BOSS/Ada: An Open Source Ada 95 Safety Kit

A Dependable open source embedded operating system for GNAT

Volkert Barr
ISTI/ FG Softwaretechnik, TU−Berlin
barr@cs.tu−berlin.de

Dr. Sergio Montenegro
Fraunhofer Gesellschaft − FIRST
sergio.montenegro@first.fhg.de

Abstract: *Ada has been successfully used in many dependable real−time applications, which have undoubtedly benefit of its major strengths: The well defined language semantics [1], the strong type checking, structuring mechanisms like packages and not to forget the Ada Semantic Interface Standard [2] supporting the development of code analysis, verification and testing tools. But an Ada system can not be more safe than the underlying operating and run time system. An dependable and certified operating system involves very high license costs and you will never see the sources. The operating system remains a black box and you become depend of the OS provider. We aim to change this by certifying our OS and Ada/GNAT interface and give it as open source public domain.*

Because of its concurrent nature, many safety critical applications increasingly using multithreading, which have a strong impact on the certification process and the resulting total development costs. This is especially true for Ada applications due to its powerful and therefore complex tasking semantics. Using the full range of Ada tasking power the complete Ada run time system has also be a subject of the certification process. This leaded to a definition of an Ada tasking subset called Ravenscar Profile, with a commercial implementations used in different avionics systems. The major goals of such a profile is to allow a runtime efficient and deterministic implementation of an Ada Runtime System with a simple internal organization and low memory usage. BOSS will support this profile for the open−source GNU Ada Translator (GNAT). Its current characteristics are: preemptive, prioritized, real time multithreading, OO−framework structure, C++ and Ada interface (in work). Thread switch time 10 microseconds on a PPC at 48 Mhz and under 1 microseconds on Pentium 500 Mhz. The time resolution is configurable 1 ms or 1 microsecond with at least 500K years linear time without overflow. The reaction time to interrupts is less than 3 microseconds PPC at 48 MHz

Due to the fact, that complexity is the first foe of safety, BOSS is intended to be as simple as possible, so it is easier to understand, to review, to use, to certificate, to port to other platforms, etc. Some parts of BOSS are being verified mathematically and formally using model checkers and theorem provers. BOSS is based on very few and simple basic functions, which can be proved very faithfully, and these few functions are used for almost every operation of the kernel. Furthermore BOSS is open−source, so that everybody can look at it and find possible errors. We are currently developing two types of Ada interfaces to BOSS GNAT:

1. An interface for a No_Ada_Tasking environment using a general Ada binding to the BOSS primitives. This allows to write multithreading applications in Ada, without the Ada languages tasking facilities.

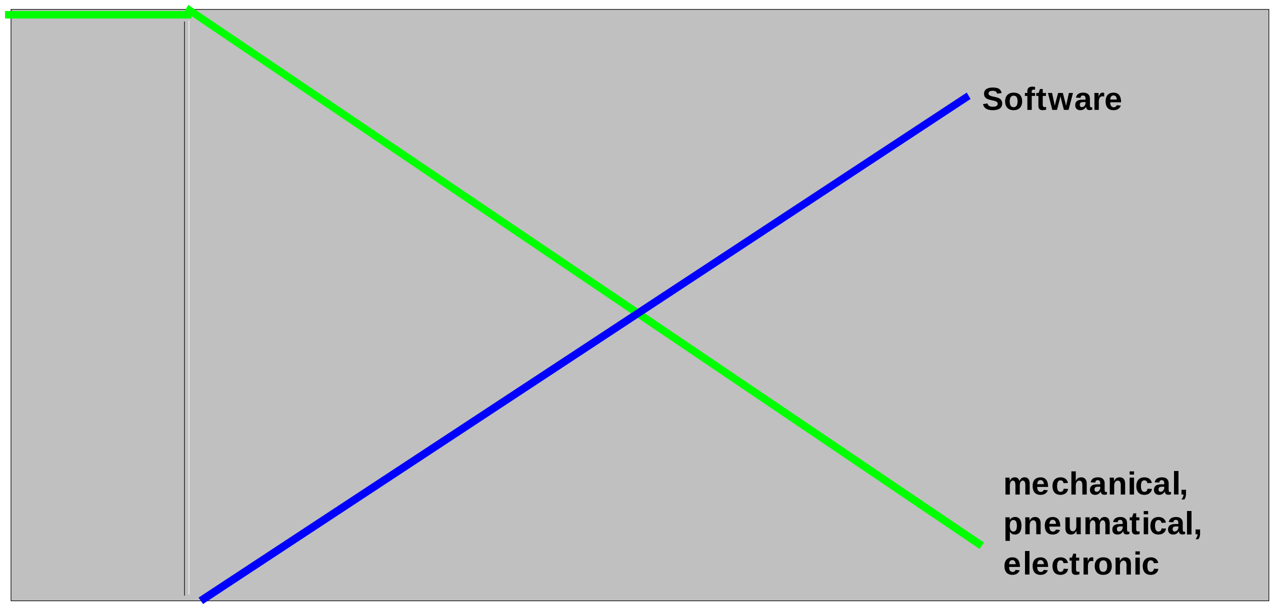
2. An Ada tasking subset using the general BOSS binding as a core implementation of an

tasking subset comparable to the Ravenscar Profile. The core implementation is integrated into to GNU Ada Runtime Architecture.

We believe that the BOSS operating system together with Ada 95 can play an important role in future safety critical applications using open−source technologies.

