Dessert, an Open-Source .NET Framework for Process-Based Discrete-Event Simulation

Giovanni Lagorio
DIBRIS - University of Genova
Genova, Italy
Email: giovanni.lagorio@unige.it

Alessio Parma
Finsa S.p.A.
Genova, Italy
Email: alessio.parma@finsa.it

***Abstract*—We present Dessert, an open-source framework for process-based discrete-event simulation, designed to retain the simplicity and flexibility of SimPy, within the strongly-typed .NET environment. Both frameworks build domain-specific languages, for simulation writing, by using existing constructs in a novel way and providing a rich library of classes. By exploiting .NET generic types and iterators, we have successfully retained, and in few places even enhanced, the lean syntax and usability of the original library, without sacrificing static type checking. Static type-safety, in addition to being a very important property by itself, facilitates runtime code optimizations; indeed, benchmarks show that our Dessert outperforms SimPy.**

***Keywords*–Discrete-event simulation; .NET; Python.**

I. Introduction

DES (Discrete-Event Simulation) is an intuitive and flexible form of modeling that enables to represent and simulate complex systems in a wide range of application domains, from logistics and supply chain management, to health care. In this paper, we present Dessert, a process-based DES framework for .NET, explaining the rationale behind its design, and discussing the technical challenges we have faced during its development. The design of Dessert has been heavily inspired by SimPy [1] [2], which exploits Python *generators* [3], a special form of coroutines [4], for writing process-based simulations cleanly and easily.

Being written in, and consumed from, Python can be seen as a double-edged sword for SimPy, since typing errors are found at runtime and the (dynamic) typechecking overhead harms simulation running times. In designing Dessert we have striven to create a first-class "citizen" in the strongly-typed .NET environment, while retaining the lean syntax and usability of SimPy. For instance, Figure 1 shows a simple example simulation in Python, using SimPy, and Figure 2 shows the same example, written in F# using Dessert. This example is described more in Section II but, as the reader can easily verify, both listings are, with the exception of small syntactic differences, quite similar and very readable; indeed, even without knowing anything about SimPy or Dessert, the meaning of the simulation can be easily inferred.

We have developed Dessert as an open-source project, readily available via both *NuGet* [5], the package-management platform for .NET, and *GitHub* [6], one of the most popular hosting service for software development projects. *Any* .NET language can be used to write simulations to be run on our engine, since it complies with the *Common Language Specification (CLS)*, a strict subset of the .NET *Common Type System* that describes how to design types that can be manipulated by any CLS consumer [7].

The paper is organized as follows: Section II gives an overview of SimPy and Dessert, Section III analyzes design and implementation issues, and Section IV compares the performance of our framework in various environments. Finally, Section V discusses related work, while Section VI outlines some concluding remarks and further work.

```
import simpy

def car(env):
  while True:
    print('Start parking at %d' % env.now)
    parking_duration = 5
    yield env.timeout(parking_duration)
    print('Start driving at %d' % env.now)
    trip_duration = 2
    yield env.timeout(trip_duration)

env = simpy.Environment()
env.process(car(env))
env.run(until=15)
```

Figure 1. A simple example of SimPy (Python).

```
open Dessert

let rec car(env:SimEnvironment)=seq<SimEvent> {
    printfn "Start parking at %g" env.Now
    let parkingDuration = 5.0
    yield upcast env.Timeout(parkingDuration)
    printfn "Start driving at %g" env.Now
    let tripDuration = 2.0
    yield upcast env.Timeout(tripDuration)
    yield! car(env)
}

let env = Sim.NewEnvironment()
env.Process(car(env)) |> ignore
env.Run(until = 15.0)
```

Figure 2. A simple example of Dessert (F#).

II. Overview of SimPy and Dessert

As mentioned in Section I, SimPy exploits Python *generators* for writing process-based simulations. Indeed, in SimPy a *process* is simply a generator function, which is used to model active components like customers, vehicles or agents.

All processes live in an *environment*, and interact with it and with each other via *events*. This is shown in Figure 1, where the function `car` is used to model a process where a car alternates between being parked and driving. More in detail, an environment `env` is created (line 12), then a process is created by passing `car(env)` to the method `process` of the environment (line 13) and the simulation is run for 15 units of time, by calling `run(until=15)` (line 14). The process defined by the function `car` enters in an "infinite" (that is, until the simulation runs) loop that consists in:

- "parking the car", simulated by suspending the process for five units of time by yielding a *timeout event*, created by calling `env.timeout` (line 7);
- "driving the car", simulated by suspending for two units of time (line 10).

