ETOS — DISCRETE EVENT SIMULATION FRAMEWORK FOCUSED ON EASIER TEAM COOPERATION*

Jiří Fišer, Jiří Škvára

Faculty of Science, J.E. Purkinje University in Usti nad Labem
400 96 Usti nad Labem, Czech Republic

(Received March 25, 2014)

The leading tool for event based discrete simulation in Python programming language is SimPy. The SimPy supports all simulation primitives and efficiently utilizes high level Python constructs. Unfortunately, SimPy's simulation processes have to be implemented as one coroutine *i.e.* in a single code unit. This solution is sufficient for simple simulations but totally inappropriate for large teams and more complex problems. Our SimPy extension ETOS makes possible separation of roles in simulation teams and simplify description of simulation. The simulation is represented as a group of XML nodes and is built from simpler Python constructs — entities. The entity is implemented as a Python class using SimPy primitives and specialized support of value context (*i.e.* random and time-dependant values relative to simulation, a process or to entity processing). The entities provide elementary control flow constructs, versatile simulation objects and elementary operations of simulated models. The entities are also responsible for a data collecting. The paper describes hierarchy of basic entities of ETOS model and it also covers case study of framework including annotated Python and XML source codes.

DOI:10.5506/APhysPolBSupp.7.271
PACS numbers: 07.05.Tp, 89.20.Ff, 07.05.Bx

1. Introduction

Two years ago, we have started participating in a project which deals with simulation of e-cars traffic. Therefore, we prepared several event-based discrete simulations [1].

These simulations were focused on behavior of e-car in relatively long term scale (weeks and more) and the main goal is an optimization of e-car external facilities (especially charging station) in the world of resource

* Presented at the Summer Solstice 2013 International Conference on Discrete Models of Complex Systems, Warszawa, Poland, June 27–29, 2013.

competition. We have preferred event based systems which are based on a program code (our students are primarily programmers), high level approach (our ideal is a domain specific language which are both specialized and flexible), and open source implementation (the open source software is not only free of charge but also it uses open and documented code). Existing traffic simulation systems do not meet these requirements (especially *ad hoc* systems implemented on very low level in C++ or Java) or complex products with huge libraries (but not with e-cas oriented support).

Finally, we have chosen SimPy as the simulation framework because Python language (the foundation of SimPy) is part of our bachelor curriculum of informatics and Python is relatively popular among our students and teachers. We also use Python and its mathematical packages NumPy and matplotlib in our mathematical courses as an alternative to commercial products for numerical analysis.

The SimPy itself has relatively clear design and its complexity is reasonable for our students. It supports all typical components of discrete-event simulation as queues, shared resources or signals. The SimPy also shares process oriented approach with other modern simulation frameworks. Last but not least, SimPy is open source project and its documentation is (for relatively small open-source project) excellent [2] (basic ideas of Simpy are presented also in [3]).

Unfortunately, the design of SimPy processes makes a cooperation on larger projects almost impossible (even on the scale of several students and teachers). SimPy processes (coroutines) are based on Python generators [4], which are primarily designed as iterator providers. The generators are not based on low-level continuation-like primitives (as, for example, continulets of PyPy [5]) and, therefore, they do not support an unrestricted dispatching among coroutines. The main dispatcher (denoted as trampoline) has to be located on top of all active coroutines and every dispatch request of coroutine at any level must be propagated through all parent coroutines to the trampoline by process which is denoted as re-yielding.

The main implication is obvious — the nested coroutines are supported only by mean of complex support code which iterates over sub-generators and passes flow of control to the calling coroutine. This supporting code has to handle all situation including several types of exceptions. Unfortunately, some exception in iterators are not exceptional but they arises during normal processing, *e.g. StopIteration* signalizing iterator's exhausting or *GeneratorExit* thrown after a leaving of nested iterators. Therefore this code, which is complex and fragile, has to be (manually) repeated in any coroutine except leaf ones. This situation is slightly improved in Python 3.3 with *yield-from* constructs but the main constraints remain.

Therefore, the code of a SimPy process is often packed in one huge coroutine. This has a lot of negative implications:

— developers share monolithic codes *i.e.* their cooperation is very difficult and a separation of developer's roles is almost unfeasible;

— the coroutine code is not robust because some recurring SimPy implementation patterns are both extremely error-prone and non-reusable (*e.g.* reneging);

— the code is not extensible or stackable;

— the behavior of exceptions (including user defined) in SimPy is often undefined or at least esoteric.

