An Extensible Framework for Real-time Task Generation and Simulation using Object and Reflection Oriented Programming

Chaitanya Belwal, Albert M.K. Cheng

Computer Science Department
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

UH-CS-11-04
June 8, 2011

Keywords: Real-time Simulation, Response Time Analysis, Real-time Systems, Task Generation, Interface Classes

Abstract

In real-time systems research, validations are usually performed by executing synthetically generated tasks through programmatic implementations of derived algorithms or theoretical results. For every new result, real-time researchers have to develop systems, several times from scratch, for generating task sets as well as implementing their derivations. Another issue arises when the results are submitted for peer review. Reviewers only have access to results in the form of numerical values given in the paper, and have no easy way of validating these values themselves. To solve these two issues, we present a new extensible system for real-time task generation and simulation. Using modern software engineering principles of object and reflection-oriented programming, we show how real-time analysis can be partitioned into sub-systems, where each such sub-system can be implemented as a run-time 'plug-in', that can be developed by independent research groups. Our system also requires every task set to persist by saving task sets in customized file formats, which can then be shared with peer reviewers. Our technique is intended to save real-time researchers the significant amount of time they spent in result validation, as well as allow reviewers easy access to the experimental setup of the submitted paper for a more efficient review process.

[*] This work is supported in part by U.S. National Science Foundation under Award no. 0720856

An Extensible Framework for Real-time Task Generation and Simulation using Object and Reflection Oriented Programming

Chaitanya Belwal and Albert M.K. Cheng
Department of Computer Science
University of Houston, TX, USA
{cbelwal, cheng}@cs.uh.edu

Abstract

In real-time systems research, validations are usually performed by executing synthetically generated tasks through programmatic implementations of derived algorithms or theoretical results. For every new result, real-time researchers have to develop systems, several times from scratch, for generating task sets as well as implementing their derivations. Another issue arises when the results are submitted for peer review. Reviewers only have access to results in the form of numerical values given in the paper, and have no easy way of validating these values themselves. To solve these two issues, we present a new extensible system for real-time task generation and simulation. Using modern software engineering principles of object and reflection-oriented programming, we show how real-time analysis can be partitioned into sub-systems, where each such sub-system can be implemented as a run-time 'plug-in', that can be developed by independent research groups. Our system also requires every task set to persist by saving task sets in customized file formats, which can then be shared with peer reviewers. Our technique is intended to save real-time researchers the significant amount of time they spent in result validation, as well as allow reviewers easy access to the experimental setup of the submitted paper for a more efficient review process.

Index Terms

Real-time Simulation, Response Time Analysis, Real-time Systems, Task Generation, Interface Classes

I. Introduction

In real-time systems research, synthetic task sets are generally used to validate Real-time Simulation, Response Time Analysis, Real-time Systems, Task Generation, Interface Classes theoretical results. These results range anywhere from schedulability tests to algorithms for multi-processor partitioning, which are all classified as *real-time research contributions (RTRCs)* in this paper. These synthetic tasks are generated using random or pseudo-random selection or by statistical methods like probability distribution. Since a large number of tasks sets are required, automated programs are generally used. These task generation programs are either written by researchers from scratch, or some of the publicly available tools are used. Once these tasks are generated, they have to be analyzed using the derived theory or RTRC. A separate program, that implements the RTRC is also be written by the re-

[†] This work is supported in part by the U.S. National Science Foundation under Award no. 0720856

searchers. This program analyzes the generated task sets and outputs numerical results which are then used to produce validation data in the form of graphs or tables in the research paper.

In this endeavor, there can be a significant duplicity of work. For example, if the results get published and the RTRC is used by another research group they will have to re-program the implementation presented in the paper. Sometime it is possible to contact the original researchers and ask for their source code or the binary executable. However, since there is no standard to which implementations of RTRC have to comply with, a learning curve is involved which reduces the time savings incurred by reusing code written by other research groups. Another duplicity of efforts is encountered when generating task sets. There have been several proven methods which can be used to generate schedulable task sets. For example, if the RTRC deals with rate-monotonic schedulability, it might be desirable to use task sets that are guaranteed to be schedulable. Liu and Layland [14] and Bini et al [6] have presented sufficient tests for schedulability in rate-monotonic schedulability, hence tasks sets which satisfy both these conditions can be generated. Researchers will have to write the programmatic implementation of these schedulability tests from scratch. Another research group that needs to use the same schedulability tests will have to write their own.

If these tests are available in a simple to use binary library, research groups can save valuable time. It should also be noted, that writing software involves much more than simply writing the programming code. The software has to be carefully analyzed and validated against task sets to guarantee it correctness. Any *bugs* in the code affect the correctness of the results, which if used in scientific papers, reduce the quality of the paper. As anyone who writes software code will certify, *debugging* is not a trivial task, and when guaranteed mathematical correctness is expected, such as that required in real-time analysis, it can be a frustrating and time-consuming experience.

Another issue comes up during the process of peer-review when a paper is submitted to some conference or journal. Currently, reviewers look for the correctness of proofs given in theorems / lemmas, to determine if the presented RTRC is valid or not. Several papers are also composed of algorithms that vary in complexity. Most papers are accompanied by experimental analysis where the authors of the paper assert that the presented theorem or algorithm has been implemented and run through experimental task sets to give the desired results. The reviewers have to rely on the authors' assertions and have no way of knowing if the implementation that generated the results is correct and if correct task sets were used. Note that these results are provided in good faith, and researchers have no intentions of misleading the reviewers. However, situations when there is a flaw in RTRC or the implementation is not correct without the authors being aware of, are quite possible. This is especially true when there are several theorems, or the algorithms are too complex to be analyzed by visual reading. Expecting the reviewers to write the implementations presented in the paper and to test them, is not practical, due to severe time constraints and deadlines present in any peer-review process.

However, if all the task sets as well as the programming implementation used to generate the results, can be provided to the reviewers, it offers another venue for critical analysis of the paper. Reviewers will be able to detect flaws in the results and alert the authors, allowing them to fix the RTRC or its implementation, thus improving the quality of the paper.

To achieve these goals, a system that allows code modules as well as task sets to be easily shared with researchers and peer-reviewers is required.

I.I. Contributions

We present an extensible framework that can be used for both task generation and simulation. This framework uses an object-oriented design that allows new binary modules developed by independent researchers to be added dynamically. The goal of this framework is to save real-time researchers the effort to implement methods that have been previously developed. It also allows peer-reviewers a system to evaluate practical implementations of the work they are reviewing.

