ScipySim: Towards Heterogeneous System Simulation for the SciPy Platform

Allan I. McInnes and Brian R. Thorne
Department of Electrical and Computer Engineering
University of Canterbury, Christchurch, New Zealand
allan.mcinnes@canterbury.ac.nz, brian.thorne@canterbury.ac.nz

Keywords: SciPy, Heterogeneous systems, Tagged signal model, Generalized Kahn theory

Abstract

The goal of the ScipySim project is to develop a heterogeneous system simulation capability for the SciPy scientific computing platform, based on an executable version of the Lee/Sangiovanni-Vincentelli tagged signal model that was proposed by Caspi *et al.* Instead of using a centralized simulation engine, ScipySim simulations are composed of autonomous actors that interact by passing tagged events through first-in/first-out queues. The resulting simulation system is completely decentralized, and well-suited to distributed execution. We describe the design and operation of ScipySim, its current status, and prospects for the future.

1. INTRODUCTION

ScipySim is a tool for heterogeneous system simulation developed using the Python programming language [1], and intended to run on top of the SciPy [2] platform for scientific computing. Python is freely available on many platforms. SciPy is also freely available on a similar number of platforms, and is becoming increasingly popular as a scientific and engineering computing platform in both academia and industry. However, there is little support for system simulation in Python or SciPy, and none of the simulators that we are aware of support the simulation of heterogeneous systems, i.e., systems composed of components that operate in a variety of domains or implement different models of computation (e.g., discrete-time, continuous-time, or Petri nets). The two primary motivations for the ScipySim project were:

- To develop a block-diagram-based heterogeneous system simulation tool for the SciPy platform.
- To explore the implementation of simulators based on the "actors without directors" generalized-Kahn approach proposed by Caspi *et al.* [3].

In this paper, we describe the design of ScipySim (sec. 3.1.), which extends the ideas proposed by Caspi *et al.* beyond a proof-of-concept into a working simulator. We also describe the decentralized way in which ScipySim simulations are executed (sec. 3.2.), discuss how ScipySim may be developed into a distributed simulation system (sec. 3.3.2.), and outline some of the remaining problems in supporting general continuous-time models (sec. 3.3.3.). We begin with a brief overview of the tagged signal model (sec. 2.), and close with a review of related work (sec. 4.).

2. BACKGROUND

Lee and Sangiovanni-Vincentelli developed the "tagged signal model" [4] as a denotational framework for describing the behavior of heterogeneous systems. It encompasses ideas from both signals & systems theory and computer science.

The starting point for the tagged signal model is a set of *values*, V, and a partially-ordered set of *tags*, $(T, \leq)$. An *event* is a tag-value pair $e = (t, v) \in T \times V$. A *signal* is a set of events $s \subseteq T \times V$. A *process*, or *actor*, P, is a function from a set of input signals, S_i, to a set of output signals, S_o, i.e., $P : S_i \longrightarrow S_o$. Different choices of tag and value sets (partial or total order, finite or infinite) lead to different models of computation. The resulting formalism is similar to the I/O relation framework of Zeigler *et al.* [5], but addresses the behavior of coupled systems without introducing states or transitions.

The tagged signal model is intended to allow the relationships between models of computation to be understood and defined. The tagged signal semantics is denotational rather operational, meaning that it describes the behavior a system will produce without describing how the system is executed. Heterogeneous system simulators, such as Ptolemy II [6], can be understood in terms of the tagged signal model, but their execution is governed by different semantics. For example, execution in Ptolemy II uses "directors" that regulate the execution of a domain or model of computation.

In an effort to bring the denotational and operational semantics of heterogeneous system simulation closer together, Caspi *et al.* [3] proposed a generalization of Kahn Process Networks [7] that builds upon tagged signals to provide an executable model of heterogeneous systems. In their model, systems are composed of autonomous actors that interact by sending tag-value pairs through first-in-first-out (FIFO) queues. The actors are similar to Kahn processes, in that they block when one or more of their input queues is empty, and each actor only knows the head of their input queues. Tag-conversion actors manage interactions between different domains. Caspi *et al.* claim that their approach does not require centralized execution control, naturally supports heterogeneous models, and is well-suited to distributed execution.

