Composing Different Models of Computation in Kepler and Ptolemy II

Antoon Goderis[1], Christopher Brooks[2], Ilkay Altintas[3], Edward A. Lee[4], and Carole Goble[5]

[1] School of Computer Science, University of Manchester, UK
goderisa@cs.man.ac.uk
[2] Department of EECS, UC Berkeley, USA
cxh@eecs.berkeley.edu
[3] San Diego Supercomputer Center, UC San Diego, USA
altintas@sdsc.edu
[4] Department of EECS, UC Berkeley, USA
eal@eecs.berkeley.edu
[5] Shool of Computer Science, University of Manchester, UK
carole@cs.man.ac.uk

Abstract. A model of computation (MoC) is a formal abstraction of execution in a computer. There is a need for composing MoCs in e-science. Kepler, which is based on Ptolemy II, is a scientific workflow environment that allows for MoC composition. This paper explains how MoCs are combined in Kepler and Ptolemy II and analyzes which combinations of MoCs are currently possible and useful. It demonstrates the approach by combining MoCs involving dataflow and finite state machines. The resulting classification should be relevant to other workflow environments wishing to combine multiple MoCs.

Keywords: Model of computation, scientific workflow, Kepler, Ptolemy II.

1 The Need for Composing Models of Computation in E-Science

E-scientists design on-line (in silico) experiments by orchestrating components on the Web or Grid. On-line experiments are often orchestrated based on a scientific workflow environment. Scientific workflow environments typically offer support for the design, enactment and provenance recording of computational experiments.

Most workflow environments fix the model of computation (MoC, or the formal abstraction of computational execution) available to an e-scientist. They leave little flexibility to change MoC as the experiment evolves. Different experiments are modeled more cleanly with different MoCs because of their relative expressiveness and efficiency. Different uses of MoCs for scientific workflows include dataflow for pipeline compositions, e.g. gene annotation pipelines; continuous-time ordinary differential equation solvers, e.g. for Lattice-Boltzmann simulations in fluid dynamics; and finite state machines for modelling sequential control logic, e.g. in clinical protocols or instrument interaction.

Y. Shi et al. (Eds.): ICCS 2007, Part III, LNCS 4489, pp. 182–190, 2007.

There are also scenarios where a combination of MoCs is useful, e.g. a mixture of a time dependent differential equation model with dataflow. Most environments do not support experiments that mix multiple MoCs. This interferes with intra and inter-disciplinary collaboration**.** For example, in genomic biology, gene annotation pipelines provide useful input to systems biology simulation models. Candidates for drug development found in cheminformatics simulations are plugged into bioinformatics annotation pipelines to retrieve the candidates' hazardous interactions within cells. The inability to mix MoCs also makes it more difficult to mix software workflows with physical systems such as sensor networks and electron microscopes, which have continuous dynamics. Moreover, mixing specialized MoCs for visualization (e.g. for animation) with, for example, time-based simulation, makes for more efficient execution and for better models. In addition, if we can mix MoCs, then we can design workflows that manage the execution of models and workflows. Representative use cases include: (i) selective extraction and analysis of proteins from public databases, combining finite state machines and dataflow and (ii) dynamically adapting model control parameters of Lattice-Boltzmann simulations in fluid dynamics by combining finite state machines and continuous-time ODE solvers. In such scenarios, using an integrated environment that supports mixing MoCs enables integrated provenance collection**.** In the fluid dynamics example, the provenance includes dynamic changes in the overall model as well as parameter sweeps within each model, covering the full range and variability.

2 Paper Contribution and Overview

To date, little is known about how models of computation are joined. Kepler, which is based on Ptolemy II, is a scientific workflow environment that allows for MoC composition. The paper explains how MoCs are combined in Kepler and Ptolemy II, and analyzes which combinations of MoCs are possible and useful. The resulting classification should be relevant to other environments wishing to combine MoCs.

Kepler/Ptolemy II comes with a wide range of MoCs, which are implemented as directors. Section 3 introduces the notion of hierarchy as the key concept for mixing MoCs in a workflows. Section 4 provides an overview of MoCs in Kepler/Ptolemy II. For a scientific workflow developer, determining which MoC combinations are legal is non trivial. Section 5 establishes MoC compatibility, based on the notion of actor abstract semantics and presents a classification of MoCs combinations. Section 6 discusses the validity of the approach and demonstrates successful and unsuccessful combinations of dataflow and finite state machines. We conclude in Section 7.

