Toward an error handling mechanism for timing errors with Java Pathfinder and Ptolemy II

Shanna-Shaye Forbes

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-123
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-123.html

September 7, 2010

Copyright © 2010, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

This work was conducted with support from a Jenkins Pre-doctoral Fellowship Mini Grant Award.

Toward an error handling mechanism for timing errors with Java Pathfinder and Ptolemy II

Shanna-Shaye Forbes*
University of California, Berkeley, CA 94702

Project Mentor: Johann Schumann†
SGT/NASA Ames, Moffit Field, CA 94035

Designing effective error handling systems in an embedded software system is essential for acceptable and reliable functionality in cases of errors and for the recovery from faults. Errors in the error handling system can cause catastrophic failures of the software, lead to mission failures and can endanger human life. We take a principled approach of extending a model of computation (MoC) with timing semantics for embedded systems by an error handling mechanism for timing errors in model-based design. As a apart of the mechanism we use Java PathFinder (JPF), a model checker developed at NASA Ames. This work presents our attempt during a summer project as we use JPF along with the Ptolemy II modeling and simulation framework, developed at UC Berkeley, with the goal of speeding up the design process of a correct and adequate error handling mechanism for timing errors for a model of computation with timing semantics.

I. Introduction

It is extremely important that one identifies and is capable of handling error cases before deploying embedded software. In an effort to avoid possible mistakes with a newly designed NASA rocket or spacecraft, we take a principled approach of extending a model of computation (MoC) for embedded systems, which features timing semantics, by an error handling mechanism for timing errors. We use the NASA Ames model checking tool Java PathFinder (JPF) for automatic verification.

Commonly used models of computation or programming languages do not include an error handling mechanism for timing errors. Thus, how errors and specifically timing errors are managed can vary dramatically which can easily lead to major software problems. *A model of computation, which specifies an error handling mechanism is necessary for mission critical software. An interesting option to consider is the use software model checking before code generation.*

A model of computation is used to provide execution semantics as well as timing guarantees for the specification. Augmenting a specification language or a model of computation for real-time embedded systems with error handling mechanisms, is a very important step in designing robust specifications. This is necessary to ensure correct possibly degraded behavior in the event of timing errors or other faults in the system . We prototype an implementation of the error handling mechanism for timing errors in the simulator, we generate Java code for the model and use the Java PathFinder to detect uncaught errors to allow the designer to correct them, and we generate C code[4] to run on an embedded platform. This is done in Ptolemy II for Giotto, a model of computation for the periodic execution of tasks.

---

*Department of Electrical Engineering and Computer Sciences.
This preliminary project was conducted with funding from NASA JPFP mini grant with guidance from Johann Schumann and Edward A. Lee
†Research Scientist, Robust Software Engineering,Intelligent Systems Division.

II. Goal

A. High Level Goal

The high level goal, shown in Figure 1, that extends beyond this project is to provide a cohesive environment that includes timing semantics at each level beginning with the specification level and ending at the architecture level.