After reviewing the definitions used in this paper (*Section 1.2*) and related work (*Section 1.3*) ,we

- Present the basics of object and reflection-oriented programming (*Section 2*)
- Present the categorization of sub-systems used in real-time analysis (*Section 3*)
- Present the design of our framework including class templates and logical flow (*Section 4*)
- Present the implementation of our framework in a standard programming language (*Section 5*)

And, finally conclude with a reflection on current and future work (*Section 6*).

I.II. Definitions

The formal definitions of important concepts used in this paper are as follows:

- A **task set** is a set of *n* periodic tasks
- Each task has an associated **priority level**
- The **arrival time period** is time difference between two successive jobs of a task
- The **execution time** of a task, refers to the minimum time each job of the task requires to complete execution after its release. In most schedulability studies the execution time is assumed as equal to the Worst-Case Execution Time (WCET)
- The **release offset** of a task is the time of release of its first job
- The **relative deadline** of a task is the amount of time within which each job of a task should complete execution after its release
- A **binary module** refers to a compiled and linked executable or library file
- **Persistence** is a property of data to stay valid after the program is unloaded from memory. Persistence is usually achieved by systematically storing data in secondary storage
- A **plug-in** is a binary module that can be used with another program without requiring any programming effort for integration into the main module

I.III. Related Work

Several research groups have developed systems for simulating tasks. While several algorithms for task generation have been proposed ([3],[8],[11]), we are not aware of a library or software available package dedicated to task generation. Among the systems for simulating real-time tasks the most notable are RTSIM [17] and Times-tool [22]. A system called STRESS [2] was developed at York University in the 90's, however the current availability of this tool is unknown. Times-tool features a graphical editor for designing directed graphs, which are verified for schedulability using timed automata. The tool also has a simulator which can display graphical representations of the execution trace. The simulator validates the results derived from the formal verification done using timed automata.

RTSIM , which is an open source project caters to simulation and generation of task execution traces. It follows a modular design and features four important modules. The first module called *metasim* is a library for simulation of discrete event systems. *Rtlib* is a library which can be used for analyzing tasks through scheduling algorithms. The other two libraries *ctrlib* and *jtracer* are optional and used for control systems and displaying graphical representation of the task execution trace.

While these tools are very useful in their own right, they do not have any well structured system of task generation. We have designed a fine grained model for task generation and have developed interface templates that serve as the 'protocol' for any new module to follow. Using the principle of reflection, any new user developed binary module that uses the interface templates can be used in our framework without writing any code. Hence, every user developed module behaves like a plug-in which integrates seamlessly into the framework. Every task set is also persisted by saving them into files. By persisting task sets, they can be shared with peer-reviewers or used in other experiments.

The commonality between RTSIM / Times-tool and our framework in the area of task simulation. Our simulation framework is also a separate module that can be called automatically after task generation. We allow researchers (*users*) to write their own simulation modules, and require their implementations to conform to a template defined in the framework. Every simulation module is expected to return a task execution trace which can also be written into a file. The users can define their own trace file formats to allow analysis in separate third party programs, such as RTSIM's *jtracer* module. Note, that each of the modules required by our framework can be written in isolation without having access to the source code of the framework.

In this respect, being an open source project, RTSIM also is extensible and has an object oriented structure. However, RTSIM is not explicitly designed for easy extensibility and does not require have any defined 'protocol' which new modules have to comply with. Integrating new modules in RTSIM requires serious programming effort, as well as a learning curve to understand the code design.

II. Programming Paradigms

In this section, we review the object and reflection-oriented programming paradigms. A detailed description of these paradigms is outside the scope of this work and can be find in respective texts ([4],[9],[19],[20]).

II.I. Object-Oriented

The object-oriented programming paradigm has been in existence for a long time and is widely used in industry and academia. The important concepts of this paradigm relevant to our design are given below.

- Classes

Classes are the basic building blocks in object-oriented programming. They provide a template for building objects, which are actual data representation of the class. A class consists of *variables*, *properties* and *functions*, which are called class members. While a value can be read from and written to a variable, properties are special kind of variables to which read / write access can be controlled. Objects are actual memory representations of the class and though they will have the same template, the data in each object can be different. Any number of objects can be created from a single class.

All functionality of our framework is implemented in classes. We use a special type of class called *Interface* which contains declarations of functions but not their implementation. In this paper, *user classes* refer to those classes which are not part of the framework, but are developed by real-time researchers as extensions to the framework.

- Encapsulation

This property allows classes to hide implementation details from other classes. Using the notion of *public* and *private* variables, properties and functions in a class can either be hidden or made visible to other classes. In our context, encapsulation allows different task generation and simulation classes to be used through their publicly available members, without knowledge of their implementation details.

- Inheritance

This is an important property for extensibility in object-oriented programming. Inheritance allows a class to be *derived* from another class, which is called the *base* class. The derived class retains most of the functionality of the base class, and has additional functionality of its own. Derived classes can also overwrite implementations of methods which were used in the base class. All functional classes that are used in our system have to be derived from Interface classes provided by the framework. Classes which inherit from an Interface class are said to *implement* that interface.

- Polymorphism

Polymorphism gives the functionality for an object of the base class to call methods in the derived class. Only those functions can be called which are also declared in the base class. In our framework, all user classes have to be derived from one of the Interface classes, and functions inside these classes are called using objects of the Interface classes. This allows our framework to use the functionality in user classes without having to know their exact name or other implementation details.

II.II. Reflection-Oriented

Reflection is the ability of a programming language to analyze and modify the programming logic at run-time. Originally proposed in Smith's Ph.D. Thesis [19] as an extension to Lisp, it is now available in several commercial languages. Reflection allows extension modules to be independently developed and compiled into executable or library files. These can then be loaded by the executable of other programs, and functions inside the extension modules can be called. This allows extension modules to be distributed as binary files and attached to the main module by a simple process like copying these modules to the file system.

Reflection is implemented by maintaining separate meta-data information of the executable. In the object-oriented programming context, meta-data contains information on all variables, properties and functions that are used by each class. For detailed information on implementation aspects of reflection, readers can refer to ([9], [20]).

Reflection is important for the ease of integration of extension modules in our framework. A classical approach for extending capability of software is to write code and compile them into object modules or dynamic link libraries. After making required modifications in the main routine to integrate the object modules or libraries, the main module needs to be compiled and linked again to use the extended functionality. In this approach, even a simple program extension requires changes to the source code, and compilation and linking steps have to be replicated. This model is time consuming due to several reasons. First, integrating new modules requires development and testing effort. Second, it requires a development environment like operating system, compiler or linker version which might not be readily available to researchers or reviewers who will have expend considerable effort in setting them up. Finally, changing code which is not self-authored has a learning curve associated with it.

