InWest – section E-mobility

Report 2012-05

group: simulation of traffic of e-cars
coordinator: Jiří Fišer

Jiří Fišer, KI PřF UJEP
ithil@jf.cz

May 2, 2012

1. phase (February - March, finished)

Searching for existing tool for process-based discrete-event simulation (for Python 3).

1.1. requirements

- object oriented design and interface
- simulation based on coroutines (generators)
- support of exceptional events

1.2. solution

SimPy i.e. object-oriented, process-based discrete-event simulation language for Python [1] (Python 3 supported from version 2.3, 25 December 2011). The *SimPy* have been tested in small simulation and taught in three InWest seminars.

1.3. problems

1. complicated (=tricky) factoring out of code of PEM (*Process Execution Method*, see [1]) methods (mechanism *yield from* – planed for Python 3.3 [2] – is only partial solution)
2. obfuscated and error-prone expressions, which have to utilize values, which are randomized (random variable) or values which are function of time (relative time of simulation, process or activity)
3. weak separation of roles (especially role of processes and actors)
4. the simulation process is coded in undeclarative and hard-wired fashion (procedural code intermixed with *yield* statements)

2. phase (April, finished)

Design and initial implementation of high-level simulation library, which uses SimPy as low-level engine (but is usable in all coroutine based system)

2.1. requirements

- XML declarative specification of model data (including random variables)
- strict separation of roles (transaction, action, actor)

- simple model of unified access to values (including random variables, and function of simulation time) bound to context of simulation entities
- safe implementations of reentrant code in environment of cooperative multithreading (coroutines [3]), partially also in real parallel environment [4] (parallel run of several simulations with shared XML declarative base)
- reduction of number of created objects in simulations step (reusing of existing object from pools)

2.2. solution

New Python 3 library with name ETOS (*Entity based Transactions Of Simulation*).

The library supports four main classes of simulations objects:

Entity is parametrized elementary action of a simulation transactions. Entities cover (1) per-transaction actions, which don't require cooperation with other transactions (e.g. waitings depended only on absolute time) and (2) transaction cooperated with others transactions on limited resources (e.g. refuel stations). The entity must be identifiable by unambiguous identifier (crucial for cooperated [shared] entities).
The user code of entities (i.e. overriding methods) is divided into:

constructors = processing of XML element and setting of entity-level context (e.g. time of waiting) via *XValue* subsystem

action method = part of PEM method of transaction, typically contains *yield* coroutine constructs

Transaction is simulation process in which Actor executes entity actions (subclass of SimPy simulation process). User code on transactions can be concentrated in constructor (in specialized subclass). The transaction contains owns *XValue* context (*transaction context*). The transaction class is subclass of *SimPy.Process* class and define PEM method (chain of action of its entities)

Simulation is lightweight extension of *SimPy.Simulation* class. It contains own XValue context (simulation context, including crucial simulation time) and container of resource objects, each of which is shared by all shared entities with the same identifier.

Actor is representation of entity, which performs transactions (a by them entity actions). In the simpler case, the actor performs just one transaction but this is not limitation (for example a car may perform several routes). The actor posses another context (*actor context*) including owns relative time.
Derived classes (specialized models of real actors) should be override constructor and define sets of attributes (in actor context)

The typical implementation of action method is relatively simple. For example the implementation of simple waiting on passage from one point to another have this form (only expired time is modeled):

```
t = self.distance / self.velocity #s/v
yield self.hold(t)
```

The precise meaning of expression s/v depends of context of attributes of distance and velocity. For example the meaning $\bar{s}/v(t_s)$ is possible where $\bar{s}$ is random variable (represents estimation of average distance) and velocity is function of simulation time (e.g. average car velocity depends on day time).

The simplified UML class diagram of *ETOS* library is presented in figure 1. The library classes are yellow colored, classes of base libraries (*xml.ETree* and *SimPy*) are in green. Sample classes of real application have orange hue.

The diagram is self-explainable except the relation between *SharedEntity* class and *SharedObject*. The instances of class *SharedEntity* with the same identifier cooperate on the single object. This shared object represents a shared resource (object of subclass of *SharedObjec*t, for example refuel station) which is unique on the whole simulation level. But the instance of class *SharedEntity* are not unique and represent the concrete using of the resource in some transaction (e.g. concrete refueling). Moreover, the using of resource is potentially repetitive in transaction (i.e. there is possibility of existence of several instances of shared entity with identical identifier in the one transaction).