Why open Source

You may think you can safe money by moving mechanical functionality to software, BUT you may save 2 Euros in mechanical parts and pay 3 Dollars for operating system licensees, like in picture 1; Or you have to reinvent the operating system; this this done by many companies once and again.



Picture 1: Cost Distribution and losses

We are working on open source software packages, including operating system, an Ada environment and validations/verification tools. This will allow many companies and institutes to create reliable systems in a short time and lower costs, without having to reinvent the operating system for each project. The operating system BOSS will be certified by the aeronautic council and made open source and public domain. This was done never before! And this could safe the EU billions (and you thousands) of Euros of OS−licenses every year.

BOSS: a dependable open source embedded operating system

BOSS targets a principle which the world forgot a long time ago: *Simplicity.*

BOSS was designed to be a dependable real time embedded operating system which can be easily certified by different safety organizations.
Due to the fact, that complexity is the first foe of safety, BOSS is intended to be as simple as possible, so it is easier to understand, to review, to use etc. The whole kernel can be printed in a few pages. Some parts of BOSS are being verified mathematically and formally using model checker and theorem provers. With the current state of the art on formal verification, complex systems cannot be verified formally, but BOSS can be. BOSS

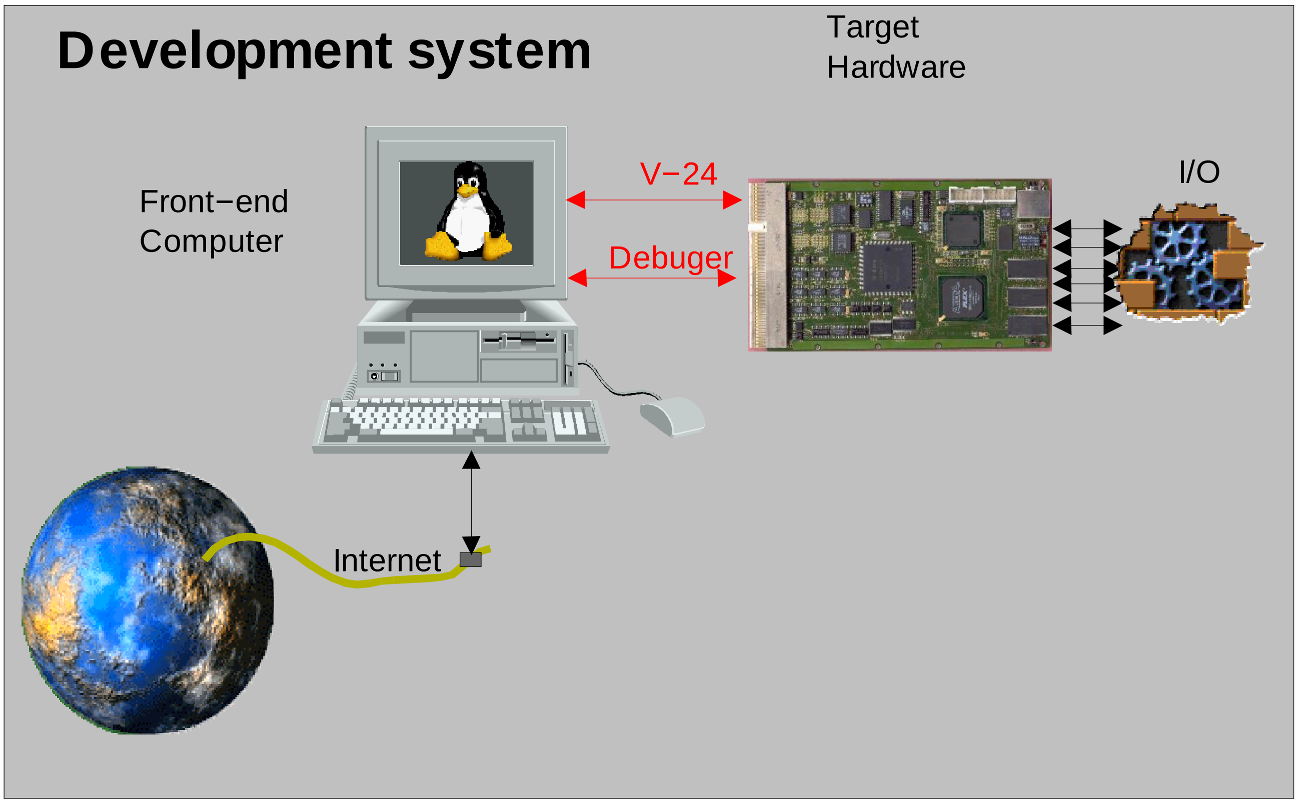
is based on very few and simple basic functions, which can be proved very faithfully, and these functions are used for almost every operation of the kernel. Furthermore BOSS is open−source, so that everybody can look at it and find possible errors.

Currently there are 3 implementations of BOSS on different platforms: powerPC, x86, and an on−top−of−LINUX implementation. Applications written on BOSS can run without changes on any of these platforms. The on−top−of−LINUX implementation helps the developer to work locally on his workstation without having to use the target system. To move to the target he has only to recompile the code. The behaviour will be the same except for timing requirements and time resolution which on LINUX cannot be as exact as in the target systems.