The environment `env` is used both to create new events and to get the *current time*, given in simulation units, by accessing `env.now` (lines 5 and 8). It is up to the simulation writers to decide what a *unit of time* corresponds to; for some simulations using seconds is a sensible choice, for others it makes more sense to use minutes and so on. While in SimPy simulations can be performed "as fast as possible", in real time or by manually stepping through the events, the current version of Dessert always run simulations at "full speed", so the simulation time never corresponds to the real (wall clock) time.

Figure 2 shows the same simulation of Figure 1, but written in F# using Dessert. As the reader can see, the former is just a little more verbose and, more importantly, contains type annotations (for instance, `env : SimEnvironment`, which declares that the parameter `env` must comply with the type `SimEnvironment`) that are statically checked by the compiler.

While *events* are obviously the central topic of DES, Dessert also provides some utility types for representing:

- *resources* (modeled by classes `Resource` and `PreemptiveResource`), which can be used by a limited number of processes at a time (e.g., a gas station with a limited number of fuel pumps);
- *containers* (`Container`), which model the production and consumption of a homogeneous, undifferentiated bulk. It may either be continuous (like water) or discrete (like apples);
- *stores* (`Store<T>` and `FilterStore<T>`), which are resources that enable the production and consumption of discrete objects of type `T`.

Moreover, other classes aid in gathering statistics about resources and processes. Given the available space we cannot detail all features, so we give an overview of the key concepts by means of the small, yet feature packed, following example.

Figure 3 contains a stripped down version of a process representing a *network switch*, which is used in the peer to peer simulation presented in Section IV-C. In this simulation, the switch waits quietly for incoming frames and, when one arrives, the switch delivers it to the right target. However, to perform its work, the switch needs to temporarily store incoming frames inside the buffer `_buffer`, which can only store `G.BufferSize` frames. When the buffer is full, any incoming frame is simply dropped, that is, thrown away; this fact is logged, inside method `Receive`, by invoking `G.Stats.DroppedFrame()`. We point out that, except for syntactic differences, this code is analogous to the one that it

```
 1  sealed class Switch : Entity {
 2     readonly Store<Frame> _buffer;
 3     Switch(SimEnvironment e, G g) : base(e, g) {
 4        var cap = G.BufferSize;
 5        _buffer = Sim.NewStore<Frame>(e, cap);
 6     }
 7     IEnumerable<SimEvent> Run() {
 8        while (true) {
 9           var getFrame = _buffer.Get();
10           yield return getFrame;
11           var f = getFrame.Value;
12           var w = WaitForSend(f, f.Len);
13           yield return Env.Call(w);
14           Send(f);
15        }
16     }
17     void Receive(Frame f) {
18        if (_buffer.Count == G.BufferSize)
19           G.Stats.DroppedFrame();
20        else
21           _buffer.Put(f);
22     }
23     void Send(Frame f) {
24        if (f.Type == FrameType.Request)
25           G.ServerOSes[p.Dst].Receive(f);
26        else
27           G.ClientOSes[p.Dst].Receive(f);
28     }
29  }
```

Figure 3. The switch process in (C#).

could have been written for SimPy. This is not just a matter of name similarity: the key point is that the usage of generators is *fully* preserved. Consider, for instance, the method `Run`, at line 7, which implements the behavior of the switch as an *infinite* generator. The body of the method just consists of an infinite loop, which contains the instructions that "animate" the switch. In particular, the first `yield return` yields an event that corresponds to the wait for an incoming frame. When an incoming frame `f` arrives, method `Receive` puts `f` in the buffer `_buffer`, awakening `Run`, that continues its execution at line 11. From there, the process calls a subroutine, `WaitForSend` (not shown), that stops the switch for the required time to send the frame `f`; then, as the final step, the frame is really sent to the proper target.