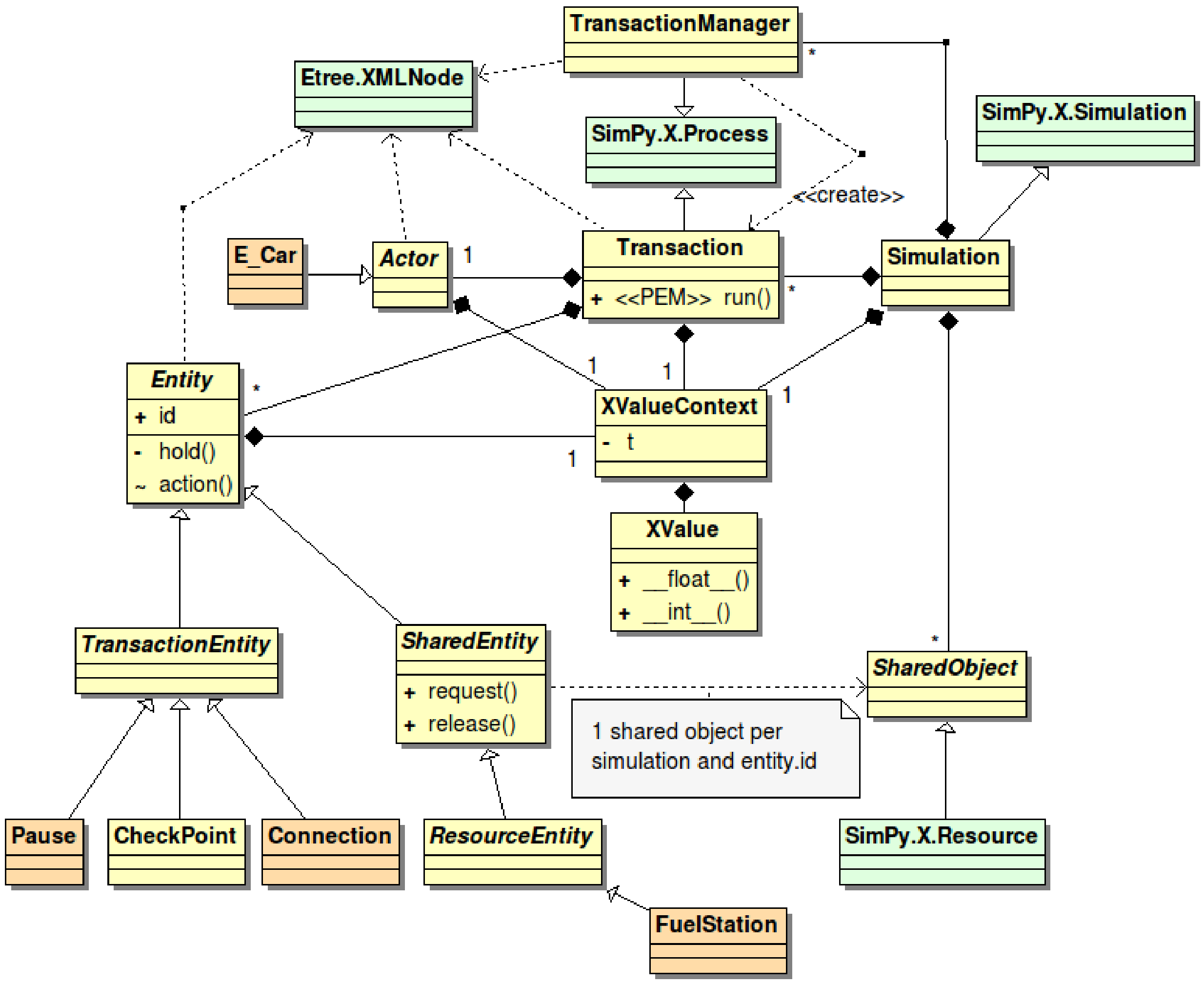

Figure 1: UML class diagram of ETOS library

The initial state of transactions and actors are read from XML files with simple structure in which the entities are represented by elements. The simple examples of this XML files are listed in fellowing listings (values are strictly in SI units).

