A PRIMITIVE EXECUTION MODEL FOR HETEROGENEOUS MODELING

Frédéric Boulanger
Supelec – Département Informatique
3 rue Joliot-Curie, 91192 Gif-sur-Yvette cedex, France
Email: Frederic.Boulanger@supelec.fr

Guy Vidal-Naquet
Supelec and Université Paris-Sud
3 rue Joliot-Curie, 91192 Gif-sur-Yvette cedex, France
Email: Guy.Vidal-Naquet@supelec.fr

Keywords: Heterogeneous modeling, Models of computation, Execution models.

Abstract: Heterogeneous modeling is modeling using several modeling methods. Since different modeling methods are used in different crafts, heterogeneous modeling is necessary to build a heterogeneous model of a system that takes the modeling habits of the designers into account.

A model of computation is a formal description of the behavioral aspect of a modeling method. It is the set of rules that allows to compute the behavior of a system by composing the behaviors of its components.

Heterogeneous modeling allows different parts of the system to be modeled using different models of computation. Therefore, some parts of the system may obey some rules while other parts obey other rules for the composition of their behaviors.

However, computing the behavior of a system which is modeled using several models of computation can be difficult if the meaning of each model of computation, and what happens at their boundary, is not well defined. In this article, we propose an execution model that provides a framework of primitive operations that should allow to express how a model of computation is interpreted to compute the behavior of a model of a system. When models of computation are “implemented” in this execution model, it becomes possible to specify exactly what is the meaning of the joint use of several models of computation in the model of a system.

1 CONTEXT

The design of most complex systems appeals to different crafts that are organized around sets of design methods. These methods are adapted to specific aspects of a craft: there are methods for designing the control of industrial plants, others for designing signal processing algorithms and others for modeling packets that travel across a network. What is important is that the people who use a design method have an intuition of its functioning that matches its effective semantics. Therefore, in order to avoid errors in the design of a system, it is better to allow each specialist of an aspect of the system to work with the tools he is used to. Imagine that you have to design a system that computes mathematical expressions using a tool in which the addition is written ‘×’: you will make much more errors than if you are allowed to use a tool that respects your habits and you intuition of the meaning of mathematical operators.

There are many cases where, in the same company, the same system has as many models as there are teams working on it. Each team needs to model the system or a part of it with its usual modeling tools, but the different models are disconnected and there is no way to guarantee their consistency.

When integrating the different parts of a system, we generally translate each part into a common low level formalism, or even into a common implementation language like C, Java or Ada. However, by doing so, we loose all the information that tells how we went from the specification of the subsystem to its model. Therefore, when building the whole system, we cannot take advantage of the different choices of realization offered by the model, since they have been “frozen” in the low level implementation. For instance, the high-level model may describe two independent tasks that may be executed in any order, but happen to be executed one after another in the implementation, and it might be very difficult to see that the order in which they are executed can be changed, just by looking at the implementation.

Another issue is that, when validating the behavior of the whole system, it will be difficult to find what should be changed in the model of a subsystem to make a global property of the system true, since the
low level implementation does not carry enough information about the design of the subsystem.

Heterogeneous modeling tries to overcome these issues by allowing to describe the whole system as a composition of subsystems that are designed according to different methods (\cite{noteref}). Another way to say this is that heterogeneous modeling allows to compute the behavior of a system by composing the behaviors of its subsystems even when they are expressed using different models of computation.

Heterogeneous modeling does not provide a greater expressive power than other modeling techniques, but it allows the different teams that work on the design of a system to share a common model of the system while using their usual modeling techniques, instead of developing their own separate models of the system.

2 ACTORS, PORTS, PROPERTIES AND RELATIONS

Our approach to modeling is based on the actor paradigm: a system is built from components named actors. Actors have properties and communicate through ports. Ports have properties and are linked by relations which also have properties. So, building a model of a system amounts to using some actors, to set their properties and the properties of their ports, and to build relations between the ports of the actors. The effective behavior of the model is obtained by interpreting the properties and the relations according to a model of computation.