3 Workflows and Hierarchy

Ptolemy II [2] is a Java-based environment for heterogeneous modeling, simulation, and design of concurrent systems. Ptolemy II forms the core of Kepler [5], an environment for building scientific workflows. The focus of Ptolemy II is to build models based on the composition of components called *actors* [1]. Actors are encapsulations of parameterized actions performed on input tokens to produce output

tokens. Inputs and outputs are communicated through ports within the actors. They provide the common abstraction used to wrap different types of software components, including sub-workflows, Web and Grid services.

The interaction between the actors is defined by a Model of Computation. The MoC specifies the communication semantics among ports and the flow of control and data among actors. *Directors* are responsible for implementing particular MoCs, and thus define the "orchestration semantics" for workflows. By selecting the director, one selects the scheduling and execution semantics of a workflow. Many actors can work with several directors, adapting their behaviors to match the semantics of the director [5]. The models of computation implemented in Ptolemy as directors are described in detail in [2, Vol. 3]. A subset of them, including dataflow, time and event dependent directors, is available in Kepler. Key to mixing MoCs in a workflow is the notion of hierarchical abstraction. Figure 1 shows a Kepler chemistry workflow using the PN director, which implements a process networks MoC [7]. This workflow contains a *composite actor* (a.k.a. sub-workflow) named Babel. The implementation of Babel actor is another workflow that contains another director, the SDF director, which implements a synchronous dataflow MoC. This example mixes two MoCs in a single, hierarchical workflow.

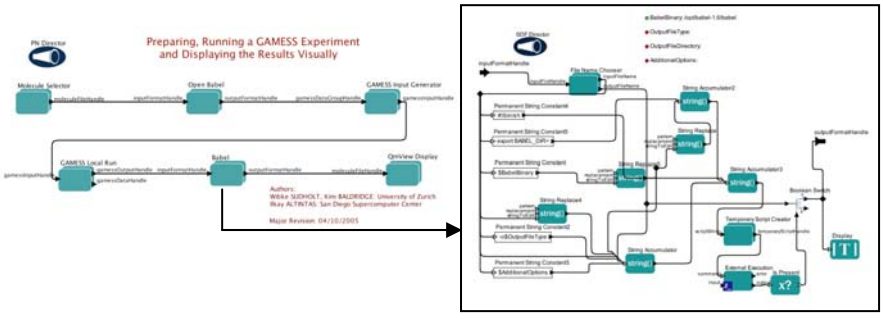

Fig. 1. A Kepler workflow from chemistry combining the PN and SDF director [7]

In Ptolemy/Kepler, hierarchy can serve either of two roles. First, it can be simply an organizational tool in building workflows, permitting a workflow designer to aggregate portions of a workflow and create conceptual abstractions. In this usage, the composite actor does not contain a director, and is called a transparent composite actor. The hierarchy has no semantic consequences; it is just a syntactic device. A second use of hierarchy is to use a workflow to define an actor. The Babel example is of this type. The fact that it has a director makes it function within the top level workflow exactly as if it were an *atomic actor*. A composite actor that contains a director is called an *opaque composite actor*, because its internal structure is neither visible nor relevant to the outside director.

For an opaque composite actor to function externally as if it were an ordinary actor, the director must be able to execute the inside workflow in a manner that emulates an actor. We examine below what that means, but before we can do that, we explain a few of the MoCs in enough detail that they can serve as illustrative examples.

4 Models of Computation in Ptolemy II and Kepler

One of the main objectives of the Ptolemy Project has been the exploration of models of computation. For this reason, many distinct directors have been created by various researchers, some realizing fairly mature and well-understood models of computation, and some that are much more experimental. Kepler has adopted Ptolemy's rich MoC architecture and focused principally on a few of the more mature ones, described here.

Process Networks (PN). In PN, each actor executes in a Java thread, and all actors execute concurrently. An actor can read input data (which is encapsulated in tokens) from input ports, and write data (wrapped in tokens) to output ports. Normally, when it reads from an input port, the read blocks until an input token is available. Writes do not block. The PN director includes sophisticated scheduling policies to ensure that buffers for tokens remain bounded, and also detects deadlock, which is where all actors are blocked attempting to read data. See [3] and [6]. Most of the scientific workflows (composite actors) built with Kepler to date have been based on PN.

