Monitoring of Real Time Systems: a case for Reflection?

Ricardo Barbosa, Luís M. Pinho
IPP-HURRAY Research Group
Polytechnic Institute of Porto
Rua Dr. António Bernardino de Almeida, 431
4200-072, Porto, Portugal
rambarbosa@iol.pt, lpinho@dei.isep.ipp.pt

ABSTRACT

As industry evolves, embedded software becomes an intrinsic part of any system. Examples of this type of systems spread out from dish washing machines to advanced combat airplanes. As it is obvious, some systems have more critical requirements than others in terms of failure consequences. This is the main reason why all these kinds of systems must be monitored at all times, both during the development process and especially after deployment.
Hard real time systems are very difficult to monitor, either intrusively or non-intrusively, not only due to their inherent timeliness requirements, but also because of their embedded nature. In order to adequately observe its run-time behaviour it is necessary to give particular attention to the impact of any additional monitoring instrumentation placed inside the system, so that it does not interfere with the system's behaviour (or at least that this interference is deterministic).
In this paper we discuss several approaches developed to deal with this problem, paving the way to introduce the use of Reflection as a prime technology for this purpose. Reflection allows a component to provide observation and control of its own internal structure and behaviour to the outside world, thus limiting the impact that the monitoring mechanisms may have in the monitored system. We discuss the different approaches that can be used for this purpose, presenting its advantages and impairment, leading to a concrete propose for a Reflection-based monitoring framework.

1. Introduction

Not too long ago, machines were controlled by mechanical systems. Today, almost all airplanes are controlled through *fly-by-wire* systems and cars have already started to integrate automated driving control systems. The mechanical instruments that once controlled these systems are increasingly being replaced by complex pieces of software [1]. The problem is that the same reliability and safety that the mechanical parts

provided is also expected from this software. This in turn becomes even worst when safety/mission critical systems demand that faultless software is developed for the price to pay for failure is unbearable.

To fight this increase on demand for fault tolerant and reliable software systems, in the last few years an effort was made to create new tools and theories that approach these problems in straightforward way. Fields of research go from testing techniques to software development standards. From all these research fields, one that is particularly important, and that is still much unexploited, is monitoring [1].

In order to not only perform testing for verification and validation of critical software, but also to observe the runtime behaviour of the system after deployment, monitoring services are needed that provide sufficient information about the state of the system [2]. Although this is true, very little attention is given to the subject of monitoring and profiling.

If we, as an example, consider that safety critical software requires that the rate of failure is less than 10^{-9} failures/hour [1] (which, in "normal" numbers is something near to one failure each 114 thousand years!), and that to test systems with these safety critical requirements requires approximately 150 years [1], (considering that testing for software below 10^{-4} failures/hour is infeasible [3]) we can conclude that new approaches are needed and much more effort must be made in research for this area. Monitoring must be considered as a key factor in real-time systems, both during the development and deployment phases.

Therefore, in this paper we propose the use of Reflection as an enabling technology for real-time systems monitoring. For this purpose, we start by providing a description of the different types of existent monitoring technologies, as a basis for the discussion in

Section 3 of the different types of Reflection-based systems which may be used. Section 4 presents the proposed framework, also detailing its main advantages. Finally, some conclusions are drawn in Section 5.

2. Monitoring

"The price of freedom is eternal vigilance."
Thomas Jefferson

According to [4], monitoring a system means to collect runtime information about the system under test that cannot be obtained by static analysis. In other words, a monitor is a system that is used to observe an underlying system (usually called *target system*).

A term also very used when talking about monitoring is *intrusiveness*. It is important that the act of observing the system does not disturb, in any way, the system being observed. A system is said to be intrusively monitored if the act of monitoring uses any resources of the system that is monitoring.[1]

Another important definition to bear in mind is the non-deterministic effect of intrusively observe the system through the addition of code lines (software) to the target system in order to observe it (C language printf's, assertions, etc). This is called the *Heisenberg uncertainty in software* or the *probe-effect* [1]. This is particularly exacerbated if it is necessary to dynamically change the monitored components of the system.

It is very important to deem which information is necessary for monitoring of the system. If we try to extract a great amount of information we must consider the fact that we are loading the target system with extra burden, and intrusive issues may arise [5].

[1] From this definition we can see that it is utopic the pursue for non-intrusively monitor a system, but the intrusive symptoms can be minimized to a state of *quasi-non-intrusive* as we will see further ahead on this document. Of course that this varies depending on the boundary of the system being monitoring, but generally the system always "pays a price" for monitoring intrusion.

Of course that if too little information is extracted from the system, there will be a lack of precision on the observations and it may not be enough to make a consistent judgement of how and why the system behaves as it does [5].

The information that can (or must) be monitored can be divided into three groups: Data Flow (internal and external), Control Flow (execution and timing) and Resources (memory and execution resources) [1].