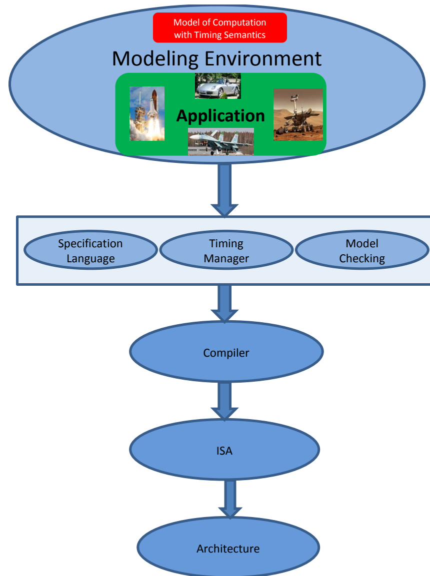


Figure 1. Timing Semantics at all levels

B. Project Goal

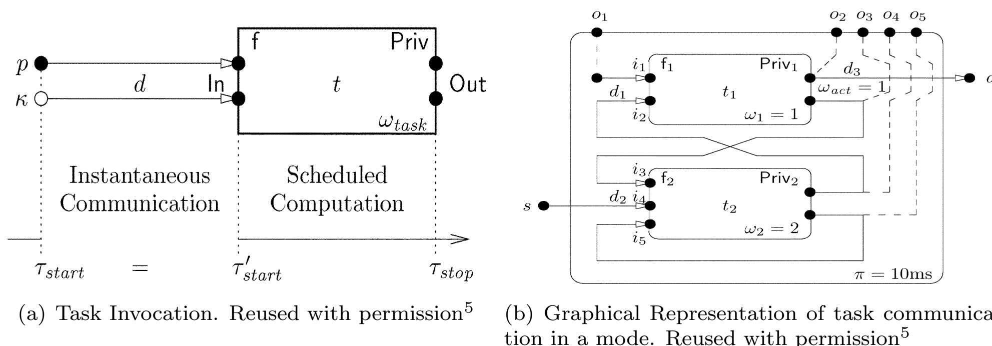
Our objective in this project is to provide the model checking component of our high level goal. In this project we use the Java PathFinder model checker developed at NASA Ames.

III. Background

We are working with an implementation of the Giotto model of computation implemented in Ptolemy II, we use Ptolemy II as our model-based design environment, and we use Java PathFinder to model check the specification.

A. Giotto

Giotto is a programming model for embedded control systems,[5] and is most useful for hard real-time specifications that are periodic and feature multi-modal behavior. Examples include fly-by-wire or brake-by-wire where the responses of the system must be periodic and there are multiple modes of operation. Examples of common modes of operation in the automobiles include startup, cruise control, normal operation, and a degraded operation mode in case of partial equipment failure.



(a) Task Invocation. Reused with permission[5]

(b) Graphical Representation of task communication in a mode. Reused with permission[5]

Figure 2. Giotto Mode and Tasks

In Giotto, a task, graphically represented in Figure 2(a), is the basic functional unit. If multiple tasks need to be run concurrently, they are put in a mode, as are $t1$ and $t2$ shown in Figure 2(b). Each task has a set of ports P, with input ports $(In \subseteq P)$, output ports$(Out \subseteq P)$, and maintains its state through private ports $(Priv \subseteq P)$. Tasks in a mode, i.e. t_i, t_j, have distinct input ports, $In(t_i) \cap In(t_j) = \emptyset$ for each two tasks t_i and t_j of a mode. All input ports are distinct from all other input ports; however, tasks may share output ports as long as they are not invoked in the same mode. Task inputs are read only at the beginning

of a period and task outputs are written only at the end of the period. The output can be computed at any time but should only be provided to other tasks or to the outside world at specified times.[5] The task's function f specifies what should be done with the inputs to produce the outputs. If several tasks are to be executed concurrently, they are included in a mode, and a system can change from one mode of operation to another at multiple points during the model's execution. Each mode has a period (π) as well as a frequency (ω_t) that specifies how many times the task should be executed within a mode period. In the mode shown in Figure 2(b), task 1 executes once every 10ms, and task 2 executes once every 5ms. Each mode also has guards that specify a mode switch frequency. The mode switch frequency specifies how many times within a period the possibility of a mode switch should be checked and taken if possible. Since it is possible to interrupt a task's execution when switching from one mode to another, one must ensure that a task that can be interrupted in one mode is present in the next mode so it can continue its execution.

Communication between tasks is done via ports in Giotto. Ports are specific locations in memory or variables dedicated to storing information. Drivers specify which output values should be copied to specific ports and which port values should be copied from a port to an input. Ports hold their value over time until they are updated by a driver. Sensor ports are updated by the environment, and actuator ports and task ports are updated by the program.[5] Task ports communicate data between concurrent tasks and can also be used to communicate between modes in the event of a mode switch. Mode ports are given a value every time the mode is entered.[5] In Figure 2(b), $o_1 - o_5$ are mode ports. o_1 is read when the mode begins and $o_2 - o_5$ are updated every 10ms and can be used to pass information from one mode to another in the event of a mode switch. In Figure 2(b), f_i specifies the actions that should be taken on the inputs to produce outputs, w_{act} specifies actuation frequency, and d_i specifies drivers which move information from ports to inputs, or from outputs to an actuator.

B. Ptolemy II

Ptolemy II[8] is an open source modeling and simulation framework being developed at the University of California-Berkeley that supports model-based design. Ptolemy II facilitates actor oriented and object oriented modeling. Actor oriented modeling is an alternative to the established object oriented methodology where objects are manipulated. Instead actors allow actions to take place on the evolving data which flows through them.[3] Ptolemy II facilitates the modeling and simulation of the design of systems whose behavior is governed by directors implemented in the Ptolemy II framework.

