Time-Embedded-Reactive RTOS for Embedded Application's

Pallavi Joshi
Department of Computer Science and Engineering
Vidhya Vikas Institute of Engineering and Technology, Mysore, India

Abstract— Software development for small, real-time and resource controlled, embedded systems is becoming increasingly intricate. To be able to guarantee robustness and consistency, the underlying groundwork should not be based upon adhoc solutions. In this paper we identify three tactical features of a minimalistic Real-Time Operating System (RTOS), and presents the run-time system of Timber, a reactive deadline-driven programming language. We analyse the functionalities of the run-time system in the light of real-time requirements, and highlight the significance of integrating an adequate concept of time, both semantically in the programming interface as well as part of the run-time system.

Key words: RTOS, "Time - embedded– Reactive"

I. BACKGROUND

Software development for small, real-time and resource constrained, embedded systems is becoming more and more complex. In order to guarantee robustness and reliability, the underlying groundwork should not be based upon ad hoc solutions. For this reason real-time operating system (RTOS) features for small embedded systems have gained recent interest.

We have identified a set of anticipated key features. A minimalistic RTOS should at least:

- supply suffecient infrastructure for reactive parallel Programming,
- preserve state integrity, and
- realize real-time constraints.

In addition, it is beneficial if the RTOS and its programming interface provides the ability of formal reasoning about system properties, which would be useful towards not dangerous and minimal system dimensioning. It is also necessary if this can be consummate without limiting the expressive power of the programming interface.

RTOS's in general are too substantial weighted for small embedded devices, but a few proposals have been presented. Among them are systems like Contiki[10], PicoOS[3], FreeRTOS[2], Nucleus RTOS[1], and TinyOS[12,11,5]. All of these systems have their matchless characteristics, but we will, in short, only present the last one. TinyOS is a minimal operating system suitable for the smallest embedded devices. It is based upon a module structure and a reactive event based concurrency model. Its core footmark is only about 500 bytes. Robustness is partly succeeded by a static analyser for data race discovery. The key problem that still remains unsolved in TinyOS, as well as in all the others, is the awareness of real-time constraints in run-time. To be able to semantically guarantee certain degree of robustness, TinyOS has restrictions in terms of expressive power. For instance, dynamic storage allocation is not allowed.

II. TIMBER - A TRANSITORY INTRODUCTION

Timber, "Time - embedded– Reactive", Timber is a general programming language specifically aimed at the construction of complex event-driven systems. It allows programs to be accessibly structured in terms of objects and reactions, and the real-time behaviour of reactions can additionally be exactly so controlled via platform-independent timing constraints. This property makes Timber particularly suited to both the specification and the implementation of real-time embedded systems.

Timber is a direct descendant of O'Haskell, which was technologically advanced at Chalmers University of Technology in 1999. O'Haskell extended the lazy functional programming language Haskell with object-oriented perceptions such as methods, classes and subtyping, while retaining the purely functional execution model of its ancestor. It also introduced the characteristic notion of parallel reactive objects that Timber subsequently has adopted.

In brief, the language is based upon parallel executing reactive objects .The inter-object communication is message-based by means of synchronous and asynchronous message sends. A message send is equivalent to pleasing a method of the recipient object. Even though Timber is a general purpose language , it is primarily designed to target embedded systems, and we will discuss the facets of the language in the context of embedded programming.

A. Records and Objects

Besides primitive data types such as integers, floating point numbers, etc., Timber embraces user-defined records and primitive types to support object-orientation. Records can either be used to define incontrovertible data or to designate interfaces to objects. Timber objects basically consists of two parts, an internal state and a communication interface. An object is instantiated by a template construct, which in turn defines the early state of the object and its communication interface.

The primitive object-oriented types are Action, Request, and Template, which all are subtypes of Cmd. The meaning of the Action and Request types are asynchronous and synchronous message sends, respectively. These actions and requests are communally called methods. Template is the type defining the template command from which objects are created. A Timber program has to take in a specific main template. The communication interface of this template is system dependent. For embedded devices the input interface usually comprises a reset method and bindings from interrupts to actions. The output interface is the environment in which the Timber program will operate. In the context of embedded devices, it shall at least provide methods to read from and write to ports. At system startup, the main template command will be executed, creating an occurrence

of the object main and then executing the reset method. The system will supply the main object with its environment.