These are not actors in the strict meaning of the actor model defined by Gul Agha in the 1980s (\cite{noteref}) since our actors may or may not have their own flow of control and may communicate through other means than asynchronous message passing. However, they can be considered as actors since they are autonomous opaque entities that provide services by communicating with the actors to which they are connected (their acquaintances).

Actors, ports, properties and relations are the elements of the abstract syntax of actor-oriented modeling. Models of computation are semantics for this abstract syntax: a model of computation is a way to interpret the relations between the ports of actors and the properties of these relations. For example, the model on figure 1 may be interpreted either as a two state automaton with a transition triggered by the occurrence of symbol “i”, or as a network of two dataflow operators, the first one producing i data samples each time it is triggered.

In the first case, each actor represents a state of the automaton, relations represent transition between states, and the behavior of the system is the behavior of its current state. In the state-machine model of computation, each actor has two ports: an input port which is used for transitions that lead to the state, and an output port which is used for transitions that leave the state. The relation that represents a transition has several properties: guard, actions to be executed when the transition is taken, initializations for the target state (to ensure continuity in hybrid systems for instance).

In the second case, each actor represents an operator which consumes data from its input ports and produces data to its output ports. Relations denote connections between inputs and output an therefore describe the flow of data between operators. Depending on the exact semantics of the model of computation, properties of the relations may indicate how many data samples are exchanged atomically, or properties of the ports may tell how many data samples are produced or consumed atomically on a port.

3 ROLES OF A MODEL OF COMPUTATION

A model of computation allows to compute the observable behavior of a model of a system from the individual observable behaviors of its components. For instance, on figure 2, the MoC_{ext} model of computation compute the behavior of the top-level model that contains actors A and B. This model of computation is in charge of computing the status (availability of data and value) of B_{in} from the status of A_{out}. The status of A_{out} is determined from the status of A_{in} by the behavior of A. In the example, this behavior is described by a model that contains three actors and is governed by the MoC_{int} model of computation.
MoC\textsubscript{ext} is the “external model of computation” of actor A. It is the model of computation that combines its behavior with the behavior of other actors to compute the behavior of a model. MoC\textsubscript{int} is the “internal model of computation” of A. It is the model that computes the behavior of A by combining the behaviors of the actors of the model of A.

Information is produced by observing the output ports of actors, so the model of computation must tell how information goes from output to input ports, and when it is available on input ports. We can consider the first aspect (routing data from output ports to input ports) as communication, and the second aspect (telling when data is available) as control or synchronization.

The communication aspect consists generally in copying data samples from output ports to input ports. In models of computation where several output ports may be in relation with a single input port, some operation may be performed to combine several data samples into one. For instance, the presence of several data samples at the same time on the same channel could produce a “collision” data sample on the corresponding input port, or it could produce a multiplexed sample.

The synchronization aspect can be more complex because it defines the type of causality used by the model of computation. In some models of computation, data produced on an output port is available immediately on all input ports that are in relation with that output port. In other models of computation, a notion of “tick” is used to relax causality: a data sample produced on an output port is available on all inputs ports that are in relation with it, but only at or after the next tick. In other models of computation, a notion of time is introduced to label data samples with a time-stamp which tells the date at which (or after which) they are (will be) available.

The communication and synchronization aspects must be described precisely to define a model of computation. There are actually very few tools or languages for describing models of computation in this way. Most of the time, one has to implement a model of computation in a generic language like C++ or Java. Projects like KerMeta \textsuperscript{(7)} or Rosetta \textsuperscript{(8,9)} are related to our works, but focus on different objectives: KerMeta defines behaviors for the elements of the Meta-Object Facility (MOF) of the OMG, while Rosetta defines the combination of models of computation and uses a hierarchy of compatible models of computation. Our objective is to provide a framework in which any model of computation may be used to compute the interactions of components or subsystems of a system. In this paper, we focus on the steps that are required to compute the behavior of a model of a system.