In Ptolemy II, when an actor fires, its behavior for a particular model of computation is governed by a director which is specified for the particular model of computation. A designer can select the director that specifies the desired behavior of a model, and then create and simulate their model. In Ptolemy II actors are governed by an abstract semantics, which are rules that dictate how an actor should behave. A simplification of the abstract semantics as it applies to Giotto includes pre-initialization, initialization, firing, and wrap-up. During pre-initialization in simulation, the Giotto director determines whether the frequencies specified by the actors are permissible. During initialization the ports and actors are assigned their specified default values. After initialization, actors are executed in the firing stage.

In addition to being a modeling and simulation framework, Ptolemy II also features an extensible C and Java code generation framework for multiple models of computation.

1. Giotto in Ptolemy II

The Giotto model of computation is implemented as a domain in the Ptolemy II simulation and modeling environment. A Giotto model is created with a Giotto director in Ptolemy II. The period π of the mode is specified as the period parameter to the director and the frequency of each task ω_t is specified as a frequency parameter to each Ptolemy II actor. If no values for the period and actor frequencies are provided as parameters, default values are assumed.[2]

In[5] a mode in Giotto consists of all tasks to be run concurrently with a particular period. In Ptolemy II, a mode is slightly different but allows all models expressible in.[5] Ptolemy II allows the use of hierarchy that proves to be very convenient in the specification of control behavior. In addition it also reduces the number of distinct mode combination specifications that are necessary in .[5] A Ptolemy II mode is specified inside a finite state machine modal model and improves the flattened specification present in[5] with the use of hierarchy. In Ptolemy II, tasks, which are referred to as actors, at the same level of hierarchy execute concurrently and a modal model contains tasks that should be switched when a guard is enabled. If it is

desirable to have three tasks: A, B, and C, where task A is always running and task C should replace task B when a certain condition is met, a designer could specify that in Ptolemy II as is shown in Figure 4. The lower portion of Figure 4 shows how the model is specified with Ptolemy II and the upper portion of the figure shows the logical execution times of each task based on their frequencies, and on the period parameter π of the Giotto director.

Ptolemy II allows hierarchy through the use of composite actors. A composite actor contains actors and in some cases a director. If no director is present inside the composite actor the actor is transparent. If however there is a director present inside a composite actor the frequencies of the tasks inside the composite actor are all interpreted to be relative to the frequency of the composite actor itself. If a composite actor with frequency 2 contains a Giotto director, and a task with frequency 3, the interpreted frequency of the task inside a composite actor is 6.

Each Giotto model is expected to specify a period as an attribute to the Giotto director, the frequency of each task as an attribute to each actor, as well as initial values for outputs. If Giotto directors are used inside a composite actor, the period of the top most Giotto director is used, but the frequencies of the tasks inside the composite actor are relative to the frequency of the composite actor.

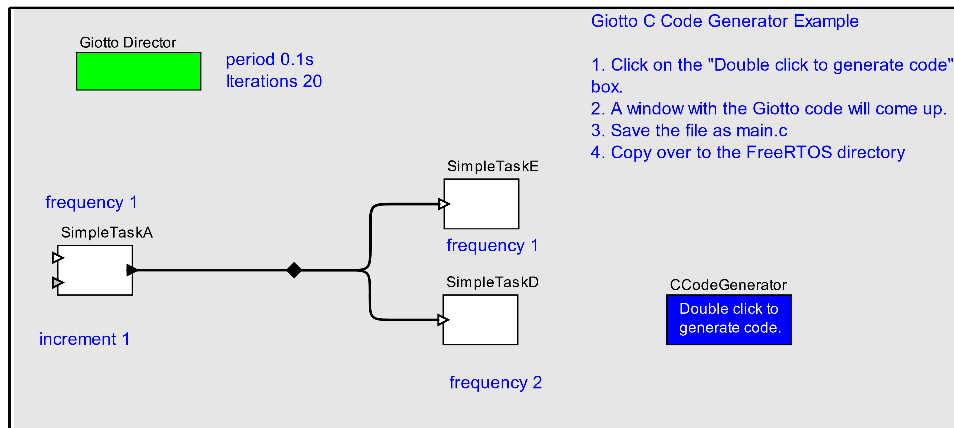


Figure 3. Simple Giotto model in Ptolemy II with three actors

For the purposes of simulation it is also possible to set the number of times you wish to have the model run. This can be specified as a parameter called *iterations* to the Giotto director. It should be noted that since Ptolemy II allows hierarchical models, if another Giotto director is specified within a composite actor, only the topmost Giotto director's *period* parameter is used along with the *frequency* parameters of each director and actor.

