Kelb — A Real-Time Programming Environment for the Sony AIBO

Erik Cedheim Ramzi Ferchichi Anders Jonsson Dan Lind
Henrik Nyman Olof Sivertsson Andreas Widenfalk Jöns Åkerlund
Leonid Mokrushin Paul Pettersson

Department of Information Technology, Uppsala University
P.O. Box 337, S-751 05 Uppsala, Sweden.
Email: {leonid.mokrushin,paul.pettersson}@it.uu.se

Abstract. Kelb is a new real-time programming environment developed at Uppsala University for the Sony AIBO ERS-210. It is aimed to provide efficiency by introducing a notion of light-weight tasks executing according to well-known real-time scheduling algorithms and resource protocols, while still allowing applications to be developed in a high-level abstract programming language. In this paper we give an overview of the design of Kelb and describe the status of the environment, currently including: a real-time programming language and compiler extending gcc for MIPS with support for time- and event-triggered tasks, a runtime library with support for static and dynamic preemptive scheduling algorithms (e.g. fixed priority and earliest deadline first), and a prototype connection to the Times tool allowing Kelb designs to be analysed for schedulabilty.

1 Introduction

Sony Four Legged is one of the six official leagues in the Robocup 2004 competition. Besides challenging the competing teams with problems commonly found in robotics applications such as image recognition, planing and other AI-problems, the four legged league also restricts the teams to a specific hardware, namely the Sony AIBO[1] ERS-210 or ERS-7. This means that any participating development team need to consider not only the challenging problem of developing a collaborating team of soccer playing autonomous robots, but also how to make best possible use of the hardware constraints imposed by the chosen platform.

In order to allow for teams to develop their own applications for the Sony AIBO, Sony Computer Science Laboratory provide a software development kit called OPEN-R SDK, which is based on the gcc C^{++}-compiler for MIPS. It includes several AIBO specific libraries and an abstraction of processes called OPEN-R object. The OPEN-R objects are executed by an underlying reflective object-oriented operating system, called AperiOS, in which every object encapsulates state, methods and a virtual execution processor. An object is further

[1] AIBO is an acronym for Artificial Intelligence roBOt. It also means "friend" in Japanese.

associated with a set of meta-objects (metaspace) that defines the behavior (object model) of the object.

The aim of the Kelb project is to build a real-time operating system kernel that provides developers of Sony AIBO applications with the kind of light-weight processes (or tasks) found in the literature of real-time operating systems [4]. As such, it should provide the developers with efficient and predictable services for context-switching, scheduling and semaphore blocking when accessing shared resources. However, due to the proprietary nature of AperiOS it is not possible to modify levels lower than that of OPEN-R. As a consequence the current solution executes a Kelb program as two OPEN-R objects comprising a runtime system with a scheduler, an event handler etc, and the compiled source program. A similar solution is used in the Tekkotsu framework [1].

The Kelb programming language is a real-time programming language that extends OPEN-R with a notion of tasks, and a set of language construct to describe the conditions at which tasks are being released for executions. These may be composed by boolean combination of timer events, other internal events, or external events, and a set of other C-like control statements. To support analysis, the language also includes real-time annotations, inspired by the stereotypes found in the recent UML profile Scheduling, Performance and Time [5]. Using the annotation, the programmer can specify tasks parameters such as priority, worst-case execution time, deadline, scheduling policy etc. and use them to check e.g. schedulability of an application, i.e. that all tasks are guaranteed to complete their execution before the given deadlines. The programming language of Kelb is highly influenced by the graphical input language used in the analysis tool TIMES[2, 3] developed at Uppsala University. To perform analysis of a Kelb program, the Kelb compiler derives (in addition to the binary code to be executed on AperiOS) a model of timed automata that can be further analysed in TIMES.

The rest of this paper is organised as follows: in the next section we describe the design process adopted in Kelb. Sections 3 and 4 present the Kelb input language and the runtime system respectively. We conclude the paper with some final remarks and a discussion about future work in Section 5.

2 Programming with Kelb

In this section we overview the design process and the components of a real-time application developed using Kelb language. We also describe a link to the TIMES tool, which makes it possible to perform schedulabilty analysis of Kelb applications.