2. ETOS — design principles

We have considered two possible solutions: using a more abstract formalism together with a high level framework (if possible, in Python) or design and implementation of a new framework. The most perspective candidate was Discrete Event System Specification (DEVS) [6], *de facto* standard formal representation of discrete event simulation models (even with Python implementation — DEVSimPy). Unfortunately, this formalism is relatively complex and very different from SimPy. DEVSimPy also uses graphics user interface, while we prefer textual representation by domain specific language. Moreover, the documentation of DEVSimPy is almost non-existent.

Finally, we have chosen implementation of new framework on top of SimPy that utilizes our know-how and makes possible early involvement of our students.

The main design objective of our framework is separation of simulation code to two basic levels of abstraction:

— **Declarative description** of parameters and behavior of simulated objects by high level XML-like structured language;

— **Procedural representation** of basic actions by extended SimPy (that is Python) code in the form of generator coroutines.

The framework should prefer declarative descriptions over a low level SimPy code. The procedural code should be limited to concise, simple and clear (co)routines which hide details of coroutine dispatching (even the yielding constructs have to be simplified and clarified). This also makes possible to reuse a repetitive code by means of an object oriented inheritance or even by a metaprogramming.

The efficiency of the framework strongly depends on efficiency of SimPy. Unfortunately, Python and especially SimPy are not systems focused on time or memory efficiency [7]. Basic Python objects are represented by tens or hundreds bytes, object creations need milliseconds and coroutine yieldings are also relatively slow. Therefore, the framework should optimize object creation by re-usability of auxiliary objects (only one instance of auxiliary object per simulation) and minimize slow operations (especially multiple re-yieldings, creations of coroutines).

The framework is designed with three main parts (see Fig. 1). The core part is a transaction engine which extends SimPy processes (based on Python generators). The simulation process (transaction in ETOS terminology) is controlled by a declarative program that is read from XML documents by parsers. The declarative program utilizes entities (*i.e.* instances of Python classes which represent basic simulation activities, control flow of transactions or they collect outputs of a simulation).

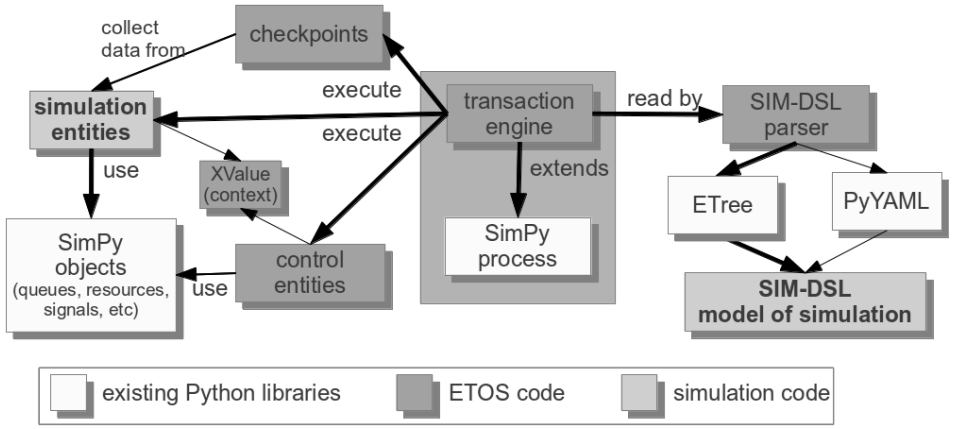

Fig. 1. Structure of ETOS framework.

The declarative notation is based on a more abstract domain specific language (denoted as SIM-DSL). This notation unifies structured simulation data with high-level transaction code. This unification is similar to Lisp programming language with its homoiconicity. The SIM-DSL should support hierarchical trees of nodes together with their complex attributes and extended values including random numbers with specific probability distribution and simple functions of simulation time. The SIM-DSL is representable by XML or YAML (structural language with more compact syntax than XML [8]).

The ETOS framework offers five concepts for end users (note: the names of this concepts slightly differ from classical simulation nomenclature [9]).

The top level concept is **simulation** which represents a global state of simulated system, including shared resources or data collectors. The simulation is represented by set of interlinked SIM-DSL documents and by instance of class *Etos.Simulation* (derived from class *SimPy.Simulation*) in Python code. ETOS supports multiple simulation instances that can be executed in multiple Python threads or processes (*e.g.* in simple cluster based on Python multiprocessing package [10]). We tested this approach in a small *ad hoc* cluster of tens student notebooks connected by Wi-Fi network (alternative approach see in [11]).