Dataflow (DDF and SDF). In dataflow MoCs, instead of having a thread associated with each actor, the director "fires" actors when input tokens are available to them. We discuss two variants of dataflow here, dynamic dataflow (DDF) and synchronous dataflow (SDF). In the case of DDF, the director dynamically decides which actor to fire next, and hence constructs the firing schedule dynamically at run time. In the case of SDF, the director uses static information about the actor to construct a schedule of firings before the workflow is executed, and then repeatedly executes the schedule. SDF is very efficient in that very little decision making is made at run time. PN is semantically a superset of DDF, in that the repeated firings of an actor in DDF can be viewed as (or even implemented as) a thread. Every DDF workflow can be executed using a PN director. DDF in turn is a superset of SDF, in that every SDF workflow can be executed identically with a DDF director. In SDF, a fixed number of tokens are consumed and produced in each firing. The token consumption and production rates allow for the computation of a fixed schedule. In SDF, deadlock and boundedness of communication buffers are decidable. With DDF, actors need not have a fixed token production or consumption rate, the schedule is determined at runtime. In DDF, deadlock and boundedness are not decidable. In a DDF model, an actor has a set of firing rules (patterns) and the actor is fired if one of the firing rules forms a prefix of unconsumed tokens at the actor's input ports.

Continuous Time (CT). In CT, the communication between actors is (conceptually) via continuous-time signals (signals defined everywhere on a time line). The CT director includes a numerical solver for ordinary differential equations (ODEs). A typical actor used in CT is an Integrator, whose output is the integral from zero to the current time of the input signal. The CT director advances time in discrete steps that are small enough to ensure accurate approximations to "true" continuous-time behaviour. For reasons of space we omit discussion of **Discrete Events (DE)** and **Synchronous/Reactive (SR),** two other mature time-based directors.

Finite State Machines (FSM) and Modal Models. An FSM composite actor is very different from the above. The components in an FSM composite actor are not actors, but rather are states. The FSM director starts with an initial state. If that state has a

refinement, then the FSM director "fires" that refinement. It then evaluates guards on all outgoing transitions, and if a guard evaluates to true, then it takes the transition, making the destination state of the transition the new current state. A state machine where the states have refinements is called a Modal Model. A *Modal Model* is an opaque composite actor containing an FSM, each state of which may contain an opaque composite actor. In a modal model, the refinement of the current state defines the current behavior of the state machine. The refinement of a state need not have the same type of director as the workflow containing the modal model.

There are many other MoCs implemented in Ptolemy II, but the above set is sufficient to illustrate our key points. Akin to choosing between programming languages to tackle a problem, often different directors can be chosen to model a given phenomenon. A suitable director does not impose unnecessary constraints, and at the same time is constrained enough to result in useful derived properties (such as efficient execution or deadlock detection). The misinformed use of directors also leads to actors that cannot be embedded in others, as explained in the next section.

5 Composing Models of Computation in Kepler/Ptolemy II

Although prior work has offered formalisms for comparing MoCs, e.g. [4], MoC compositions have not been well treated. To address the void, we develop a classification of valid MoC combinations in Kepler/Ptolemy II.

In the Kepler environment, opaque composite actors can be put into workflows with a different type of director, thereby combining different models of computation in one workflow. In the workflow in Figure 1, the Babel actor is part of a network of actors orchestrated by the PN director. The Babel actor internally uses an SDF director. In the example, SDF is nested inside PN, which is a valid combination, as we will explain below. Nesting PN inside of SDF would have been invalid in most cases, however. The choice of director determines whether a given actor can be put on the inside or outside of other actors.

To determine which combinations are possible, we need to know two things about a director:

1. What properties it assumes of the actors under its control, and
2. What properties it exports via the opaque composite actor in which it is placed.

If a director's exported properties match those assumed by another director, then it can be used within that other director. Otherwise, it cannot. In the example of Figure 1, the SDF director exports properties that match those assumed by the PN director, and hence SDF can be used inside PN. The properties in question can be formulated in terms of actor abstract semantics.

