ScipySim: Towards Distributed 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 distributed heterogeneous system simulation capability for the SciPy scientific computing platform. It began as an experiment in implementing Caspi *et al.*'s generalized Kahn theory for executable heterogeneous system semantics. 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 decentralized simulation system is, in principle, well-suited to distributed execution. However, in practice, simultaneously achieving efficiency and liveness proves to be difficult. We describe the current design of ScipySim, some of the difficulties we have encountered in implementing a simulator using the generalized Kahn approach, and plans for the future.

1. INTRODUCTION

ScipySim is a heterogeneous system simulator developed using the Python programming language [1]. A heterogeneous system is composed of subsystems that operate in a variety of domains or implement different models of computation (e.g., discrete-time, continuous-time, discrete-event, or Petri nets). ScipySim originated as an exploration of the practicalities of implementing a heterogeneous system simulator based on the generalized Kahn theory of heterogeneous system semantics proposed by Caspi *et al.* [2], and has helped to illuminate several difficulties with that approach. We are now working to evolve ScipySim into a distributed simulation tool running on top of the SciPy [3] scientific computing platform.

Our choice of Python and SciPy as our development platform was motivated by Python's support for rapid implementation of complex software, and the useful numerical and signal processing functions provided by SciPy. Python is freely available, runs on a wide variety of platforms, includes a large number of library modules, and supports bindings to many GUI toolkits. SciPy, and its underlying NumPy package, provide a library of numerical and scientific functionality that includes access to mature code available from netlib and other public-domain repositories [4]. SciPy is becoming increasingly popular as a scientific and engineering platform in both academia and industry, often filling the same niche as MathWorks' MATLAB®. However, although SciPy allows basic simulation of linear time-invariant systems it lacks anything resembling MathWorks' Simulink® tool [5], and few other Python-based simulation tools are available. None of the Python simulation tools of which we are aware support heterogeneous simulation. ScipySim is intended to provide such a capability to SciPy users.

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.), and outline some of the problems we have encountered (sec. 3.3.) in trying to implement Caspi *et al.*'s proposal. We begin with a brief overview of the tagged signal model and the generalized Kahn theory that was developed from it (sec. 2.), and close with a brief review of related work (sec. 4.).

2. BACKGROUND

Lee and Sangiovanni-Vincentelli developed the "tagged signal model" [6] as a denotational framework for describing the behavior of heterogeneous systems. 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 lead to different models of computation. The resulting formalism is similar to the I/O relation framework of Zeigler *et al.* [7], 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 [8], can in principle be understood in terms of the tagged signal model, but in practice their execution is often governed by quite different semantics.

In an effort to bring the denotational and operational semantics of heterogeneous system simulation closer together, Caspi *et al.* [2] proposed a generalization of Kahn Process Networks [9] 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 its input queues. Heterogeneous systems are constructed by connecting subsystems having different tag sets by means of tag-conversion actors that convert signals from one tag set to another. The generalized Kahn approach thus avoids the need to incorporate Ptolemy-style directors, and describes a system entirely in terms of actors. Caspi *et al.* claim that their approach does not require centralized execution control, naturally supports heterogeneous models, and provides "distribution for free".

3. SCIPYSIM

Caspi *et al.* [2] give several examples of executable actors based on their generalized Kahn theory. Further examples, and a Haskell simulator prototype, appear in a technical report by Benveniste *et al.* [10]. However, the simulator prototype is a limited proof-of-concept, and does not explore issues such as distributed execution. Our goal for ScipySim was to move beyond a proof-of-concept and examine how the generalized Kahn approach works as the basis for a simulator.

3.1. Simulator Design

ScipySim is implemented in standard Python. While we currently use only basic SciPy functionality, such as simple functions and linear spaces, the rest of the SciPy package is available for developing new actors. Graphical output, such as line graphs and stem plots, is presently produced using the matplotlib library [11], although this is shielded from the user and can be changed to support other libraries.