3. SCIPYSIM

Caspi *et al.* [3] give several examples of executable actor models based on the generalized Kahn theory. Further examples, and a small simulator prototype written in the Haskell programming language, appear in a technical report by Beneniste *et al.* [8]. However, the examples and simulator prototype are a fairly limited proof-of-concept, and do not explore issues such as distributed execution. Our goal in developing ScipySim is to move beyond a proof-of-concept implementation of the generalized Kahn approach.

3.1. Simulator Design

ScipySim is nominally built on top of SciPy, although it currently uses only a handful of SciPy features. Most of ScipySim is implemented in standard Python. Graphical output, such as line graphs and stem plots, is produced using the matplotlib plotting library [9].

ScipySim consists of a core set of classes that define foundational simulation entities such as `Actors`, `Events`, `Channels`, and `Models`, as well as a library of `Actor` subclasses that implement different kinds of simulation components. The current library of actors includes:

- Signal source actors (constant, ramp, step, and sinusoid).
- Basic mathematical operation actors (summation, scaling, and absolute values).
- Signal manipulation actors (delay, split, boolean filter, interpolator, and decimator).
- Numerical integration actors (discrete-time and continuous-time).
- Display actors that translate signals into matplotlib plots.

3.1.1. Events

In keeping with the tagged signal model, `Event` objects contain a tag and a value. The `Event` class is designed to make `Event` objects immutable, which prevents possible simulation errors resulting from actors modifying a shared `Event`. Thus

```
e = Event(1.0, 3.5)
```

will create a new event with fields `e.tag` and `e.value` respectively equal to 1.0 and 3.5, but an attempt to assign a new tag (e.g., `e.tag = 2.0`) will cause an exception to be raised. The terminal event in a signal is marked by having its `last` field set to `True`.

3.1.2. Actors

In keeping with the concurrent nature of the generalized Kahn networks proposed by Caspi *et al.*, every `Actor` is a Python thread. A slightly simplified version of the `Actor` class appears in Fig. 1. The arguments to the `Actor` constructor are a list of input channels and a list of output channels. The `Actor` class itself is essentially a thin wrapper around the Python `Thread` class that overrides the `Thread.run()` method, which specifies the functionality that should be executed in a separate thread of control, to continually call the `process()` method.

```
class Actor(threading.Thread):
    def __init__(self, input_channel=None,
                       output_channel=None):
        # Initialization of internal fields
    def run(self):
        while not self.stop:
            self.process()

    def process(self):
        raise NoProcessFunctionDefined()
```

Figure 1. ScipySim `Actor` class

```
def process(self):
    event = self.input_channel.get()
    if event.last:
        self.stop = True
        self.output_channel.put(event)
        return
    out_event = # Some computation
    self.output_channel.put(out_event)
```

Figure 2. Example `process()` method

The `Actor` base class is abstract, and thus provides no implementation of `process()`. Derived classes implement their own `process()` methods to provide appropriate actor behavior. The typical form of a `process()` method is to block until an event is available on all input channels, compute a new output once all input events are available, and send the output through any output channels. For example, a single-input/single-output actor would have a `process()` method like that shown in Fig. 2. This pattern is captured in our `Siso` class, which sets

```
out_event = self.siso_process(event)
```

and can be subclassed to define a single-input/single-output actor by implementing the `siso_process()` method. A `siso_process()` for a simple actor is shown in Fig. 3.