4 EXECUTION MODEL

Even if the precise semantics of a model of computation still has to be coded in a generic implementation language, this is generally done in the context of a framework like Ptolemy II \textsuperscript{(10)} that provides support for the abstract syntax of the models, editors for graphical concrete syntaxes (data-flow diagrams, state-machines), and parsers for textual concrete syntaxes. In order for such a framework to support an open set of models of computation, it must consider them as black boxes that compute the availability and values of data on input ports from the availability and values of data on output ports. A model of computation may also produce a schedule for the observation of data on output ports since it knows the causality relations between outputs and inputs. However, the framework can provide a generic execution model that can be used with any model of computation.

In the following, we define such a generic execution model that relies on models of computation to determine in which order the actors of a model should be observed, and how the values on input ports are computed form the values observed on output ports. This execution model has matured from previous works \textsuperscript{(7,8)} based on the Ptolemy framework. It is a first step toward the definition of a meta-model for the modeling of models of computation based on the Eclipse Modeling Framework \textsuperscript{(7)}.

4.1 What is an execution?

Before defining our generic execution model, we must define what we call an execution of a model of a system. In order not to compromise the generic aspect of our approach and its ability to work with any model of computation, we deliberately ignore the internal mechanisms of an actor. What is important is what an actor produces on its output ports and when, not how it produces it. The “how” is important only when we need to compute the behavior of this actor, that is to say, when we have to execute its internal model.

Therefore, in the following, we will never “trigger” the behavior of an actor, we will just observe its output ports. The behavior of an actor can occur at any-time and may be a continuous process that runs during the whole execution of the model, but it must provide us with a coherent view each time we observe its ports. Since a model is composed of several actors, we must have a way to observe them simultaneously and we call such an observation a snapshot.

We consider that an execution of a model is a sequence of snapshots of the values available on the ports of the model. In a given snapshot, each port has one well defined value (it cannot have an undefined value and cannot take several values during a given snapshot).
The exact nature of a snapshot depends on the model of computation. The execution model only tells the actors that a new snapshot is going to be taken, and then makes them approve the value of their ports as they appear on the snapshot. To make an analogy with photography, we first tell the actors “stay still”, and then we ask them if they are pleased by the resulting photograph.

This definition of an execution implies that we are only interested in discrete behaviors. This is because our goal is not to describe behaviors, but to compute them. For instance, even when we consider a model of a physical system, as a system of ordinary differential equations, we are not interested in finding properties of these ODEs but in computing the value of outputs of the system at a discrete set of instants.

4.2 Observations versus activities

We define the execution of a model in terms of observations instead of activities because:

- we don’t need to know how an actor computes its output to be able to use these outputs;
- ignoring how an actor computes its outputs allows to abstract out the model of computation used to model its internal behavior. Each model of computation is insulated from the others, and there is no need to define the composition of any pair of models of computation.

When we tell an actor that a snapshot is beginning, all we need to know is that we can now observe its outputs. We don’t need to know if the behavior of this actor runs in a separate thread of control or if it was only activated by our telling of the beginning of the snapshot. To consider an extreme case, if an actor is a sensor which acquires information from the external world, all we want to know is the result of the measurement, not how the real world has influenced the sensor to produce this result.

4.3 Types of actors

To define a generic execution model, we must consider the different ways by which an actor may produce its outputs. The simplest kind of actors needs to know the value of all its inputs to determine the value of all its outputs at once. We call such actors strict actors. For instance, an actor that computes the sum of its inputs must know the value of all of them before being able to determine the value of its output. With strict actors, a model cannot contain instantaneous causality loops because a strict actor cannot have an input that depends (even indirectly) on the value of one of its outputs in the same snapshot. It is therefore possible to observe the outputs of each actor only once, in an order that is compatible with the causality relation induced by the connections between ports, and to determine the value of all ports by propagating the values of outputs to the inputs according to the model of computation.

Other actors can determine some of their outputs when they know the value of only some of their inputs. We call such actors non-strict actors. A very simple example of a non-strict actor is a delay: the value of its output depends only on its state, which in turn depends on the value that its input had in the previous snapshot. A logical OR gate is also a non-strict actor since when one of its input is true, it can set its output to be true without knowing the value of the other input.