Implementing extensibility through reflection overcomes all these issues and allows for a simple and easy integration of extension modules.

III. Real-time Analysis Sub-Systems

In this section, we define import sub-systems for real-time analysis. These sub-systems have been classified under the categories of task generation and simulation. Each of these sub-systems can be independently developed by separate researchers. The breakdown of real-time analysis into such sub-system offers a fine-grained control and offers several options for setting the configuration of the analysis.

III.I. Task Generation

The process of creating periods and execution times, generating task sets which conform to certain schedulability and satisfiability conditions and writing every task set to files for persisting data is classified under task generation. The following summarizes each of these important sub-systems.

- Period Generator

Generates all the task arrival periods for use in task sets. Periods can be generated based on various criterions like selection within a bounded range using random or pseudo-random algorithms. Since several schedulability analysis models are based on the feasibility interval, which is dependant on the LCM of all periods present in the task set, it might be desirable to select periods with low LCM values. Methods to generate periods with low LCM values have been shown in ([3],[11]), and these methods can be easily implemented to work with our framework.

- Prime Number Generator

Prime numbers are required by some period generator algorithms such as the ones in ([3],[11]). Prime number generation is one of the classical problems in numerical theory for which no polynomial time algorithm exists. However, there are several algorithms available some of which have complex implementations. By defining prime number generation as a separate sub-system we allow researchers to share their implementations of efficient prime number algorithms, which can then be used for period generation.

- Execution Time Generator

Like task periods, execution times are used for creating task sets and can be generated using random or pseudo-random algorithms. A fixed set of execution times can also be used.

- Release Offset Generator

In a *synchronous* system, the release offset is 0, while in an asynchronous system the release offsets will be different. A release offset generator can be developed for synchronous system that sets all release offsets to 0, and various approaches like random selection can be used to generate the release offsets for *asynchronous* systems.

- Task Set Generator

This sub-system generates the actual tasks based on the periods, execution times and release offsets. Tasks can be generated by combining task periods with execution times, randomly or based on certain algorithms like the ‘UUniFast ‘algorithm [8].

The Task Set generator can also use its own internal system for creating periods, execution times and release offsets. Hence, it is possible to generate task sets based on probability theory like Poisson's distribution.

- Schedulability Tests

Several schedulability tests for different execution models have been developed over the past several years. The schedulability tests can be divided into necessary, sufficient or exact types. A task set that fails the necessary test, is guaranteed to be unschedulable, while a task set that passes a sufficient test is guaranteed to be schedulable. However, even if the task passes the necessary test it can be unschedulable, or if it fails the sufficient test it can still be schedulable. An exact test gives a guaranteed result as to the schedulability of the task set. However, unlike sufficient or necessary test, exact tests are computationally intensive. Among the popular schedulability tests are Liu and Layland's [13] and Hyperbolic bound [6]. Our framework allows all three kinds of tests to be used in the framework.

- Satisfiability Tests

Under satisfiability condition we allow researchers to define restrictions on task sets that do not fall under any of the formal schedulability tests. For example, if the combined utilization of task set should be more than a certain value, or there should be a minimum difference between task periods, then these conditions are classified under satisfiability tests.

- Formatted File Writer

Writing task set to files, fulfills an important requirement of our framework to persist task set data for archival and use in the peer-review process. The task sets files are written by this sub-system. Defining the process of writing files as separate sub-system, gives the capability to users to define their own file formats for use in other propriety programs.

III.II. Task Simulation

Once the tasks have been generated they can be simulated. The sub-systems required for task simulations are described below.

- Formatted File Reader

In our framework, task sets cannot be passed from memory directly to the simulation. This has been done to enforce persistence of task sets. While the task set data is written to files using the formatted file writer , a sub-component of this file writer is the formatted file reader. The file reader reads the contents of the files and converts it to task set data. For every formatted file writer there is a corresponding file reader which reads task set data in the same format written by the file writer.

- Execution Model

This sub-system defines the execution semantics of real-time tasks. The most common model is the preemptive model where upon release, higher priority tasks can preempt any executing lower priority tasks and the preempted tasks can resume execution from the point they were preempted. In the non-preemptive execution model, lower priority tasks cannot be preempted, while in the P-FRP [13] execution model, the lower priority tasks are preempted. Lock-free execution [1] and Transactional memory [10] are other well-known execution models. The execution model in our framework computes the execution of a task at each discrete time interval till a maximum specified time.

- Execution Trace Writer

The output generated by the task simulator is a list that shows the task executing at each discrete time. This is called the *execution trace*. Our framework allows researchers to generate customized output file for the execution trace. The main motivation for doing this is to use the execution trace for building task graphs in external drawing programs. For example, if we write trace files in the format required by RTSIM, we can use RTSIM's *jtracer* module to build execution graphs for task sets that were simulated in our framework.

