2010:043 CIV

MASTER'S THESIS

Modeling and Simulation of Embedded Real-Time Software

Andrey Kruglyak

Luleå University of Technology

MSc Programmes in Engineering
Computer Science and Engineering
Department of Computer Science and Electrical Engineering
Division of EISLAB

2010:043 CIV - ISSN: 1402-1617 - ISRN: LTU-EX--10/043--SE

Abstract

Model-based design has been successfully used for development of general-purpose software systems and it is known to reduce the number of errors, shorten development cycles and thereby decrease overall development costs. As the complexity of embedded software systems has been steadily increasing, attempts have been made to introduce model-based design into the realm of embedded system development. However, this process has been slow and ridden with difficulties.

It can be argued that the factors adversely affecting the method's applicability include the absence of models capable of capturing functionality of both software and hardware, as well as the absence of adequate modeling of timing behavior. There is thus a clear need for a new modeling approach that would address these shortcomings.

One solution is offered by the modeling and programming language Timber. This reactive, highly concurrent, object-oriented language is primarily targeting development of real-time embedded systems. It can be used to specify and model functional and timing behavior of a system, including both its software and hardware parts. Timing requirements can be incorporated in a Timber model in the form of timing constraints on reactions at both object and system levels.

The present work investigates modeling of embedded real-time software using executable Timber models, and suggests a method for simulation of Timber models in Simulink, a widely used tool for modeling and simulation of multi-domain dynamic systems. Applicability of the proposed approach is demonstrated in a case study of an anti-lock braking system controller.

The results presented in this thesis open for simulation of Timber models of embedded software together with a wide range of simulators of discrete and continuous systems that can be co-simulated with a Simulink model, and even together with real hardware (so-called hardware-in-the-loop simulation). In the proposed simulation setting, the specification of timing behavior of a software system as expressed in its Timber model can be observed, allowing to adjust the specification to achieve intended system behavior. This introduction of verifiable timing properties into simulation, together with the true identity between the code used for simulation and the code used for realization (both can be derived from the same executable Timber model) should greatly simplify the design process and make the resulting system more reliable.

Preface

The results presented in this thesis were obtained in September 2006 - May 2007, during my time as research trainee at Luleå university of technology. They were presented in May 2007, but for a variety of reasons the publication has been delayed until 2010. As my views on many issues have changed since the work was done, the text has been substantially re-written.

I would like to thank my supervisor, Prof. Per Lidgren, for his support and patience; and special thanks are due to my friend, Nils Granlund, whose help and support have been invaluable.

Luleå, 2010-03-05

Table of Contents

Introduction

In recent years, the complexity of embedded systems has been steadily increasing. This has led to a recognition of the inadequacy of ad-hoc design techniques for embedded software (when a new technique is used for each new software system) and a nearly universal acceptance of the need for model-based design (MBD) as a means of handling system complexity. It is well-known that MBD allows to simulate system behavior at early stages of development, thus reducing the number of errors, shortening development cycles and thereby decreasing overall development costs. However, introduction of MBD, already well-established in development of general-purpose software systems, into the realm of embedded system development, has been slow and ridden with difficulties. One can argue that it has been hampered by the absence of models capable of capturing functionality of both software and hardware, when interaction between the two is crucial for most embedded systems. Another factor could be the absence of adequate modeling of timing behavior of software, which is necessary since most embedded systems are real-time systems.

There is thus a clear need for a new modeling approach to embedded software that would address these shortcomings. Such an approach should also provide a close coupling between a model and its realization, so that results of simulation of a model (including the timing behavior) would hold for its realization as well. The present work suggests using executable models expressed in the Timber language for modeling embedded software and its interaction with hardware. We also describe how such models can be simulated in Simulink, a simulation environment for discrete and continuous systems that is often used in industry in development of control systems of various kinds. Our approach to modeling and simulation is demonstrated in a case study of a software controller for an anti-lock braking system (ABS).

In the next chapter, we present the Timber language and a translation of a Timber model to C code, and we discuss the Simulink simulation environment. The following chapter describes how Timber can be used to model embedded real-time systems, followed by a chapter detailing the simulation setup necessary for simulating a Timber model in Simulink. A separate chapter presents the conducted case study. In conclusion, we discuss the contribution of the present work to modeling and simulation of embedded real-time software.