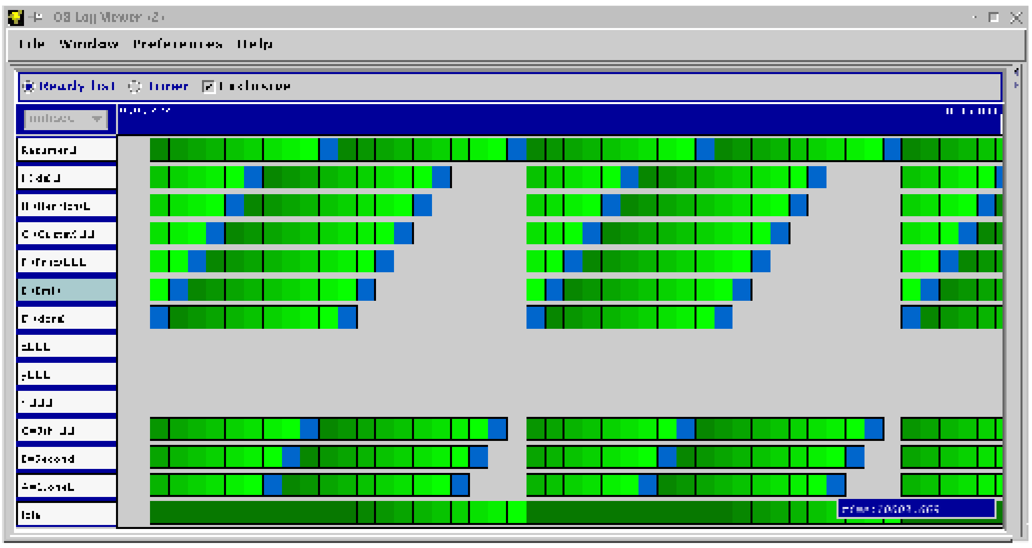
BOSS is implemented in C++, it provides an C++ interface and an Ada interface is being developed.

Development Environment

The application development and execution of a simulated system can be done on a LINUX workstation. For the execution on the target system we provide a debugger interface and serial connections to load and debug your system. Using LINUX as front−end to the target system the whole work can be done remotely using internet. It is possible to capture log files of the activities in BOSS in the target system and there are tools to visualise the internal timings of your system.



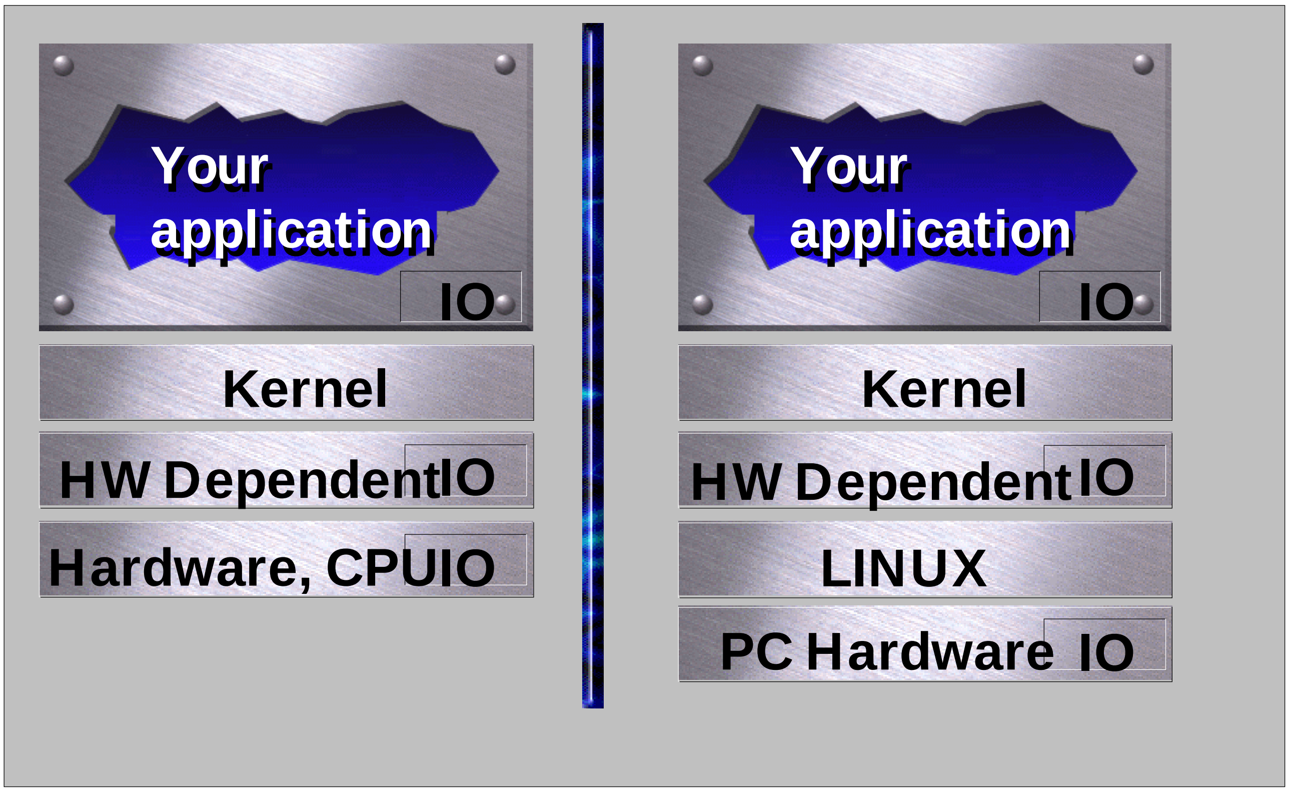
Picture 2: Development System



Picture 3: example of Task Visualization

Macro structure

BOSS is structured in layers from hardware up to the final application, each providing an virtual view from the lower layer. This virtual view is always the same even if the more lower layers are substituted. The bottom layer is the target hardware CPU, IO devices and other Hardware units or a LINUX−platform simulation of the devices. This layer is for the kernel and applications (except speed and timing granularity) transparent and can be changed very easily.