5.1 Actor Abstract Semantics

All models of computation in Kepler and Ptolemy II share a common abstraction that we call the actor abstract semantics. Actors and directors are instances of Java classes

that implement the Executable interface, which defines *action methods*. The action methods include two distinct initialization methods:

1. preinitialize(): invoked prior to any static analysis performed on the workflow (such as scheduling, type inference, checking for deadlock, etc.).
2. initialize(): invoked to initialize an actor or director to its initial conditions. This is invoked after all static analysis has been performed, but it can also be invoked during execution to reinitialize an actor.

The action methods also include three distinct execution methods that are invoked in sequence repeatedly during an execution of the workflow:

3. prefire(): invoked to check whether an actor is ready to fire (for example, an actor may return false if there are not enough input data tokens).
4. fire(): In this method, the actor should read input tokens from input ports and write tokens to output ports, but it should not change its state. That is, if the fire() method is invoked repeatedly with the same input tokens, then the resulting output tokens should be the same.
5. postfire(): In this method, the actor can read input tokens and update its state.

Finally, there is a finalization method:

6. wrapup(): invoked for each actor just prior to finishing execution of a workflow.

All of the methods are required to be finite (they must eventually return).

The method definitions specify a contract, but not all actors obey this contract. Any actor that strictly conforms to this contract is said to be *domain polymorphic*, and the actor may be used by any director that operates on actors (which is all the directors above except FSM, which operates on states).

Actors that do not obey the contract are more specialized, and may only work with specific directors. They are not domain polymorphic (strictly obeying the actor abstract semantics) and come in two flavors. The first flavor obeys a looser version of the abstract semantics where the fire() method provides no assurance that the state of the actor is unchanged. The second is still looser in that it that also provides no assurance that any of these methods is finite. Based on these three levels of conformance to actor abstract semantics, we can now classify the directors.

5.2 Abstract Semantics Assumed by a Director of the Actors Under Its Control

The **PN** director only assumes the loosest of these abstract semantics. It does not require that any method be finite because it invokes all of these methods, in order, in a thread that belongs entirely to the actor. If an actor chooses to run forever in the preinitialize() method, that does not create any problems for the director. The director will let it run. **Dataflow** directors require that actors conform with the loose actor semantics, where all methods are finite. But they do not require that actors leave the state unchanged in the fire() method. **CT** requires that actors obey the strictest form of the semantics. The director iteratively fires actors until some condition is satisfied. The strict actor semantics ensures that the answers will always be the same given the same inputs. **FSM** requires loose actor semantics. A firing of an FSM in Ptolemy II consists of a firing of the refinement of the current state (if there is one), followed by

evaluation of the guards and a state transition. Clearly, the firing of the refinement must be finite for this to be useful.

5.3 Abstract Semantics Exported by a Director Via the Actor in Which It Is Placed

A director also implements the *Executable* interface. If a director conforms to the strict actor semantics, then an opaque composite actor containing that director also conforms to the contract. Such an actor can be used safely within any workflow. In the current version of Ptolemy II (version 6.0), only the **SR** director conforms to the strict actor semantics, although in principle **CT** and **DE** can be made to conform. Currently these and the **dataflow** directors conform to the looser abstract semantics, but still guarantee that all methods return after finite time. **PN** only conforms to the loosest version, providing no guarantees about methods ever returning. The **FSM** director exports whatever the state refinements export.

5.4 Director Compatibility

We classify directors according to two criteria. They require that the actors they control are either *strict*, *looser*, or *loosest*, depending on whether they must conform with the strictest, looser, or loosest form of abstract semantics. They similarly export either *strict*, *looser*, or *loosest*. Ideally, any director should export the same version of the contract it assumes or a stricter version, but this is not the case in currently.

The current status of the directors is given in Table 1. The rule applied to determine director compatibility is that exported abstract semantics should be stricter than or equal to required abstract semantics. The table is on-line and will evolve (see http://www.mygrid.org.uk/wiki/Papers/IccsPaper2007).

Table 1. Rules for hierarchically mixing directors in Kepler and Ptolemy II

Inner director ↓ (**exports X**)	Outer director ↓ (*requires Y*)				
	PN (*loosest*)	SDF (*loose*)	DDF (*loose*)	CT (*strict*)	FSM (*loose*)
PN (**loosest**)	Yes	No	No	No	No
SDF (**loose**)	Yes	Yes	Yes	No	Yes
DDF (**loose**)	Yes	Yes	Yes	No	Yes
CT (**loose**)	Yes	Yes	Yes	No	Yes
FSM (**refinement**)	Yes if the refinement is stricter than or equal to *Y*				

6 Discussion of PN, Dataflow and FSM Directors

A key question may arise at this point. If actors can be made domain polymorphic by conforming to the strict actor semantics, then why not design all directors to conform?