2.1 Design Process

Kelb is a real-time extension of C^{++} used in OPEN-R SDK for AIBO robots. The design process of Kelb application is shown on Fig.1. In the first place, the designer writes a Kelb program. Syntactically, a Kelb program is a superset of

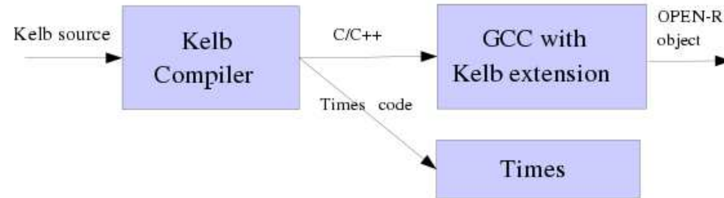

Fig. 1. Design process of a Kelb application.

OPEN-R program, extended with parameterized real-time tasks[2], task execution patterns, and a chosen scheduling policy. The program is then compiled using the Kelb compiler that produces two types of output: application binaries for AperiOS, and the abstract TIMES model of the Kelb application.

The TIMES tool can used the generated model of a Kelb program to check for schedulabilty, i.e. if in all possible runs of the application all released tasks are guaranteed to always meet their deadlines.

The Kelb compiler is a modified gcc C^{++}-compiler for MIPS processor. The compiler produces binaries for the application itself (i.e. task code and task release patterns) along with an automatically generated Kelb runtime system. The runtime system consists of a task scheduler implementing the given scheduling policy, and the module responsible for event handling, command handling and external communication.

2.2 System Components

As any OPEN-R application, Kelb applications consist of OPEN-R objects. However, programmers develop their applications on the abstraction level of tasks and task release patterns. In order to reduce inter-object communication overhead and hence increase the performance of a real-time application, every Kelb program is compiled into exactly two OPEN-R objects called MAIN and CORE.

The components of a Kelb system are illustrated in Fig.2. The MAIN object contains all the application tasks together with a generated task scheduler. The CORE object implements parts of the runtime system for communication between tasks (communication handler) and with the hardware (event handler and command handler).

3 The Kelb Language

The Kelb programming language has been designed to capture timing properties of programs, and produce predictable code for real-time applications. The common way of building real-time applications is from a number of tasks.

[2] Tasks in a Kelb program can be annotated with parameters such as priority, worst-case execution time, deadline etc. Parameters are further used for the analysis and generation of Kelb runtime system.

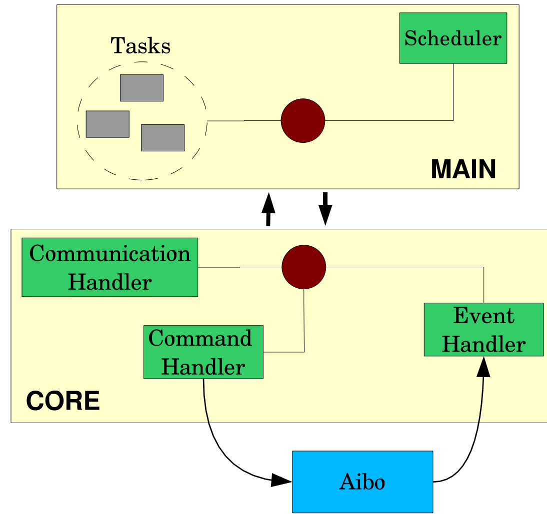

Fig. 2. The components of a Kelb application.

3.1 Real-Time Tasks

A *task* in Kelb language is written in OPEN-R code but annotated with parameters such as priority, deadline, worst-case execution time etc. Naturally, tasks take some time to complete execution depending on their complexity.

Periodic tasks are usually used when a similar computation has to be performed with some periodicity, e.g. computing an average temperature using a thermometer. *Event-triggered tasks* are used when a computation has to be a reaction upon some event, e.g. parsing a data packet received from a network. In Kelb language all tasks are event-triggered since periodic tasks are merely tasks triggered by timer events.

At the arrival of a given event(s), a task is *released* for execution and placed in the ready queue of a scheduler. The scheduler decides on the order in which tasks should be executed based on some pre-defined scheduling policy, resource availability, and possibly other constraints. If the higher priority task is released, the currently executed task may be *preempted* meaning that its execution is postponed and the processor is allocated for execution of a higher priority task.