ScipySim consists of a core set of classes that define basic simulation entities such as `Actors`, `Events`, `Channels`, and `Models`, and a library of `Actor` subclasses that implement specific components. The current actor library includes:

- Signal source actors (constant, ramp, step, and sinusoid).
- Mathematical actors (sum, 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 encapsulate the translation of signals into plots.

```
class Actor(object):
    def __init__(self, input_channel=None,
                       output_channel=None):
        # Initialization of internal fields
    def start(self):
        self.thread = Thread(target=self.run)
        self.thread.start()

    def join(self): self.thread.join()

    def run(self):
        while not self.stop:
            self.process()

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

Figure 1. ScipySim `Actor` class

3.1.1. Events

Following the tagged signal model, `Event` objects contain a tag and a value. The `Event` class makes `Event` objects immutable, which prevents simulation errors resulting from actors modifying a shared `Event`. Thus the event `e = Event(1.0, 3.5)` is created 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`) causes 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

Caspi *et al.* claim that the generalized Kahn approach naturally supports asynchronous actor execution. Thus, in our current implementation every `Actor` contains a Python thread. The `run()` method of the `Actor` class (Fig. 1) specifies the behavior of the thread. At present, `Actor.run()` simply continually calls the `process()` method, which is consistent with Caspi *et al.*'s notion of "chaotic" actor execution. Changes in the simulation protocol can be introduced by modifying `Actor.run()`. Due to inheritance, these modifications flow to all derived actor components. However, some protocols may require more far-reaching changes to the simulator design (see sec.3.3.).

The `process()` method represents a single actor execution step. The `Actor` class is abstract, and does not implement `process()`. Specific actor behavior is defined in derived classes that implement an appropriate `process()` method. The typical form of a `process()` method is to wait until an event is available on all input channels, then compute an output event that is sent through any output channels. For example, a single-input/single-output actor would have a `process()` method like that shown in Fig. 2.

```
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

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.

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()` 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 [2].

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 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

A model is a collection of actors and the channels that connect them. In ScipySim models are defined by creating a derivative of the `Model` class that specifies the actors and channels that form the model (see Fig. 4 for an example). The derived `Model` class only specifies the model structure. Indeed, the text of a derived `Model` class can be seen as a model structure description that happens to use Python syntax, and thus is not tied to any particular execution approach. However, through inheritance any derived `Model` object 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.

```
class Model(Actor):
    def __init__(self, *args, **kwargs):
        # Initialization of internal fields
    def process(self): pass

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

Figure 3. ScipySim `Model` class

Although a `Model` is an `Actor`, it does not define a `process()` method. Instead, as shown in Fig. 3, the behavior of a `Model` is generated by overriding the `run()` method to start execution of the component actors contained within the `Model`. Thus the execution of a model is determined entirely by the execution of its constituent actors.

3.2. Simulator Operation

The current implementation of ScipySim does not include a separate simulation program that regulates the execution of a simulation model. Instead, execution of the simulation is effectively self-regulating due to the nature of generalized Kahn theory [2]. Actors execute when they are able to, and otherwise block awaiting input events. As a consequence, a simulation model is both a definition of the `Actors` and connecting `Channels` that make up the system to be simulated, and an executable Python program that carries out the simulation.

Fig. 4 illustrates the basic elements 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

The connections between actors are defined by passing the appropriate channels to each actor as it is created. To enable execution of the model, all of the actors are added to the `components` list.

```
class SinRampSum( Model ):
    def __init__( self ):
        ch1, ch2, ch3 = MakeChans(3)
        src1 = Ramp(ch1)
        src2 = CTSinGenerator(ch2,
                    amplitude=1.0, freq=1.0)
        sum = Summer([ch1, ch2], ch3)
        dst = Plotter(ch3, title="Sin+Ramp")
        self.components = [src1, src2,
                           sum, dst]

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

Figure 4. An example of a simulation model

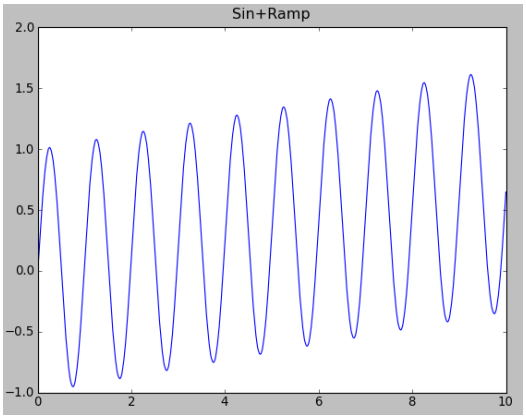

Figure 5. The output of the example simulation model

As described in sec. 3.1.4., calling `run()` on a `Model` causes all actors contained in the model to start executing. Execution of the simulation is driven by the source actors, which autonomously produce events in accordance with the kind of signal they represent. The other actors run until they receive a terminal event from a source or from another actor, at which point they output terminal events, then terminate. The simulation ends when all actors have terminated. The simulation output produced by the `Plotter` is shown in Fig. 5.