Background

The Timber language

Timber [1] is a programming language under development at Luleå university of technology (Luleå, Sweden). This high-level general-purpose programming language primarily targets real-time systems. In the next chapter we will show that it can be used to create executable models of embedded real-time software as well as hardware. By an executable model we mean a model that (a) can be "executed" in the sense that it can process input data as the realized system would, and calculate a result (e.g., in a simulation environment), and (b) can be deployed on the target platform after some automatic transformations, possibly together with some platform-specific but application-independent code (the second part of the definition only refers to the parts of the model realized in software). Here we limit ourselves to a brief summary of the relevant features of the language: object-orientation, reactivity, message passing between objects, and object-level concurrency. More details are available in [1, 2, 3, 4].

Object-orientation. Timber is an *object-oriented language*, where system state is always encapsulated in objects[1] and functionality can be expressed either using functions or using methods of objects [5]. Functions return values that only depend on the arguments; they are stateless, i.e. no information is preserved or shared between different invocations, and they cannot produce any side-effects (such as changing the value of a state variable or performing output). An object's method, on the other hand, can operate on the object's state variables and perform output, thus producing side-effects in addition to returning a value.

Reactivity. Timber adopts a *reactive programming paradigm*, i.e. functionality of the system is defined in terms of reactions to discrete external events; "discrete" means that each event has a specific time reference [5]. Moreover, Timber allows specifying timing requirements for each reaction in terms of a permissible execution window: a *baseline*, the earliest time when the reaction may start execution, defined relative to the time reference of the triggering event, and a *deadline*, the latest time when the reaction should complete execution, defined relative to the reaction's baseline (see Figure 1) [2]. This information is preserved in the code and can therefore be used at runtime to guide scheduling of execution. Note that these time constraints only specify the correct behavior of the system, not what happens if a deadline is missed. Moreover, the question of whether the system can be scheduled on a particular hardware platform has to be addressed separately, based on the worst-case execution times on the platform in question and the chosen scheduling algorithm. This approach is reasonable for hard real-time systems [W1], where missing a single deadline is equivalent to a system failure.

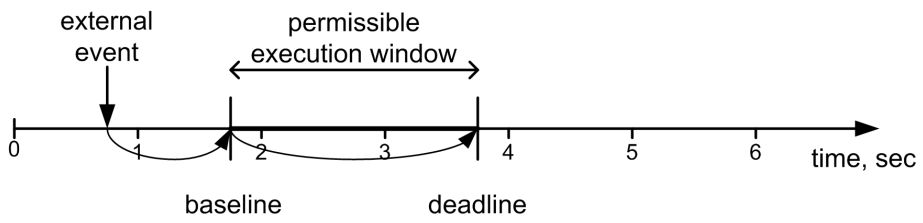

Figure 1. A permissible execution window for a reaction with a baseline offset *1 sec* and a deadline offset *2 sec*.

[1] Encapsulation means that an object's state variables can only be accessed via its methods.

Message passing between objects. Timber objects interact by posting *synchronous* and *asynchronous messages* to each other, with each message specifying which method should be executed [2]. Posting a synchronous message is very much similar to a classic method call, i.e. the caller waits for the invoked method to complete and optionally return a value. A synchronously called method always inherits the permissible execution window of the caller, which is the only logical choice since the caller cannot continue execution until the invoked method returns.

On the other hand, an object posting an asynchronous message may continue execution immediately after posting the message. Posting an asynchronous message is considered an internal event, and its execution – the recipient's reaction to this event. As is the case with reactions to external events, it is possible to define a new permissible execution window for each such reaction, i.e. for each posted asynchronous message. In cases when this window overlaps the window of the original reaction, this gives rise to concurrency (with parallel or interleaving execution of the reactions).

The baseline of a reaction to an internal event can either be inherited (Figure 2a), or defined relative to the baseline of the original reaction ("**after** *t*" in Timber, $t \geq 0$) (Figure 2b). In the latter case the baseline is not allowed to have passed, in which case it is adjusted to the current time (Figure 2c). For example, writing "**after** 0" forces the baseline to be set to the time when the message is posted. The deadline can either be inherited, or defined relative to the new baseline ("**before** *t*" in Timber, $t > 0$). The assumption here is that the system can be scheduled on the chosen hardware platform so that no deadline is missed.

