Reflective Real-Time Systems*

John A. Stankovic
Department of Computer Science
University of Massachusetts

June 28, 1993

Abstract

This paper describes how the notion of a reflective architecture can serve as a central principle for building complex and flexible real-time systems, contributing to making them more dependable. By identifying reflective information, exposing it to application code, and retaining it at run time, a system is capable of providing predictable performance with respect to timing constraints, of reacting in a flexible manner to changing dynamics in the environment including faults, to be more robust to violations of initial assumptions, to better evolve over time, and even for better monitoring, debugging, and understanding of the system. Advantages of this approach are given and details on the specific implementation of a reflective architecture are provided. Several open questions are also presented.

KEYWORDS: Fault tolerance, adaptability, flexibility, dependability, real-time, scheduling, critical tasks, and reflective architectures.

1 Introduction

Building real-time systems for critical applications and showing that they meet functional, fault, and timing requirements are complex tasks. At the heart of this complexity there exist several opposing factors. These include:

- the desire for predictability versus the need for flexibility to handle non-deterministic and complex environments, failures, and system evolution,
- the need for efficient performance and low cost versus understandability,
- the need for abstraction to handle complexity versus the need to include implementation details in order to assess timing properties, and
- the need for autonomy (to better deal with faults and scaling) versus the need for cooperation (to achieve application semantics).

*This work has been supported, in part, by NSF under grants IRI 9208920 and CDA 8922572, by ONR under grant N00014-92-J-1048, and by CNR-IEI under the PDCS project.

It is also a common understanding that we want integrated system-wide solutions so that design, implementation, testing, monitoring, dependability, and validation (both functional and timing) are all addressed. This paper argues that one good approach is a reflective system architecture that **exposes** the **correct** meta-level information. Exactly what this information should be and how it should be supported will be the subject of research for many years. However, in this paper we provide details on our current view of this information and its implementation structures. Once a system has this information, it can be dynamically altered as system conditions change (increasing the coverage of the system dynamically), or as the system evolves over time, or is ported to another platform. In particular, the system is built with the idea that it cannot know the environment completely, but must nevertheless be dependable. With a reflective architecture the necessary flexibility for complex real-time systems can be attained. Further, we argue that this powerful type of flexibility contributes to many other things that are required including understandability, analyzability (and therefore predictability), high performance, monitoring, debugging, and dependability[1]. Finally, we note that much of what is discussed here has been implemented [18] and is now being used on a robotic automated assembly application in complicated environments.

In Section 2 we briefly discuss the notion of reflective computing and present the state of the art. In Section 3 we outline a methodology for complex real-time systems where reflection is a central principle. In Section 4 we identify ingredients of such a reflective architecture and provide details on its implementation. In Section 5 we summarize our contributions, the status of the implementation, and identify required future work.

2 State of the Art – Reflection

Reflection is defined as the process of reasoning about and acting upon the system itself. Part of this action may be altering the systems own structure from within. Normally, a computation acts to solve a problem such as sorting a file or filtering important signals out of radar returns. If, in addition, there is computation about the sorting process or filtering process itself, then we have a meta-level structure and reflective possibilities. Consequently, if using a reflective architecture, we can consider a system to have a computational part and a reflective part. The reflective part is a self-representation of the system, exposing the system state and semantics of the application. Such information can then be monitored, used in decision making during system operation, and easily changed thereby supporting many flexibility features. Much research is still required to identify what meta-level information is important including timing and dependability information, how to represent it, and determining the performance implications including predictability of the meta-level.

Reflection is not new and many systems have some reflective information in them. However, identifying reflection as a key architectural principle and exploiting it is less common. Reflection has appeared in AI systems, e.g., within the context of rule based and logic based languages [10, 17]. More recently it is being used in object-oriented languages and object-oriented databases in order to increase their flexibility [5, 11]. Combining object oriented programming with reflection seems to be very important since object oriented programming

[1] In this paper we do not consider all aspects of dependability; we address reliability, availability, and safety, but not security.

supports abstraction and good design and adding reflection supplies flexibility. Reflection also has been touted as valuable to distributed systems for supporting transparency and flexibility [22]. While reflection is not new, using reflection in real-time systems is new. In fact, the only work we are aware of is our own work [19] that reported on a preliminary step in developing a reflective architecture for real-time systems. This paper extends that work with a more complete identification of reflective information as a central principle, demonstrating its implementation support in a functioning system, and extending it for adaptive fault tolerance.