When a model of computation supports non-strict actors, the values of the ports are determined iteratively. First, actors are provided with known inputs and they determine part of their outputs. These newly determined outputs allow to compute new values for inputs according to the model of computation. These newly determined inputs allow actors to determine more outputs, and so on until all the ports have a known value. With such models of computation, it is necessary to tell actors when new inputs become available.

Some actors, independently of their strict or non-strict nature, may not agree with the value they have computed for their outputs for the current snapshot. An example of such an actor is a level-crossing detector. It monitors a signal and produces an event when this signal crosses a threshold. If the signal is computed by numerical integration of differential equations, the integration step is adjusted so that the value of the signal is computed with a given precision. However, the level-crossing detector is interested in the precision on the date at which the signal crosses the threshold. If the signal varies slowly, the integration step required to obtain a precise enough value of the signal may be much larger than the precision required on the date of the level-crossing event. Therefore, when the signal crosses the threshold, the level-crossing detector will produce an event, but it will also detect that this event should have been produced between the previous snapshot and the current one, and that the time elapsed between these snapshots is greater than the precision required for the date of the event. The only solution is to execute the snapshot again, but with a smaller integration step. Therefore, a snapshot is considered valid only if all the actors of the model agree with the value assigned to their ports.
5 THE GENERIC EXECUTION MODEL

The taxonomy of actors presented above, and the fact that we are only interested in observations of the ports of actors, not in the activity of the actors, allows us to define a generic execution model that we believe is capable of executing models that obey any reasonable model of computation. To attain such universality, we made as few assumptions about actors as possible, and we rely on an operational description of the model of computation to schedule observations and to compute the value and availability of data on input ports from the data available on output ports.

The previous sentence may seem strange since we are used to have outputs computed from inputs, not the reverse. The key is to consider that if actors produce their outputs from their inputs, the model of computation interprets the relation between ports to determine what is available on inputs ports from what is available on output ports.

We can now describe the steps that will be taken by our execution model to compute a snapshot of the execution of a model of a system and define the primitive operations that an actor must provide:

1. the start_of_snapshot operation is invoked on each actor of the model. In response to this invocation, an actor prepares for the snapshot. For instance, an actor that acquires information from the environment of the system (reading data from a file, sampling a sensor) should do it during this step.

2. the reset operation is invoked on each actor of the model. In response to this invocation, an actor should reset all its outputs to the "unknown" state.

3. the update operation is invoked on each actor of the model. If the model of computation is able to compute a schedule for the actors, the actors are processed according to this schedule. In response to this invocation, an actor makes data available on its output ports. If the actor is strict, it makes data available on all its output ports. If it is non-strict, it makes data available only on the ports it can determine.

4. the operational description of the model of computation is used to compute the status (availability of data and value of the data) of the input ports from the status of the output ports.

5. if the value of all ports has been determined, go to step 6, else, go back to step 3.

6. the validate_snapshot operation is invoked on each actor of the model. In response to this operation, an actor considers the data available on its ports as definitive for this snapshot and tells whether it considers it as correct or not. If it does not validate the data, it should change some property of the model of computation (e.g. the integration step in our example with the level-crossing detector) so that a new computation of the snapshot will compute data that it may validate.

7. if all the actors of the model have validated the snapshot, go to step 8, else go back to step 2 to compute the snapshot again with the new parameters of the model of computation that have been set by the actors which have not validated the snapshot.

8. when all the actors of the model have validated the snapshot, the end_of_snapshot operation is invoked on each actor of the model. This operation tells the actors that the snapshot is valid and that they can use the data available on their ports in their own activity or to update their internal state if any. Actors that provide data to the environment of the system (writing data to a file, driving an actuator) should do so during this step.