B. Methods

A method is invoked by a message send command, any an asynchronous action or a synchronous request. A Timber program running on an embedded device can be seen as a set of concurrent objects, all awaiting external inducements initially instigated by interrupts. A method can basically do three things, it can update the state of the object, create new objects, and invoke methods of other objects. After an external stimulus, the chain of reactions will eventually fade out and the system will return to the state of waiting for new inducements. We will refer to the time when the whole system is inactive and passive as the idle state, or idle time.

C. Objects as Concurrent Reactive Processes

Objects in Timber has its own execution context, or thread of control. Inter-object communication is attained by message passing. Only one method within an object can be active at a time, and the object state is only reachable through its methods. This results in mutual exclusion of state changes, frequently referred to as state integrity. Furthermore, a method cannot block the object thread indefinitely which leads to a controllable receptiveness of every object.

III. THE RUN-TIME SYSTEM OF TIMBER

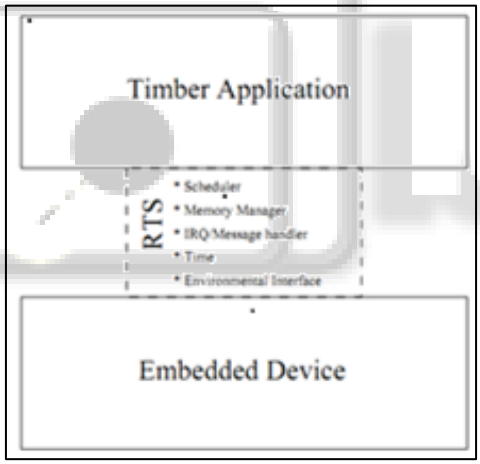

Fig. 1: Overview of Timber Run Time System

In fig(a), The functionalities of the run-time system is directly reflected by the semantics of the language. The key features that requirements to be simplified in terms of functionalities of the run-time system are as follows:

1) Scheduling

The fundamental functionality of the run-time system to attain concurrency between Timber objects, with scheduling based on the baselines and deadlines of their methods.

2) Message-passing

Providing sufficient infrastructure for the inter-object communication.

Threading: Facilitating the unique execution contexts for Timber objects.

3) Time

Capability to supply sufficient time information to make baselines and deadlines evocative.

4) Interrupt Handling

Functionality for receiving and distributing interrupts all over the system.

Environment interface: Implementation of the interface to the environment.

B. Automatic Memory Management

Timber does not depend on on explicit allocations and deallocations of dynamic data and needs an automatic memory manager to serve with junk collection.

The semantics of Timber is extremely dependent on the capability to allocate memory storage dynamically. Furthermore, as mentioned, the language does not include any explicit allocation/deallocation commands. It is thus critical to include a garbage amasser in the run-time system. A prototype of a reference counting garbage gatherer has been implemented for the run-time system of Timber and currently a copying collector is under development. The work has so far shown a set of collector characteristics but due to space limitations they are only addressed in short. A thorough description of the collector is forthcoming.

- The memory manager will reclaim garbage memory when the system is in its indolent state.
- The time needed for bookkeeping for the duration of execution of any method of the program is kept constant and small.
- The time it takes to perform a whole garbage collection cycle will at worst be proportional to the maximum amount of simultaneously live memory.
- The amount of contiguous free heap storage has to be at least the maximum amount of consecutively live memory. This however should not be misunderstood as a essential for twice as much memory than a system based on explicit allocations/deallocations. It is a rather trivial fact that, ina multi threaded system, avoidance of deallocating semantically live data will most of the times result in a lot of semantically dead data to be kept alive. This is almost impossible to measure but if such a measurement would have been talented, it would not be astonishing if it showed a factor much greater than two between the amount of data

IV. LIMITATIONS

- Compiler error messages, particularly those resulting from type errors, are far from acceptable.
- Recursive type class instances are naively engendered, sometimes resulting in extensive work duplication.
- Garbage collection of objects with a state comprising arrays is sensitive to an (unlikely) race condition that may cause state updates to be forgotten.
- The abort primitive will sometimes fail to abort a message that has been scheduled to start nonetheless yet not been received.
- Deadlock detection is currently not implemented.

