the semantics. As for the former, it shares a lot with the original syntax of πDemos, a language demonstrated by Birtwistle and Tofts, and we direct the readers looking for a more detailed description to their work [2]. For our purposes, it would suffice to point out only the differences introduced by our implementation.

Commands

As the consequence of using `apply` (see example 1), POKer's commands should be now represented as compound objects (tuples), and therefore are subject to the native Python syntax rules for such objects [16]. A summary of the syntax changes in POKer commands as contrasted against πDemos commands is given in table 2.

Processes

In contrast to the process structure proposed by Birtwistle and Tofts (example 2), the nesting introduced by a compound object `PD` is removed, and a new object (attribute `data`) is added to support data-flow of a process. This attribute holds a reference to any intermediate data that is currently associated with the process (for example, the address of the file with output data produced by the previous activity with an environment resource). Additionally, the order of objects in the process is changed (example 1):

example 2: `(id, PD(classDef, attr, evt))`
example 3: `(evt, id, classDef, attr, data)`

6. Example of a sample POKer program

In this section we present an example of a POKer program to demonstrate the usage of the semantics rules and step by step execution of such a program. This program creates a generic process class `PD` and three instances (`P1, P2, P3`) of that class are scheduled on the `EL` at simulation times `1, 2, 3` respectively. Each of the processes acquires, uses, and releases the same resource (`B`) with capacity `2`. For simplicity, the processes simulate activities with the resource (gateway mechanism is not used) by `hold(3)`.

The state of the system is:

```
EL = [(0,'P0',PD0,[],[])],
R = {'B':([],2,'true')},
D = {},
```

where:

```
PD0 = [ (decP, ('PD',[(getR,('B',)),
                      (hold,(3,)),
                      (putR,('B',)),
                      (close,())])),
        (newP, ('P1','PD',1)),
        (newP, ('P2','PD',2)),
        (newP, ('P3','PD',3)),
        (close,()) ]
```

The execution begins with process `P0` declaring a process body `PD` as a class (see table 3). Next is to create instances of processes and schedule them after all processes already existing on the `EL` (*lines 2-4*). Finally `P0` calls `close()`. Execution of `P0` unrolls in a consecutive manner, for it does not include tasks that would either reschedule (`hold`) or possibly delay (`getR`) the process. After termination of `P0`, `EL` contains three processes `P1, P2, P3`. Notice, that `P1` and `P2` both were allowed to use resource concurrently (*6 and 8*) while `P3` became blocked (*line 10*) and how the state of the resource is affected. This demonstrates the effect of capacity being added to the resource. For the rest execution of the processes is similar and obvious.

7. Server POKer

7.1. Extending POKer

The application of POKer is intended to operate in the environment as controller in a way it is described in subsection 2.1. In order to enable POKer for such an undertaking, it has to be amended with additional functionality. This is accomplished in a special version of POKer, called Server POKer. This advanced version of POKer is based on the original POKer and extends its functionality by wrapping it in a class.

This class does not affect the state of the system, nor provides any direct functionality to the controlling of the environment. Its development and implementation was not based on formal operational semantics, but rather on usual programming practice, and available standard and 3rd party libraries with the needed functionality. As such, it is more of a helper class that bridges the gap between the controller's core (language run-time system) and the rest of the environment components it has to interact with. This class is packed into a separate Python module named `ServerPoker`.

Lines	*Run-time messages*			
01	(0,P0,[])	Declares class PD	==> done	
02	(0,P0,[])	Creates instance of PD	==> (1,P1,[])	entered EL
03	(0,P0,[])	Creates instance of PD	==> (2,P2,[])	entered EL
04	(0,P0,[])	Creates instance of PD	==> (3,P3,[])	entered EL
05	(0,P0,[])	Closes itself	==> (0,P0,[])	is removed from EL
06	(1,P1,[])	Gets (B,2,T)	==> (1,P1,B)	owns (B,1,T)
07	(1,P1,[B])	waits for 3	==> (4,P1,B)	is rescheduled
08	(2,P2,[])	gets (B,1,T)	==> (2,P2,B)	owns (B,0,F)
09	(2,P2,[B])	waits for 3	==> (5,P2,B)	
10	(3,P3,[])	gets (B,0,F)	==> (N,P3,[])	is blocked by (B,0,F)
11	(4,P1,[B])	puts (B,0,F)	==> (4,P1,[])	(B,0,F) freed (4,P3,[B])
12	(4,P1,[])	closes itself	==> (4,P1,[])	is removed from EL
13	(4,P3,[B])	waits for 3	==> (7,P3,B)	
14	(5,P2,[B])	puts (B,0,F)	==> (5,P2,[])	freed (B,1,T)
15	(5,P2,[])	closes itself	==> (5,P2,[])	is removed from EL
16	(7,P3,[B])	puts (B,1,T)	==> (7,P3,[])	freed (B,2,T)
17	(7,P3,[])	closes itself	==> (7,P3,[])	is removed from EL
18	POKer: empty EL – simulation is complete			