In order for this execution model to be correct, it is very important that an actor does not update its internal state, change its activity or perform any operation that may have side effects on the environment between the start_of_snapshot and end_of_snapshot operations. The fact that a snapshot may be computed several times to converge toward a result that is accepted by all the actors of the model must not be visible outside the model. For instance, if we consider our example of a level-crossing detector, it may be necessary to compute a snapshot several times before the integration step becomes smaller than the precision required on the time-stamp of the level-crossing event, but outside the model, only the last level-crossing event must be visible because it is the only one that has been considered as correct. For the same reason, data should not be acquired several times from the environment when the snapshot is computed several times.

5.1 Discussion on the steps

The overall structure of our execution model is shown on figure 3, with "actor operations" in rectangular boxes, "model of computation operations" in ellipses, and control choices in diamond shaped boxes.

The schedule operation of the model of computation sorts the actors of the model in an order that is compatible with the way data is produced and made available in this model of computation. It may be a no-op if the actors are self-scheduling and update their outputs as soon as data becomes available on their inputs. The order returned by the schedule operation is used during the update step. The schedule operation is invoked before each occurrence of the update step, but if static scheduling is possible, the
The first and last steps of the computation of a snapshot insulate the environment of the model from the internal changes that occur in the model during the computation of the observation of its ports. They also insulate the behavior of the actors of a model from the details of the computation of a snapshot of this model, since an actor is not allowed to update its internal state before the end_of_snapshot step. For instance, an actor should not count the number of times its reset, update or validate methods are invoked and make its future behavior depend on this count.

We can consider the start_of_snapshot step as “sample the external world”, and the end_of_snapshot as “update state, act on external world”. Between these two steps, actors must have a combinational behavior. This model is therefore very close to the synchronous sequential model where registers are loaded on the ticks of a clock with the results of combinational computations. In our execution model, the clock is the series of instants at which a snapshot exists. We do not need a more elaborate model of time at this level of the execution of a model of a system. We consider that time, when needed, is to be handled by the model of computation, either as a global property of each snapshot, or as a property of each data sample on a port.

The reset – validate loop is crucial for heterogeneous modeling because it is the way by which the model of computation that is used for the internal model of an actor can influence the model of computation in which the actor is used. In a model of a system, actors are just observed, and the observations are combined by a model of computation to build an observation of the model. However, the behavior of each actor can also be described by a model of the actor (the actor is considered as a subsystem), and the model of computation used to model the behavior of the actor can be different from the first model of computation. We call “external model of computation" the model of computation that is used to combine the behavior of an actor with the behavior of other actors, and “internal model of computation” the model of computation that is used to describe the internal behavior of an actor.

In order to avoid to compute all the possible combinations of models of computation, we hide the internal model of computation to the eyes of the external one. Since the external model of computation “decides” when the ports of an actor are observed, the internal model of computation would have no control on the computation of the snapshot if it could not refuse a computation by making the actor return false to the validate request.

The update loop implements a well-known technique to compute the behavior of a model as a fixed-point. It has been implemented in Ptolemy since version II by the prefire, fire and postfire methods. prefire is the equivalent of start_of_snapshot in our execution model, fire is equivalent to update and postfire to end_of_snapshot. However, we chose different names for these operations since there is no validate – reset loop in the general execution model of Ptolemy II (even if such a validation steps exists in the “Continuous Time” model of computation), and the names of these methods denote the activation of a behavior. Our execution model deals only with observations and we do not limit the behavior of actors to the body of a fire method.

6 HETEROGENEITY

The execution model we have just presented here uses only one model of computation, so one may wonder how heterogeneous models are handled. Our approach of heterogeneity is the same as the hierarchical approach used in Ptolemy (1), and our execution model does not depend on the models of computation used to compute the behavior of the actors of a model. It is therefore possible to define the behavior of actors using internal models of computation that differ from their external model of computation.

An issue still subsists: how data produced according to the internal model of computation of an actor will be interpreted in the context of its external model
of computation? The behavior of an actor may be expressed using properties that have no meaning in the external model of computation. For instance, an actor may produce time-stamped data samples because its behavior is defined using a timed model of computation, and these samples may be read in a model of computation that has no notion of time. In this case, the series of time-stamped data samples can be viewed as a sequence of data samples just by discarding the time-stamps, but in the reverse case, when data with no time-stamp is produced in a timed model of computation, a time-stamp must be created for each data sample, and this requires additional information.