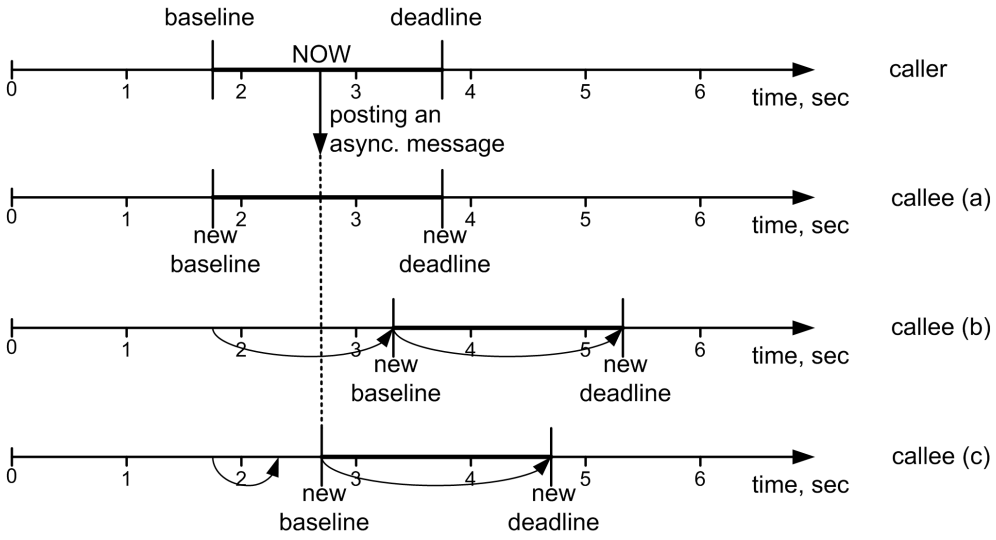

Figure 2. Possible permissible execution windows for a new asynchronous message: inherited baseline (a), a new baseline set in the future (b), and a new baseline set in the past and hence adjusted to the time when the message is posted (c). The new deadline is either inherited (a), or defined relative to the new baseline (b, c).

Object-level concurrency. Execution of messages has to be scheduled since there may be multiple reactions active at the same time. In fact, concurrent execution of different reactions (by means of true parallelism on a multi-CPU system, or by means of interleaving execution on a single CPU) is essential for a system continuously interacting with its environment. That is why Timber supports *object-level concurrency*, which results in an increased flexibility and better resource utilization [5].

A common problem with concurrency is that it gives rise to race conditions [W2], when the same state variable or hardware resource is shared between two reactions. To protect state variables inside objects from concurrent modification and thus ensure the consistency of system state, only one method of an object is allowed to be active at any given time. The same mechanism can be used to synchronize access to hardware resources, provided that such access is performed through a dedicated object. At the same time, semantics of Timber allows concurrent execution of methods of different objects, striking a balance between state protection and flexibility of scheduling.

For synchronous messages, this approach results in a possibility of a deadlock (if an object posts a synchronous message to itself, either directly or by way of other objects). While in some cases it may not be possible to verify absence of deadlocks statically (at compile time), it is always possible to detect them at run time.

This also means that a Timber kernel deployed alongside application code (a platform-specific micro-kernel [W4] handling scheduling of execution and communication between objects) has to support queuing of messages, both synchronous and asynchronous. For an object posting a synchronous message, queueing may lead to a delay before the message is allowed to execute, during which time the sender remains locked as it is waiting for the message to return.

Note that no specific scheduling policy is defined in the language itself, since any scheduling that leads to complying with the timing specification (in the form of permissible execution windows for reactions) is allowed. For example, both EDF (earliest deadline first) and EDF with SRP (stack resource policy) [6] scheduling can be used. In any case, schedulability of the system can be verified for a given hardware platform only together with a scheduling algorithm implemented in a Timber kernel.

Translation of Timber into C: tinyTimber

A Timber model can be compiled to C code with the Timber compiler [1]; the resulting C code can be combined with a platform-specific Timber kernel, and then compiled to binary code for a particular hardware platform using a suitable compiler. However, a system modeled using Timber does not have to be realized in Timber. While, as a programming language, Timber is certainly suitable for writing embedded software, and it is most certainly advantageous to avoid translation of the model into another language, other considerations may preclude this. For example, there may not (yet) exist a Timber kernel for the chosen hardware platform, or the platform may be so resource-constrained that the code produced by the current version of the Timber compiler cannot be scheduled on it so that the system is guaranteed to meet its deadlines. Hence the question arises how a Timber model can be (manually) translated into another programming language, and tinyTimber is in fact such a translation into the C language.