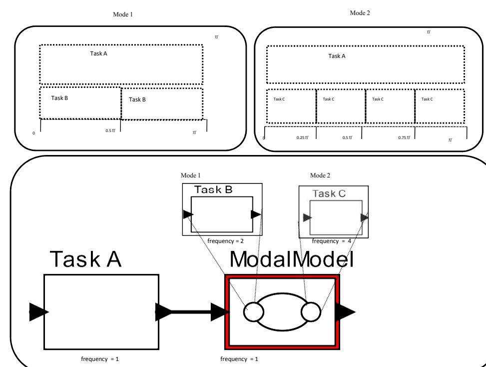


Figure 4. Hierarchy

C. Model Checking

In the past few years, model checking has been used to determine more causes of errors than traditional testing. In traditional testing, inputs are provided to a program and the the input is run through a single path in the program being tested. Model checking allows you to test automatically if a model of a system meets a certain specification[9] by exploring all possible paths in a program. Model checking is often used to check a program for deadlocks as well as other potentially crippling issues.

A well known use of model checking is the identification of possible deadlock in an implementation of the dining philosophers problem. In the dining philosopher problem, shown in Figure 5, a philosopher's three tasks are to eat, sleep, and think. Each philosopher needs two utensils to eat and they release their utensil after they finish eating. If each philosopher picks up the utensil to their right and then the utensil to their left, and all the philosophers attempt to eat at the same time, there would be a deadlock. No philosopher will release their utensil until after they have eaten and as a result all will starve. The use of a model checker on this problem will check all possible interleavings to decide if the order of the tasks each philosopher does as well as the order in which they acquire utensils can lead to a deadlock.

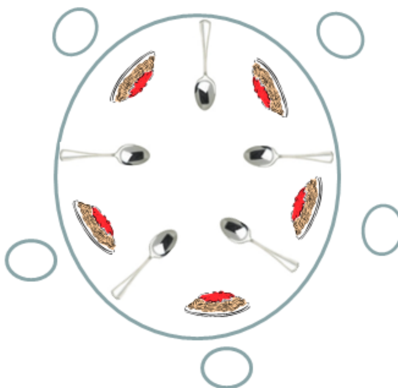


Figure 5. Dining Philosophers

D. Java PathFinder(JPF)

Java PathFinder (JPF)[a],developed at NASA Ames, is an explicit state software model checker for Java programs, which automatically checks safety and liveness properties of the code, e.g., deadlock freedom or absence of race conditions in multi-threaded programs. JPF is highly flexible and can be customized for many purposes including automatic generation of test cases, testing of graphical user interfaces, or symbolic execution and is actively used within NASA and in industry. JPF supports the full functionality of the Java programming language; and, as is seen with other model checkers and verification tools, large problems yield a large state space to explore. Since we target embedded platforms, we use the RTEmbed JPF extension since it has features particular to an embedded platform.

1. JPF-RTEmbed extension

JPF RTEmbed is an extension to the Java PathFinder core developed by Pavel Parzek as a specific extension of JPF for embedded systems using Real-Time Specification for Java. RTEmbed circumvents some of the issues that lead to state explosion in general Java programs by limiting the thread yield points to be checked to the general yield points used in embedded systems.[7] This is possible because they use the green threading model[7] which allows fined grained control. As a result, instead of having to treat every Java byte code as a thread yield point which must be checked, it is able to limit thread yield points to "the acquisition and release of monitors, calls of methods of the thread class, as well as calls of wait and notify".[7] RTEmbed also has enforcement of thread priorities, but does not have a precise model of real-time and execution time of instructions on a particular platform.[7]

Though it does not have a precise model of real-time, its current timing model provides a good first step in our approach.

[a]`http://javapathfinder.sourceforge.net`

IV. Theory and Design

A. Theory

Our goal in designing the Giotto timing manager is to provide capabilities not generally present in the MoC specification as well as to introduce real time to model time generally present in the MoC specification. Giotto does not specify what should happen if a task takes more than the mode's period (π) divided by the task's frequency (w_t) to execute. We provide the user with a means of modifying the execution behavior if timing constraints are violated, if not it behalves as previously specified.