Transaction XML (denote one transaction)

```
<?xml version="1.0" encoding="UTF-8"?>
<transaction>
    <pause><duration><normal mu="10000" sigma="600"/></duration></pause>
    <refuel id="UL"/>
    <connection id="UL-PHA"/>
</transaction>
```

Entity XML (denote entities, which are referred from transaction XML)

```
<?xml version="1.0" encoding="UTF-8"?>

<entities>
    <connection id="UL-PHA">
        <distance>83000</distance>
        <velocity><uniform min="20" max="30"/></velocity>
    </connection>
    <refuel id="UL">
        <capacity>5</capacity>
    </refuel>
</entities>
```

For representation of values of attributes the instance of *XValue* class (*x-value*) are utilized. The instances are instantiated from fixed float and integral values, parameterless functions (typically by generators of random variables with a distribution from *random* module of Python standard library) or by function of time (= function with arity equals 1).

The x-value is read-only value but is easily interoperable with and convertible to normal numeric values (the integral or float type is, if possible, strictly conserved and in some situation checked).

The each x-value is controlled by context (instance of *XValueContext* class). The context determines the maximal[1] lifetime and derived scope of value (for example the lifetime of value of entity value is execution of entity's action method i.e. scope is body of this method) the timescale of the time depended values (especially epoch), which is expressed by parameterless method (typically this is a closure based on actual simulation time and start simulation time of context lifetime). The context provides auxiliary property (getter) which provides x-value representing context time (implemented as identity function of time $f(t) = t$).

The sample of code of *XValueContext* class is listed bellow.

```
class XValueContext:
    "context␣for␣x-values␣(support␣of␣scope␣and␣local␣timescale)"
    def __init__(self, timeFunc = None):
        self.time = timeFunc #function of timescale
        self.values = weakref.WeakSet() #set of weak reference for all values of context

    def addValue(self, value):
        "add␣value␣to␣context"
        self.values.add(value)

    def resetContext(self):
        "reset␣context␣in␣the␣beginning␣of␣new␣scope"
        for value in self.values:
            if value is not None: #if value isn't garbage-collected
                value.reset()
        return self

    def __enter__(self): #implementation of context manager protocol
        self.resetContext()

    def __exit__(self, *exc): #implementation of context manager protocol
        return True
```

[1]the real lifetime is possible shorter, therefore the weak references are used for link from context to x-value

```
    @property
    def t(self):  #return auxiliary x-value represented context time
        return XValue(lambda t:t, self)
```

2.3. Problems

1. the *TransactionManager* object is, in current version, only simple trigger (starter) of transactions with random frequency (only interval is parametrized). This model is very simple and insufficiently configurable. The more complex scenarios of transactions are partly representable by system of several *TransactionManagers*. (potential solutions: representation of managers as transaction, which are configurable by XML elements or different model based on XML denotations).
2. the reusing of objects is supported (by resettable XValue contexts) but is not implemented (the pool of entities is necessary, but it must ensure sequential allocating in non-sequential coroutine context)
3. the current actor⟷transaction mapping very simple (one actor per transaction)
4. the system is not exhaustively tested (at lest simple unit testing module must be implemented)

3. phase (May, planned)

1. pilot implementation of e-car model in *ETOS* library
2. module for collecting of simulated data (if possible, based on *Numpy* library for efficiency) (sources: “checkpoints” in transactions, shared objects via SimPy interface)
3. visualization of simulated data by *matplotlib* library
4. finalizations of base implementation (support of remaining distribution)
5. testing, testing, testing, . . .

3.1. problems

- selection of real model with optimal complexity
- obtaining real attributes of actors (e-cars, fuel cells) and resources (refuel stations)

References

[1] Klaus Müller, Tony Vignaux, Ontje Lünsdorf, Stefan Scherfke. *SimPy manuals* [online]. c2012 [cit. 2012-05-01]. http://simpy.sourceforge.net/SimPy_Manual/Manuals.html

[2] Gregory Ewing. *PEP 380 – Syntax for Delegating to a Subgenerator* [online]. Version: fe8867efc2bb (2012-01-13) [cit. 2012-05-01]. http://www.python.org/dev/peps/pep-0380/

[3] David Beazley. *A Curious Course on Coroutines and Concurrency* [online]. 2009. [cit. 2012-05-01]. http://www.dabeaz.com/coroutines/

[4] *Python 3.2.3 Documentation. Multiprocessing – Process-based parallelism* [online]. May 01, 2012 [cit. 2012-05-01]. http://docs.python.org/py3k/library/multiprocessing.html