Picture 4: Operating System Layers

The second layer is the (only) hardware dependent layer. This layer implements the functionality which is different on each platform, e.g. CPU−register load/store, low level Hardware−Drivers and basic interrupt management. To move from one platform to another one only this layer has to be rewritten. A small problem is the IO. The basic IO handling is implemented in the second layer, but there are so many different devices with different and sometimes very complex protocols. Such complex protocols should be implemented in the application layer. An abstraction which would cover all possible devices would bring a very high complexity and lower flexibility. BOSS provides only a basic support for IO handling, like event propagation.

The third layer is the kernel which implements the interface for the applications. It manages threads and resources; it will be explained later.

The fourth layer is the biggest and most complex, it implements your applications which can do anything you need!

BOSS Kernel

The Kernel is so simple that it can be totally explained in one page:

The basic class is Thread: it produces executable objects with context, stack and own data. They can run, be suspended, reactivated and react to time, internal and external events.

The basic operations are lists−management. All resources are managed (sorted) in chained lists, therefore there is no limit to their length. No element can be in two lists at the same time. Before inserting an element in any list, it will be removed from any other list. If it is required to have an element in several lists it is possible to use reference entries.

There are two basic operations on any list:

insert: inserts an entry in the proper place according to a sort field. The sort field will be compared and the new entry will be inserted before the first with higher value. e.g. if other entries with the same value are found (e.g. same priority) the new one will be added after these (first come first serve on same priorities).

remove: removes an entry from a list (if already there). Most of the times the remove operation is applied to the first of the list.

The kernel lists are:

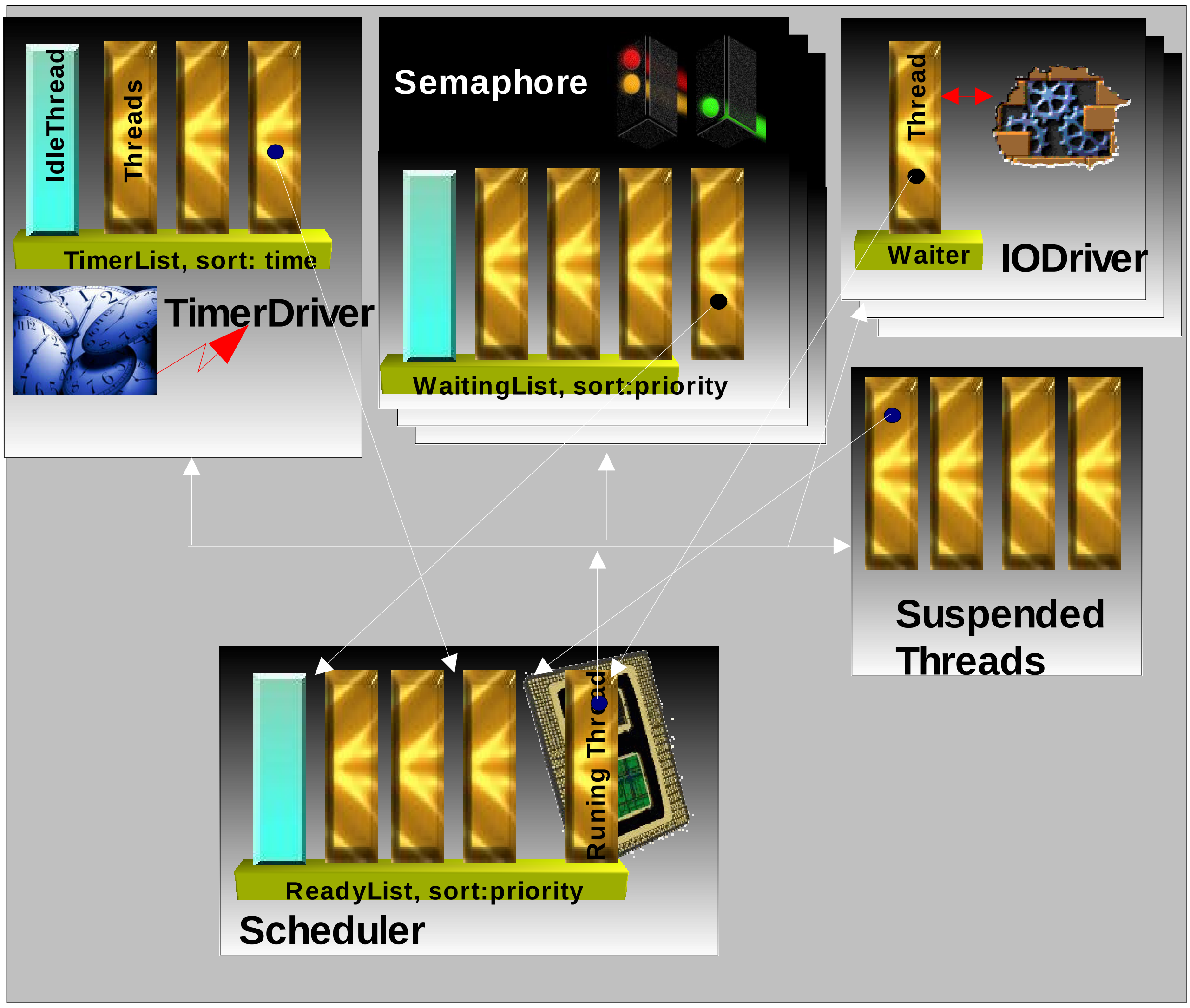
Ready list: List of threads which are ready to use CPU time, sorted (inverted) by priorities. The first in the list has currently the CPU −− is running now. All other are waiting. The priority is inverted internally so that the higher priority is first and 0 at the end of the list. If the list is empty the Idle thread gets the control, which consumes all unused CPU time.