3.1.3. Channels

Actors communicate via `Channels` that carry `Events`. The `Channel` class is an extension of Python's thread-safe `Queue` class, and follows the principles of Kahn Process Networks: the size of the queue is nominally infinite, so `put()`

```
def siso_process(self, event):
    new_value = event.value * self.gain
    return Event(event.tag, new_value)
```

Figure 3. A simple gain

```
class Model(Actor):
    components = []

    def __init__(self, *args, **kwargs):
        super(Model, self).__init__(*args,
                                    **kwargs)
    def process(self):
        pass

    def run(self):
        assert hasattr(self, 'components')
        [c.start() for c in self.components]
        while True:
            if not any([c.isAlive()
                for c in self.components]):
                break
            else:
                sleep(1)
```

Figure 4. The `Model` class (with `KeyboardInterrupt`-handling removed for clarity)

operations should always be non-blocking, and a thread attempting a `get()` operation when the queue is empty will be blocked until an `Event` is available. The `Channel` class also adds the ability to inspect the head of the queue without removing it, which is useful for processing inputs from multiple channels in the appropriate tag order but does require careful use if the continuity of the actor is to be preserved [3].

The events in a channel must have monotonically increasing tags, so that actors receiving events from the channel can rely on never receiving an input earlier than the one they are currently processing. However, `Channel` objects do not enforce this order. They provide a strict FIFO ordering on events, and assume that the actors adding events to the queue maintain the tag ordering. As a consequence, only a single actor should be responsible for adding events to a given channel.

3.1.4. Models

In ScipySim, a model is defined as collection of actors. A `Model` object can be used to specify a set of actors that form a model for simulation. It is also a fully-fledged actor in its own right. `Model` objects can therefore be used to impose a hierarchy on complex actor systems, and to create composite subsystems made up of lower-level actors.

Although a `Model` is an `Actor`, it does not define a `process()` method. Instead, as shown in Fig. 4, the behavior of a `Model` is generated by overriding the `run()` method to start execution of all of the components contained in the `Model`. Specific models are usually created by subclassing the `Model` base class, and populating the `components` list with actors in the constructor of the derived class. An example of creating a model using this technique appears in Fig. 5.

3.2. Simulator Operation

Unlike many simulators, ScipySim does not include a separate simulation program that regulates the execution of a simulation model. Instead, the execution of the simulation is effectively self-regulating due to the nature of generalized Kahn theory [3]. Actors will execute when they are able to, and otherwise block awaiting input events. As a consequence, a simulation model is both a definition of the `Actor`s and connecting `Channel`s that make up a the system to be simulated, and a directly executable Python program that carries out the system simulation.

Fig. 5 shows a simple example of a ScipySim model. Like all ScipySim models, it is a subclass of the `Model` class. The system to be simulated is defined within the constructor of the model. In this case, the system consists of:

- A `Ramp` actor, which is a signal source producing a linearly increasing output
- A `CTSinGenerator` actor, which is a continuous-time sinusoidal signal source
- A `Summer` actor, which takes inputs from the ramp and sin sources, and outputs their sum
- A `Plotter` actor, which generates a matplotlib plot from the events it receives

`MakeChans()` is a utility function used to create a collection of channels. Once the necessary number of channels have been created, the connections between actors are defined by passing the appropriate channels to each actor as it is created. Finally, all of the actors are added to the `components` list of the model. Execution of the simulation is triggered by calling the `run()` method of the model, resulting in an output from the `Plotter` like that shown in Fig. 6.

As described in sec. 3.1.4., calling the `run()` method of a `Model` causes all of the actors contained within the model to start executing. During execution the progress of the simulation is driven by the source actors, which autonomously produce events in accordance with the kind of signal they represent. Each actor runs until it receives a terminal event, either from a signal source or from another actor, at which point it attempts to terminate. The simulation run ends when all actors have terminated.

```
# Various imports from the scipysim
# library appear here
class SinRampSum( Model ):
    def __init__( self ):
        chan1, chan2, chan3 = MakeChans( 3 )
        src1 = Ramp( chan1 )
        src2 = CTSinGenerator( chan2,
                   amplitude=1.0, freq=1.0 )
        summer = Summer( [chan1, chan2],
                         chan3 )
        dst = Plotter( chan3,
                   title="Sin/Ramp Sum" )
        self.components = [src1, src2,
                           summer, dst]

if __name__ == '__main__':
    SinRampSum().run()
```