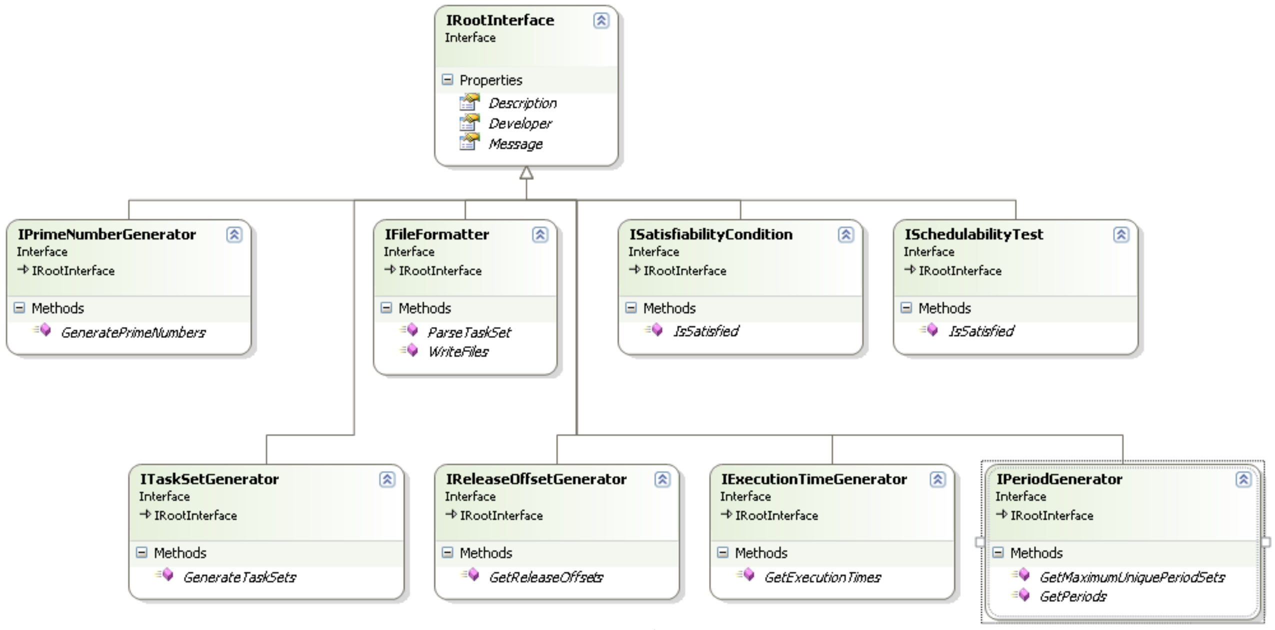


Figure 1. UML Diagram of Task Generation classes

IV. Framework Design

In this section, we present the framework design that allows for integration of extensible modules for each real-time sub-system presented in the previous section. We first describe the Interface classes for each sub-system and show how using reflection, the interface classes are used to implement each sub-system as a plug-in. We also present the logical flow of our framework.

IV.I. Sub-System Interfaces

Our framework defines Interface classes for each real-time sub-system. In this section, we discuss the template of each Interface class. Descriptions on the templates are supplemented with UML class diagrams of Interface in *figs. 1* and *2*. The UML diagrams have been generated using Microsoft Visual Studio.

- Root Interface

Certain properties are required in each sub-system plug-in. To allow this, we define a root interface from which other sub-system interfaces are derived. The root interface has three properties *description*, *developer* and *message*. The *description* property contains a short description of the module and *developer* consists of the author of that module. The *message* is used to pass error or other message to the main user interface of the framework. *IRootInterface* is the identifier of this interface in the class diagram.

- Period Generator

Identified by *IPeriodGenerator*, it consists of two functions. While 'GetPeriods' generates all the periods 'GetMaximumUniquePeriod' sets returns the maximum number of unique period sets possible for the specified configuration.

- Prime Number Generator

Identified by *IPrimeNumberGenerator*, it has a single function that returns the prime numbers generated by the method implementation.

- Execution Time Generator

Identified by *IExecutionTimeGenerator*, it has a single function that returns the execution times generated by the method implementation.

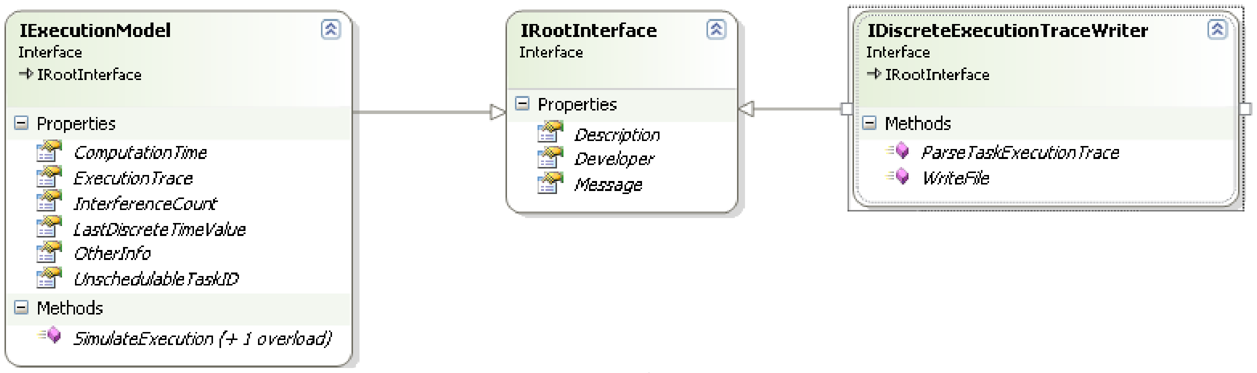


Figure 2. UML Diagram of Task Simulation classes

- Release Offset Generator

Identified by *IReleaseOffsetGenerator*, it has a single function that returns the release offset generated by the method implementation.

- Task Set Generator

Identified by *ITaskSetGenerator*, it has a single function to which the generated period, execution times and release offset are passed. The task set generator creates task sets based on these parameters and passes every generated task set to the framework. The framework then runs the task set through the specified schedulability and satisfiability conditions and either accepts or rejects the task set. If the task set is rejected the task set generator will generate another task set.

- Schedulability Tests

Identified by *ISchedulabilityTest*, it has a single function that takes a task set and returns a Boolean true/false value depending whether the task set passes the test.

- Satisfiability Tests

Identified by *ISatisfiabilityTest*, it as a single function that takes a task set and returns a Boolean true/false value depending on the task set passing the test.

- Formatted File Writer

Identified by *IFileFormatter*, it has two main functions. 'WriteFiles' takes a set of task sets and writes their data into files. Task sets can be written into any format depending on the implementation, and each task set is required to be written in separate files. 'ParseTaskSet' is the formatted file reader sub-system and takes a input string containing a task set file which was written by the 'WriteFiles' function and converts it to a task set data object. Note that both the formatted file writer and formatted file reader are implemented in the same class since their functions are complementary.

- Execution Model

Identified by *IExecutionModel*, it has a single function that takes a task set and simulates it based on the implementation. The time till which the simulation should be run is also passed to the 'SimulateExecution' function. It returns a Boolean value of true/false to denote schedulability of the task set. Besides this, it has several properties that define the state of the simulation. The *ExecutionTrace* property contains execution trace data showing the tasks which execute at discrete time. The *LastDiscreteTimeValue* is used to denote the last discrete time when the task was found unschedulable or schedulable. *OtherInfo* is an array that can contain information specific to the simulation that an implementation can use.

Note that it is not required that all the properties to have values. This allows flexibility in the simulation. For example, if the simulation is not based on discrete times the *ExecutionTrace* and *LastDiscreteTimeValue* can be left blank.

