# Implementing Metropolis Quantity Managers in Ptolemy II

Haibo Zeng

EECS 290N Report
December 17, 2004

University of California at Berkeley
Berkeley, CA, 94720, USA

zenghb@eecs. berkeley.edu

## Abstract

*This paper proposes an approach to implement Metropolis quantity manager mechanism in Ptolemy II, by utilizing the AspectJ which adds to Java aspect-oriented programming (AOP) capabilities. The benefits and limitations of this approach are also discussed.*

## 1 Introduction

Ptolemy II is a set of Java packages supporting heterogeneous, concurrent modeling and design. It is mainly focusing on the research related to embedded software. Therefore, it emphasizes very much on the execution (simulation) of different models of computation[2]. Metropolis, on the other hand, will be addressing synthesis (software/hardware), verification and architecture exploration in addition to simulation. Currently, what is available in Metropolis design environment includes the Metropolis meta-model (MMM) language, a Metropolis compiler, a simulation backend, and a SPIN based formal verification backend[1].

AspectJ [4] is a simple and practical extension to the Java programming language that adds to Java aspect-oriented programming (AOP) capabilities. AOP allows developers to reap the benefits of modularity for concerns that cut across the natural units of modularity. In object-oriented programs like Java, the natural unit of modularity is the class. In AspectJ, aspects modularize concerns that affect more than one class. The user can compile her/his program using the AspectJ compiler and then run it, supplying a small runtime library. The AspectJ technologies include a compiler (ajc), a debugger (ajdb), a documentation generator (ajdoc), a program structure browser (ajbrowser), and integration with many Java tools such as Eclipse, Sun-ONE/Netbeans, GNU Emacs/XEmacs, JBuilder, and Ant.

## 2 Metropolis Quantity Manager Mechanism

One of the key innovations in Metropolis is the notion of *quantity managers*[1]. An important question to be addressed in a design is to specify performance constraints, and to do it, behaviors must first be annotated with metrics, such as time, power, or other Qualities of Service. In the Metropolis meta-model, such physical annotations are defined with a concept called *quantity*. In addition, quantities is also defined for something like the arbitration of shared resources. Each quantity has an associated type, for example, double for time. For each quantity there is an object called a *quantity manager* which is responsible for assigning annotation to behaviors, or restricting the execution of the behaviors.

Quantity managers contain codes that are invoked when a request is made to annotate a process transition with the quantity it is responbile. Such requests are made, for example, on certain points in the program which is defined as *events*, such as the beginning or ending of the execution of a statement or a communication action over a medium. A formal definition of events is given in Section 3. As an example, a special case of quantity manager is the global time manager, which is used to implement a discrete event semantics, where events are processed in chronological order. To model that the delay between two events $e_1$ and $e_2$ is 10, the process makes a request that a time-stamp of $e_2$ must be equal to the time-stamp of $e_1$ plus 10. It is the responsibility of a quantity manager to collect all requests and satisfy them. If a request cannot be satisfied, the manager must

disable the event for which that request was made. For example, if one process wants to execute event $e_1$ and request for it time-stamp 10, and the other process wants to execute $e_2$ with time-stamp 20, time manager must set the current time to 10, let $e_1$ occur, and disable $e_2$. This example also demonstrates that quantity managers not only annotate events, but also determine which events should occur, i.e. they schedule the execution of the model. That is why the collection of quantity managers (and some other related objects) is called the *scheduling network*, and the network it schedules is referred to as the *scheduled network*.

In addition to imperative implementation of a quantity manager, the user may also use certain logic formula to express constraints on the sequences of scheduled network execution. In this way, a given sequence of annotated state transitions is a legal behavior of a meta-model restriction if and only if:

1. it satisfies all the constraints specified by logic formulas,
2. it can be generated by the execution of the scheduled network restricted by the imperative code of scheduling network.

### 2.1 Operational Semantics

In Metropolis, processes in the scheduled network are active, in that each of them has its own thread of control. A key reason for this choice is the desire to model several aspects of a system, including particularly its performance when mapped to hardware resources. The separate threads of control model parallel hardware systems well. A natural question to implement the quantity manager mechanism in this type of system is when and how to let the quantity managers annotate and restrict the execution of the processes and media in the scheduled networks. In Metropolis, the operational semantics of quantity manager mechanism is the quantity request/resolution network execution model, which defines the following two-phase execution sequence of a network containing a scheduled network and a scheduling network:

1. *Quantity Request*: each process in the scheduled network runs to an event $e$ (meaning this process is stalled there), and finds all the quantity constraints on $e$.
2. *Quantity Resolution*: find a set of candidate events and quantities annotated with each of these events, such that all the axioms of the quantity types are satisfied.

To generate an annotation request for an event it is associated with, some piece of codes is inserted within the enclosure of this event. This type of codes is also called *request making code*, or *RM code* for short. It typically involves some computation, and then a call to the *request()* function of some quantity manager. Thus in Metropolis, a quantity manager has to implement the interface function

*void request(event e, RequestClass rc);*

where *RequestClass* is a container used to package requests for various quantity managers.