It has to be noted that, since in tinyTimber we are limited by what can be expressed in C, a substantial part of static verifications conducted by the Timber compiler cannot be performed on tinyTimber code, most notably there is no *type safety* [W4, 7] in tinyTimber (while Timber boasts an advanced type system with a combination of static type checking [W5, 7] and (optional) type inference [W5, W6]). Using tinyTimber requires a special variant of the Timber kernel; we call it a tinyTimber kernel.

Let us now describe how different aspects of a Timber model can be implemented in tinyTimber. For more information and examples of systems implemented in tinyTimber, see [8, 9, 10, 11].

Objects and methods. Following the common praxis of encoding object-orientation in C, each Timber object is implemented as a struct containing its state variables, with the object's methods defined as top-level functions that take a reference to the object as their first argument. It is the responsibility of the programmer to ensure that state variables of an object (i.e. members of the corresponding struct) are only accessed from within the methods of that object. Moreover, no global variables are allowed in the user code.

Scheduling. Recall that in Timber, only one method of each object can be active at any given time. In order to preserve this mutual exclusion between methods of the same object and allow a tinyTimber kernel to schedule execution based on the permissible execution window for each reaction, two conditions have to be met. Firstly, each object should be assigned a mutex lock [12, W7] that has to be acquired before a method of the object starts execution and released once it has completed execution (locking and unlocking is handled automatically by the tinyTimber kernel). Secondly, interaction between objects must always use special macros that create a synchronous or asynchronous message (with optional baseline and/or deadline offsets for an asynchronous message) and post it to the kernel. Thus no direct method calls between objects are allowed; however, stateless functions that do not belong to any object, i.e. do not operate on an object's state variables and do not produce side-effects, can be defined and called directly. For simplicity of implementation of a tinyTimber kernel, all objects' methods are limited to taking exactly one argument (beside the obligatory reference to the object), which can be a 32-bit integer or a 32-bit pointer.

Implementing concurrency. Asynchronous messages can be executed concurrently by the tinyTimber kernel by using multiple threads [W8]. To improve performance, a shared pool of reusable threads [W9] is used instead of creating and destroying a new thread for each message. Not only does this remove an unnecessary duplication of work, but it also allows us to limit the size of the thread pool and to statically assign memory regions to each thread's stack. This is especially important for resource-constrained systems where the amount of memory is a common limitation.

An asynchronous message is assigned a thread not earlier than its baseline has passed. However, this is only done if there is a thread available in the pool, and this limitation should be taken into account if a schedulability analysis is performed. The thread may be temporarily blocked during execution if it fails to acquire the lock on the object it should execute on (which may only happen if another message is being executed on the same object). Note that when a synchronous message is posted, the caller object remains locked, and hence the posted message can be executed on the same thread (as is done with function calls). Finally, the thread is released back into the pool when the original asynchronous message has completed execution.

Interaction with the environment. Reading and writing to the environment can easily be encoded as reading and writing to memory locations (at least for memory-mapped I/O [W10]), and external events that the system should react to are typically manifested as hardware interrupts. It is thus necessary to define interrupt handlers [W11], which are executed with the highest priority outside "regular" scheduling of messages. The interrupt handlers may perform reading and writing to the environment and post asynchronous messages to objects, but they are not allowed to post synchronous messages.

Simulink simulation environment

Simulink [13] is a well-known extension to Matlab. It includes a graphical environment for constructing models of various physical processes and control systems, and allows to simulate the models in virtual time.

A Simulink model is comprised by a number of blocks; each block has its inputs, outputs, and optionally state variables. Mathematical functions are used to describe the relationship between the inputs and old values of state variables, on the one hand, and outputs and new values of state variables, on the other hand; the latter are updated at each simulation point.

Simulink blocks are connected to each other, and the order of simulation is determined automatically so that all inputs are known when the outputs and new values of state variables are calculated for a certain block. This requires that any loops (circular dependencies) between blocks are not combinatorial, i.e. at least one block in a loop delays its output by one simulation step.

In Simulink, physical processes and control systems can be simulated either using discrete models, when each simulation point is separated from the previous one by a fixed time interval, or as continuous systems. In the latter case, the time intervals between simulation points vary, which allows accurately capturing the points in time when the system transitions from one state to another; such transitions can normally be detected by comparing a certain variable to a predetermined threshold value (so-called *zero-crossing detection*). Different solvers can be used to find such simulation points, and they differ in accuracy of zero-crossing detection as well as in computational cost (performance of each solver is highly dependent on the type of model being simulated).