- Execution Trace Writer

Identified by *IDiscreteExecutionTraceWriter*, it has two main functions. 'WriteFile' takes as input the execution trace data and writes it to a file based on some file format chosen by the implementation. The 'ParseTaskExecutionTrace' reads an execution trace file in the same format as used by 'WriteFile' and converts it to execution trace data. Currently, our framework does not utilize the execution trace data. However, the execution trace data can be used to build task graphs through other programs. In future work, we will build a module to plot the execution trace for which the 'ParseTaskExecutionTrace' function will be used.

IV.II. Sub-System Plug-Ins

In the previous section, we have presented the Interface classes that form the foundation of the framework. These Interface classes do not implement any method, but define a standard or protocol to which very new module has to comply with. New modules which derive from these Interface classes can be compiled into a binary file and placed in the same directory as the executable. Using reflection our system can find these modules and use the functionality implemented in them

For example, consider a developer who wants to write a new schedulability test for use in our framework. The developer will write a class and inherit it from the *ISchedulabilityTest* interface. Because of this inheritance, the 'IsSatisfied' function will have to be implemented. Since *ISchedulabilityTest* inherits from *IRootInterface* the developer will also have to define the *description*, *developer* and *message* properties. If the method or any of the properties are not declared, the module will not be compiled. After writing the implementation for the 'IsSatisfied' function the developer will compile the binary module and attach it to our framework. Based on the implementation for the framework, a module can be attached by a simple process like placing the binary module in the same file system as the framework.

When the framework is activated, it has no information on the number or type of binary modules that are available for use. Using reflection, the framework queries all the binary modules attached to it and creates objects for the classes in the modules based on the Interface they derive from. These objects are then used for the task generation and simulation purposes. This way the 'IsSatisfied' method in the implementation of the *ISchedulablityTest* interface will be called.

By using reflection and writing classes that derive from the sub-system Interfaces, modules written by different developers can be used in the framework without any development effort. Hence, each class written by users that derives from our framework's interface has all properties required from a plug-in, and these classes will be referred to *sub-system plug-ins* in the rest of this paper.

IV.III. Logical Flow

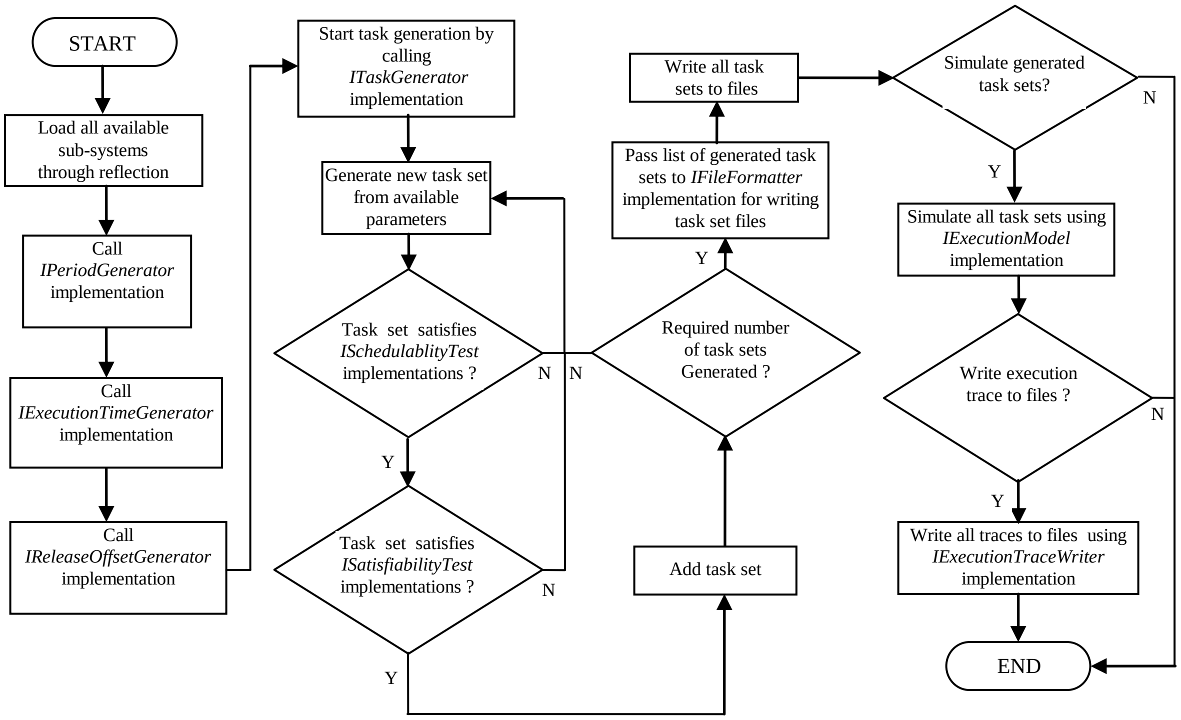
We have shown how our framework identifies sub-system plug-ins and integrates them. In this section, we show the sequence of steps in which various sub-systems are called to generate task sets as well as simulate them.

Fig. 3 shows the flow chart for these base steps. First the task sets are generated and for this the periods, execution times and release offsets sub-system plug-ins are used. These values are then passed to the task generator which combines the parameters to generate task sets. Every generated task set is verified through available schedulability and satisfiability test plug-ins. If the task set satisfies the tests it is added to the master list, else a new task set is generated. When the required number of task sets has been added to the master set, the task generation process stops and the task set files are written to the file system using the implementation of *IFileFormatter*.

Due to high computation times, running the simulation after task generation is an optional feature of the framework. If the task sets have to be simulated the required implementation of *IExecutionModel* is used. Since, execution traces can be large and take disk space writing execution trace generated by the simulation is also an optional feature. If enabled, execution trace files are written after the simulation is done.

IV.IV. Support Classes

Besides the Interface classes, the framework also provided several classes for use by sub-system plug-ins. These classes deal with task, task sets, periods, configuration of task generation as well as common mathematical functions required in real-time analysis like lowest common multiple (LCM) and combinatorial arithmetic. We



START
Load all available sub-systems through reflection
Call IPeriodGenerator implementation
Call IExecutionTimeGenerator implementation
Call IReleaseOffsetGenerator implementation
Start task generation by calling ITaskGenerator implementation
Generate new task set from available parameters
Task set satisfies ISchedulablityTest implementations ?
Y
N
Task set satisfies ISatisfiabilityTest implementations ?
Y
N
Add task set
Required number of task sets Generated ?
N
Y
Pass list of generated task sets to IFileFormatter implementation for writing task set files
Write all task sets to files
Simulate generated task sets?
N
Y
Simulate all task sets using IExecutionModel implementation
Write execution trace to files ?
N
Y
Write all traces to files using IExecutionTraceWriter implementation
END

Figure 3. Logical Flow for Task Generation and Simulation

briefly describe some of these important classes. Detailed layouts of these class are provided as UML diagrams in *fig. 4*.