Data Flow information concerns the inputs and outputs of each component of the system, also allowing determining what are the intermediately computed values and/or program state that are not visible through the predefined interface. Control Flow information, allows to determine at what time and in what order are events received and handled in the system, to determine which, when and in what order are tasks staring, pre-empting and finishing, and to access the kernel specific scheduling and overheads. Finally, Resources information allows determining the kernel internal state, and the utilization of memory, CPU, and other particular system resources.

2.1. Levels of Abstraction

Another important aspect to have in mind is the level of abstraction. If we look at the previous section carefully, we can see that all information that can be retrieved is of low level of abstraction (like thread information). If we consider an object oriented framework, it would be useful to have information regarding objects, methods and classes [6], meaning, a higher level of abstraction.

Also, another concept important when speaking about levels of abstraction is the concept of *activity* [7]. An activity is a sequence of possibly nested method invocations. It branches from a top-level object and return to the same top-level object.

A tool that provides a good example of an activity is the *gprof* profiling tool [8]. The higher abstraction level of this tool is a single thread running on a single CPU. In the lowest level, the tool generates a call graph representing the breakdown of the total execution time, per function, including all nested call relationships.

The highest abstraction level on which monitoring should be focused is objects and activities, while at the lowest level is located the information regarding threads, interrupts, etc [6].

2.2. When to monitor

Another factor to consider is when to collect data from the system, in terms of system state. The collection of data can be triggered by events since the system under monitoring can be described as a series of events. Events like thread creation or termination and context switch can be used to trigger data collection [5]. Events can then be grouped into categories. Depending on the level of precision needed, only small subsets of events from each category are used during monitoring. An instrumented system is one that contains probes. When this instrumented system is executed, event traces are produced.

Another technique used for collecting data is *sampling*. This technique is time based, where a small part of the system state is captured and recorded with a certain sampling frequency. It is, however, used only when the desired result of the monitoring can be obtained by a statistical analysis, due to the large amount of data needed for sufficient coverage and precision [5]. Normally, due to this, event based triggers are used to collect information.

If we relate the concept of events with the concept of abstraction levels, we can combine events from the various abstraction levels to obtain information about the system.

2.3. Monitoring Approaches

After deeming when and which information is needed to observe in the target system, the question of how to minimize intrusive behaviour is raised. There are basically three approaches to monitor a system, that are, in some form or another intrusive to the system: hardware, software and hybrid.

The optimal solution to monitoring in terms of low interference with the target system is a hardware monitoring approach. The application of special hardware that allows the monitoring of the processor's signal lines such as data, address and control buses, makes monitoring passive to the underlying system [1][5]. As it can be noticed, this solution separates the monitoring task from the target system's workload. Although at first a glimpse seems like the perfect solution, it presents major drawbacks.

It is argued that hardware monitors have reached their maturity due to the fact that today's systems use extensively memory management units, on-chip caches, etc, and as industry evolves, and with it the integration of functionality, this becomes even harder to accomplice [1][5]. Another issue that is raised with the use of this type of monitors is that only low-level observations are available. If we want to know the internal state of the system, this approach cannot provide data with a higher level of abstraction [5]. Also, the cost of the development of this special (and very specific) hardware is significant and also puts in question issues like portability and scalability [1].

This approach, however, is been applied before in performance measurements [9], execution monitoring of multiprocessor systems [10] and in real time systems. Some research is also being done through the use of modern processor's *boundary scan* instrumentation for the monitoring purpose [11].

A suggestion for limiting the use of the hardware approach is a hybrid approach. This approach uses software probes that trigger recording of data when specific events occur. The software probe, for example, writes to specific addresses that are memory mapped with the monitoring hardware, or use special co-processor instructions [1]. This approach diminishes the use of the special hardware, but the burden of non-portability still remains [1].

The software solution allows eliminating most of the problems presented by the previous solutions. Actually, with the software approach we can observe significantly more than possible with the hardware approach. With this approach, in order to minimize the intrusive behaviour of the probes, we just need to make the probes part of the design, by allocating it resources, and use the execution time analysis and scheduling theory to analyse the system. Although we can eliminate the probe effect by adding the probes to the target system, accidents have proven that leaving non functional code in the system can be hazardous [6].

There are three types of software probes, each of which differ by the location where placed in the system: kernel-probes, inline-probes and probe-tasks. Kernel probes, are usually not accessible by the application programmer, they are made available by the kernel as an internal infrastructure. Inline probes are added to tasks to provide auxiliary outputs of internal behaviour. (e.g. extra printf's in the middle of the code, etc). Finally, probe-tasks have the form of auxiliary tasks that collect information from kernel-probes, inline-probes and/or other probe-tasks.

3. Introducing Computational Reflection

"Objectively a world of objects and relations between things springs into being (...). The laws governing these objects are indeed gradually discovered by man, but even so they confront him as invisible forces that generate their own power."

Georg Luckacs

Although the concept of computational reflection was not only idealized for the object-oriented paradigm, it is in this area where the most significant work is being done. Computational reflection has already been used in several domains like concurrent programming [12], distributed systems [13], artificial intelligence, expert systems. It is a recent technology (late 80's) and it is increasingly evolving to become the solution to problems in various areas of computer science [14], mainly through the use of inheritance strategies [15].