Figure 5. An example of a simulation model

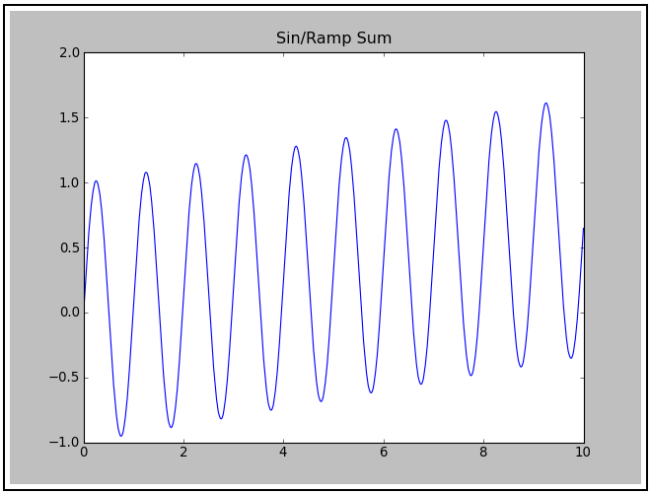

Figure 6. The output of the example simulation model

3.3. Current Status and Future Directions

We have fulfilled our goal of moving implementation of a simulator based on generalized Kahn theory beyond the proof-of-concept described by Benveniste *et al.* [8]. We have implemented the core classes described in sec. 3.1., along with a small library of actors modeling various signal sources and basic operations on signals. We have also created a collection of simulation models to test and demonstrate various aspects of ScipySim. These models include a pulse-width modulated signal generator, a numerically integrated approximation of a ballistic trajectory, and a simple discrete-time infinite impulse response (IIR) filter. All of these models can be found in the code repository at http://code.google.com/p/scipy-sim/.

Although implemented beyond a proof-of-concept, ScipySim is still very much a work in progress. It was originally envisaged as a graphical tool in which models would be specified by drawing block diagrams. But our implementation efforts have focused on the underlying simulation infrastructure, so ScipySim currently lacks a graphical user interface. A great deal of work still remains to be done on the simulation infrastructure as well, particularly in the areas of event and channel type-checking, actor implementation and execution, and modeling of continuous-time systems.

3.3.1. Events and Channels

At present `Event` objects are untyped, or rather they all share the same type. The domain of an `Event` is implicit in the kind of tag it carries, for example a floating-point number for a continuous-time event, or an integer for a discrete-time event. `Channel` objects do include a `domain` field, which is a string identifying the kind of events the channel is expected to carry. However, this field is largely unused at the moment, and channels do not do any checking on the domain of the events put into them. This lack of checking makes it possible for model-builders to construct nonsensical models, and feed erroneous events to actors. One potential path for future improvement is to include domain type-tags as part of each `Event` object, and perform runtime checking on these types to ensure that the simulation model is properly constructed.

3.3.2. Actor Implementation

Although we already have a library of actors in place, this library must be expanded to cover more domains and more complex models. In particular, we need to implement actors for untimed domains such as Kahn Process Networks and Petri nets, and tag conversion actors for connecting these domains to existing domains. We also need to implement actors such as continuous-time and discrete-time transfer functions.

More fundamentally, we need to reconsider how actors should be implemented. Making each actor its own thread is a heavyweight solution that adds overhead to simulations. On the positive side, it helps to ensure that the none of the simulation components rely on global information, and are thus ready for distributed execution. Ultimately, we will probably move to a scheme that uses some form of green threads layered over a process pool. This approach would eliminate the overheads of native threads, enable us to avoid the bottleneck of the notorious Python “Global Interpreter Lock” [10], and allow us to implement distributed simulations.

3.3.3. Continuous-Time

Benveniste *et al.* [8] discuss some preliminary ideas for modeling continuous-time systems in their generalized Kahn framework. They present a proof-of-concept numerical integration scheme that performs Euler integration with a simple

step adaptation for closed-loop integrals (i.e., those in which the derivative is a function of the integral). However, they do not discuss open-loop integration. Furthermore, they note that their approach leads to difficulties with detecting threshold-crossing events since it requires a variable time-step numerical integration scheme that is able to roll back integration when a threshold is missed. Such integration schemes are not Kahn-order preserving, and are difficult to implement in an asynchronous simulation that lacks a global clock.