- *CTask*
  This class contains task parameters such a execution times, periods, release offsets and a unique identifier. The utilization property returns the ratio of execution time to arrival period.

- *CTaskSet*
  Is the container for *CTask* objects. The properties of Hyper-period and Utilization return the LCM of periods of all tasks and their combined utilization factors respectively. It has functions to return tasks in priority order as well as to change the priority order to rate-monotonic. Functions to change priority order based on a different criterion can also be added by extending this class. Objects of this class are created by the *ITaskGenerator* implementation, and it is also an input parameter to *IExecutionModel*.

- *CPeriodSet*
  Is the container for sets of all periods generated by the *IPeriodGenerator* implementation. Provides functions to add or remove periods as well get LCM of periods.

- *CConfiguration*
  This class contains all the user-defined attributes required for task generation and simulation. These include bounded ranges for task period, execution times and release offsets, the number of task sets that have to generated and the number of tasks in each set. There is only one instance of the *CConfiguration* object which is passed as a function parameter to several sub-system plug-ins, and the plug-ins have freedom to use any of the user defined properties.

V. Implementation

The framework design presented in this paper, can be implemented in any of the popular object-oriented programming languages which support the reflection capability. However, due to wide use the choice essentially comes down to C++, Java or a language in Microsoft's .NET framework. A detailed discussion on relevant merits of each of these platforms is outside the scope of this paper. While there are reflection libraries available for C++,

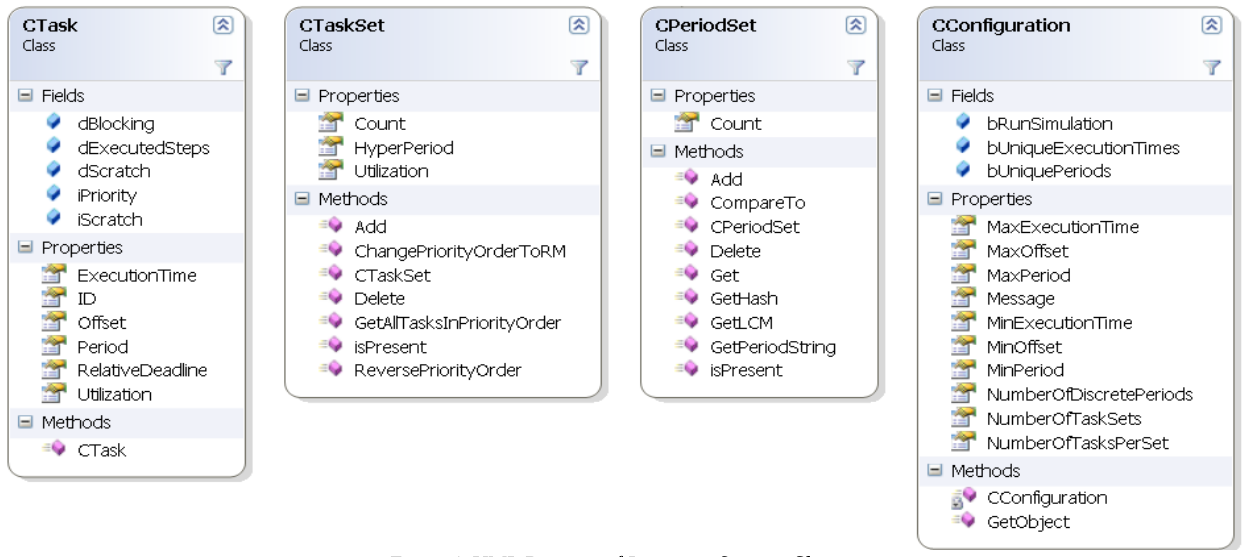


Figure 4. UML Diagram of Important Support Classes

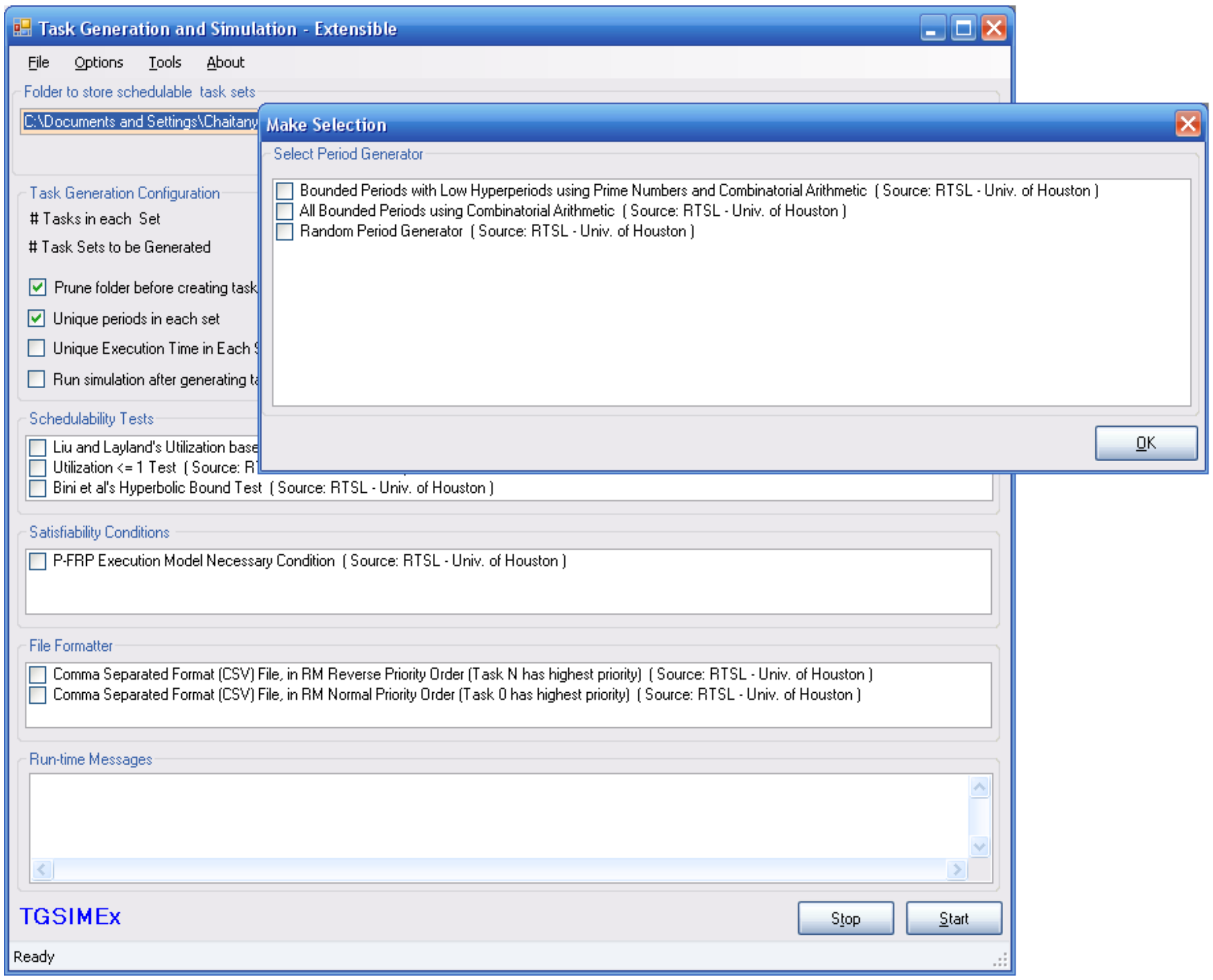
we feel that the reflection capabilities of C++ are not in par with those of Java or .NET. Java has an inbuilt library 'java.language.reflect' that provides the classes and functions to use reflection. After careful analysis, based on the choice of several programming languages, a powerful and well-document capability for reflection, and powerful user interface features we selected to implement our framework in the language C# running on Microsoft's .NET framework. Like Java, .NET programs are compiled into an intermediate language and executed by a just-in-time compiler called common language runtime (CLR). While there is a small performance cost as compared to machine-language compiled code, this cost in negligible for an application used for offline task generation and simulation. An extension to .NET [15] is also available for the Linux operating system.