3.1. Reflection and Reification

Within computational reflection, two concepts are of importance. Reflection is a concept by which a component provides observation and control of its own internal structure and behaviour to the outside world [15] [14]. It is also defined as the capability of a program to manage, as data, the structural and computational structures of it self, at run time [16] or turning into data the structure and behaviour of classes and objects of a system [17].

To perform these types of management operations, there are two important considerations to have in mind: *introspection* and *effectuation* [14]: Introspection is the ability of a system to observe and reason about its own state, and it is used to find out what tools are provided by classes to be implemented on objects: effectuation is the ability of a system to modify its own interpretation and/or behaviour. An important aspect when speaking about effectuation is that it allows to add functionality to a

system, temporarily. Another important aspect is the ability to make modifications to the system behaviour without making modifications to the source code.

When speaking of reflection we must consider two levels of information: *base level* and *meta* or *reflective level* [15] [14]. The structural and behavioural information of an object model is called *reified information* or *meta information*. This information is handled by the *meta objects*. The set of meta objects in a reflective system is called the *meta level* and any changes on the handling of this information by the meta objects is *reflected* to the associated object computation. The set of objects in a reflective system is called the *base level* [15].

There must be a causal connection between the two levels of information. When changes occur at any of the levels, either at meta-level or at base-level, they are propagated to the other level [14].

3.2. Reflection Models

At least two reflection models where identified within the object-oriented paradigm [14]: *Structural* [18] and *Behavioural Reflection* [19]. In the structural model, the meta-level is constituted by meta-classes. Meta-classes have the structural description of the objects at base level, therefore when this information is modified, the structure of the objects at the base level are modified accordingly. This allows [20] [14] to operate in a class instance of a metaclass as an object instance of a common class. In this model, all instances of a class have the same behaviour. It is not possible to assign a different orthogonal property to a single object, only to classes of objects [14].

In the behavioural model, objects at meta-level are called *metaobjects* [19]. These metaobjects are similar to normal objects and contain all reflective information. A class

of a metaobject is called *metaobject class*. A metaobject is activated when the corresponding base level object, called *reflected object*, is invoked. When this happens, the associated metaobject executes the corresponding *metamethod*. This method determines the actions to be developed and passes control to the base level object. This allows [14][21] objects of the same class to have associated different metaobjects and, for instance, to build a N metalevel architecture.

3.3. Meta-object Protocol

In order that both levels work correctly together, there must be a definition or protocol of how to communicate (Figure 1). This "communication" protocol between the metaobject and object is called *metaobject protocol*. This protocol must be predefined and any interaction made between this object and the correspondent metaobject is fixed and made through this protocol [15]. The link between the metaobject and the object can be made in two ways; can be fixed statically at compile time or load time, or use a more flexible approach and be defined at runtime [15][14].

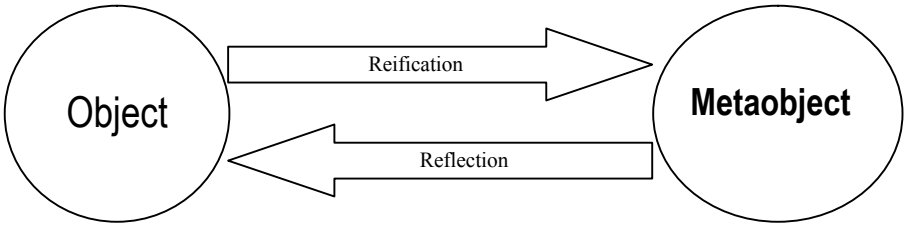

Figure 1.Relation between Object and MetaObject

The procedure of the reflective mechanism is the following: every time an invocation is made to a base level object, this redirects control to the thread of control of the associated metaobject. At the metalevel, the metaobject develops the reflected behaviour needed and them returns the control back to the base level object.

It is called *reified information* to the computational behaviour that is transformed into data. These reified entities constitute the *meta-information* to which the reflection

mechanisms (metaobjects) perform operations [14][15]. There is also a meta-information protocol whose function is to provide the programmer with information if the program components that are known and treated by the compiler or interpreter [14]. When changes are made to this information, they are reflected to the base level objects [14] [15]

Message interception between objects at the base level is the most usual mechanism used to activate metaobjects. When a message is intercepted, it is delegated to the associated metaobject [14]. There are three types of association: run-time reflection, load-time reflection and compile-time reflection. Here a trade-off between flexibility and efficiency must be considered.

In runtime reflection, a metaobject is an instance of a class and exists at runtime. Although it seams that metaobject and object are separated runtime entities, this not always happens (they may coexist within the same address space) [14] [15]

The advantages of run time reflection are the high flexibility and adaptation capabilities. The drawback of this approach is the overhead caused by the reflective computation at run time.