In this example, the buffer is represented as store of `Frame`, `Store<Frame>`, allowing us to use its blocking operations (`Get`, in this case), to stop the process until the buffer contains something to get. While the API of Dessert resembles the one of SimPy, there are some important differences. On the one hand, as we detail in Section II, everything, from events to stores, is strongly typed in Dessert so, for instance, local variable `f`, in line 12, has (inferred) static type `Frame`, since `_buffer` has static type `Store<Frame>`.

On the other hand, our goal was not to make a *straight* "clone" of SimPy, but keeping what we liked (*a lot* of design choices and features) while trying to improve the usability even more, by making some changes and additions. One addition is the introduction of a new type of events, the *call events*. In the example, a call event is used at line 14, and expresses a "call" to a *subgenerator*. While newer versions of Python, since version 3.3, elegantly handle this situation by using the new `yield from` expression [8], previous versions of Python and all mainstream .NET languages do not offer such a feature.

```
def f(): # f is a generator function
  two = yield 1 # two gets the argument of send
  yield 3
  # ...

g = f() # gets the generator
one = g.next() # gets the 1st yielded value
three = g.send(2) # gets the 2nd yielded value
```

Figure 4. Example of Python generators.

A possible workaround, not particularly elegant nor intuitive, is to iterate through the result of a subgenerator call, $v_1, v_2, \ldots$, yielding each value v_i. We think that expressing these calls through our *call events* makes the code more readable and intuitive.

III. Design and implementation issues

In this section, we first describe a couple of prerequisites, common to any implementation of a DES engine, and then we focus on typing issues.

The common prerequisite are: an efficient priority queue, to store the *(pending) event set*, and a random number generator, able to deal with various probability distributions. Curiously, both are absent in the .NET standard library. While there is not a single data structure that is the best choice for storing the event set in all situations, an *heap* is a fairly reasonable choice [9]. For this reason, we have implemented, and experimented with, various kinds of heaps (array, binary, binomial, Fibonacci, and pairing) and finally settled with a *skew heap* [10] that, in our experiments, outperformed all the other kinds. The standard .NET `System.Random` class only provides the uniform number distributions; fortunately, we have found, and used, an excellent free library [11], that supports four different random number generators and many discrete and continuous probability distributions.

When "translating" the idea of modeling a process as *generator function* yielding *events*, one of the major issues we had to face has been the fact that in Python the *yield* construct is an expression, while in .NET the corresponding construct is a statement. Moreover, in our settings, the type of such a value should be statically determined. More in detail, in Python a function f containing an `yield` expression e, that is, $e \equiv$ `yield` e', is a *generator function*, which returns an iterator, known as a *generator* g, which is an automatically generated object that permits to iterate through the values generated by evaluating the yield-expressions. The evaluation of e suspends the execution of f, which is then resumed when a method (as `next` or `send`) is invoked on g. The resulting value of e depends on the method which resumed the execution; for instance, consider the snippet of code shown in Figure 4: the call `g.next()` evaluates the body of `f` until it yields the value `1`, which is assigned to `one`. The subsequent call `g.send(2)` resumes the evaluation of `f`, that assigns `2` to `two` and yields the value `3`, which is finally assigned to `three`.

This passing values back-and-forth works very well for SimPy, where triggered events can "return" values (by `send`ing them to the generator). We tried to translate this idea in .NET as close as possible; unfortunately, in all mainstream .NET languages there are some critical differences in how generators work. Terminology aside (we stick with the Python terminology, double-quoting Python terms when used in place of the .NET terms, to make the comparison easier to follow), a method m containing a `yield return` statement s is a "generator function", whose invocation returns a "generator" g. As in Python, the evaluation of s suspends the execution of m, which is resumed when the parameterless method `MoveNext()` is called on g. The key difference is that `yield return` is a *statement*, so there is no way to pass a value v to be used as the "resulting value" of evaluating s (since s, being a statement, does not evaluate to a value!). Since a process yields only events, we introduced the (read-only) property `Value` in `SimEvent`, the supertype of all event types in Dessert, to emulate the Python behavior: when the execution of a generator function f is resumed, after an invocation of `MoveNext()` on the corresponding generator, f can retrieve the "returned/sent" value by reading the property `Value` of the yielded event object; see, for instance, lines 10 and 11 of Figure 3.