The second concept is **actor**. Actor represents a simulated object during its lifetime (which can exceed the lifetime of process, *i.e.* actor may be carried out by several processes). The simplest actors are only containers of attributes and, therefore, they are representable only by a declarative code in SIM-DSL. More sophisticated actors are implementable by Python's objects.

The **transaction** is actor's carrier *i.e.* transaction simulates its lifetime (or part of its lifetime). The transaction is represented by a cooperating system of declarative (SIM-DSL) and procedural (Python) code. The declarative code describes the high-level flow of control, the procedural one defines the basic steps of simulations. The transactions in EOS can be nested. However, sub-transactions are only instruments for mapping one logical task to several SimPy processes *i.e.* to several generator coroutines.

The **entities** are basic buildings blocks of ETOS program forming individual executing steps in transactions. ETOS provides four types of entities:

1. simple activity of modeled object (end user entities),
2. entity which provides interface to shared SimPy services, *e.g.* shared limited resources with waiting queues (parking, fuel stations),
3. control entities *e.g.* branching, loops, try blocks, *etc.*,
4. subtransactions.

The shared service entity manages shared objects which are referred from two or more entities. This makes possible cooperation or competition of several transaction on limited resources. The shared objects are identified by common identifier of entity.

The extended values used by entities (in roles of parameters or states) are encapsulated in special objects (we called them *x-values*). The main characteristics of any x-value is its context. Random or time dependent values of x-values are always related to explicit context. The ETOS uses four types of contexts:

— *simulation context*
Time is relative to global simulation time, random values are generated only once per simulation.

— *actor's context*
Epoch of time is starting by the first transaction of this actor and random value is generated only once per actor.

— *transaction context*
This auxiliary context is suitable for system with simple actors (1:1 mapping between actor and transaction). Only top level transactions provide this context.

— *entity context*
Context of every instance of an action (*e.g.* new context in every iteration of action in a loop).

3. Simplified example

The ETOS as an event-based system is suitable especially for simulation of dynamic system with stochastic behavior (especially with randomly distributed time intervals) and with shared and limited resources (completion for limited resources), *e.g.* queuing theory problems. We have used discrete event-based simulations also for discrete-time Markov chains (results have been presented in [12]) and Markov chains are representable by ETOS using basic entities (*e.g. withProbability* constructs).

Currently, we are exploring potential of ETOS for simulation of multi-agent system especially for process of formation of agent's coalition (cooperating with Mashkov [13]). In this simulation, agents are represented by ETOS actors (only part of agent real power should be simulated by ETOS — timing and communication protocol during coalition formation).

Real multi-agent systems are (almost) strictly distributed system *i.e.* there are not shared data in the system. Therefore, the natural representation of a multi-agent system does not contains shared (resource) objects and the agents are represented by actors (and their transactions) which have to communicate only by message passing. This model is supported by SimPy (by communication via signals) and it is representable in ETOS (by two additional entities and by embedded mechanism of transaction identification).

However, this model makes simulation of more complex multi-agent systems very extensive and almost unmanageable (the process of successful coalition formation requires hundreds of messages). Fortunately, some central agents (managers) can be transformed to (complex) shared entities

which (temporally) join several agents and they support inter-agent communication (directly because shared entities hold references to their temporary clients).

Unfortunately, the standard shared entities of event-based discrete systems (resources, storages, and containers) in SimPy support only competitive clients. On the other hand, the formation of coalitions in MAS is predominantly a cooperative task and the design and implementation of a whole new category of entities is necessary.

In the following example, we present simplified multiagent system, with several agents and with one auxiliary shared entity — "barrier" in which agent transactions wait for a formation attempt *i.e.* an attempt to form new coalition (analogy of *synchronization barrier*). In this simplified example, the attempt is the accomplishment of a certain number of waiting agents (specified by per shared object basis). The lifetime of new coalition is specified by common (typically random) value (in more realistic scenarios, the success and lifetime have to depend at least on a common goal and agent's predispositions).