A quantity manager in Metropolis also needs to implement functions

*void resolve();*

*void postcond();*

*booelan isStable();*

Typically, in the *resolve()* function, a quantity manager looks at the pending requests, and decides if they can or cannot be granted. If the request cannot be granted, the managers disables the corresponding events. For example, if several events request a time-stamp, the time manager must set the current time to the lowest of all request, and it must disable all the events requesting a higher time-stamp.

Repeatedly calling *resolve()* functions of all the quantity managers will decrease the number of enabled events. Eventually, there will be a single enabled event for each process. At this point, the vector of events that will occur has been set, and the quantity managers can assign annotations to these events. This is the primary of the function *postcond()*. Other typically uses of this function is to clean up data used in the resolution.

Since many managers need to cooperate in selecting an event vector that can be annotated consistently with all made requests, the function *resolve()* is often called many times. If during a particular call, the quantity manager disables some event, or makes some annotation, then a subsequent call to function *isStable()* should return **false**; otherwise it returns **true**. In other words, *isStable()* is useful to decide if the resolution process has converged, i.e. it has reached the point at which further calls to *resolve()* will not change enabled events.

## 3 Implementing Quantity Managers in Ptolemy

Basically the quantity manager mechanism involves specification of system level quantitative concerns. In Object-Oriented language such as ordinary Java, it is difficult to modularize design concerns like system-wide error-handling, distribution concerns, feature variations, and context-sensitive behaviors[4]. The code for these concerns tends to be spread out across the system. Because these concerns won't stay inside of any one module boundary, meaning that they *crosscut* the system's modularity. In Metropolis meta-model, to support exploring design concerns in quantity manager mechanism which are typically crosscutting, some new concepts and built-in syntax are added, such as *action*, *events*, and *Request-Making code*,

as briefly introduced in Section 2. However, this will require the insertion of event declaration and related request-making codes in a process, which may break the *domain-polymorphism*[2] of the actors in Ptolemy. AspectJ adds constructs to Java which is the implementation language of Ptolemy II, thus enabling the modular implementation of crosscutting concerns[4] and keeping the actors domain-polymorphic.

The remaining of this section is organized as following: in Section3.1 the necessary constructs of AspectJ to support implementation of quantity manager mechanism are explained and compared with similar concepts in Metropolis. In Section 3.2, an implementation of quantity manager mechanism in Ptolemy utilizing AspectJ is proposed, and the limitations of this proposal is discussed in Section 3.3.

### 3.1 AspectJ Constructs

In Metropolis, a typical request-making code inserted in a process are like the following syntax:

*labela*{$
[**beg**{$< begin\_code >$}]
[{statement}]
[**end**{$< end\_code >$}]
$}

By their position in the code, these inserts are always associated with an *action* labelled as *labela*. More precisely, *actions* are executions of the pieces of code in the scheduled network. Function calls to media are predifined observable actions, and each statement in the code of the scheduled network can be claimed as actions by labelling it as in the example. With each action $a$ there are two type of *events* associated with it, $a^+$ indicates the start of an execution of $a$, and $a^-$ indicating the end. The request-making code $< begin\_code >$ is associated with the begin event of the action *labela*, and $< end\_code >$ associated with the end event. Both of these codes make annotation requests for their respective events.

Similar to the concept of *actions* in Metropolis, in AspectJ a *join point* is a well-defined point in the program flow, such as reading or writing a field, calling or executing an exception handler, method or constructor. A *pointcut* picks out certain join points and values at those points. A piece of *advice* is the code executed before, after or around a join point is reached. As the name suggests, *before advice* runs before the join point executes, which is similar to the begin event and its associated request-making code in Metropolis. *After advice* executes after the join point, which is similar to the end event and its associated request-making code. The power of advice comes from its ability to access values in the corresponding join points, although advice is in different modular class from these joint points.

### 3.2 Implementing Quantity Manager using AspectJ

In Ptolemy[3], a similar concept to Quantity Managers is *Directors*. A composition of actors is guided by a director, which represents a specific *Model of Computation(MoC)*. In Ptolemy II, a Model of Computation is also called a *domain*. A director may control the execution of actors through an *Executable interface*. A director, together with all *receivers* which is contained by input ports and implements the communication mechanisms among actors, defines a *framework*. To obey a specific model of computation, a director and receivers must match.

An important Ptolemy framework design choice is *hierarchical heterogeneity*. This approach constrains each level of interconnected actors to be locally homogeneous, while allowing different models of computation to be specified at different levels in the hierarchy. A well-defined model of computation at the same level improves the understandability of the system, and may allow certain parts of the system to be correct by construction, because of the formal properties obtained by that specific MoC.