In order to statically type these values, we would have liked to introduce a generic type `SimEvent<TVal>`, exposing a property `Value` of type `TVal`, to represent events that "return" values of type `TVal`. Unfortunately, the situation is more complex than that: each event type E must also expose a collection of callbacks `Callbacks`, which are invoked when the event is triggered. In .NET the standard type to model a strongly-typed callback, that gets an object of type E as the only argument, is `Action<`E`>`. If we try to implement this common interface in `SimEvent<...>` we stumble in an inherent recursion: if E is an event type returning values of type T, then E should be a subtype of `SimEvent<`T`>`, which, in turn, should expose a collection of `Action<`E`>`. This is a known and recurring situation [12] that can be solved by introducing a second type-argument; indeed, we have defined `SimEvent<TEv,TVal>` where `TEv` is the type of the event, and `TVal` is the type of the "returned" values, as described above. In this way, inside `SimEvent<TEv,TVal>`, we can declare a collection of strongly-typed callbacks as `ICollection<Action<TEv>>`. For instance, consider `Timeout<T>`, which represents events that are scheduled with a certain delay and return values of type `T`. Such a type (indirectly) extends `SimEvent<Timeout<T>,T>`; so when we create a timeout event of type, say, `Timeout<double>` we obtain an object that exposes the collection `Callbacks` of type `ICollection<Action<Timeout<double>>>`. This guarantees that the callbacks of `Timeout<double>` are, correctly, a collection of `Action<Timeout<double>>`.

In SimPy, and so in Dessert, events can be combined together to form *event conditions*, that is, events that are triggered when some *condition* becomes true. For instance, given two events e_1 and e_2, we could create a new condition event $e_{\texttt{AND}}$ that is triggered when both e_1 and e_2 are triggered, or create another event $e_{\texttt{OR}}$ that is triggered when *any* of them is triggered, and so on. In general, any number of events and any predicate p can be specified, allowing simulation authors to build arbitrarily complex conditions that are triggered whenever p becomes true on the given events. A common use of event combinations is implementing timeout policies; for instance, given a certain event e, obtaining a new event that corresponds to waiting for e *or* the expiration of a timeout event.

Differently from SimPy, in our strongly-typed settings, the result of combining arbitrary events, of types $t_1, \ldots, t_n$,

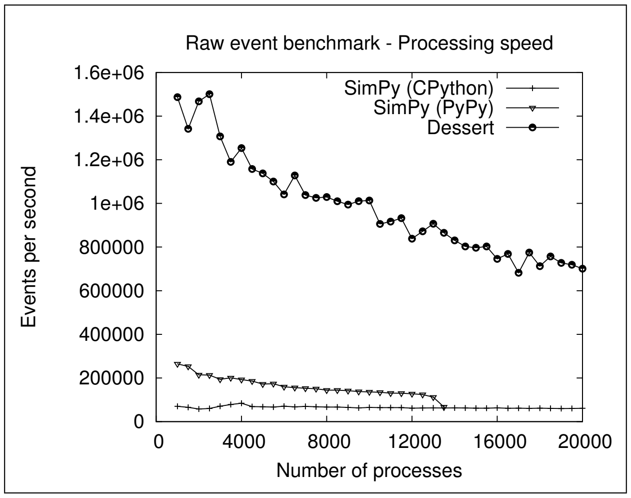

Figure 5. Events per second in timeout benchmark.

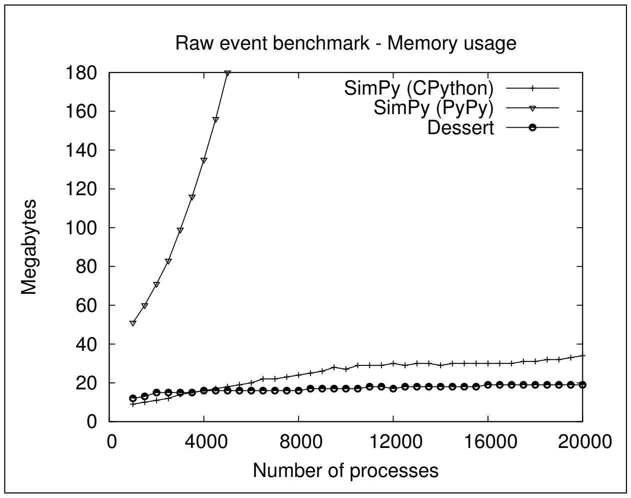

Figure 7. Memory usage in timeout benchmark.