Timer list: List of threads which are waiting for a time point (time event), sorted by time. The next thread to be awaked is at the front of the list.

Semaphores: List of threads sorted by priority. Semaphores are used to implement monitors: to protect exclusive sections and to implement synchronisation. All threads which are waiting to enter an exclusive section are managed in a semaphore list. The thread with the highest priority will get the resources first.

Messages and communication lists: Messages from one thread to other are written in a list sorted by priorities. Usually all messages have the same priority, so that the list works like a fifo. A thread attempting to read from an empty message box will be suspended.

User lists: Applications can use the list−classes to build any kind of list for their own uses.



Picture 5: Most important Lists of Threads

Use of lists

All threads operations are implemented just by inserting and removing threads from lists:
start: creates the thread context and inserts the new thread into the ready list.
suspend: removes the thread from the ready list
resume: inserts the thread into the ready list
wakeAt: removes the thread from the ready list and inserts it into the timer list

The scheduler manages the ready list and activates the first thread found in the list. But a small help is required from the hardware dependent layer:
transfer(): to save the context of the running thread and load the context of other thread to give it the CPU control. The scheduler is pre−emptive. At any time if a thread (B) with higher priority than the running thread (A) is resumed, the CPU will be taken from thread A and passed to thread B (Transfer A −> B).

The time manager manages the hardware time gauge or interrupts and resumes threads which are waiting for time points.
IO Drivers do the same, but with IO events/interrupts.

Some Extra Support Classes

Following classes are not needed for the basic thread management, but they include functions which are required very often, therefore they are supplied here.

For communication among threads there are two possibilities: 1. Synchronously: the receiver thread will be suspended until the data arrive. For this kind of communication fifos (message lists) and signal−boxes are used.
2. Asynchronous communication: if no message is ready the receiver is notified and can do something else or the receiver will get the last written data. Most of the cases a thread needs only to get the newest status information of other threads and not the whole history. In this cases you can use an asynchronous communication buffer. All messages get a time stamp to know the age of the received message.

Some other useful classes:
pool management
flash−memory & rom management
logwriter (for timing visualisation on the host computer)
nameserver (to find objects using names)
Timecontrol (to create loops in time)

All this classes are thread safe. If you need more than that, you can use the gnu template library, but watch out, they are not thread safe.