In compile time reflection, objects exist only at compile time. It can be seen as pre processing of source code. The advantages are run time efficiency, since the overhead only occurs at compile time. Of course that compile time is not enough when it is needed to change object behaviour and or attach metaobjects to objects at run time [14] [15]

Load time reflection is done when loading and linking the program (before executing it). As we can see, this as the same drawbacks as the compile time reflection. Here the overhead only exists at the program start up time [14].

3.4. Flexibility

The main difference between the structural model and the behavioural model is that in the first case the association is made between classes and metaclasses and in the second case the association is made between objects and metaobjects. After reviewing both concepts it is clear that the behavioural reflection model is more flexible than the structural one. One important issue to have in mind when considering the use of reflection is the fact that there must be a balance between flexibility and efficiency. Serious reasoning regarding the overhead caused by the reflective computation must be taken into account for hard real time systems.

4. A Reflection-based Framework

"Imagination is more important than knowledge"
Albert Einstein

After discussing the basic concepts of monitoring and computational reflection, we can now start to deem a new monitoring approach directed to hard real time systems. As stated in [22], *"we cannot control what we do not measure"*. When considering hard real time systems, as seen before, this is particularly more difficult since most of them are embedded. Due to this, and many other similar aspects, we must bare in mind that we must *"see"* what is happening within the system, and communication between the system and the outside world must happen at some point.

Monitoring by itself brings extra burden to the system being monitored, independently of how it is monitored. The dilemma of balancing between the amount of interference with the system and the amount of information driven from the system under observation will always be present. From the several factors present when considering monitoring, the following requirements must be carefully considered:

1. What (levels of abstraction) and when (triggering) to monitor,
2. How to monitor (intrusive and probe effect considerations) and,
3. How to obtain accuracy (balance between interference and information).

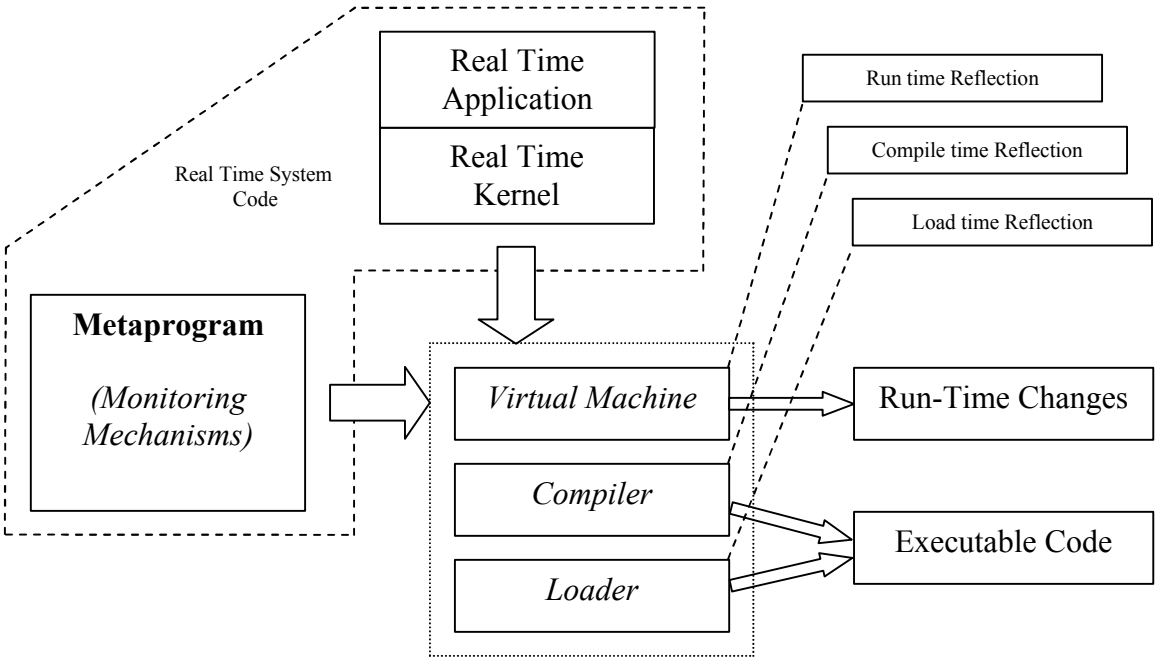

Figure 2. Proposed Framework

The proposed framework (Figure 2) takes advantage of the computational reflection mechanisms to abstract the concept of monitoring from the rest of the real time system, thus separating the system implementation from the these requirements. The main objectives are to give emphasis to the concept of monitoring, providing a conceptual separation between the real time kernel code and the monitoring code and to open the discussion about the integration of computational reflection in hard real time systems.

In order to monitor the system, the meta program contains the monitoring mechanisms and reflects them to the underlying real time kernel. As we have seen before, reflective mechanisms must be provided either by the programming language or by the kernel on which the system is being developed. These mechanisms are solely responsible for

getting the required information and passing it to the outside world. At this point we only consider the information retrieval process. Controlling the monitoring system itself is still a topic of ongoing research. As we can see, these monitoring mechanisms must contain knowledge about "how to get the information out", namely, information about output channels of the system. Even if the system has no output channels, one can always consider logging on the system for posterior analysis.