Unfortunately this approach is inconsistent with the assumption under quantity manager mechanism, thus disabling most of its power. In quantity manager mechanism, it allows multiple quantity managers in the same hierarchical level. Each individual quantity manager can be written independently from each other, without knowing the existence of the other quantity managers. The cooperation among quantity managers are done through the current set of enabled events, which is stored as the state of the scheduled network and can be accessed by the scheduling network through the connection between them. In each call of the *resolve()* function, a quantity manager looks at the pending requests associated with the current set of enabled events, and decides if they can or cannot be granted. If the request cannot be granted, the quantity manager disables the corresponding events, meaning it will delete these events from the state of the scheduled network. The following resolution of quantity managers will realize the decision of the previous ones by basing their decision on the current set of enabled events only. The design choice in Metropolis is to keep more power than hierarchical heterogeneity by allowing multiple quantity managers in the same hierarchical level, assuming that the user has a good understanding of the system she/he wants to model and follows the guideline of quantity manager mechanism.

To implement the Metropolis quantity manager mechanism unchanged in Ptolemy, it is natural to keep the interface functions a quantity manager must implement as in Section 2.1 and implementation of each quantity manager the same as in Metropolis. In addition, an AspectJ program is written to take care of the rest of the work, which in-

cludes:

1. Mimic the event naming and associated request-making codes by pointcuts and advices
2. Record and update the state of each actors
3. Alternate between quantity request/resolution phases

### 3.3 Discussion

There are certain limitations to this approach imposed by the different design choices in Metropolis and Ptolemy. In Ptolemy, the abstract semantics of control is based on the *Executable interface*, and the abstract semantics of communication is based on the *IOPort/Receiver interface*. The abstract semantics that binds most Ptolemy domains (with the exception of PN and CSP) is significantly different from Metropolis. Instead of processes, components are actors with three-phase firings. The objective of this abstract semantics is to be able to define *domain polymorphic* actors, meaning that they can interact with other components within a wide variety of domains. As can be seen, different domains impose different requirements for actors. Some actors, however, can work in multiple domains. These actors are called domain-polymorphic actors. The Metropolis meta-model is more fixed about control semantics, i.e. components are process based, but allows a richer interface to communications media, since a media can expose arbitrary interfaces to components. This design choice makes communication refinement easier since the interface to a media can change, but makes it more difficult to build domain-polymorphic components since media often have different interfaces. Metropolis also emphasizes process-oriented concurrency that often exists at the top-level of a system. The quantity manager mechanism fits well with the Metropolis framework, for example, to disable an event, the quantity manager just stalls the thread of the corresponding process.

Metropolis quantity managers can disable events, thus stalling the thread. But not all the domains in Ptolemy are thread-oriented, such as SDF, Giotto, and DE. In these domains, the meaning of stalling a thread is ambiguous, since there might be only one thread in the whole model. The only thread-oriented domains in Ptolemy are PN, CSP, and DDE.

Second, in Ptolemy components are actors with three-phase firings. To keep a consistent state among actors, it is not always safe or possible to stall their execution even if they have their own thread of control. In many models of computation, there is a natural time[3]: between iterations, which is before *prefire()* or after *postfire()*. These are the only types of safe advice the user can assume for actors in all Ptolemy domains.

Please notice that this discussion is only imposed on the quantity managers that may stall the execution of actors. For quantity managers that only annotate performance numbers to events such as a power manager which computes the power consumption of the system, the approach in Section 3.2 has no limitation on the applicable domains or joint points.

In PN domain, actors can have a richer set of joint points: all threads must be stalled on read, or on write to full buffers, or block themselves with a wait() function. The current implementation of SDF domain in Ptolemy is single-threaded. As an example, an alternative is to take a model in the multi-thread PN domain, which is also compatible with SDF domain, i.e. statically schedulable. An AspectJ program is written to stall the actors before *prefire()*; And the sequence of actor iteration follows the execution queue the SDFScheduler in the kernel of Ptolemy, meaning that each time an actor finishes its iteration, the next actor in the execution queue gets notified.

## 4. Conclusion

This paper proposed an approach utilizing AspectJ to implement the quantity manager mechanism in Ptolemy which allows multiple quantity managers coordinate in the same hierarchical level, without touching the code of the actors in Ptolemy II, thus keeping the design property in Ptolemy II: domain-polymorphic actors.

Certain limitations are also imposed by different design choices in Metropolis and Ptolemy: this approach is safely applicable only to thread-oriented domains i.e. PN, CSP, DDE and large granulated joint points in the actors.

## References


[1] The Metropolis Project Team. The Metropolis Meta Model Version 0.4. Technical Memorandum No. UCB/ERL M04/38, University of California, Berkeley, CA 94720, USA, September 14, 2004.

[2] Edward A. Lee. Overview of the Ptolemy Project. Technical Memorandum No. UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2, 2003.

[3] Jie Liu, Johan Eker, Xiaojun Liu, John Reekie, and Edward A. Lee. Actor-Oriented Control System Design: A Responsible Framework Perspective. IEEE Transactions on Control System Technology, Pages 250-262, Volume 12, Issue 2, March, 2004.

[4] AspectJ Project Website. http://eclipse.org/aspectj/, the AspectJ Team.