One problem is that reflection is a term used somewhat differently in three different areas: AI, object-oriented programming, and object-oriented databases. The differences emerge from two issues: one, it is a matter of degree as to what is considered reflective and what is considered computational; not always easy to identify, and two, some definitions require reflection to be able to generate code at run time similar to how LISP has a duality between code and data. In this paper we identify what we consider reflective and computational for complex, dependable, real-time systems, and prohibit newly generated code because any such dynamic code generation would be difficult to dynamically analyze with respect to timing and dependability properties.

3 Methodology

While most of this paper deals with reflection, here we briefly present an overview of a design methodology for complex real-time systems. We do this, in part, to emphasize that reflection by itself is **not** sufficient to address the problems raised in the Introduction, and, in part, to place what it does provide in perspective. Reflection can be used independently of the overall design paradigm, e.g., it can be used with an object oriented design and implementation or with a more classical procedure oriented design and implementation. Because of the obvious advantages of an object oriented design at the functional level, we briefly discuss a vision of how a complex real-time system could be developed under this approach. This would contribute to dependability. However, significant performance questions still exist for object oriented real-time systems. In particular, concurrency is not addressed well enough yet, overheads for support of objects are still too high, and predictability of these run time structures has not been established nor even seriously considered. Because of these reasons, in the actual prototype system we have developed [12, 13, 18], we currently employ a procedure based design and programming paradigm which is extended to integrate with reflection.

A vision for the design of complex real-time systems might begin with using a concurrent object oriented approach subject to the integrating theme that the system is reflective. The object oriented aspects provides modularity, abstraction, information hiding, library modules with different properties, etc. The hierarchies created within object classes can contain the reflective information and proceed from more abstract levels down to specific implementations where the implementations may have varying runtime costs and different fault semantics as a function of the actual compliers, operating systems, and hardware used. In this way the objects can encapsulate both the local timing and fault tolerance requirements. We believe that the object oriented paradigm may provide the structure necessary to constrain builders of critical applications, even when we allow a high degree of adaptability (supported by reflection). See the Conclusions for a brief statement about open questions in this regard. System-wide

composition of objects to meet system-wide timing and fault requirements is still required and can be facilitated by the reflective architecture as shown in Section 4.1. Concurrent objects can be used to model the large numbers of concurrent activities found in complex real-time systems. In addition, properties of objects which should be exposed to run time monitoring and modification need to be specified.

Since the reflective paradigm should be used across all levels of the system, what information is reflective will be somewhat application dependent. However, most complex real-time systems will require features found in operating systems, so we can apply this architecture to the operating system in a more generic manner than for higher level application code; we do this in the remainder of this paper.

4 A Reflective Architecture

Can a flexible real-time system architecture be significantly different than a general timesharing system architecture. It can, and, it must be, to handle the added complexity of time constraints, flexible operation, and dependability. As the systems get large we can no longer hand craft solutions, but require architectures that support algorithmic analysis along functional, time, and dependability dimensions. This analysis can and should be done for a dynamic system, as a completely static approach relies too much on *a priori* identification of fault and load hypotheses which are invariably wrong in complex, critical applications (See Section 4.1).

One problem we have in real-time systems is that the time dimension reaches across all levels of abstraction. Reflection has an advantage here in that at design time, at programming time, and at run time, one can reach across those layers. For example, if the programmer knows that the system retains the task importance, worst case execution time as a function of a system state, and fault semantics for use at run time, then that user may program policies that make adaptive use of this information including exact execution time costs. This allows for more efficient use of resources, all information can be dynamically altered, and new policies added more easily, than *a priori* choosing one policy and mapping to a priority where all information on how that priority was achieved is lost!