The success of Simulink has been mainly due to the fact that for an important class of problems, it is sufficient to simulate system behavior at certain pivotal points (such as when a pendulum is at its turning points) to create an accurate picture of the system behavior as a whole. Simulink also allows to combine discrete and continuous models, and more generally, models with different solvers and different simulation rates, in a single simulation. It is widely used in industry, especially in design of various control systems. Importantly, there exist a substantial number of plugins allowing co-simulation of Simulink models and models in domain-specific simulators. The program CarSim [14] used in the case study below to simulate vehicle dynamics is one example.

Since many embedded systems are control systems, realized in software and developed using Simulink, there is a clear interest in incorporating a more realistic model of software into Simulink models, and in bringing simulation results closer to behavior of software realizations, particularly with respect to timing behavior. The present thesis demonstrates how this can be done using Timber executable models if they are either compiled to C code with the Timber compiler or manually translated to tinyTimber – a special subset of the C language.

Modeling Embedded Software Systems Using Timber

There exist two distinct kinds of models: models of natural phenomena and models of artifacts. Natural phenomena are modeled with the purpose of understanding what cannot be observed directly. The common assumption is that if a model behaves essentially similar to a natural phenomenon, then the model's (known) inner structure is a good guess for the (unknown) inner workings of the phenomenon. Apart from purely cognitive purpose, such models are often developed to predict the behavior of a phenomenon in the future.

Models of artifacts, on the other hand, serve a very different purpose. They are often created as a part of a system design process, on the way from some vague idea on how the system should work to its complete specification and realization. Multiple variants of the same model are typically created during this process, as we are filling in the details, adding or correcting information. If several models are created to reflect different aspects of the system, they need to be reconciled sooner or later, as the result of the development should be a single system. This approach to system development is known as *model-based design*, and one of its essential advantages is that already at early stages of development, we can use the model to predict (either by formal methods or by simulation) future behavior of the system in different circumstances and compare it to our idea of desired system behavior, correcting the model as needed. Thus modeling can be used to quickly explore different design choices and their consequences for system functionality and for other properties of the system. This normally leads to a much more effective and hence shorter and cheaper development cycles.

Models of embedded systems, including models of embedded software, are clearly models of artifacts. As they often need to be simulated together with models of natural phenomena, we need a single simulation environment that would allow us to build models of both kinds and simulate them together; one such environment is Simulink.

The traditional way to model software systems in Simulink is to focus on the control algorithm that will later be implemented in software. The algorithm is usually expressed as a (continuous or discrete) mathematical function. The problem with this approach is that it opens up a gap between the algorithm's behavior observed during simulation and the behavior of the implemented software system, as the implementation process is far from straightforward for all but the simplest of systems. This gap becomes increasingly difficult to bridge when implementation is substantially different from the model, for example, when the system is implemented partly in software and partly in hardware, or when resource constraints (processing power, amount of memory, energy consumption, etc.) significantly affect the implementation of the system.

The suggested modeling of embedded real-time software in the Timber language attempts to bridge this gap, on the one hand, being sufficiently abstract to remain useful at early stages of system design, and on the other hand, being directly coupled to a possible implementation on a target platform. Let us now discuss the requirements that we would like our model to meet, and demonstrate that Timber models do indeed satisfy these requirements.

Reactive approach to modeling

By definition, an embedded (software) system is a part of a larger system that has some other purpose than calculations per se. As calculations play an essential, yet subordinate role in such systems, the software's

interaction with the environment is of utmost importance. In fact, it is the external signals (originating from a human user, or from sensors detecting changes in the environment) that often bring the software system to life, demanding a reaction. It would therefore be beneficial to be able *to express system functionality in the model in terms of reactions to external events.*

This requirement is easily met by a Timber model thanks to the reactive programming paradigm of the language. All that is required is to identify external events (user input, changes in the environment, etc.) that the software system should react to, and to define the reactions, i.e. which methods of which objects are triggered by these events.

Independent specification of independent reactions

It is quite typical for embedded software systems to have multiple channels of interaction with the environment, and it is often possible to define a correct system response to each external event separately. Hence it would be desirable to support *independent specification of each reaction* in the model, which should be attempted unless there is a direct interdependence between them, such as using a shared resource. This approach would allow us to avoid imposing an unnecessary sequencing of reactions and increase responsiveness of the system.