SIM-XML code of simulation

```
<simulation>
<main_transaction>
<counted_loop count="100" sim:shared_objects="#shared">
   <start_transaction> <!--start of independent transaction-->
     <pause duration="logarithmic: {lambda : 60}">
     <infinity_loop restartBy="fail">
        <infinity_loop>
          <barrier id="main"/>
          <pause duration="a.taskLifeTime"/>
          <with probability="0.01">
            <!--repair-->
            <pause duration="normal:{mu:'8m30s' sigma:'2m'}"/>
            <exception type="fail"/>
           </with>
        </infinity_loop>
     </infinity_loop>
   </start_transaction>
</counted_loop>
<main_transaction>
<shared>
  <barrier id="main" size="10"/>
</shared>
</simulation>
```

The initial code is concentrated in *main_ transaction* element (names of transaction elements are arbitrary). *Start_ transaction* entity inside counted loop creates 100 agents transactions, which are controlled by embedded SIM-DSL code. Agent's life begins by random pause and continues in inner infinite loop with two actions: agents enter barrier (waiting room) and after coalition formation they process a task (represented by pause). This simple behavior is slightly complicated by simulation of agent's failures. With probability of 1% the inner loop is interrupted but after some repair period the agent is restarted (by outer infinite loop).

The implementation of the barrier in Python is not trivial because it does not use built-in resource managers and, therefore, it must synchronize agents by signals. This fragile and error prone code has to be placed in each auxiliary entity. We plan to move this code into lower levels of the ETOS implementation by extension of yielding messages.

We are investigating two solutions:

1. Extension of SimPy engine. SimPy is an open source project but its internals are underdocumented. Furthermore, the SimPy is a living project and, in some versions, the internals have been completely rewritten from scratch (including last major version from October 2013, we are in process of adapting of ETOS for this version).

2. An auxiliary adapter between the SimPy engine (trampoline) and ETOS code, which translate extended messages to SimPy counterparts (using especially sequences of SimPy signals). This solution has been partially implemented for some simple shared entities but long term maintainability requires a careful design of minimal set of elementary messages supporting all agent's demands (*e.g.* client entry points, awakening of set of transactions *etc.*).

4. ETOS in practice

The ETOS has been used for several simulation of e-car traffic. For this purpose, we have implemented library of specialized simulation entities (e-cars, refuel stations with typical recharging characteristics, *etc.*). The simulations were executed in clusters of notebooks and the results (collected by checkpoints and gathered by parallel map construct provided by *multiprocessing* library). The data processing has been implemented by NumPy framework and visualization has been supported by matplotlib library (2D graphing library for Python). The matplotlib supports the creation of animated objects and, therefore, it is possible to animate a simulation or a gathering process.

For example, Fig. 2 is a contour diagram of results of one from our simulation. The actor of this simulation is an e-car which on daily basis recharges at home (low-voltage charging) or in a shopping center (high-voltage charging).

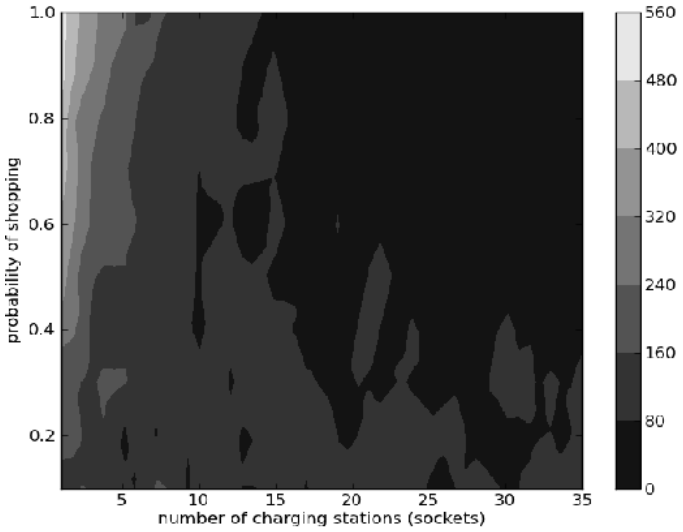

Fig. 2. Sample of simulation results.

The simulation is parameterized by number of charging stations (sockets) in a shopping center parking place (x-axis) and with probability of a shop visit (y-axis). The result quantity is a number of out-of-charge events (the event is relatively rare — the 20000 car-days are simulated for every combination of values).

The graph depicts two types of dependency. The first dependency (very strong) is visible horizontally (especially on the top). The insufficient supply of charging sockets significantly increases the probability of out-of-charge event. The second result is more obscure, the higher probability of home charging (which is more limited by circuit breakers) slightly increases the probability of out-of-charge event (vertical transition on the right-hand side).

The separation of descriptive and procedural part of simulation and the mapping of actions to several entity classes has made possible separation of role in our simulations teams. Figure 3 illustrates estimated roles (in our small teams some roles were shared).