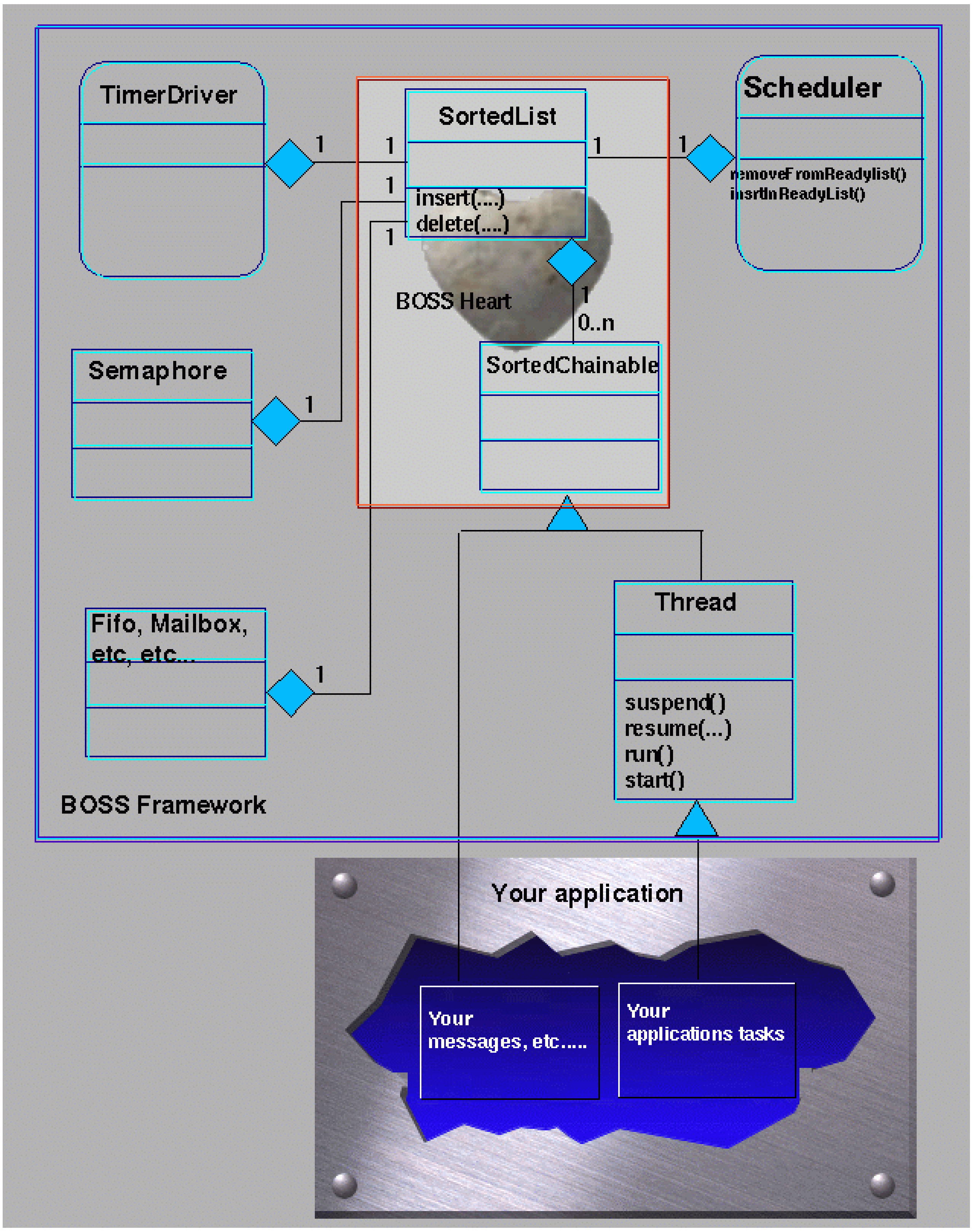
Building your application

BOSS is build as a frame work which you can specialise to fulfil your requirements. The framework technology is a further step following the object oriented technology where the functionality is provided by OO methods in classes, the user can employ each class as it is, or he can tailor (adapt) the class to his own needs, by means of inheritance and methods/operators overloading.

The framework technology offers complete adaptable structures of classes. A framework is composed of several classes in a structure with different relationships: inheritance, references and contention. The whole structure has a specific functionality. The user can adapt its functionality to his needs as follows: Some classes in the structure provide the adaptation interface for the user. Other classes offer a function interface or support the whole framework function. To adapt the functionality of the framework to his need, the user writes new classes, which inherit from the adaptation interface classes (subclasses). The adaptation interface methods should be overloaded with the user methods and functionality in order to integrate the user functionality into the framework. The new (user) subclasses are integrated automatically (by inheritance) into the structure. The framework functionality is thus extended by the desired functionality and adapted to the user’s needs.

The most important adaptation interface class is the "Thread" class in BOSS. With this class the user can implement his tasks with own context and stack. The user should write his desired functionality in a subclass and overload the method "run()" which else does nothing per default. After the thread is started it will be written in the ready list and will run as soon as possible, according to its priority and other ready threads (its position in the list).

The TimerDriver should not be extended, it can be used to resume (reactivate) threads at a determined time point. The EventManager and IODrivers are like the TimeDriver, they resume threads after the expected event arrived, e.g. software events, timer events, external events like interrupts etc.



Picture 6: BOSS Framework

Example of use

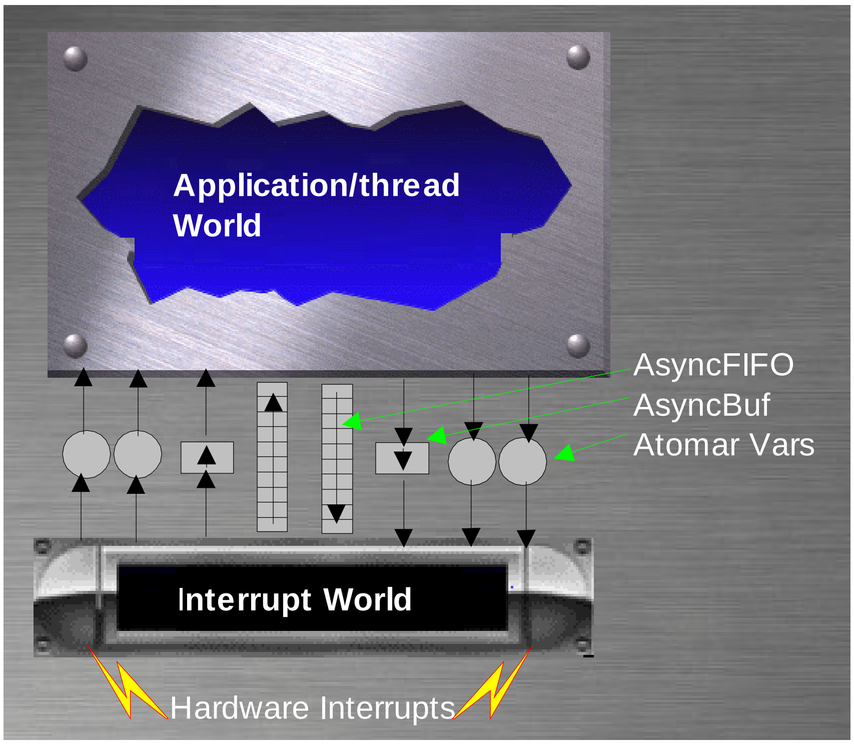
```
external Thread xx;
class TestThread: public Thread {     // Thread produces an active object
        void run () {
                while(1) {
                        {.... do something }
                        yield();                // other thread, same priority will get the CPU
                        {.... do something }
                        suspend();              // not run anymore until someone resumes me
                        {.... do something }
                        suspendFor(1000);       // TimerDriver will resume me in 1000 ms.
                        resume(xx);             // I resume the Thread xx, which I know.
                }
        }
};
/** Another example: **/

class OtherTestThread : public Thread {
    void run () {
        Semaphore monitor;
        TimeControl timeControl;   //To implement time loops
        timeControl.startAt(5000); // Time point for the first time
        timeControl.every(100);    // Cyclus time
        while(1) {
            timeControl.wait();    // wait according to start and cyclus
            {.... do something }
            monitor.enter();       // protected area,
            {.... do something }
            monitor.leave();
        }
    }
};
/** Create 6 threads or applications ***/

TestThread       a, b, xx;
OtherTestThread  x, y, z;
```