3.2 Task Release Patterns

As mentioned, Kelb tasks are reactions on events, but in order to make the language more expressive, an additional structural level called *release patterns* (or just patterns) is provided to describe how tasks are released. A pattern is a rule specifying the conditions and the order in which tasks should be released upon arrival of certain events. Patterns can be simple (e.g. periodic execution

```
task moveLeftForeLeg {
    properties:
        wcet = 2;        // worst-case execution time = 2 seconds
        deadline = 3;    // relative deadline = 3 seconds
        priority = 10;
     }
    body:
    void moveLeftForeLeg(int angle) {
        commandHandler->setJointValue(PRIM_FORE_LEFT_J2, angle, angle);
        commandHandler->sendRegion();
    }
};
```

Table 1. A Kelb task moving leg limb to a given position.

of a task with a given period), or rather involved with conditions and branching of control flow. Events are generated by the event handler (see Section 4.2) and captured in patterns using triggers. A *trigger* is a boolean expression over sensor and timer values evaluated by the event handler. A pattern can capture an event by registering a corresponding trigger in the event handler and using non-busy wait statement. Triggers can also be combined in *multi-triggers* using boolean operators.

In each Kelb-program there is a main pattern that spawns other patterns. Besides standard flow control and integer arithmetic, patterns can release/terminate tasks, start/stop patterns, and wait for events.

3.3 Shared Resources

Tasks in Kelb may share resources using the locking mechanism implemented in the language. If a task has to access a resource exclusively, it should acquire the lock on a corresponding mutex. When the resource is no longer needed, the task unlocks a mutex allowing other tasks to use it.

In order to prevent deadlocks caused by the higher priority tasks accessing a shared resource locked by the lower priority tasks, Kelb scheduler uses a priority inheritance protocols. In case of fixed priority scheduling, the access to mutexes is scheduled with immediate ceiling priority protocol (ICPP) [6].

4 The Runtime System

The runtime system of a Kelb application is generated automatically when a Kelb program is compiled to binary code. The components of the runtime system are located in the OPEN-R objects MAIN and CORE, as illustrated in Fig. 2. The runtime system consists of the following components:

- Scheduler - keeps track of the tasks released for execution, and at any point in time selects the task to be executed based on a given scheduling policy

```
pattern main {
    int i = 0;
    while ( i < 10 ) {
        release(moveLeftForeLeg(i*60));  // release the task
        wait(timer(5));                  // wait for 5 seconds
        i++;
    }
}
```

Table 2. A Kelb pattern releasing ten instances of task `moveForeLeg`.

and availability of resources. This component is located in the MAIN object along with all the tasks declared in the application.

- Event Handler - monitors the state of the hardware, and checks the enabling condition for every trigger registered by task release patterns. This component and is a part of the CORE object as well as the following two.
- Command Handler - passes commands generated from the tasks to the physical hardware actuators (motors, lights etc.).
- Communication Handler - passes messages between tasks possibly assigned to different AIBOs.

4.1 The Scheduler

The scheduler like the Kelb language has a general notion of a task more like threads than processes to increase efficiency. A task can be in any of the following states:

- Released - ready for execution
- Blocked - waiting for resource
- Terminated - not running

All tasks are registered in the system when the system starts but are not released until triggered by a release pattern and then executed when the scheduler chooses to. The scheduler keeps track of the tasks in the system and schedules the released tasks according to the chosen scheduling policy. Our system provides three different scheduling policies:

- Fixed-Priority - the released task with the highest priority is executed.

[2] A process is an executing program with its own address space and cannot access the address space of other processes. A thread is a piece of code executing in the address space of a process. We can have several threads inside one process and they will then share the same address space, i.e. among other things the code and data sections. By using threads instead of processes it becomes computationally cheaper to switch between different pieces of executing code and to communicate between them (using shared memory).

– Earliest Deadline First - the released task with the earliest absolute deadline runs first.
– Round-Robin - in case of preemptive tasks each task is given an equal amount of the runtime in a round-robin fashion. In case of non-preemptive scheduling, the are executed to termination in round-robin fashion.

The scheduler also provides support for terminating tasks, and blocking and unblocking tasks upon some event registered in a trigger.

A main function of the scheduler is to perform the task switches, i.e. to change the executing task and to store and load the context of preempted and (re-)started task respectively. In our system hardware interrupts can not be used as it is built on top of AperiOS. Preemption is therefore emulated by inserting *checkpoints* in the code, where a task can be interrupted. We call such interrupts *virtual* to distinguish them from the common notion of a hardware interrupt.

The virtual interrupts are automatically inserted when using our modified version of gcc for the OPEN-R platform. The check-point insertion is described in more detail below.

A virtual interrupt is implemented as a macro. The macro checks if the scheduler has set a flag to indicate that a task switching should take place. If this is the case, the macro calls a function that performs the task switching by saving the context of the currently executing task and loading the context of the next task the scheduler has chosen to run. When this function returns it will return to the next tasks code as the runtime system changes the return address as part of the task switching process.