must have a type that "remembers" the types t_1,...,t_n. That is, if an event has some type t_1, and another event has type t_2, then the resulting type of combining them must include (some encoding of) both t_1 and t_2. For this reason, we use the "variadic-generic type" `Condition<`t_1`,...,`t_n`>` to encode the type of condition events built from events of type $t_1, \ldots t_n$. Inside an object of type `Condition<`t_1`,...,`t_n`>` the source events are available through the read-only properties named `Ev`N, where $N \in \{1, \ldots, n\}$. Note that `Condition` cannot really be single generic type, since in .NET each generic type is constrained to have a fixed number of type arguments. Handling the combination of an arbitrary number of events would require a mechanism analogous to C++ variadic templates [13] which, at the moment, is not available in C# (and the other mainstream .NET languages). So, we had to use a family of generic types (`Condition<T1>`, `Condition<T1,T2>`, `Condition<T1,T2,T3>` and so on); fortunately, this family of types can be automatically generated, for any arbitrary number of type arguments, by exploiting the T4 [14] (Text Template Transformation Toolkit) offered by Visual Studio.

IV. BENCHMARKS

In this section, we describe the benchmarks used to assess the relative performance of our Dessert, with respect to SimPy. We start, in Section IV-A, with the specifications of the machines used to run the benchmarks and the general description of the benchmark environment. Then, we describe the two kinds of benchmarks we carried out. The former, described in Section IV-B, is an artificial simulation, akin to a stress-test, where we obtain the average raw event processing time of the engines. The latter, described in Section IV-C, consists in running a real simulation of a peer-to-peer (P2P) system, thus measuring how the different engines perform on a "real-world" simulation.

```
def timeoutBenchmarkProcess(env, counter):
  while True:
    yield env.timeout(randomDelay())
    counter.increment()
```

Figure 6. Benchmark process.

A. Benchmark environment

Every benchmark has been run under a dedicated virtual machine (VM), created and run by VirtualBox [15] 4.3.2, hosted on Ubuntu 13.10 on a Intel Core 2 Duo E4700 with 4 GB or RAM. We created a Windows VM, with Windows 7 SP1 and the .NET Framework 4.5.1, and a GNU/Linux one, with Lubuntu [16], a lightweight variant of the more famous Ubuntu, and Mono 3.2. Both VMs share the same hardware profile: 2 CPU cores and 2 GB of RAM. Since DES is a strongly CPU-bound process, neither Dessert nor SimPy use secondary storage, we do not detail storage specifications. As concerns Python interpreters, we tried out both CPython 2.7, the "default" language implementation, and PyPy [17] 2.2, a recent and highly optimized alternative Python implementation. Both implementations have been run with full optimizations enabled (`-OO` flag).

In order to time our benchmarks, we started a virtual stopwatch at the beginning of each run and we stopped it at the end. On .NET we used a standard dedicated class, `System.Diagnostics.Stopwatch`, while on Python we used the facilities exposed by the general `time` module. To evaluate memory usage, we sampled the *resident set size* (RSS) of the (operating system) process at fixed intervals, by taking advantage of the standard class `System.Diagnostics.Process` on .NET, and of the library `psutil` [18] on Python.

In the following sections, for lack of space, we thoroughly analyze only the benchmarks on Windows. Anyway, the tests on GNU/Linux confirmed what we found on Windows, with a caveat: since Mono is not as optimized as .NET, Dessert still outperforms SimPy on CPython, but PyPy becomes an interesting competitor, yielding better results in the P2P simulation tests, but consuming an enormous quantity of memory, as it does on Windows. Given Mono continuous improvements, the performance of Dessert on Linux can only get better, so we hope to achieve soon the same results we already obtain on Windows.

B. Raw event processing benchmark

The goal of this benchmark is to measure the raw event processing speed of the DES engines. To do this, we designed a rather artificial simulation, in which we spawn an increasing number of extremely simple processes, as shown in Figure 6.