Our position is that there is no automatic way to convert data (or control) from a model of computation to another. Often, there are standard ways of adapting the semantics of two models of computation (for instance, periodic sampling can be used to go from continuous time to synchronous data-flow), but such transformations should not be hard-coded in the modeling framework nor applied implicitly to an heterogeneous model. The reasons for this are:

- implicit transformations are framework-dependent. This means that the same model could adopt different behaviors when executed in different modeling frameworks;
- several transformations between two models of computation may exist (for instance, when going from discrete to continuous time, it is possible to hold the last value, or to use linear or more complex interpolation). The choice of a transformation is part of the design of the system, and it should therefore appear explicitly in the model;
- even when there is only one possible transformation between two models of computation, using this transformation and setting its parameters is a design choice, and it should appear in the model of the system, with the same importance as the models of computations.

The main problem with such transformations is that if they are implemented as actors, these actors appear either in the internal or in the external model of computation. Both ways are wrong since they break modularity: if the internal model of an actor contains actors to adapt data to its external model of computation, this internal model depends on the external model of computation. This means that the design of an actor depends on the context in which it will be used. The same problem occurs when the adapting actors are placed in the external model of computation. In (2), we presented a model for domain-polymorph components that allows the adaptation between two models of computation to be done at the interface between the models. This approach turns the adaptation between the semantics of the internal and external models of computation into a property of the edge of the actor.

A last issue is that it is sometimes necessary to define actors which obey several models of computation. For instance, a sampler has a continuous input, a discrete event input (the sampling clock) and a data-flow output (the sequence of samples). A level-crossing detector has a continuous or sampled input and a discrete event output. Such actors cannot be handled directly in our execution model because only one model of computation is allowed in the model of a system. However, we have shown in (2) that a flat heterogeneous model, i.e. a model that uses several models of computation at the same level of its hierarchy, can be rewritten automatically into a hierarchical model by projecting heterogeneous actors on the models of computation they use. One may also consider that such behaviors should not me modeled as actors but as transformations between models of computation, and considered as properties of the edge of models, as evoked earlier.

7 CONCLUSION

We have presented the roles of a model of computation and the different kinds of actors it should be able to manage, and then an execution model which, by making as few assumptions as possible about actors, seems to be able to execute models that obey any reasonable model of computation. There can be no formal proof of this, but our works on the integration of the reactive synchronous approach into object-oriented programming and on the adaptation between models of computation in the Ptolemy framework make us quite confident in the universality of this model. By considering only observations on the ports of actors, and not the activity of actors, this execution model can safely ignore what happens at lower levels of the hierarchy of a model. This allows the use of different models of computation at different levels of the hierarchy of a model of a system. Moreover, by allowing an actor to veto the result of the computation of a snapshot of the model, this execution model allows inner models of computation to interact with the outer models, in addition to the usual control that the outer model has on the inner models of computation.

This execution model is the foundation of a framework that allows system designers to model each part of a system using the most suitable model of computation, and to specify explicitly how these models of computation interact to produce the behavior of the system.

This execution model requires that an actor supports a set of 5 primitive operations which mark the beginning and the end of the computation of a snapshot, the initialization of the computation of a snapshot, the update of the snapshot when new informa-
tion becomes available, and the validation of the snapshot when all data has been determined.

This execution model also requires that a model of computation is able to provide it with a schedule (eventually trivial) of the actors of a model, and to propagate to the input ports the data observed on the output ports. These two operations can be complex, and are, for the moment, implemented using generic programming languages like Java or C++. The lack of a formal description of the operations of a model of computation makes it difficult to define transformations from a model of computation to another and to handle heterogeneity in a more generic way. Our current works aim at such a description using either an extended version of the Object Constraint Language (OCL) or the Action Language of UML 2.