Building a real-time system based on a reflective architecture means that first we must identify reflective information regarding the system. This information includes:

- importance of the task, group of tasks, and how tasks' importance relates to each other and to system modes,
- time requirements (not just simple deadlines and periods and not just priorities),
- time profiles such as the worst case execution times or formulas depicting the execution time of the module,
- resource needs
- precedence constraints
- communication requirements
- objectives or goals of the system

- consistency and integrity constraints
- policies to guide system-wide scheduling (these policies contribute to availability and safety; shown in section 4.1)
- fault tolerance requirements and policies to guide adaptive fault tolerance (these policies contribute to reliability; shown in section 4.2)
- policies to guide tradeoff analyses
- performance monitoring information

Implementation structures in the operating system then retain this information and primitives allow it to be dynamically changed. We have done this in the Spring kernel [18] by defining sophisticated process control blocks where much of the above information is kept, and other data structures that keep more system-wide information such as properties of groups of processes. Therefore, when programming with real-time languages, in our case Spring-C and a System Description Language (SDL) [12], we identify reflective information, provide reflective information and write code that modifies it. Tools can use this information for analysis and design *knowing* that such information is also available at run time. Below we provide more specific examples of the reflective architecture for scheduling and fault tolerance.

4.1 Real-Time Scheduling

Most real-time kernels provide a fixed priority scheduling mechanism. This works when tasks' priorities are fixed. However, in general, this mechanism is *inadequate* because many systems require dynamic priorities and mapping a dynamic scheme onto a fixed priority mechanism can be very inefficient and significant information can be lost in the process. In other words the run time system has no information as to how the fixed priority was calculated, e.g., it might have been some weighted formula that combined importance and deadline and resource needs.

Fixed priority scheduling is also *incomplete* because it deals only with the CPU resource. Since we are interested in when a task completes we must consider all the resources that a task requires. An integrated view of resource management should be part of the reflective architecture interface including the ability to specifically identify the needed resources and to reserve them for the (future) time when they will be needed. So, reservations of sets of resources should be part of the architecture.

Fixed priority scheduling has *missing functionality* because it substitutes a single priority number for possibly a set of issues such as the semantic value of completing the task, the timing constraint of the task, and fault properties of the task. Further, a fixed priority ignores the fact that semantic information is often dynamic, i.e., a function of the state of the system.

When using priority based scheduling, the analysis either assumes a static system and shows that *logically* the system works (but assumption coverages are often unknown and if something unexpected occurs, then the system is uncategorized), or assumes a completely dynamic approach with average case performance; this is unacceptable for critical applications.

In order to deal with predictability versus flexibility, we propose the need for multi-level, multi-dimensional scheduling algorithms that explicitly categorize the performance including

under system degradation or unexpected events. It is multi-level in the sense that we categorize the tasks into critical (missing the deadline causes loss of life or total system failure), essential (these tasks have hard deadlines and no value is accrued if the deadline is missed, but the tasks are not critical), soft real-time (task values drop after the deadline but not to zero or some negative number), and non-real-time (these tasks have no deadlines). Each category has its own performance metric. Critical tasks must be shown to meet their deadlines and fault requirements in the worst case based on the most intelligent assessment of environmental conditions that we can make at deployment time, but **in addition** we must be able to dynamically borrow processing power and resources from the other classes of work so as to understand how much additional load and faults can be handled. This additional work could be categorized as *in the worst case*, e.g., where all tasks run to worst case times, and as *average case* where tasks execute to average execution times and where it is likely that even greater unexpected loads can be handled. Of course, no one wants to run the system in these difficult loads, but, if such loads exist the approach allows for categorization of what happens in excess load and likelihood of being able to continue. This is in contrast to some static solutions where any excess critical load is *guaranteed* to cause failure! It is also true that in many static designs all tasks are equated to critical tasks thereby increasing the cost of the system or even making it infeasible due to the combinatorial explosion of schedules that have to be accounted for in large, complex, dynamic, environments. We have developed and analyzed an algorithm for classes of tasks and where time can be borrowed from non critical tasks in a categorized manner [4]. This permits a degree of safety because critical tasks are guaranteed, robustness because unexpected events (of a certain class, i.e., excess load) are handled in a categorized manner, and availability because the system gracefully degrades rather than failing catastrophically. Details of the algorithm and its analysis are beyond the scope of this paper.