When a context switch occurs, all non-temporary registers are saved before execution is changed to another task. The temporary registers need not be saved as they are automatically saved when the function that does the context switch is called.

4.2 Event Handler

The event handler is the part of the system that receives input from the robot and notifies the running system. The event handler checks the conditions of any trigger are met. The triggers are stored and registered in the event handler by the individual tasks. In the runtime system a trigger consists of two parts: a *checker* that is invoked on a condition, and a *handler* that is executed afterwards. Triggers can be of three kinds:

– Event trigger - returns a signal when something changes in the robot as specified by the implementor.
– Time trigger - returns a signal at a specified time interval. These signals can be either periodic or one-shot, meaning that it only occurs once.
– Image trigger - returns a signal every time new image data is receive.

Checking for individual conditions is too limited for most applications. For this reason a method of combining checks for several conditions at once have been supplied through the use of *multi-triggers*.

A multi-trigger is simply a binary tree consisting of either a unary (ordinary) trigger or two multi-triggers composed by a boolean operator. For example, two triggers T1 and T2 can be composed with the AND condition, in case both T1 and T2 have to be activated before the trigger sends a signal.

4.3 Communication Handler

The communication handler provides support for message passing between tasks using mailboxes. Message passing is implemented using shared memory to improve performance.

Two sorts of communication is implemented in the Kelb runtime system, synchronous and asynchronous communication. When using synchronous communication a task is blocked when sending to a full mailbox or receiving from an empty mailbox. The task is later unblocked when the communication request can be handled. Timeouts for synchronous communication can also be specified.

Asynchronous communication, on the other hand, is non-blocking. If a task fails to send/receive the called function immediately returns with a return value indicating this.

A message is sent by defining its content and structure, and allocating a buffer for it, and finally passing the message to the communication handler. It is then received by a task defined to wait for the message that properly handles it.

4.4 Checkpoint Insertion

The virtual interrupts used by the core object takes the form of a checkpoint, which investigates whether a task should be preempted or not, and the scheduler writing into shared memory what task should run (as described above).

While it is possible for a programmer to insert the checkpoints manually, this is unsafe as it allows malicious code containing e.g. non-terminating loops. To automatically insert checkpoint we have extended gcc with a compile-time functionality inserting the checkpoints. It inserts checkpoints according to the following three rules:

- An inner loop has been reached. The rationale for this is that loops can be running for a long time, and it must be ensured that they do not hamper task switches.
- A call statement is reached.
- A goto statement is reached.

Checkpoint insertion is not done before statements or leaf functions as these rarely take much time.

5 Conclusion and Future Work

In this paper, we have described the development of Kelb — a programming environments for the Sony AIBO ERS-210. It was developed as a student project

during the fall of 2003 based on experiences made during similar projects in the fall of 2002 and the spring of 2003. The current version of Kelb features a real-time language, a compiler integrated with gcc, and a runtime library executing tasks according to static or dynamic scheduling policies. A prototype connection to the analysis tool Times is also available.

Future work includes performing a proper evaluation of Kelb. So far we have evaluated its performance only on smaller examples. The results are encouraging but it remains to see how the Kelb runtime system will scale up to larger systems containing many tasks. In particular, it remains to evaluate how appropriate Kelb is for robot soccer. We hope to have a group of students developing a soccer team using Kelb in the fall of 2004. We also plan to further develop the Kelb programming language as such, and the connection to the Times tool.

A particular obstacle during the development was the restriction of not being able to modify AperiOS lower than the level of OPEN-R. Lower level modifications would make it possible to improve Kelb in several ways, including more efficient task switching, true preemptive scheduling, better predictability etc. On the other hand, that would dramatically change the conditions of the Sony four legged league of Robocup.

References

1. *Tekkotsu Source Documentation*. Available at the web site `www-2.cs.cmu.edu/-~tekkotsu/`.
2. Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: A tool for modelling and implementation of embedded systems. In J.-P. Katoen and P. Stevens, editors, *Proc. of the 8*th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2280 in Lecture Notes in Computer Science, pages 460–464. Springer–Verlag, 2002.
3. Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *Proc. of 1*st *International Workshop on Formal Modeling and Analysis of Timed Systems*, Lecture Notes in Computer Science. Springer–Verlag, 2003.
4. Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable scheduling algorithms and applications*. Kluwer Academic Publishers, 1997.
5. Object Management Group Inc. *UML Profile for Schedulability, Performance, and Time Specification*, 2002.
6. L. Sha, R. Rajkumar, and J. Lehozky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Computers*, 39(9):1175–1185, 1990.