Our implementation of the framework is called TGSIMEx (Task Generation and Simulation – Extensible) and is available as a Microsoft Windows portable executable file [21]. The user interface of this implementation is shown in *Fig. 5*. The user has to select a directory to write the task files and enter various configuration parameters such as bounded ranges for execution times, arrival periods and release offsets.

TGSIMEx has a 'PlugIns' directory where all sub-system plug-in files are placed. When the application is launched it iterates through all the files placed in the 'PlugIns' directory and looks for classes that inherit from one the sub-system interfaces. The framework then creates objects for each of these classes, accesses the *description* and *source* property required by *IRootInterface* and displays it in the user interface. Since only one generator for periods, execution times, release offsets and task sets is required, we allow the user an option to select one of these among those that are available. Users are allowed to select more than one schedulability and satisfiability test.

To test the framework, we created two .NET binary modules available as dynamic link libraries (DLL's), that provides at least one implementation of each sub-system interface. The 'UHPeriodLib.dll' contains three implementations of period generators, while 'UHGeneralLib.dll*'* provides implementation for all the other sub-system interfaces. We have implemented the schedulability tests provided in ([6],[14]), and provide two implementations of formatted file writer. For execution models, we have implemented the standard preemptive execution model as well as the abort-restart execution model of P-FRP. Both these models are used in the research done by us at the Real-Time Systems Lab, University of Houston. In *fig. 5*, note the use of *description* and *source* properties to denote each plug-in that is available for use in the framework. All the schedulability tests are listed under the group 'Schedulability Tests', and a checkbox is provided for the user to make a selection.

If no schedulability or satisfiability tests are required during task generation, no selection is required. The various generators for periods, execution times etc. can be selected from the 'Options' menu. In *Fig. 5* we also show the selection window for task periods. All the three period generators implemented in 'UHPeriodLib.dll' are listed. Users are required to make at least one selection for each of the generators as well as the file formatter, otherwise the task generation will not start. Both task generation and simulation takes place in independent threads allowing both these processes to be stopped if the user desires. Exception handling routines are used to capture any error that might occur in the user developed sub-system classes. The simulation module can also be launched indepen-



Folder to store schedulable task sets

C:\Documents and Settings\Chaitany

Task Generation Configuration

Tasks in each Set

Task Sets to be Generated

- [✓] Prune folder before creating task
- [✓] Unique periods in each set
- [ ] Unique Execution Time in Each
- [ ] Run simulation after generating t

Schedulability Tests

- [ ] Liu and Layland's Utilization base
- [ ] Utilization <= 1 Test (Source: R
- [ ] Bini et al's Hyperbolic Bound Test (Source: RTSL - Univ. of Houston)

Satisfiability Conditions

- [ ] P-FRP Execution Model Necessary Condition (Source: RTSL - Univ. of Houston)

File Formatter

- [ ] Comma Separated Format (CSV) File, in RM Reverse Priority Order (Task N has highest priority) (Source: RTSL - Univ. of Houston)
- [ ] Comma Separated Format (CSV) File, in RM Normal Priority Order (Task 0 has highest priority) (Source: RTSL - Univ. of Houston)

Run-time Messages

Ready

Select Period Generator

- [ ] Bounded Periods with Low Hyperperiods using Prime Numbers and Combinatorial Arithmetic (Source: RTSL - Univ. of Houston)
- [ ] All Bounded Periods using Combinatorial Arithmetic (Source: RTSL - Univ. of Houston)
- [ ] Random Period Generator (Source: RTSL - Univ. of Houston)

Figure 5. Main user interface of TGSIMEx

dently without any task generation. For this the location to a directory containing all task set files has to be selected.

We now describe how TGSIMEx can be used to achieve the twin goals of extensibility and as an aid in the peer review process.

IV.I. Extensibility

The sub-system interfaces are available in a .NET binary module called ‘TGSIMExLib.dll’, available from [21]. Consider a case where a user has developed a RTRC which is a new exact schedulability test for rate-monotonic scheduling. The user will include the file ‘TGSIMExLib.dll’ in their project and implement a class that derives from the *ISchedulabilityTest* interface. Once all the functionality in the class has been developed, the researcher will compile it into a .NET DLL and place it in the ’PlugIns’ directory. When TGSIMEx is launched it will find the implementation of *ISchedulabilityTest* through reflection and display it under the ‘Schedulability Test’ list. The user selects this test and enters the task generation parameters. For execution model, the user selects the ‘Preemptive Execution Model’ and for file formatter the rate-monotonic formatter is selected, plug-ins for which have been already been developed by us.