Essential tasks, soft real-time tasks, and non-real-time tasks each have a probabilistic performance metric, but these can be function of load. For example, in the absence of failures and overloads it may be that 100% of essential tasks also make their deadline, but this degrades as unexpected loads occur. Interesting approaches have been developed, including [16], where dynamic arrivals are accounted for when doing static allocation of critical periodic tasks. Other possibilities include bounding the performance such as discussed in [6, 8].

The algorithms are multi-dimensional [15] in that they must consider all resources needed by a task, not just the cpu, and they must consider precedence constraints and communication requirements, not just independent tasks. Providing algorithm support for this sophisticated level of scheduling improves productivity and reduces errors compared to having a very primitive priority mechanism and requiring the designer to map tasks to priorities accounting both for worst cases blocking times over resources and interrupts. While the details of this type of algorithm are again beyond the scope of this paper, what is important about it, is that it dynamically uses reflective information about the tasks requirements (importance, deadline, precedence constraints, resource requirements, fault semantics, etc.).

We also use scheduling in planning mode as opposed to myopic scheduling[2]. When tasks arrive, the planning based scheduling algorithm uses the reflective information about active

[2]Myopic scheduling refers to those algorithms which only choose what the next task to execute should be. At run time these algorithms do not have any concept of total load, nor whether any or all of the tasks are likely to miss their deadlines.

tasks and creates a full schedule for the active tasks and predicts if one or more deadlines will be missed. If deadlines would be missed, error handling can occur before the deadline is missed, often simplifying error recovery. Further, because reflective information is available, the decision as to how to handle the predicted timing failure can be made more intelligently and as a function of the current state of the system as opposed to some *a priori* chosen policy. The planning based scheduling with its inherent advantages is implemented in the Spring kernel [18, 15] and a scheduling chip has been designed to reduce its run time overhead [3].

4.2 Fault Tolerance

Many real-time systems require adaptive fault tolerance [7] in order to operate in complex, highly variable environments, to keep costs low (rather than a brute force static approach that replicates everything regardless of environmental conditions), and so that redundancy and control can be tailored to the individual applications functions as it is required by those individual functions. The reflective architecture is a suitable structure for adaptive fault tolerance. We now briefly discuss some of the details of what constitutes our current view of a reflective architecture for adaptive fault tolerance.

In particular, we have developed a framework and notation for software implemented, adaptive fault tolerance in a real-time context, called FERT (Fault Tolerant Entity for Real-Time) [2], which adheres to the reflective architecture approach. This work extends previous fault tolerance work [1, 14, 9] in two main ways: by including in the notation features that explicitly address real-time constraints, and by a flexible and adaptable control strategy for managing redundancy. The reflective aspects of FERT can be divided into two parts: the reflective information itself and the overall structure.

Information regarding timing constraints, importance of tasks (specified as values and penalties), levels and types of redundancy, and adaptive control of redundancy are all part of the reflective information. All this information is visible at design, implementation, and runtime, allowing it to be dynamically updated and used by on-line policies. For design time, the generic design notation can specify information such as whether n-copies and a voter, or primary-backups, or imprecise computations are required, and to notify the scheduling mechanisms (both off-line and on-line algorithms) of relative importance of tasks, their timing requirements and their worst case and average case use of resources. This specification is part of the reflective architecture which links the design to both off-line analysis and on-line scheduling. Additionally, a FERT has its own structure which is exposed to all levels. The structure consists of (1) ports through which all inputs and outputs pass, (2) application modules which implement the functionality and voting or adjudication of the FERT, if needed, and (3) a control which specifies how application modules interact with each other and with the runtime system. The control part is meant to specify adaptive strategies that take into account available resources, deadlines, importance, and observed faults.