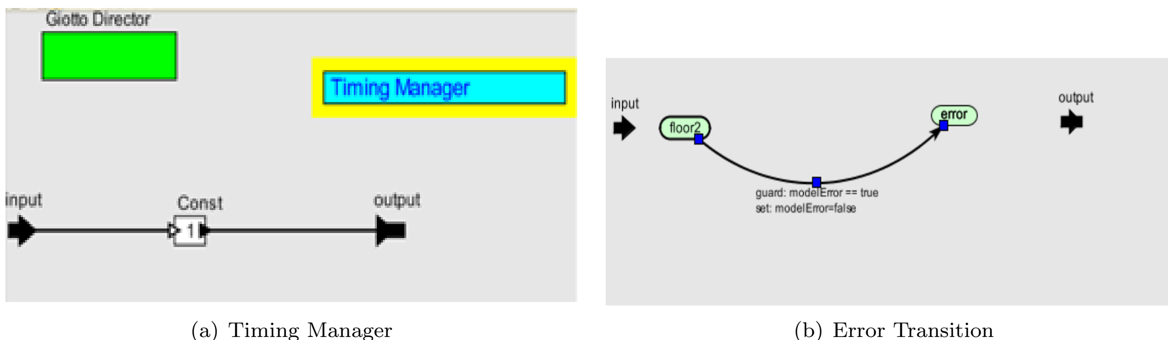
To enable this facility we explored aspect oriented programming[6] as well as the decorator pattern.[1] "Whenever two properties being programmed must compose differently and yet be coordinate, we say that they cross-cut each other. Because general procedure language provide only one composition mechanism, the programmer must do the co-composition manually, leading to complexity and tangling in the code.[6]" Generally aspects "cannot cleanly encapsulate in a generalized procedure. Aspects tend not to be units of systems functional decomposition, but rather to be properties that affect the performance or synthesis of components in systematic ways".[6] Aspect Oriented Programming is generally used instead of having the programmer do co-composition manually.

A decorator pattern adds, defines, or specifies features or attributes of components when brought into a model, and the decorated aspects of a component are removed when the decorator is removed.

B. Design

The Giotto timing manager uses a piggyback mechanism in Ptolemy II that allows the timing manager's preinitialize,initialize, fire, postfire, etc methods to be executed right before the execution of each of the respective Ptolemy methods. As a result the timing managers's prefire methods is fired before the MoC director's prefire method. We decorate all the actors seen by the timing manager with WCET, ET, and grace. In the timing managers's fire method each actor's execution time is randomly set as a variant of the initially annotated actors WCET. This allows each firing to have a different execution and also explore values above and below the WCET.

In the current timing manager, we compare the cumulative execution times(ET) of all actors fired at a particular tick to the predicted cumulative worst case execution time(WCET) of the actors after the firing of the actors. If the cumulative ET is larger than the cumulative WCET, the timing manger denotes an error in the model and calls the handle model error method. Handle model error traverses the model hierarchy to the first place it encounters a means of dealing with the error. In our case, if the Giotto Model is specified inside a modal model, the handle model error method sets the modal model's model error flag. Inside the modal model the model's designer can specify what should occur if a model error is detected by creating an error transition from the current refinement (tasks in a mode) to the error detected/degraded refinement. The user has the option to select any transition as an error transition and it is automatically annotated with the required guard and set action, shown in Figure 6(b).



(a) Timing Manager (b) Error Transition

Figure 6. Timing Manager and Error Transition

Since models can be composed hierarchically, we use model checking to determine if there are issue in the design of the model. In our case if the user forgets to specify an error transition when there is potential for a timing error. We use JPF for verification even though we later generate C code for embedded platforms

because, while Java PathFinder verifies all code, if we used the spin model checker with embedded C code we could run the embedded C code, however we would not be able to verify it. As a result, we generate Java code that mirrors the generated C code and verify the Java code with Java PathFinder.

1. Timing manager in the generated code

The timing manager manifests itself in the generated code in the portion of the code generated for the scheduler. Here we check to see if a completed task took longer than the specified time. If an error is detected and the timing manger was enclosed inside a modal model we set the model error flag for that modal model. If it is not the case that it is inside a modal model we simply throw an exception. The model checker then checks to see if there are any uncaught or unhanded exceptions in the generated Java code.