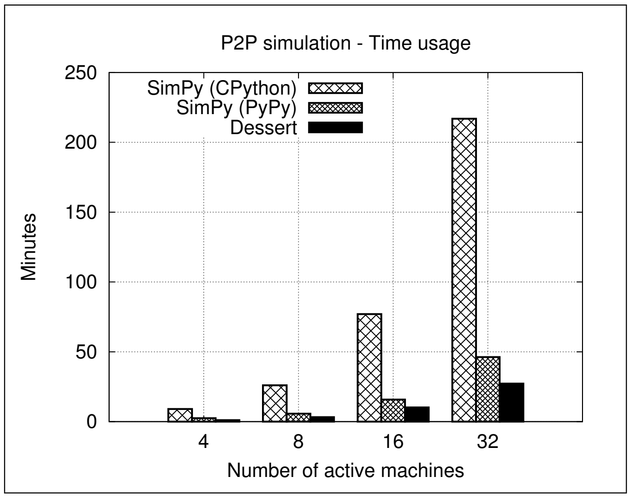

Figure 8. Time usage in P2P simulation.

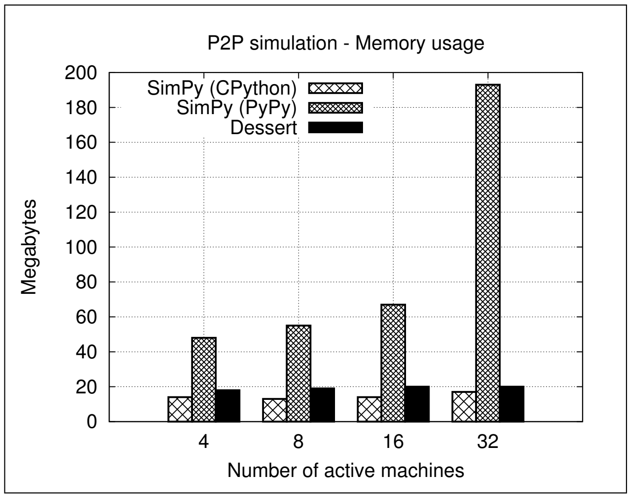

Figure 9. Memory usage in P2P simulation.

Each of those processes simply awaits a random timeout and, when woken up, increases a shared counter, which records the total amount of timeout events properly handled by the engine. So, by dividing that counter by the simulation execution (wall) time, we obtain a good approximation of the average event handling speed. Analogously, we measure how much memory each engine consumes by repeatedly running the same simulation and varying the number of processes from an already significant $1,000$, to a rather big $20,000$. In this way, we evaluate how the engines perform when heavily loaded. For any given process count, we have run twenty simulations and averaged the results.

Before discussing the results, we would like to emphasize that these benchmarks, by themselves, cannot tell us which is the fastest engine, but only which engine has the potential of being the fastest. As it is shown Figure 5, the raw event processing speed of Dessert is impressive, especially when the number of processes is (relatively) low. Anyway, as the graph clearly shows, in this benchmark Dessert *always* outperforms SimPy, even when it is run by PyPy. We also note that the graph shows the results of PyPy only to $13,000$ processes because, given its huge memory consumption, PyPy crashes after a period of uninterrupted swapping activity (that is, thrashing). This fact can be clearly seen in Figure 7, where PyPy memory consumption goes off the charts even with a small number of processes. The same figure shows that Dessert and CPython follow, more or less, the same curve, demonstrating that Dessert potentially allows users to run simulations faster without incurring on higher memory consumption. Moreover, on higher loads Dessert is faster (Figure 5) *and* consumes less memory (Figure 7).

C. P2P simulation

While the benchmarks previously discussed are useful to understand how fast the simulation engines could perform, they could not answer to a crucial question: what is the fastest engine on common, “real-world”, simulations? To answer such a question, we have simulated the execution of a peer to peer protocol based on linear network encoding [19]. Since the protocol was created solely for teaching purposes, we will describe it here very briefly.

Suppose we have n machines, each one running both a client and a server process, and set $k = \lfloor \frac{n}{2} \rfloor$. At first, each file that must be shared is first split into k parts, then other $k-1$ parts are created by linear combinations of the first k parts. Thus, every file is encoded in $n-1$ parts, and each part is stored on a different machine. The rest of the simulation consists in seeing whether the clients, that try to retrieve (parts of) the files from the servers, saturate the whole network, since all communications are routed through a single switch. In particular, each client needs to request at least k parts to recover a file, but it could do more requests to reduce wait times. Therefore, one of the goals of the simulation is to understand how many extra requests give the lowest wait times. For each combination of machine count n and extra-request count r, we run twenty simulations, so that results are pretty accurate and reliable. Therefore, since r lies in the interval $[0, k-1]$, for each n we execute $20 \cdot k = 10 \cdot n$ simulations.