In the current implementation of ScipySim we provide a first-order fixed-step numerical integration actor. This allows simple continuous-time simulation in both open- and closed-loop. We are also experimenting with discrete-event numerical integration schemes [11] derived from the DEVS formalism, which seem to offer the ability to perform variable time-step integration in an asynchronous context. Such schemes could also resolve the difficulties faced by Benveniste *et al.* in handling threshold-crossing events, since they naturally progress from one threshold (or quantum) to another.

One potential drawback of a discrete-event integration scheme is that it may not produce an output for every input, which can result in deadlocks if the integration actor is part of a feedback loop. One way to mitigate this problem is to force an output after every input, which may result in repeated output of the same quantized state value. This approach may not strictly follow the standard DEVS [5] execution semantics, in which outputs are only produced when an internal transition to a new state occurs[1]. However, since we are incorporating the discrete-event integrator into a continuous-time simulation scheme that uses a discrete-time approximation, generating an output for every input seems reasonable. We are also investigating the use of a conservative parallel discrete-event simulation scheme in the style of Chandy and Misra [12], are considering migrating to a continuous-time simulation scheme based entirely on a discrete-event approach.

Neither approach to deadlock avoidance addresses the problem of causality. Algebraic loops (feedback loops with only zero-delay elements) in a continuous-time model can result in situations where time does not advance. This is true for both for the fixed-step and discrete-event integration schemes. At present we leave it to the user to break algebraic loops by manually inserting a delay into the loop.

4. RELATED WORK

The DEVS approach [5] appears quite similar to the Lee/Sangiovanni-Vincentelli tagged signal model from the perspective of denotational semantics. However, DEVS also provides an operational description of system behavior that is suitable for simulation. In that sense, DEVS is comparable to generalized Kahn theory. Of the two, DEVS is the more mature formalism, but the generalized Kahn approach appears more directly suited to distributed execution. A possible convergence of the two formalisms can be seen in conservative parallel DEVS simulators such as that proposed by Jafer and Wainer [13], and our efforts in incorporating ideas from the DEVS community into ScipySim.

Other authors have investigated operational semantics for heterogeneous systems. The Ptolemy Project uses an approach that associates a "director" with each domain to govern execution of that domain [14]. Basu *et al.* developed a methodology based on specifying the behavior of components, possible interactions, and the priorities of interactions [15]. Their BIP framework supports multithreaded execution, but relies on a central execution engine. The Rosetta Project uses a coalgebra to define the semantics of heteogeneous systems [16], which is similar to the generalized Kahn approach. However, the focus Rosetta appears to be on developing a specification language rather than simulation tools.

Although the semantics of heterogeneous system simulation is still an active area of research, simulation tools for heterogeneous systems are already available. The MathWorks' Simulink® tool [17] is probably the most widely used of these tools. Other notable heterogeneous simulation tools are Scicos [18] and Ptolemy II [6]. These tools are generally based on an underlying discrete-time approach with a global clock, or some other form of centralized control.

There are few simulation tools for Python, and none that we are aware of support heterogeneous system simulation. SciPy provides simulation of linear time-invariant systems via `scipy.signal.lsim` [2]. SimPy is a sophisticated discrete-event simulation tool for Python [19]. PyDSTool is a simulation and analysis environment for modeling dynamical systems [20], focusing on ordinary differential equations and differential-algebraic equations.

5. CONCLUSIONS

The goal of the ScipySim project is to develop a heterogeneous system simulation capability for the SciPy scientific computing platform, based on the generalized Kahn theory developed by Caspi *et al.* [3]. The development of ScipySim has extended implementation of a simulator based on the generalized Kahn theory beyond the simple proof-of-concept developed by Benveniste *et al.* [8]. The core of the ScipySim architecture, based on autonomous actors running in their own threads and interacting through FIFO queues, is now in place. A range of example simulation models have been developed, demonstrating various features of the simulation system. However, ScipySim is still very much a work in progress, with more work needed on verifying correct model construction, efficiently implementing actor execution, and accurately modeling continuous-time systems.