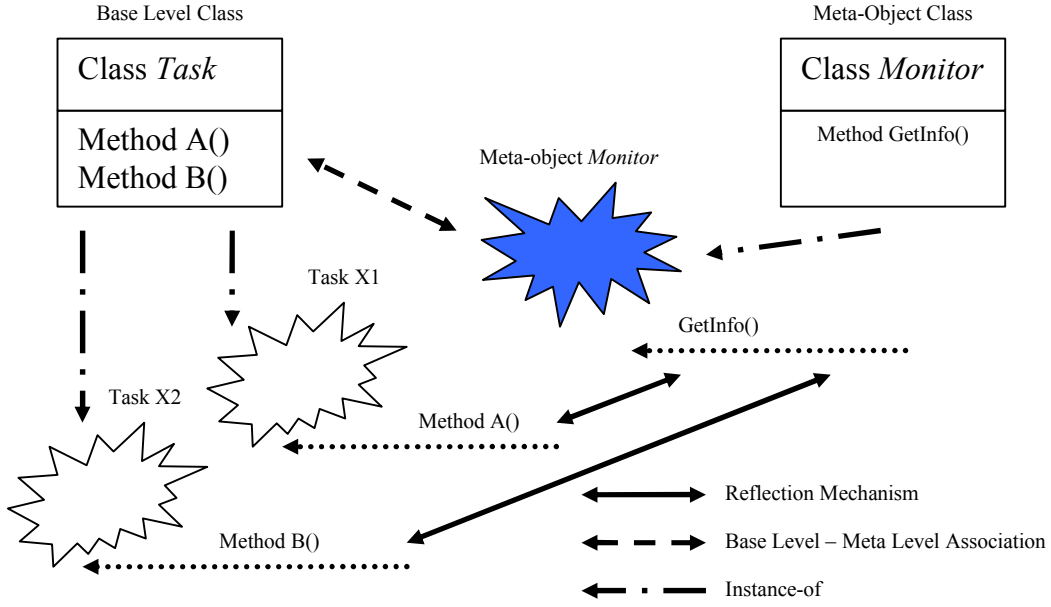

Figure 3. MetaObject example

Figure 3 provides an example of a monitoring mechanism. The meta-object class *Monitor* defines a monitoring method called *GetInfo()*. This class is implemented independently from the Task class and implements the necessary behaviour to collect the required information from the tasks. When both classes are created the association between both base level and meta level objects can be created. The meta object will collect the information regarding the two tasks each time a message is received by any of the objects. When a message is received, control is passed to the meta object which will execute its method and them passes control back to the base level object (task).

4.1. The Reflection Framework

By separating the monitoring instrumentation from the rest of the system (Figure 4), this framework introduces a certain level of abstraction and flexibility to the system. Of course that we must take into account the fact that this mechanism will consume computational resources and, consequently, they must be adequately considered in the system's analysis and design.

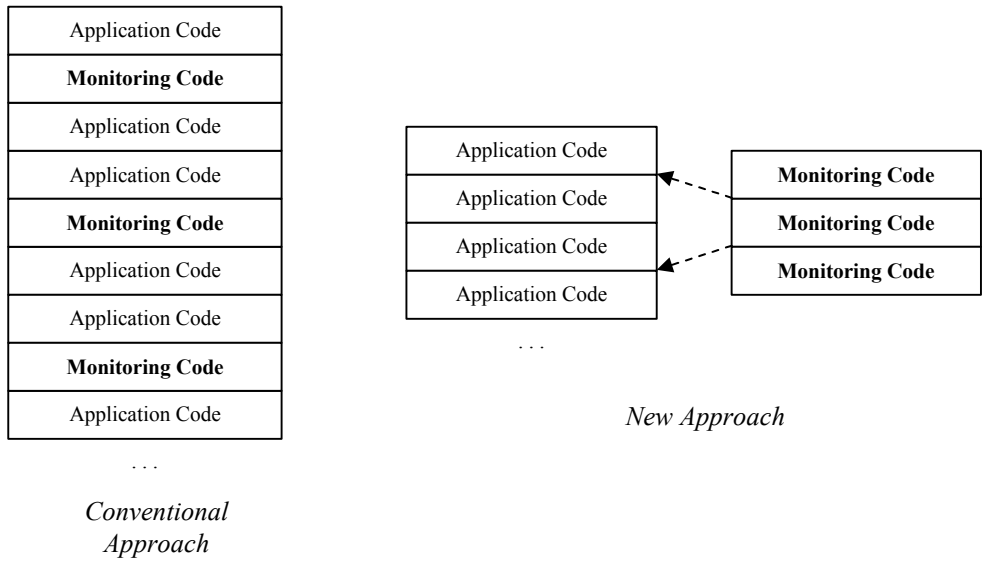

Figure 4. Conventional Monitoring vs New Approach

One of the most important aspects of hard real time systems development, is the development process itself. All hard real time system development process must be, in one way or another, certified to assure failure rates extremely low [1]. As we have seen before, this makes the development process very resource consuming and still failures, unfortunately, occur [1] [6]. With the separation of the monitoring mechanism from the mainstream development of the real time system, we further improve the focusing on the important aspects of the development without much concern with the monitoring features, introducing the Monitoring System as a separate concern in the systems' development (Figures 5 and 6).