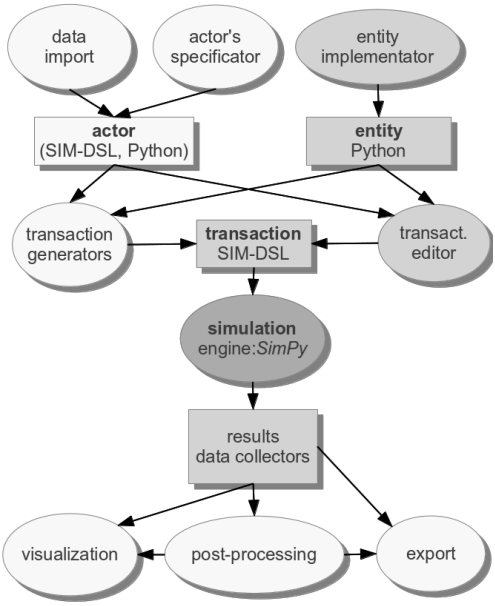

Fig. 3. Team roles.

5. Conclusions

The ETOS framework is limited in this phase of development to relatively simple simulations. The reason is obvious: the relatively small quantity of more complex entities (the development in the first phase has been focused on control and auxiliary entities). The most elaborated end-user entities encapsulate SimPy resources (including reneging and management of resource queues) but there are a few of entities modeling more complex physical processes or even more general actions.

Therefore, we are going to implement library for simulation based on directed graphs. This library provides simple but powerful representation of graphs (with support of weighted nodes and edges) and implements basic graph algorithms. This library should be integrated into ETOS by two mechanisms:

— SIM-DSL representation of weighted directed or undirected graphs,

— simulation entities, which use graph for representation of input data and graph algorithm for implementation of an action.

The design of ETOS optimizes memory and time usage but the optimization of re-yielding is crucial because the every nested entity (*e.g.* loop, branching, *etc.*) adds one level of indirect transfer of control to process of

coroutine switching. The optimization has to find an equilibrium between depth of re-yielding and number of instantiated sub-transactions. The ETOS does not support an automatic optimization and manual one (here, the inclusion of body of loop into new transaction element) is rather tricky, because the ETOS does not provide a benchmarking information. We plan to formalize the impact of re-yieldings and instantiations of sub-transactions and to implement an automatic refactoring of code.

This work was supported by grant from company *Severočeské doly a.s. Chomutov* (`http://www.sdas.cz`).

REFERENCES

[1] J. Banks, *The Future of Simulation*, Information Technology for Engineering & Manufacturing, 2000.

[2] K. Mueller, T. Vignaux, O. Luensdorf, S. Scherfke, Documentation for SimPy Version 2.3b1., `http://simpy.readthedocs.org/en/latest/api_reference/index.html`, December 2013.

[3] K. Mueller, T. Vignaux, SimPy: Simulating Systems in Python, `ONLamp.com`, Python DevCenter, 2003.

[4] G. Van Rossum, P.J. Eby, PEP 342 — Coroutines via Enhanced Generators, `http://www.python.org/dev/peps/pep-0342/`, May 10, 2005.

[5] PyPy 2.1.0 Documentation. Application-level Stackless Features, The PyPy Project, `http://doc.pypy.org/en/latest/stackless.html`, December 2013.

[6] B.P. Zeigler, *Proceedings of the IEEE* **77**, 72 (1989).

[7] E. Weingartner, H. Vom Lehn, K. Wehrle, A Performance Comparison of Recent Network Simulators, IEEE International Conference on Communications, 2009, pp. 1–5.

[8] O. Ben-Kiki, C. Evans, B. Ingerson, YAML Ain't Markup Language (YAML)$^{(tm)}$, Version 1.1, Working Draft 2008-05, 11, 2001.

[9] J. Banks, Discrete-event System Simulation, Pearson Prentice Hall, 2005, ISBN 9780131446793.

[10] Documentation of the Python Standard Library, Multiprocessing — Process-based Parallelism, Python Software Foundation, `http://docs.python.org/3/library/multiprocessing.html`, August 17, 2013.

[11] V. Castillo, Parallel Simulations of Manufacturing Processing Using Simpy, a Python-based Discrete Event Simulation Tool, Proceedings of the Winter Simulation Conference, 2006, p. 2294.

[12] V. Mashkov, J. Fišer, Generic Model for Application of Probabilistic Algorithms for System Self-diagnosis, Proceedings of ISDMCI2010 International Conference, Ukraine, 2010, pp. 215–218.

[13] V. Mashkov, J. Fišer, *J. Appl. Comput. Sci.* **18**, 19 (2010).