The control part uses four generic primitives which we believe should be visible in a reflective architecture supporting dependable computing:

- **possible**: asks whether a collection of tasks can be feasibly scheduled; multiple **possible** requests can be issued in parallel; this allows dynamic analyzability with respect to meeting timing constraints;

- **exec**: identifies a list of tasks that must be executed subject to various constraints; typically used by the control when finally deciding what should execute given the results of the **possible** queries
- **unused**: identifies resources which were planned to be used but no longer required, e.g., because the initial version of the task completed successfully and the backups are no longer required; improves performance of the system
- **output**: finally commit the produced results

In summary, the reflective architecture for supporting adaptive fault tolerance permits the designer to specify time, importance, redundancy and control information *knowing* that the run time structure will retain this information, so that the adaptive control policies implementor can use and modify such information. In particular, the interaction with the dynamic scheduler permits more flexibility while preserving timing related predictability (defined in the sense given in Section 4.1). Since the intent of this section is to focus on reflective architectures for fault tolerance, we are brief and don't discuss many other issues related to fault tolerance. Interested readers should see [2].

4.3 Summary of Reflective Architecture

Current real-time systems platforms present inadequate, incomplete, and missing functionality in their architectures including at the interface to scheduling and for fault tolerance. As a result of these poor interfaces, critical real-time systems are difficult to design, maintain, analyze, and understand. The systems tend to be inflexible and productivity is low. We need to raise the level of functionality that the runtime platform provides to better provide portability, productivity, and lower costs. The reflective architecture has potential in these areas.

Further, in order to support dependable real-time systems in complex and unfriendly environments, the current goals, policies and state of the system must be available for dynamic access and modification. However, the dynamics must be carefully controlled. Establishing designs and solutions that engineer good tradeoffs among the opposing factors listed in the Introduction will be difficult and the subject of research for many years. However, the reflective architecture has many nice properties that provide a structure for engineering those tradeoffs.

5 Conclusions

In this paper we present a view of how complex real-time systems for critical applications might be built based on a reflective architecture. We identify specific reflective information and architectural designs for both real-time and fault tolerance requirements. The reflective architecture enhances certain aspects of dependability by (i) addressing time constraints at all levels of the system so that more accurate timing analysis can be done even early in the design, (ii) facilitates the use of scheduling algorithms that are robust to violations in the initial assumptions, (iii) supports planning based scheduling that allows detection of timing errors early and application of state and time dependent recovery strategies, and (iv) supports adaptive fault tolerance so that there is flexible management of redundancy subject to time constraints.

While final verification that this approach will succeed in practice remains to be demonstrated (i.e., it has not been used on real safety critical applications), we have demonstrated many aspects of it in many ways. For example, a system description language (SDL) has been designed and implemented [12] along with extensions to the programming language C, called Spring-C. These languages provide specification of and access to reflective information. A compiler [13] has been implemented that accumulate this information and make it part of the runtime structures of the Spring kernel [18]. The kernel contains the planning based scheduling algorithms referred to in section 4.1 [15], and to lower the overhead of on-line planning, a hardware scheduling chip has been designed and is currently in production [3]. Many aspects of the algorithms and implementation details have been studied and shown to be valuable both via simulation [4, 15] and in an actual distributed testbed system composed of 3 multiprocessor nodes (15 processors) [18]. The adaptive fault tolerance aspects of the reflective architecture have not been implemented, and are still in the design stage [2].

In systems where flexibility and adaptability are supported, it is usually more difficult to understand and control. Without good constraints and guidelines a designer could program haphazardly making it almost impossible to analyze. Future work includes developing good engineering tradeoffs between what is necessary for adaptability, but what is necessary for analyzability. It is also necessary to embed the approach we discussed here in an overall design methodology, such as object oriented programming, etc.

6 Acknowledgments

I would like to thank Prof. Ramamritham for jointly developing many of the basic ideas presented here, and Doug Niehaus for his various contributions to the project including major parts of the design and implementation of the system. Many others in the Spring project also deserve thanks. I thank them all. I also thank A. Bondavalli and L. Strigini for their contributions in our joint work regarding adaptive fault tolerance.

References

[1] A. Bondavalli, F. Giandomenico, and J. Xu, A Cost Effective and Flexible Scheme for Software Fault Tolerance, Univ. of Newcastle, TR 372, Feb. 1992.