Table 3. *Run-time messages generated by the kernel.*

As the result, Server POKer (from the implementation viewpoint) is a combination of two modules: `poker`, which implements the language run-time system, and `ServerPoker`, which uses `poker` and extends it with "environment"-oriented functionality. A detailed discussion of Server POKer is beyond the scope of this paper. In following subsection 7.2 we give a brief description of the `ServerPoker` module.

7.2. Description of 'ServerPoker' module

The `ServerPoker` module consists of one class `ServerPoker`, which simplifies the task of setting up the environment controller (or server, as it is referred to in this subsection to stress its serving role). Creating the server requires several steps. First of all, one must instantiate the `ServerPoker` class, passing it the `server_address`, `state_filename` and starting up the `listener` and `kernel` threads. Then, call the `handle_request()` or `serve_forever()` method of the server object to process one or many user requests. The following is a description of the `ServerPoker` class external methods:

`servelet`(*conn, addr*)
: Process data from a socket-client (`conn, addr`). It receives the request, gets, processes, and puts the data on the internal buffer.

`server_address`
: The address on which the server is listening. The format of it is a tuple containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

`setup`(*state_filename, server_address*)
: Re-initialise the `ServerPoker` class.

`start_listener`()
: Start a Socket Server. It starts up a Socket Server by creating a `listener` thread that 'listens' for socket connections at the given `server_address`.

`state_filename`
: Full filename. This file contains the state (saved or initial) of the environment.

`terminate_listener`()
: Terminate Socket Server. It shuts down the socket binding and terminates normally the `listener` thread.

`handle_request`()
: Process a single event notice. The kernel's loop iterates ones to process the imminent event from the current process.

`POKer`()
: Process a single POKer command. This is intended to be an interactive POKer shell.

`save_state`(*state_filename*)
: Save the state of the kernel to the *filename* file

`serve_forever`()
: Handle an infinite number of events. It simply calls method `handle_request()` in an infinite loop. In addition, this subroutine complements the *timing mechanism* of POKer: whenever the POKer becomes idle, it would force the `kernel` thread into a sleep mode until it becomes timed out or awakened. The latter is caused by arrival of external events signalling that the wall-clock time has reaches the next event time (`time`) or availability of new processes on the internal buffer.

`server_activate`()
: Activate POKer as server. It starts up the `kernel` thread that executes processes from the `EL` and looks-up the internal buffer for any new requests from users.

`server_deactivate`([*save=1, state_filename*])
: Deactivate POKer. It handles the normal termination of the `kernel` thread and calls the following methods in order: `terminate_listener()`, and (optionally) `save_state()`.

8. Final remarks

Obviously, the ideas behind POKer are not at all new but based on the research reported in [Birtwistle and Tofts 1994]. Novel are, however, our implementation in the scripting language Python and the use of the resulting process-oriented discrete-event kernel in the real-world management and control of our collaborative environment in development. We have chosen for a simulator as the core of the collaborative environment in order to have all the time full overview on all that is happening in the environment. This should facilitate later on formal analysis of the environment, resource management, virtual product life cycle [17], etc.

In this paper we have presented the operational semantics definition of the process-interaction language commands and the implementation of the run-time system. Despite the fact, that conversion of the semantics into an imperative language was not as straightforward as it would have been in the case of a functional one, the implementation benefited from the available strong data-types and interactivity of the scripting language. Further, understanding the semantics of the execution of POKer programs

helped us when proving the correctness of the implementation by testing. In summary, having a good semantic background provided us with a clear, short and unambiguous understanding of the language constructs, and thus facilitated the implementation and testing phase. Later on it might facilitate reasoning about the properties of POKer, as new features are being added.