V. CONCLUSION

We have shown how the run-time system of Timber facilitates the key infrastructure for reactive concurrent real-time software development. The integration of time is reliable and meaningful throughout the whole system, from the system specifications in the source code into each

scheduling decision made in run-time. The combination of reactiveobjects and controlled use of mutable state eliminates the risk of data races and preserves state integrity. Dynamic memory storage allocation and garbage collection dismisses the programmer from the error-prone task of manual memory management. The graph of objects may also, due to the capability to create objects dynamically, change rapidly over time, resulting in a more agile system than the case where the run-time structure is motionless. In contrast to TinyOS, Timber does not require any limitations in expressive power to avoid race conditions, this is instead guaranteed by the language semantics. Furthermore, the core of the run-time system has a greater potency (scheduling, memory management, etc.) than the corresponding parts of TinyOS, without familiarizing any unsafe real-time characteristics..

Timber mainly lacks two things in comparison to other minimalistic RTOS's such as TinyOS. First of all, TinyOS is well adopted, well experienced, and well understood by practitioners. It has been obtainable for several years and many practitioners have joined in. Second of all, the heritage of TinyOS is a lot more known by general practitioners due to the wellknown imperative language C [13]. We cannot even start to compare C with Haskell in terms of how many well-skilled practitioners each programming paradigm has.

REFERENCES

[1] Accelerated Technology official homepage, http://www.acceleratedtechnology.com/, 2005.
[2] FreeRTOS official homepage, http://www.freertos.org/, 2005.
[3] PicoOS official homepage, http://picoos.sourceforge.net/, 2005.
[4] Timber website at Lule ˙a University of Technology, http://www.csee.ltu.se/index.php?subject=timber,2005.
[5] TinyOS official homepage, http://www.tinyos.net/,2005.
[6] T. P. Baker. A stack-based resource allocation policyfor realtime processes. In Proceedings of the IEEEReal-Time Systems Symposium, pages 191– 200, 1990.
[7] T. P. Baker. Stack-based scheduling of real-timeprocesses. Advances in Real-Time Systems, pages64–96, 1993.
[8] A. Black, M. Carlsson, M. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems, 2002.
[9] M. Carlsson, J. Nordlander, and D. Kieburtz. The semantic layers of timber. The First Asian Symposium on Programming Languages and Systems (APLAS), Beijing, 2003. C Springer-Verlag., 2003.
[10] A. Dunkels, B. Gr¨onvall, and T. Voigt. Contiki – a lightweight and flexible operating system for tiny networked sensors. In Proceedings of the First IEEE Workshop on Embedded Networked Sensors, Tampa, Florida, USA, 2004.
[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In In ACM SIGPLAN Conference on Programming Language Design and Implementation, 2003.
[12] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler,and K. S. J. Pister. System architecture directions for networked sensors. In Architectural Support for Programming Languages and Operating Systems, pages 93–104, 2000.
[13] B. W. Kernighan and D. M. Ritchie. The C programming language. Prentice-Hall, Inc., 1978.
[14] J. Mattsson. Garbage collection with hard real-time requirements. Master's thesis, Lule ˙a University of Technology, 2004.
[15] J. Nordlander. Reactive Objects and Functional Programming. PhD thesis, Chalmers University of Technology, 1999.
[16] J. Nordlander and M. Carlsson. Reactive objects in a functional language – an escape from the evil, 1997.
[17] J. Nordlander, M. Carlsson, M. Jones, and J Jonsson. Programming with time-constrained reactions, 2005.
[18] L. Svensson, J. Eriksson, P. Lindgren, and J. Nordlander. Language-based WCET analysis of reactive programs, 2005.
[19] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized live heap bound analysis. In Proc. 4th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI), volume 2575 of Lecture Notes in Computer Science,pages 70–85. Springer-Verlag, jan 2003.