Interrupt Structure

The threads can be synchronized among them by using semaphores, suspends, resumes and THREAD_ATOMAR constructs. But Hardware interrupts can occur at any time. It is possible to lock/disable interrupts, but this should be avoid. Normally the interrupts are allowed at any time to ensure the fastest interrupt reaction.

To avoid problems and data inconstancies when interrupts occurs, the system is divides in two independent work areas (worlds): The threads world, where the applications reside and the interrupt area, which is activated asynchronously from the threads and applications, by hardware events, like timers, i/o devices, alarms etc.

Communication between this two worlds is performed by using extra asynchronous communications buffers and fifos. This elements has two sides: one to write data and one to read data. This elements are implemented in such a manner, that both sides can be used asynchronously from each other. No explicit lock or synchronization is required. To pass complex data from one world to the other such elements are used. They are extra design to couple the interrupt world and the threads world, but they can be used among threads too. Simple data which can be written/read atomic (like an integer or a character), can be used in a manner that one side only writes and the other only reads each of this variable. This is a safe asynchronous communication too.



Picture 7: Interrupt − Thread Worlds

The Thread world will never call or interrupt the interrupt world, which has the highest priority in the system. The thread world can be interrupted at any time by interrupts and after the interrupt execution the interrupt server may "return" (jump) to the dispatcher instead than to the interrupted place if a redispatch is required. The interrupted address will be saved as part of the context of the interrupted thread. The implementation of this jump from the interrupt world to the dispatcher is a critical operation and is hardware dependent...

The Ada Environment

Why Ada?

As mentioned above, Ada offers a reasonable set of tools to write safer software including

- the strong typing capability and a rich set of arithmetic types,
- access types with no pointer arithmetic (no footshooting with uncontrolled memory access)
- defining external interfaces (formal definitions/representation clauses for memory/hardware access) and a standardized way to interface languages like C and FORTRAN,
- no need of object−oriented programming: In the safety community is widely accepted that the timing and resource usage of object−oriented programs are less deterministic and hard to analyse. Due to Ada's orthogonal language structure it is easy to restrict or even abandon object−oriented concepts like inheritance and polymorphism (in Ada terms run−time dispatching),
- Ada offers a highly useable set of decomposition mechanisms using generics, packages, public and private child packages,
- Last not least, the Ada Semantical Interface Standard [2] makes it possible to

implement code analysis tools to provide language subset checkers and annotation systems.

For safety critical software Ada is and should be the language of choice. For this reason, we decided to develop an Ada Binding to BOSS.

The Ada BOSS Binding

BOSS was designed with object−oriented decomposition in mind, but makes very few usage of complicated C++ features (e.g. for method redefinition and dynamic dispatching) to make it understandable and analysable. For this reason, the mapping to Ada was straight forward. Mainly all important OS Kernel abstractions were lifted to the Ada environment using the C/C++ interfaces facilities of the GNAT Compiler 3.13p. Our GNAT Compiler is restricted not to use Ada Tasking (No_Run_Time).

BOSS package structure

All tagged types directly map onto the corresponding C++ classes. Some BOSS classes are completely hidden to Ada programs. Our present package structure consists of the following ones:

- **Boss_Kernel**:Consists of a event_loop method to start the event processing (system start),
- **Boss_Kernel.Base_Type**: Declares a basic set of data types for time, thread priorities.
- **Boss_Kernel.Event_Type**: Declares an event type used for event trigger identification,
- **Boss_Kernel.Thread_Type**: Declares a tagged type thread with primitive operations for priority setting and reading, thread suspension and resuming. An additional controlled thread type can be used for controlled initialization. At package level we offer typical operations to control relative and absolute delays and thread yielding.
- **Boss_Kernel.Time_Control_Type**: The declared Time_Control type can be used to synchronize threads with a time raster for starting a thread *in*, *at* and *every* point of time,
- **Boss_Kernel.Time_Manager_Type**: The time manager type administrates the timerlist of threads waiting for a timepoint. The package also offers features to access a system clock, different converters between timer and clock schemes.

System Initialization/Restrictions

The following initialization scheme is used for system start up: All threads must be declared only at library level and threads can be initialized while package elaboration (controlled type) or explicitly (elaborate package body or method calling from the main unit). All threads staying forever and do not terminate. Dynamic allocation of threads is forbidden. All these restrictions are similar to the ones defined for the Ravanscar profile [3]. After elaborating the packages, from the main unit an event handler routine must be called to start systems work.