All the run-time traces provided in this paper were generated by the presented kernel. An on-line demonstration of POKer is available from the Internet [5]. There is also another on-line demo [6] that shows the ability of Server POKer to handle processes from multiple web-users on the same set of shared resources. However, for that purpose the server must be started up before one can take part in a demonstration. Should a reader be interested, please send your request via e-mail to the first author of this paper.

Acknowledgement

The research reported in this article is done in the framework of the NanoComp-project [7], sponsored by the TU-Delft.

References

[1] O. Balci, *The implementation of four conceptual frameworks for simulation modeling in high-level languages*. Proceedings of the 1988 Winter Simulation Conference, pp. 287 – 295.

[2] G. Birtwistle and C. Tofts, "An operational semantics of process-oriented simulation languages: Part 1 πDemos", Trans. Soc. Comput. Simul., 10(4), December 1994, pp. 299 – 333.

[3] A.C.-H. Chow, Parallel DEVS: A parallel, hierarchical, modular modelling formalism and its distributed simulator. TRANSACTIONS of the Society for Computer Simulation International, Volume 13, No. 2, 1996, pp. 55 – 67.

[4] Yoke-Hean Low, Chu-Cheow Lim, Wentong Cai, Shell-Ying Huang, Wen-Jing Hsu, Sanjay Jain, and Stephen J. Turner, "Survey of Languages and Runtime Libraries for Parallel Discrete-Event Simulation", Simulation 72:3, 1999, pp. 170 – 186.

[5] On-line demo of POKer: http://www.kbs.twi.tudelft.nl/Research/Projects/CollSim/Software/bin/PI/

[6] On-line demo of Server POKer: http://www.kbs.twi.tudelft.nl/Research/Projects/CollSim/Software/bin/index2.cgi

[7] NanoComp project homepage: http://nanocom.et.tudelft.nl/

[8] Tkinter homepage: http://www.python.org/topics/tkinter/

[9] A. Levytskyy and E.J.H. Kerckhoffs, "Towards a Prototype Web-Based Collaborative Simulation Environment", SCS: paper of the 5th Euromedia Conference, May 2000, pp. 60 – 66.

[10] A. Levytskyy and E.J.H. Kerckhoffs, "A Plain Python Simulator to Control a Collaborative Environment", SCS: paper of the 14th European Simulation Multi-conference (ESM), May 2000, pp. 719 – 727.

[11] Mark Lutz, "Programming Python", O'Reilly & Associates, Inc., October 1996.

[12] John K. Ousterhout, "Tcl and the Tk Toolkit", Addison-Wesley Publishing Co., 1996.

[13] Python homepage: http://www.python.org

[14] Jerry Banks (Ed.), "Handbook Of Simulation. Principles, Methodology, Advances, Applications, and Practice", A Wiley-Interscience Publication, 1998, pp. 813 – 833.

[15] Jon G. Riecke and Ramesh Viswanathan, "Isolating Side Effects in Sequential Languages", Papers of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1995, pp. 1 – 12.

[16] Guido van Rossum, Python Tutorial, Release 1.5.2, March 2000: http://www.python.org/doc/current/tut/tut.html

[17] Vangheluwe and Vansteenkiste, Ghislain and Visipkov, Vladimir and Merkuryev, Yuri and Merkuryeva, Galina and Teilans, Artis, "Design of a User Friendly Modelling and Simulation Environment", International European Simulation Multi-conference (ESM), June 1994, pp. 282 – 286.

[18] B.P. Zeigler, H. Praehofer, T.G. Kim, Theory of Modeling and Simulation. Second Edition. Integrating Discrete Event and Continuous Complex Dynamic Systems, Academic Press, 2000.

[19] Ronald C. Van Wagenen, Charles R. Harrell, Achieving Reliability in Simulation Software. Proceedings of the 1994 Winter Simulation Conference, pp. 695 – 699.

[20] J.S. Carson, Modeling And Simulation Worldviews. Proceedings of the 1993 Winter Simulation Conference, pp. 18 – 23.