[2] A. Bondavalli, J. Stankovic, and L. Strigini, Adaptable Fault Tolerance for Real-Time Systems, to appear *Proc. Third International Workshop on Responsive Computer Systems*, Sept. 1993.

[3] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J. Stankovic, Wallace, and C. Weems, The Spring Scheduling Co-Processor: A Scheduling Accelerator, to appear *Proc. ICCD*, Cambridge, MA, October 1993.

[4] G. Buttazzo and J. Stankovic, RED: A Robust Earliest Deadline Scheduling Algorithm, *Proc. Responsive Systems Workshop* to appear Sept. 1993.

[5] M. Ibrahim, editor, *Proc. OOPLSA 91 Workshop on Reflection and Metalevel Architectures in Object Oriented Programming*, Oct. 1991.

[6] R. Karp, On-line Versus Off-line Algorithms: How Much is it Worth to Know the Future, *Information Processing*, North Holland, Vol. 1, 1992.

[7] K. Kim and T. Lawrence, Adaptive Fault Tolerance: Issues and Approaches, *Proc. Second IEEE Workshop on Future Trends in Distributed Computing Systems*, Cairo, Egypt, 1990.

[8] G. Le Lann, Why Should We Keep Using Precambrian Design Approaches at the Dawn of the Third Millennium?, *Proc. Workshop on Large, Distributed, Parallel Architecture Real-Time Systems*, March 1993.

[9] C. Liu, A General Framework for Software Fault Tolerance, PDCS Second Year Project Report, ESPRIT Project 3092, 1991.

[10] P. Maes, Concepts and Experiments in Computational Reflection, *OOPSLA 87*, Sigplan Notices, Vol. 22, No. 12, pp. 147-155, 1987.

[11] S. Matsuoka, T. Watanabe, Y. Ichisugi, and A. Yonezawa, Object Oriented Concurrent Reflective Architectures, 1992.

[12] D. Niehaus, J. Stankovic, and K. Ramamritham, The Spring System Description Language, UMASS TR-93-08, 1993.

[13] D. Niehaus, Program Representation and Translation for Predictable Real-Time Systems, *Proc. RTSS*, pp. 43-52, Dec. 1991.

[14] D. Powell, et. al., The Delta-4 Approach to Dependability on Open Distributed Computing Systems, *Proc. 18th Int. Symp. on Fault Tolerant Computing*, Tokyo, Japan, 1988.

[15] K. Ramamritham, J. Stankovic, and P. Shiah, Efficient Scheduling Algorithms For Real-Time Multiprocessor Systems, *IEEE Transactions on Parallel and Distributed Computing*, Vol. 1, No. 2, pp. 184-194, April 1990.

[16] K. Ramamritham and J. Adan, Providing for Dynamic Arrivals During the Static Allocation and Scheduling of Periodic Tasks, UMASS TR-90-107, Oct. 1990.

[17] B. Smith, Reflection and Semantics in a Procedural Language, MIT TR 272, 1982.

Predictability for Real-Time

[18] J. Stankovic and K. Ramamritham, The Spring Kernel: A New Paradigm for Real-Time Systems, *IEEE Software*, Vol. 8, No. 3, pp. 62-72, May 1991.

[19] J. Stankovic, On the Reflective Nature of the Spring Kernel, *invited paper*, *Proc. Process Control Systems '91*, Feb. 1991.

[20] J. Stankovic, Real-Time Kernel Interfaces, *Real-Time Systems Journal*, Vol. 5, No. 1, pp. 5-8, March 1993.

[21] J. Stankovic, Resource Allocation in Real-Time Systems, *Real-Time Systems Journal*, Vol. 5, No. 2-3, May 1993.

[22] R. Stroud, Transparency and Reflection in Distributed Systems, position paper for *Fifth ACM SIGOPS European Workshop*, April 1992.

[23] B. Wyatt, K. Kavi, and S. Hufnagel, Parallelism in Object Oriented Languages: A Survey, *IEEE Software*, Vol. 6, Nov. 1992.