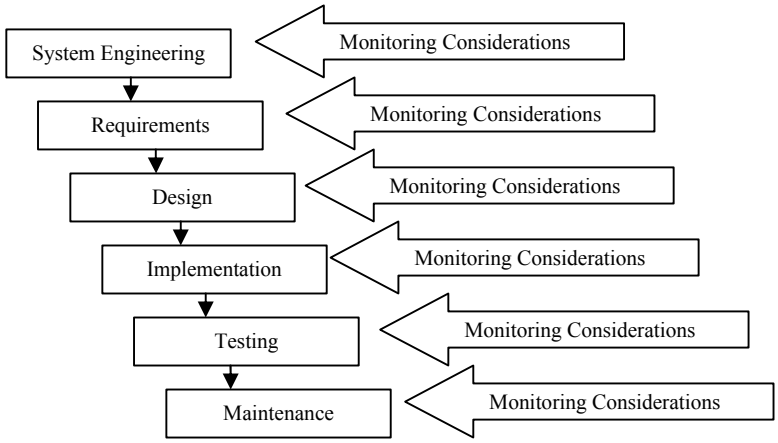

Figure 5. Real time system development

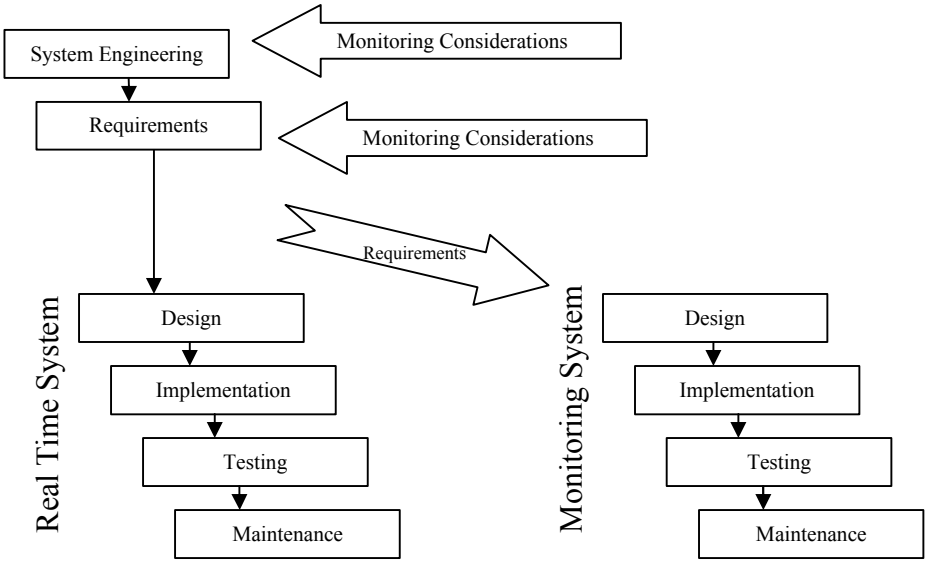

Figure 6. Separation Monitoring/Monitored System

While part of the development resources are focused on developing the system with real time concerns, another part is concerned with monitoring, that may or may not have real time restrictions, and is focused in assuring and verifying that the real time system fulfils its requirements.

Another advantage comes from the fact that all communication issues are removed from the real time system context and pass directly to the monitoring system, making the system even more versatile and clean.

Even after a monitoring mechanism has been developed for a particular real time system, with this approach, it can be adapted to other systems, without the need for application code changing. This is a major advantage if we consider the quantity of commercially available real time operating systems and application specific solutions. This way, we only need to know what we want to observe and tailor the monitoring mechanism accordingly. To some extent, we can even make the monitoring mechanisms self adaptable to the new system through the dynamic properties of computational reflection.

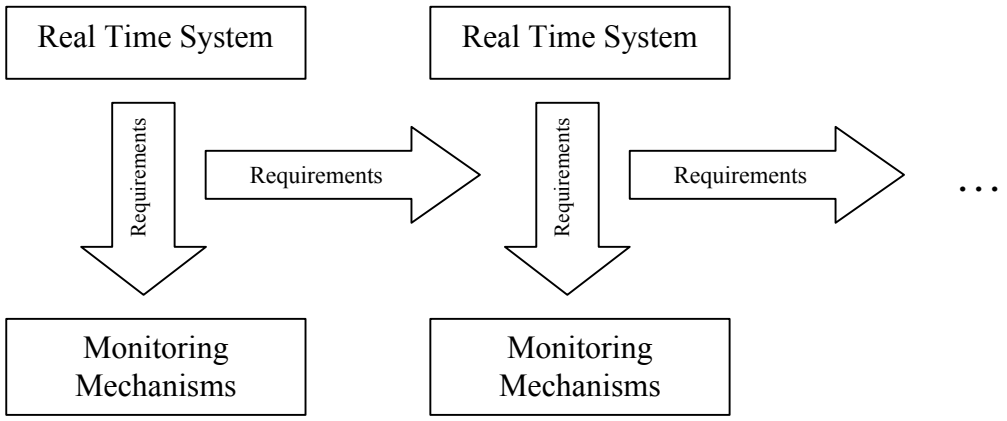

Figure 7.Augmenting monitoring capabilities

By adapting the monitoring mechanisms to other systems we are increasingly improving the monitoring capabilities when adding more monitoring points to the original system (Figure 7).