Examples of use

Let us now see some small examples of typical real−time abstractions and how to map them to BOSS features. An application specific thread can simply defined by extending the base type **Boss_Kernel.Threads_Types.Thread** and overriding its primitive operation Run():

```
type Application_Task is new
  Boss_Kernel.Thread_Types.Thread with ... end;

procedure Run
  (Self: in out Application_Task); −− redefinition of run
```

- A periodic task with absolute delay (Ada style)

```
procedure Run
  (Self : in out Application_Task)
is
  Period     : Time := Milliseconds(50);
  Next_Start : Time;
begin
   Next_Start := Get_Time;
   loop
      Next_Start := Next_Start + Period;
      −− do something
      Suspend_Until(Next_Start); −−delay until Next_Start
   end loop;
end;
```

- A periodic task using a TimeControl and a Semaphore−Object.

```
procedure Run
  (Self : in out Application_Task)
is
  TC  : TimeControl;
  Sem : Semaphore;
begin
  −−                 offset       period
  Start_At_Every(TC, Seconds(60), Milliseconds(500));
  loop
     Wait(TC);
     Enter(Sem);
     −− do something
     Leave(Sem);
  end loop;
end;
```

- A sporadic task waiting for an event of a IODriver−Object.

```
procedure Run
  (Self : in out Application_Task)
is
  IO : IODriver;
begin
  Open(IO, „GPS“);

  loop
     Read(IO, Characters);
```

```
          -- do something

        end loop;
      end;
```

Interfacing C++ from Ada

Currently there exist no standardized way to map C++ classes from the Ada world. Therefore we need to experiment to understand how the mapping of the GNAT Compiler works [4]. GNAT offers some pragmas to interface C++ at class level (pragma CPP_Class, CPP_Constructor, CPP_Vtable, Import, ...). The name mangling problem was solved by hand.

We had some minor problems to overcome:

Run−Time Dispatching calls trigged from C++ does not work. This is comprehensible, because the C++ dispatching table does not know of any redefined operation on the Ada side. To overcome this problem, we decided to export a base operation of an Ada tagged type to the C++ world and do all the dispatching on the Ada side.

Forward declaration does not work: Some BOSS C++ classes uses attributes in a forward declared manner to model references to objects of its own type. The mapping of such a structure to Ada was refused by the GNAT Compiler (looks like a compiler bug).

Keeping C++ class layout: The need of keeping the C++ class storage layout of all non static class attributes, leads to tagged types where all − the public and the private ones − class attributes were accessible by the type clients. Because of the different visibility schemes of C++ and Ada (Ada at package level, C++ at class level) this is understandable, but not very nice. Hiding those attributes works with a further indirection. We leave this open.

Status and Future Work

Currently our Ada programs are running on the BOSS Simulation Environment (see above) which is hosted on a Linux/x86 workstation. We now have to finish our GNAT crosscompiler for PowerPC to compile our programs for a fault tolerant multi board system. This board is succesfully used in the satellite project BIRD [5]. When succeeding the next step is to understand and adapt the GNAT Runtime Architecture (GNARL/GNULL) to integrate the current BOSS interface to fulfill the semantics of the Ravenscar profile.

The portability of both the BOSS kernel and the GNAT compiler leads to an interesting open source development platform for safety related applications. With some additional tool support, we think it could be a real "Open Source Ada Safety Kit". What coming into mind is the application of some formal techniques for example model checking behavior abstractions of the Ravenscar tasking model (see [6, 7]) and the simulation of real−time and scheduling aspects (Ada Software Simulation considering scheduling, execution times and task attributes) [8, 9] together with continuos components descriptions modeling physical processes [10] controlled by the embedded system under development. Such a simulation system is currently developed in his ph.d. work by one of the authors.

Literature

[1] Taft, S.T. and Duff, R.A., eds (1995). *Ada 95 Reference Manual.* LNCS 1246, Springer Verlag.

[2] International Standards Organization. *ISO/IEC 15291 Ada Semantic Interface Specification (ASIS)*, 1999.

[3] Alan Burns and Brian Dobbing. *The Ravenscar Profile for high integrity real−time programms.* Proceedings of ACM SigAda Annual Conference, ACM Press.

[4] Ada Core Technologies Inc., *GNAT Reference Manual*, Version 3.13p, http://www.gnat.com.

[5] Sergio Montenegro, Wolfgang Bärwald. *BIRD− Spacecraft Bus Controller.* Small satellites Workshop 2001, IAA 2001.

[6] R. Alur and D.L. Dill. *A theory of timed automata.* Theoretical Computer Science, Volume 126, 1994.

[7] Alan Burns, Andy Wellings. *How to Verify Concurrent Ada Programs*. *The Application of Model Checking*, Ada Letters, June 1999.

[8] Sven Lutz, Volkert Barr. *Durchgängige Modellierung objektorientierter Echtzeitsysteme.* in Ada und Software−Qualität, Ada Deutschland Tagung 2001, Shaker Verlag, 2001.

[9] Frank Oppenheimer and Guido Schumacher. *OOCOSIM − objektorientierte Spezifikation und Simulation eingebetteter Realzeitsysteme*, Workshop Objektorientierung und sichere Software mit Ada, Institut für Angewandte Informatik, FZI Karlsruhe, 1999.

[10] T. Ernst, C. Klein−Robbenhaar, A. Nordwig, T. Schrag. *Modellierung und Simulation hybrider Systeme mit Smile.* Informatik Forschung und Entwicklung 15 (2000), Springer−Verlag, Berlin, Heidelberg, 2000.