As it is shown in Figure 8, Dessert can execute these simulations faster than SimPy, especially when the number of machines gets higher. Under these particular settings, Dessert is *five* times faster than SimPy run on CPython, and twice as fast than SimPy when run on PyPy, which we deem as a good result. Results on memory consumption, shown in Figure 9, confirm PyPy memory problems and the fact that Dessert and SimPy, when run on CPython, have nearly the same footprint, although in this case the one of Dessert is slightly higher.

V. Related work

On the one hand, many libraries enable to write discrete-event simulations, using a variety of programming languages and environments. Indeed, as we have already said, our work has been greatly inspired by SimPy [1][2][20], which is written in, and usable from, Python. On the other hand, the .NET framework has been somewhat neglected by DES library authors, so there are very few free options (some commercial options are: Micro Saint® Sharp [21] and Sage®, the successor of HighMAST™ [22]) to choose from.

In particular, as far as we know, our Dessert is the *first* open-source (complete) project, on the .NET framework, for writing discrete event simulations following the *process oriented paradigm*. In this paradigm, simulations consist of interacting *processes*, that is sequences of events and activities.

This approach allows users to write simple and readable simulation code; the relationships between various paradigms and, especially, the challenges associated with modeling problems with different aspects best represented by different paradigms is a topic of on-going research [23].

Focusing on the .NET framework, the only free options that we have found implement a different paradigm or are incomplete and, apparently, abandoned. SharpSim [24] is an open-source library, written in C#, that implements the *event-oriented paradigm*. In this paradigm, users model the systems in terms of events. Implementations of this paradigm can be very efficient, but simulation code following this style is less modular, and harder to write and understand [25]. React.NET [26] is another open-source library written in C#, which shares our paradigm and general goals. Unfortunately, the project seems dead, since there are no stable releases and it has not been updated since 2006. Finally, DotNetSim [27] is described as a prototype that exploits .NET for writing fully object-orientated components that cross programming languages, packages and platforms and link them in a single application. However, this seems to be another dead project, since we could not find any prototype to download and evaluate.

VI. Conclusion and Future Work

We have presented Dessert, a fully managed .NET engine for process-based discrete-event simulation. On the one hand, Dessert has been heavily inspired, and tries to follow, the simplicity and leanness of SimPy, in the strongly typed .NET world. On the other hand, Dessert is not, and was not supposed to be, a *straight* "clone" of SimPy: we kept what we liked, which is *a lot*, but we have also tried to improve the usability even more, by making some changes and additions. Moreover, by leveraging the .NET framework, and adhering to Common Language Specification, Dessert allows user to write simulations in a variety of different programming languages. For these reasons, Dessert yields better performances both at *development* and *execution* time, since static typing is beneficial for both catching many problems early on, and permitting the use of refactoring and context-aware code completion tools, like Visual Studio IntelliSense. Indeed, the speed-up offered by Dessert is impressive, especially when SimPy is interpreted through CPython, the "default" Python interpreter, and has nearly the same memory footprint. We also benchmarked SimPy on PyPy, a recent and highly optimized alternative Python implementation, which closes the gap in running times, but at the cost of a huge memory consumption. For this reason, SimPy on PyPy crashes on "big" simulations that our Dessert handles effortlessly.

Since Dessert is completely open-source, its future developments are somewhat unpredictable. We plan to develop a library of higher level abstractions, like network components and elements of stocking chains, to ease the development of complex simulations. Moreover, we would like to address the loss of performance when running on Mono. Another direction for further work is the development of a proper domain-specific language, to write simulations even more easily, which could be compiled and run on our engine.

References

[1] "SimPy," 2014, URL: https://pypi.python.org/pypi/simpy [accessed: 2014-03-08].

[2] K. Müller, "Advanced systems simulation capabilities in SimPy," 2004, europython 2004, URL: http://simpy.sourceforge.net/old/images/Advanced_Systems_Simulation_Capabilities_in%20SimPy_Fallback_Last.pdf [accessed: 2014-03-09].