Furthermore, through modular development, we can adapt only the monitoring parts a system requires instead of the whole monitoring system. In a way we will have a benchmark of monitoring features we can use and the capability to create more based on the system requirements.

Monitoring features' grouping is also a concept to consider (Figure 8). If monitoring mechanisms are grouped with similar ones, we can achieve higher levels of abstraction

when thinking of monitoring requirements. We can even manage to select which mechanisms we want to include in the application.

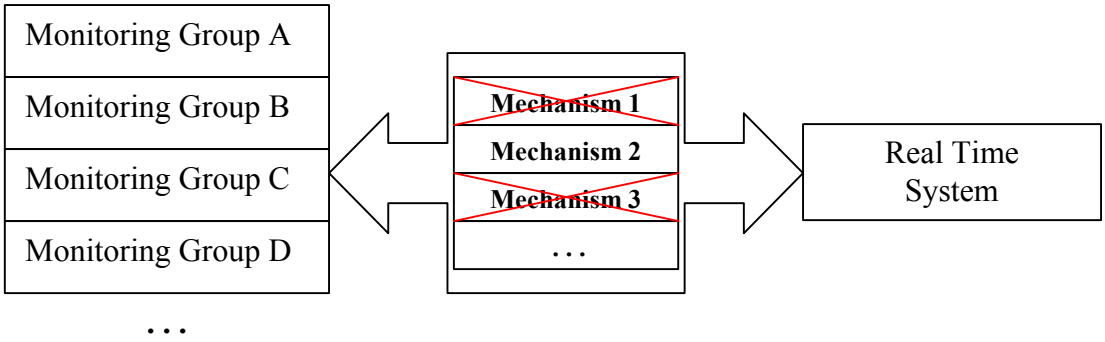

Figure 8. Grouping

An interesting concept of computational reflection is the ability to change its behaviour at run time. With this we can *"transform"* our monitoring mechanisms into something different or change their place in the system. For instance, if we are monitoring a specific task, and we want to monitor a different task (Figure 9), we just reflect the desired mechanism to the specific task and change and adapt as required.

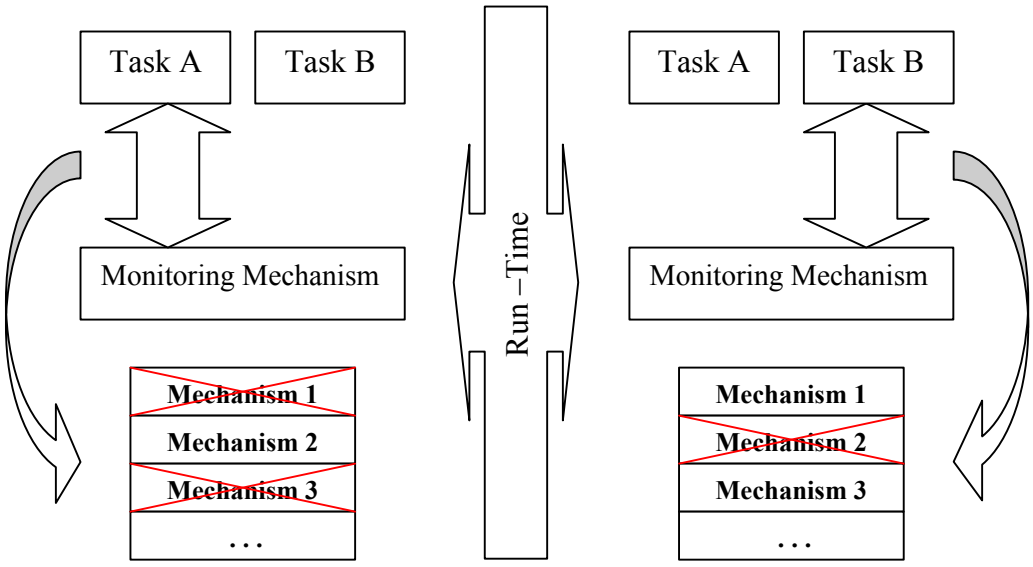

Figure 9. Use of dynamic properties

4.2. Discussion

Traditional approaches to monitoring have a considerable impact on the entire development process and consequently on the system being developed. We have presented a non exhaustive list of advantages when relating computational reflection with hard real time systems monitoring. Nevertheless, many aspects like the overhead and determinism effect caused by the computation resources required by the reflective mechanisms must be taken into account.