In some cases, the semantics of the MoC precludes this. In other cases, it would simply be too costly. We examine some of these cases.

PN. The PN director is apparently the least restrictive in the actors it can manage, but also the least useful in an opaque composite actor. The reason for this is very fundamental. If the PN director were to define a finite fire() method, what should that method do? Each of the actors under its control is executing in its own thread of control. How much execution should be performed? One possible answer is "as little as possible," but this would result in nondeterminate execution. If two actors have threads that are able to perform computation, which should be allowed to perform computation? The only other obvious answer is "as much as possible." This can be made determinate, but typical PN workflows can execute forever if given the opportunity. Hence, this yields an infinite execution. PN is sufficiently expressive that determining whether this execution is infinite is equivalent to solving the famous halting problem in computation, and hence is undecidable.

For example, the workflow of Figure 1 with PN on the outside would be hard to reuse inside others. It follows that, when a workflow has potential to be reused inside others, PN should be avoided and, if possible, replaced by a more reusable director.

DDF. DDF is as expressive as PN, and hence potentially suffers from the same limitation. However, DDF has an advantage. It assumes that all actors under its control have finite firings. Thus, it is relatively easy for the designer of a workflow to specify how many firings of the component actors constitute a single firing of the enclosing opaque composite actor. The DDF director assumes a simple default if these numbers are not given by the workflow designer: one firing of a DDF opaque composite actor constitutes at most one firing of each component actor. The actor is fired if possible, and not fired if not, given the available input data. This yields a simple, finite, and determinate notion of a finite firing for the director to export.

SDF. SDF is still simpler in that it is not as expressive as PN, and there is a simple unique finite firing that is natural and easy to define. However, for both DDF and SDF, it is difficult to define a fire() of an opaque composite actor that does not update the state of the workflow because data values stored on buffers change during the firing of the component actors. In order for SDF and DDF to export the strict actor semantics, they would have to backtrack or restore the state of these buffers on repeated invocations of the fire() method.

FSM. A particularly interesting case is FSM and modal models. Modal models always use opaque composite actors as state refinements, and these must at a minimum have finite firings to be useful (given the semantics of an FSM in Ptolemy II discussed before). Hence, it does not make sense to use PN inside the refinement of a state. But any other of the directors described above can be used in a Modal Model.

7 Conclusions

There are scenarios in e-science that rely on composing models of composition. Based on the notion of hierarchy and actor abstract semantics, we give a classification of models of computation available in Kepler and Ptolemy II. The classification shows

which combinations are useful. Notwithstanding several restrictions, many combinations are possible. Time-based simulations can be mixed with dataflow and finite state machines can be combined with almost anything. Our exploration of combinations of models of computation should be useful for other e-science systems.

Acknowledgments. Thanks to B. Ludaescher, T. Feng, G. Zhou, and J. Brooke. A. Goderis visited Kepler/Ptolemy based on the Link-Up grant EPSRC GR/ R67743.

References

1. G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, (1986).
2. C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng (eds.) "Heterogeneous Concurrent Modeling and Design in Java", Vol. 1-3, Tech. Report UCB/ERL M05/21, University of California, Berkeley. July 15, (2005)
3. G. Kahn and D. B. MacQueen, "*Coroutines and Networks of Parallel Processes*," Information Processing 77, B. Gilchrist, editor, North-Holland Publishing Co., (1977).
4. E. A. Lee and A. Sangiovanni-Vincentelli, "*A Framework for Comparing Models of Computation*," IEEE Transactions on CAD, Vol. 17, No. 12, December (1998).
5. B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, Y. Zhao, "*Scientific Workflow Management and the KEPLER System*," Concurrency & Computation: Practice & Experience, Special issue on scientific workflows (2005).
6. T. M. Parks, *Bounded Scheduling of Process Networks*, UCB/ERL-95-105, University of California, Berkeley, December (1995).
7. W. Sudholt, I. Altintas, K.K. Baldridge, "*A Scientific Workflow Infrastructure for Computational Chemistry on the Grid*," 1st Int. Workshop on Computational Chemistry and Its Application in e-Science in conjunction with ICCS (2006).