3.3. Discussion

ScipySim goes beyond the proof-of-concept described by Benveniste *et al.* [10]. We have implemented the core classes described in sec. 3.1., and a small library of actors modeling signal sources and basic signal operations in the discrete-time and continuous-time domains. In addition to the simple model of Fig. 4 we have created a collection of other models to test and demonstrate various aspects of ScipySim, including a pulse-width modulated signal generator, a numerically integrated ballistic trajectory, and a 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. In attempting to implement a simulator based on the generalized Kahn approach we have learned some useful lessons about how well that approach works in practice. Caspi *et al.* and Benvensite *et al.* describe their approach as a generalization of Kahn Process Networks, and present a few simple example actors and models with a sequential prototype simulator. Our concurrent simulator and more complex models help to illuminate the fact that the generalized Kahn approach can also be seen as a form of conservative distributed discrete-event simulation [12] in which each actor is a "logical process" and tag-conversion actors manage the interactions between different kinds of time-bases. In implementing ScipySim we have encountered problems with synchronization and deadlock that are well-known to the distributed discrete-event simulation community, although they were not apparent in the simple examples provided by Benveniste *et al.* These problems belie claims that the generalized Kahn approach provides "distribution for free", and indicate that the generalized Kahn semantics contains some unresolved flaws. As a consequence, we are now working to evolve ScipySim away from its roots as a simulator based entirely on the generalized Kahn approach, and towards a more general tool for distributed heterogeneous system simulation.

3.3.1. Continuous-Time

Many of the problems we have encountered in implementing ScipySim manifest themselves in the continuous-time domain. Benveniste *et al.* [10] discuss some preliminary ideas for modeling continuous-time systems in their generalized Kahn framework. They present a proof-of-concept scheme that performs step-adaptive Euler integration 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 or the inclusion of integrators in larger systems of actors. Furthermore, they note that their approach has difficulties with detecting threshold-crossing events since it requires a variable time-step integration scheme that is able to roll back the integration when a threshold is missed. Such integration schemes are not Kahn-order preserving, and are difficult to implement in an asynchronous simulation.

The current implementation of ScipySim includes a first-order, fixed-step numerical integration actor. This allows simple open- and closed-loop continuous-time simulation, but is inefficient for systems that require small time-steps and cannot be used for exact threshold-crossing detection. Consequently, we are experimenting with discrete-event numerical integration schemes [13] derived from the DEVS formalism, which offer variable time-step integration in an asynchronous

context. Such schemes also resolve the difficulties faced by Benveniste *et al.* in handling threshold-crossing events, since they naturally progress from one threshold to another.

A drawback of the 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. The same potential for deadlock also exists in some of the discrete-event actors described by Benveniste *et al.*, although it is neither remarked upon nor addressed. Both the fixed-step and discrete-event integration schemes also suffer from problems with causality. Feedback loops that contain only zero-delay elements can result in situations where time does not advance. At present we leave it to the user to break these loops by manually inserting a delay into the loop. A number of alternatives for dealing both with deadlock and causality have been developed by the distributed discrete-event simulation community [12, 14, 15], but implementing these alternative is likely to require changes to the way actors are implemented.

3.3.2. Actor Implementation

Although we have so far implemented only a small library of actors operating in a limited set of domains, the existing library has already helped us to identify some problems with the generalized Kahn style of simulation. These problems are driving us to consider alternative simulation protocols. The separation between actor execution behavior (the `run()` method) and actor model definition (the `process()` method) means that alternative conservative execution schemes, such as those involving deadlock recovery or synchronous execution [12], can be implemented by modifying the `Actor` base class and the `Channel` class to manage control signals and barriers, without requiring changes to existing actor implementations or models. However, implementing optimistic protocols, or combined conservative/optimistic protocols such as that developed by Praehofer and Reisinger [14], may require changes to our actor design to support rollback and memory management. We are presently investigating what changes may be required.

Making each actor have its own thread has allowed us to experiment with concurrent execution of generalized Kahn networks. However, while we have not yet made a comprehensive performance evaluation, the naive thread-per-actor approach appears to add overhead to simulations. We are planning to move to a scheme that uses green threads layered over a process pool. This approach eliminates the overhead of native threads, enables us to avoid the bottleneck of the Python "Global Interpreter Lock" [16], and allows us to implement distributed simulations.