Aspects like, design separation, portability, constant refinement of monitoring requirements, selection of desired monitoring mechanisms and dynamic behaviour changing, make this approach a very interesting approach to real time monitoring. They allow removing some of the burden imposed to the system by separating the monitoring mechanisms from the mainstream application code.

5. Conclusions

Monitoring is a very important aspect to consider when developing hard real time systems. The need to develop new theories about monitoring is increasing as real time critical systems are more used worldwide, namely, a consistent approach to the subject is a obligation. Research is also trying to solidify the knowledge into something more usable, but the quest for a solid and secure theory on monitoring is yet to come.

This paper proposes the use of a Reflection-based monitoring framework, in order to deal with this problem. Reflection may limit the impact that monitoring mechanisms have in the monitored system, an important issue for real-time and critical systems. With this in mind, we present different approaches for real-time systems monitoring, discussing its merits and drawbacks, and providing a framework for the integration of

the Reflection technology within the developed systems. Reflection is a promising approach to the problem, enabling to obtain a powerful solution for monitoring.

References

[1] Thane H., *Monitoring, Testing and Debugging of Distributed Real Time Systems*, Ph.D. Thesis, MRTC Report 00/15;

[2] Farnam Jahanian, *Run Time Monitoring of Real Time Systems*;

[3] Butler, R.W. and Finelli, G.B. *The infeasibility of quantifying the reliability of life-critical real-time software*. IEEE Transactions on Software Engineering, (19): 3-12, January, 1993;

[4] Tsai, J., Bi, Y., Yang, S., Smith, R., *Distributed Real-Time Systems - Monitoring, Visualization, Debugging, and Analysis,* John Wiley & Sons, New York, USA, 1996.

[5] Gergeleit M., *A Monitoring-based Approach to Object Oriented Real Time Computing*, Ph.D. Thesis, December 2001;

[6] Leveson N. and Turner C. *An investigation of the Therac-25 accidents.* IEEE Computer, 26(7):18-41, July 1993;

[7] Michael Mock, Martin Gergeleit, Edgar Nett, *Monitoring Distributed Real Time Activities in DCOM*, Proceedings of ISORC'2K, 2000;

[8] S. L. Graham, P. B. Kessler, and M. K. McKusick, *gprof: a call graph execution profiler* , Proc. SIGPLAN'82 Symp. Compiler Construction, 1982, pp. 120-126;

[9] Audsley N. C., Burns A., Davis R. I., Tindell K. W. *Fixed Priority Pre-emptive Scheduling: A Historical Perspective*. Real-Time Systems journal, Vol.8(2/3), March/May, Kluwer A.P., 1995;

[10] Hatton L.. *Unexpected (and sometimes unpleasant) Lessons from Data in Real Software Systems*. 12th Annual SR Workshop, Bruges 12-15 September 1995. Proceedings, pp. 251- 259. Springer. ISBN 3-540-76034-2;

[11] Barbosa R., Maia R, Pinho L. M., *Monitoring and Profiling Business and Missions Critical Real Time Systems*, WIP Session Proceedings, Euromicro Conference on Real Time Systems, July 2003. ISBN 972-8688-11-3;

[12] H. Masuhara, S. Matsouka, and A. Yonezawa. *Designing an OO Reflective Language for Massively-Parallel Processors.* Proc. of Workshop OOPSLA'93;

[13] S. Chiba. *A Metaobject Protocol for C++*. In R. Wirfs-Brock, pages 285-299;

[14] Claudia A. Marcos, *Design Patterns as First Class Entities*, Ph.D. Thesis, UNICEN University, 2001;

[15] J.C. Fabre, *Object Orientation And Fault Tolerant Systems – An Overview and Some Examples*, LAAS Report 98088, CNRS, 1998;

[16] R. Gabriel. *The failure of pattern languages*. Journal of Object Oriented Programming, JOOP, pages 84-88, February 1994;

[17] M. Campo. *Compreensão Visual de Frameworks através da Introspecção de Exemplos*. Ph. D. Thesis (in Portuguese), UFRGS, Brazil, 1997;

[18] P. Cointe. *A Tutorial Introduction to Metaclass Architecture as provided by Class Oriented Languages*. Proc. of the International Conference on Fifth Generation Computer Systems, pages 592-608, 1988, ICOT editor.

[19] P. Maes. "*Concepts and Experiments in Computational Reflection"*, in Proc. Of OOPSLA'87, Orlando USA, 1987, pp. 147-155.

[20] M. Dahchour, A. Pirotte, and E. Zimanyi. *Metaclass Implementation of Materialization*. TEEE Trans. on Knowledge and Data Engineering, 1998,

[21] M. Lisboa. *MOTF: Meta-Objetos para Tolerância a Falhas*. PhD. Thesis (in Portuguese), UFRGS, Brazil 1995.

[22] DeMarco, Tom, and Timothy Lister. *Peopleware: Productive Projects and Teams*, New York: Dorset House Publishing Company, 1987.