V. Results

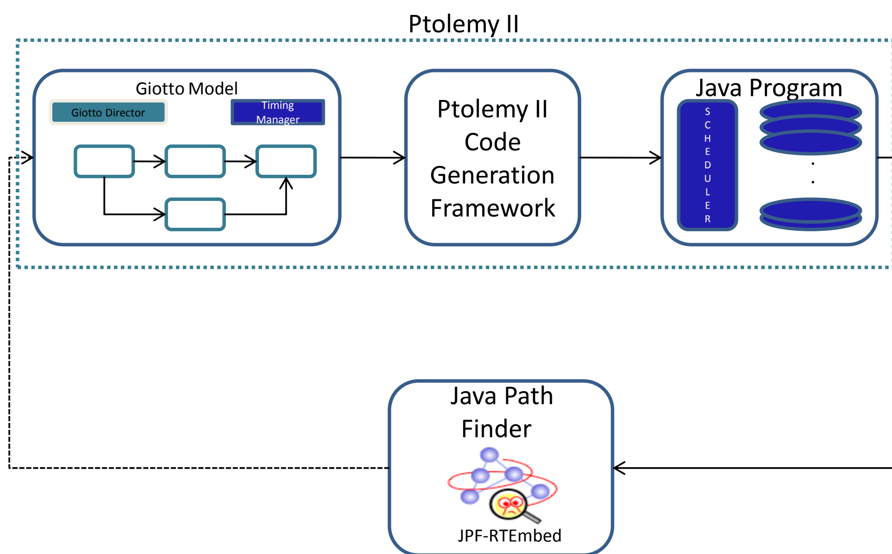


Figure 7. Ptolemy II to Java PathFinder Workflow

We created the timing manager to add real time to model time present in the Giotto MoC, and we extended the Ptolemy II Java code generation framework to generate Java code for the Giotto MoC as well as for modal models. We also augmented the code generation framework to generate Java code able to run on JPF-RTEmbed. After making these changes were were able to do preliminary runs as well as later running a very simplified experiment. This project took the first steps toward completing the Ptolemy II to Java PathFinder work flow shown in Figure 7. Currently the only unimplemented portion is the reporting of an error detected by the model checker back to the user.

A. Discussion, and Conclusions

In addition to extending the framework shown in Figure 7, we successfully checked a small and simple model shown in Figure 8 for possible timing errors.

The exploration of JPF as the model checking arm of the MoC extension indicated possible usefulness of a model checker in the extension. During our experiments we discovered that:

- The notion of timing in JPF-RTEmbed was not what we initially expected
- Due to the notion of timing present we discovered it was possible for our scheduler to starve task threads
- If there was no direct communication between threads the model checker often opted not the run the thread not directly communicated with

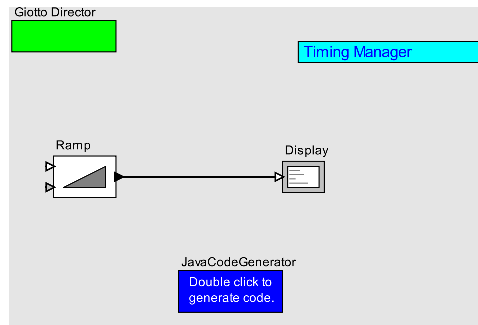


Figure 8. Simple Example with two actors

Though our experiments did not turn out as expected, the present work shows a lot of potential and we will continue extending our work even after the completion of the mini grant and attempt to resolve some of the issues which occurred.

References

[1]Decorator pattern. http://c2.com/cgi/wiki?DecoratorPattern.

[2]C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in java (volume 3: Ptolemy ii domains). Technical Report UCB/EECS-2008-37, EECS Department, University of California, Berkeley, Apr 2008.

[3]Edward A. Lee. Center for Hybrid and Embedded Software Systems Seminar on Model Engineering, October 21 2008.

[4]S.-S. Forbes. Real-time c code generation in ptolemy ii for the giotto model of computation. Master's thesis, EECS Department, University of California, Berkeley, May 2009.

[5]T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. volume 91(1) of *Proceedings of the IEEE*, pages 84–99, 2003.

[6]G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming(ECOOP), Finland*, LNCS(1241), June 1997.

[7]Pavel Parzek. The RTEmbed Extension for JPF: Checking Programs for Real-Time and Embeded Systems, October 21 2009.

[8]The Ptolemy Project. http://ptolemy.eecs.berkeley.edu/. Accessed : April 14, 2010.

[9]Wikipedia. Model Checking. http://en.wikipedia.org/wiki/Model_checking, 2010.