4. RELATED WORK

There are few tools that support simulation development in Python, and none that we are aware of support heterogeneous system simulation. The most prominent Python-based simulation tools appear to be SimPy and PyDSTool. SimPy is a sophisticated discrete-event simulation tool for Python [17]. PyDSTool is a simulation environment for dynamical systems [18], focusing on ordinary differential equations and differential-algebraic equations. Although both tools are far more mature than ScipySim, neither project appears to have plans for supporting simulation of heterogeneous systems.

Although the semantics of heterogeneous system simulation is still an active area of research [19, 20], a number of simulation tools for heterogeneous systems are already available. The MathWorks' Simulink® tool [5] is probably the most widely used of these tools, although a number of other tools are available [21]. Most of these tools focus on simulating hybrid systems, i.e. systems containing both continuous-time dynamics and discrete state transitions, rather than more general modeling of heterogeneous systems made up of many different models of computation.

The closest tool in spirit to ScipySim is Ptolemy II [8, 20], which supports a wide variety of models of computation. We originally envisioned ScipySim as differing from Ptolemy by building on the simplified semantic model proposed by Caspi *et al.*, and taking advantage of "distribution for free" that was claimed to come with the generalized Kahn approach to develop a multithreaded simulator. Unfortunately, our initial results show that distribution is not as free as it seemed, while Ptolemy now has support for multithreaded execution through the ThreadedComposite actor [22]. However, ScipySim continues to be more fundamentally based on a concurrent execution model than Ptolemy II, which should help with developing more complex distributed simulations.

5. CONCLUSIONS

The goal of the ScipySim project is to develop a heterogeneous system simulation capability for the SciPy scientific computing platform. Our initial work on ScipySim was aimed at developing a simulator based on Caspi *et al.*'s generalized Kahn theory [2], which was claimed to offer a simplified semantic account of heterogeneous systems and a decentralized execution architecture supporting "distribution for free". In doing so, we have found that implementing a simulator based on the generalized Kahn approach results in difficulties with synchronization and deadlock that are well-known to the distributed discrete-event simulation community, making distributed simulation less free than Caspi *et al.* claimed. However, the existing implementation of ScipySim does allow the simulation of models involving various domains, as our example models demonstrate. ScipySim thus provides a basic simulation capability for researchers wishing to develop simulations that are programmed in Python or incorporate functionality from SciPy. We are presently working to modify the underlying ScipySim actor implementation to support dis-

tributed execution, and to incorporate ideas from the discrete-event simulation community to support efficient distributed execution of heterogeneous system simulations.

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] P. Caspi, A. Benveniste, R. Lublinerman, and S. Tripakis, "Actors Without Directors: A Kahnian View of Heterogeneous Systems," in *Proc. 12th Int. Conf. on Hybrid Systems: Computation and Control*, San Francisco, CA, 2009, pp. 46–60.

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

[4] T. Oliphant, "Python for Scientific Computing," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 10–20, 2007.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*. New York, NY: Wiley-Interscience, 2000.

[13] 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.

[14] H. Praehofer and G. Reisinger, "Distributed Simulation of DEVS-based Multiformalism Models," in *Proc. 5th Annual Conf. on AI, Simulation, and Planning in High Autonomy Systems*, Dec. 1994, pp. 150–156.

[15] J. Nutaro and H. Sarjoughian, "Design of Distributed Simulation Environments: A Unified System-Theoretic and Logical Processes Approach," *SIMULATION*, vol. 80, no. 11, pp. 577–589, 2004.

[16] D. Hellmann, "Multi-Processing Techniques in Python," *Python Magazine*, vol. 1, no. 10, October 2007.

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

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

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

[20] E. A. Lee, "Disciplined Heterogeneous Modeling," in *Proc. 13th Int. Conf. on Model Driven Engineering, Languages, and Systems*, Oslo, Norway, October 2010, pp. 273–287.

[21] L. Carloni, R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli, "Languages and Tools for Hybrid Systems Design," *Foundations and Trends in Electronic Design Automation*, vol. 1, no. 1/2, July 2006.

[22] E. A. Lee, "ThreadedComposite: A Mechanism for Building Concurrent and Parallel Ptolemy II Models," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-151, Dec 2008.

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, and system simulation.*