Once the task generation process starts, only those task sets will be generated which satisfy the exact schedulability tests developed by the user. Once the simulation is complete the researcher can check to see that none of the task sets are unschedulable in the simulation. If all tasks sets are schedulable in the simulation, it provides the necessary proof for the developed exact test . To test the negation of the test, the user can create another implementation of *ISchedulabilityTest* which only clears those task sets which fail the exact test. Using simulation it can be verified that all the task sets generated are actually unschedulable. The .NET DLL file that has the implementation of the schedulability test can then be shared with other researchers who only have to place the file in the ‘PlugIns’ directory to use its functionality. Similarly, different types of RTRC can be developed and tested in TGSIMEx.

Note that users can also incorporate their own graphical user interface (GUI) features if the GUI provided by TGSIMEx is not sufficient. This can be done by embedding them inside the classes and displaying the GUI component from inside the public which are called by the framework. Hence, TGSIMEx can be extended both in terms of functionality and user-interface and the extended modules can be easily used without writing any code.

IV.II. Peer-Review Process

Consider the RTRC example considered in the previous section. Let’s assume a paper on this RTRC has been submitted for peer-review in some conference, and the paper contains the results of experimental validation of the RTRC done through TGSIMEx. Since, TGSIMEx requires all task sets to be persisted in files, the developers will give reviewers access to all of the task set files, along with the files containing the plug-in. The files can be made available either through the author’s website or a separate submission system in the conference web site. The reviewers will download the task sets files to some directory and save the *ISchedulabilityTest* plug-in to the ‘PlugIns’ directory. This process can be done in a few minutes.

When TGSIMEx is started, the reviewers will run the task sets through the simulator to check their schedulability. The reviewers will then use the new schedulability test to generate additional task sets to check the correctness of the *ISchedulabilityTest* plug-in. This way correctness of both the task sets and the plug-in can be verified, and in case of any discrepancy, the authors can be notified. Experimental validation of the RTRC by the reviewers gives them more confidence on the contributions made by the paper and helps them make an informed decision.

While there is a small learning curve associated with any first time use, the cost of it is minimal if TGSIMEx or another implementation of our framework becomes a common tool for task generation and analysis.

VI. Conclusions and Future Work

We have presented a framework for use in experiments and validation of real-time research. The design philosophy of this framework is easy extensibility which allows real-time researchers to share their implemented methods with each other without any separate integration effort. This is achieved by using modern software engineering principles of object and reflection-oriented programming. Another intended goal of our framework is to aid in the peer-review process. This is done by an built-in mechanism to store task sets as well as allow RTRC’s to be distributed as sub-system plug-ins.

An implementation of this framework has been done in Microsoft’s .NET framework. This framework called TGSIMEx is available for use by real-time researchers. We have also developed several plug-ins for each subsystem. The development of TGSIMEx clearly demonstrates that the framework design is practically feasible.

We believe this framework design can be a starting point for a more standard model that can be used by real-time researchers for task generation / analysis as well as improving the peer-review process. Such a model leads to better quality research both in terms of saving time and allowing greater scrutiny of presented work. Future work on framework design will involve interface specification for drawing task charts, as well as specifications that can be used for complex real-time analysis. Extensions to TGSIMEx will be done to make it more user-friendly and provide a good quality user documentation.

References

[1] J. H. Anderson, S. Ramamurthy, K. Jeffay. “Real-time computing with Lock-free Shared Objects”. *ACM Transactions on Comp.Sys. 5(6), pp.388-395,* 1997

[2] N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings, ‘‘STRESS: A Simulator For Hard Real-Time System’’. *RTRG/91/106, Real-Time Research Group, Department of Computer Science, University of York* ,1991

[3] C. Belwal, A.M.K. Cheng. “Generating Bounded Task Periods for Experimental Schedulability Analysis”. *Manuscript under consideration*

[4] G. Booch. “Object-Oriented Analysis and Design with Applications”, *2nd Edition, Addison Wessely*, 1992

[5] N. Audsley, A. Burns, M. Richardson, K. Tindell, A. Wellings. “Applying new scheduling theory to static priority preemptive scheduling”. *Software Engineering Journal 8(5), pp. 284-292*, 1993

[6] E. Bini, G. Buttazzo, and G.Buttazzo. “A Hyperbolic Bound for the Rate Monotonic Algorithm”. *Proceedings of ECRTS’01*, 2001

[7] E. Bini and G. Buttazzo “Measuring the Performance of Schedulability Tests”. *Real-Time Syst. 30, 1-2, pp. 129-15*, 2005

[8] E. Bini E. and G. Buttazzo. “Biasing Effects in Schedulability Measures”. *Proceedings of ECRTS’04, pp. 196-203*, 2004

[9] W. Cazzola, R. J. Stroud, F. Tisato (eds.). “Reflection and Software Engineering”. *Springer LNCS*, 2000

[10] S.F. Fahmy, B. Ravindran, E.D. Jensen. “Response time analysis of software transactional memory-based distributed real-time systems”. *ACM SAC Operating Systems*, 2009

[11] J. Goossens, C. Macq. “Limitation of Hyper-Period in Real-Time Periodic Task Set Generation”. *RTS Embedded System (RTS’01), pp. 133-147*, 2001

[12] C. Han and H. Tyan. “A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms”. *Proceedings of RTSS’97,* 1997

[13] R. Kaiabachev, W. Taha, A. Zhu. “E-FRP with Priorities”. *EMSOFT’07 , pp. 221-230* , 2007

[14] C. L. Liu, L. W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. *Journal of the ACM (Volume 20 Issue 1), pp. 46 - 61* , 1973

[15] *http://www.mono-project.com*

[16] *http://www.microsoft.com/net/*

[17] *http://rtsim.sssup.it/*

[18] L. Sha, R. Rajkumar, J. P. Lehoczky. “Priority Inheritance Protocols: An approach to Real Time Synchronization”. *Transactions on Computers Volume 39, Issue 9, pp.1175 – 1185,* 1990

[19] B. C. Smith. "Procedural Reflection in Programming Languages", Department of Electrical Engineering and Computer Science. *Massachusetts Institute of Technology, PhD Thesis*, 1982

[20] J.M.Sobel, D. P. Friedman, "An Introduction to Reflection-Oriented Programming". *http://www.cs.indiana.edu/~jsobel/rop.html*, 1996

[21] TGSIMEx: *http://www2.cs.uh.edu/~cbelwal/TGSIMEx.zip*

[22] *http://www.timestool.com/*