[1] Although arguably the *internal* unquantized state of the integrator changes simply with the passage of time that is marked by reception of a new input

ScipySim is an open source project licensed under the GNU General Public License. Complete source code and packaged releases are available from the project website, at http://code.google.com/p/scipy-sim/.

6. REFERENCES

[1] G. van Rossum *et al.*, "Python," 1989–. [Online]. Available: http://www.python.org/

[2] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[3] P. Caspi, A. Benveniste, R. Lublinerman, and S. Tripakis, "Actors Without Directors: A Kahnian View of Heterogeneous Systems," in *Proc. 12th International Conference on Hybrid Systems: Computation and Control*, San Francisco, CA, 2009, pp. 46–60. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00602-9_4

[4] E. A. Lee and A. Sangiovanni-Vincentelli, "The Tagged Signal Model - A Preliminary Version of a Denotational Framework for Comparing Models of Computation," EECS, University of California, Berkeley, CA, Memorandum UCB/ERL M96/33, June 1996.

[5] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation, Second Edition*, 2nd ed. New York, NY: Academic Press, January 2000.

[6] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong, "Taming Heterogeneity - The Ptolemy Approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, January 2003.

[7] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. IFIP Congress*, ser. Information Processing, no. 74, August 1974, pp. 471–475.

[8] A. Benveniste, P. Caspi, R. Lublinerman, and S. Tripakis, "Actors Without Directors: A Kahnian View of Heterogeneous Systems," Verimag, Research Report 2008-6, September 2008.

[9] J. D. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, May-Jun 2007.

[10] D. Hellmann, "Multi-Processing Techniques in Python," *Python Magazine*, vol. 1, no. 10, October 2007. [Online]. Available: http://www.doughellmann.com/articles/pythonmagazine/completely-different/2007-10-multiprocessing/index.html

[11] J. Nutaro, "Discrete-Event Simulation of Continuous Systems," in *Handbook of Dynamic Systems Modeling*, P. A. Fishwick, Ed. Boca Raton, FL: Chapman and Hall/CRC, June 2007, ch. 11. [Online]. Available: http://dx.doi.org/10.1201/9781420010855.ch11

[12] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, September 1979.

[13] S. Jafer and G. Wainer, "Conservative DEVS - A Novel Protocol for Parallel Conservative Simulation of DEVS and Cell-DEVS Models," in *Proc. 2010 Spring Simulation Conference (SpringSim10)*, Orlando, FL, April 2010, pp. 168–175.

[14] E. A. Lee, "Disciplined Heterogeneous Modeling," in *Proc. ACM/IEEE 13th International Conference on Model Driven Engineering, Languages, and Systems (MODELS)*. IEEE, October 2010, pp. 273–287. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/679.html

[15] A. Basu, M. Bozga, and J. Sifakis, "Modeling Heterogeneous Real-time Components in BIP," in *Proc. 4th IEEE International Conference on Software Engineering and Formal Methods*, Pune, September 2006, pp. 3–12.

[16] J. Streb, G. Kimmell, N. Frisby, and P. Alexander, "Domain Specific Model Composition Using a Lattice Of Coalgebras," in *Proc. 6th OOPSLA Workshop on Domain Specific Modeling*, Portland, OR, October 2006.

[17] The MathWorks, "Simulink," 2010. [Online]. Available: http://www.mathworks.com/products/simulink/

[18] INRIA Metalau Project, "Scicos: Block diagram modeler/simulator," 2010. [Online]. Available: http://www.scicos.org/

[19] K. Müller, T. Vignaux *et al.*, "SimPy," 2010. [Online]. Available: http://simpy.sourceforge.net/

[20] R. Clewley *et al.*, "PyDSTool," 2010. [Online]. Available: http://sourceforge.net/projects/pydstool/

Allan McInnes *is a lecturer in Electrical and Computer Engineering at the University of Canterbury. His research interests are in the areas of networked embedded systems, heterogeneous system simulation, and mobile robotics.*

Brian Thorne *is a mechatronics engineer from the University of Canterbury. His research interests are in machine learning, computer vision, concurrent systems and simulations.*