[3] N. Schemenauer, T. Peters, and M. L. Hetland, "Simple generators," 2001, python Enhancement Proposal 255 URL: http://www.python.org/dev/peps/pep-0255/ [accessed: 2014-03-08].

[4] A. L. D. Moura and R. Ierusalimschy, "Revisiting coroutines," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 31, no. 2, Feb. 2009, pp. 6:1–6:31.

[5] "Dessert on NuGet," 2014, URL: https://www.nuget.org/packages/Dessert/ [accessed: 2014-03-08].

[6] "Dessert on GitHub," 2014, URL: https://github.com/pomma89/Dessert [accessed: 2014-03-08].

[7] J. Hamilton, "Language integration in the common language runtime," ACM Sigplan Notices, vol. 38, no. 2, 2003, pp. 19–28.

[8] "What is new in Python 3.3," 2014, URL: http://docs.python.org/3.3/whatsnew/3.3.html [accessed: 2014-03-08].

[9] R. Rönngren and R. Ayani, "A comparative study of parallel and sequential priority queue algorithms," ACM Transactions on Modeling and Computer Simulation (TOMACS), vol. 7, no. 2, 1997, pp. 157–209.

[10] D. D. Sleator and R. E. Tarjan, "Self-adjusting heaps," SIAM Journal on Computing, vol. 15, no. 1, 1986, pp. 52–69.

[11] S. Troschuetz, ".NET random number generators and distributions," 2014, URL: http://www.codeproject.com/articles/15102/net-random-number-generators-and-distributions [accessed: 2014-03-08].

[12] J. O. Coplien, "Curiously recurring template patterns," C++ Report, vol. 7, no. 2, 1995, pp. 24–27.

[13] D. Gregor and J. Järvi, "Variadic templates for C++0x." Journal of Object Technology, vol. 7, no. 2, 2008, pp. 31–51.

[14] "Code generation and t4 text templates," 2014, URL: http://msdn.microsoft.com/en-us/library/bb126445.aspx [accessed: 2014-03-08].

[15] "VirtualBox," 2014, URL: https://www.virtualbox.org/ [accessed: 2014-03-08].

[16] "Lubuntu," 2014, URL: http://lubuntu.net/ [accessed: 2014-03-08].

[17] "PyPy," 2014, URL: http://pypy.org/ [accessed: 2014-03-08].

[18] "psutil," 2014, URL: https://code.google.com/p/psutil/ [accessed: 2014-03-08].

[19] S.-Y. Li, R. W. Yeung, and N. Cai, "Linear network coding," Information Theory, IEEE Transactions on, vol. 49, no. 2, 2003, pp. 371–381.

[20] K. Müller and T. Vignaux, "SimPy: Simulating systems in Python," 2003, ONLamp.com Python DevCenter, URL: http://www.onlamp.com/pub/a/python/2003/02/27/simpy.html [accessed: 2014-03-08].

[21] W. K. Bloechle and D. Schunk, "Micro saint® sharp simulation software," in Proceedings of the 35th conference on Winter simulation: driving innovation. Winter Simulation Conference, 2003, pp. 182–187.

[22] P. C. Bosch, "Simulations on .NET using HighPoint's highmast™ simulation toolkit," in Simulation Conference, 2003. Proceedings of the 2003 Winter, vol. 2. IEEE, 2003, pp. 1852–1859.

[23] S. K. Heath, A. Buss, S. C. Brailsford, and C. M. Macal, "Cross-paradigm simulation modeling: challenges and successes," in Proceedings of the Winter Simulation Conference, 2011, pp. 2788–2802.

[24] "SharpSim," 2014, URL: http://sharpsim.codeplex.com/ [accessed: 2014-03-08].

[25] N. Matloff, "Introduction to Discrete-Event Simulation and the SimPy language," 2008, URL: http://heather.cs.ucdavis.edu/~matloff/156/PLN/DESimIntro.pdf [accessed: 2014-03-08].

[26] "React.NET," 2014, URL: http://reactnet.sourceforge.net/ [accessed: 2014-03-08].

[27] M. Pidd and A. Carvalho, "Simulation software: not the same yesterday, today or forever," Journal of Simulation, vol. 1, no. 1, 2006, pp. 7–20.