In Timber, independence of reactions can be achieved if methods triggered by different external events belong to different Timber objects, since the methods can then be executed concurrently. At the same time, when two reactions are not independent (e.g., they both modify the same state variables or share a hardware resource), this can be expressed by assigning the methods to the same Timber object, which guarantees mutual exclusion between them. Since a reaction in Timber may involve more than one object, it is equally possible to express a partial dependence between two reactions (Figure 3).

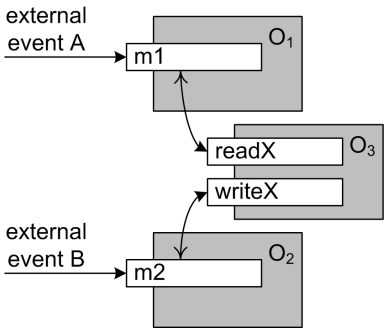

Figure 3. Method *m1* of object O_1 is triggered by an external event *A*, and method *m2* of object O_2 is triggered by an external event *B*. Both methods post synchronous messages to object O_3. As methods *readX* and *writeX* are mutually exclusive, the reactions are only partially independent.

Consistent modeling of hardware and software

A characteristic feature of embedded systems is that they typically manifest a close integration between hardware and software. Executing a single function of the system may involve going back and forth between them, e.g., when some process is controlled in a feedback loop, which involves both hardware (sensors, actuators) and software (calculation of output values). This situation can make it very difficult to specify and develop software separately from hardware, which calls for *consistent modeling of both hardware and software*.

This can be achieved if apart from modeling software, Timber is also used to model hardware parts of an embedded system. Interestingly, the reactive approach that we use to model software is very close to the traditional understanding of hardware. Moreover, hardware exhibits a strong compartmentalization that is easily expressed using Timber objects. Hardware is also characterized by a high level of concurrency, which is defined in Timber at the object level. Hence the use of Timber to model hardware is quite straightforward.

Incorporating timing requirements into a functional model

For a majority of embedded systems, meeting *timing requirements* is central for correct operation. Their importance is such that *the desired timing behavior needs to be incorporated into a functional model of the system* from the start. If this requirement is not met, i.e. separate models are used to describe functionality and timing behavior, it often leads to problems when the models are consolidated, leading to the necessity to redesign parts of the system.

This requirement is met in a Timber model by using permissible execution windows that can be defined for each reaction. While this approach imposes apparent limitations on what kinds of timing requirements can be expressed in the model, our study of various embedded systems indicates that in a vast majority of cases this approach is sufficient.

A high level of abstraction

As models are supposed to facilitate system design, they need to be *sufficiently abstract* to make system functionality and structure expressed in the model easily discernible, and changes to the model easy to make. In particular, we would like our models to be *free from platform-specific implementation details*.

Timber's object-orientation as such goes a long way towards achieving a good level of abstraction in the model. In addition, Timber as a high-level language allows to describe system functionality in a compact and abstract way. It also allows to describe system functionality in a platform-independent manner, with all platform-specific features (e.g., device drivers) concentrated in separate modules with a clear interface to the main application.

Executable models

In order to strengthen the link between a model and its implementation and ensure that the key properties of the model are preserved in the implementation, the modeling approach should enable *execution of models* and *their automatic realization* for a specific target platform.

Compilation of a Timber model to C using the Timber compiler makes the model truly executable, as the resulting application code only needs to be combined with a platform-specific Timber kernel and an implementation of low-level interaction with the environment. When a Timber model is manually translated to tinyTimber (a subset of C), the degree of automatization is, of course, much lower; however, since the translation from Timber to tinyTimber follows a number of very specific and rather simple rules (see Background), even in this case we can speak about an executable model.

We have now shown that all the specific requirements identified for models of embedded real-time software can be met using Timber as the modeling language. In the next chapter, we will turn our attention to how

Timber models (compiled to C code with the Timber compiler or manually translated into tinyTimber) can be simulated in Simulink.

Simulation of Timber Executable Models in Simulink

Let us now consider one very important part of model-based design – simulation of models. In particular, we will describe how a Timber model of an embedded real-time software system can be simulated in Simulink.

We assume that the goal of such simulation is to verify and if necessary correct the following aspects of a software model: (a) functionality, i.e. the calculated values, and (b) specification of timing behavior, i.e. *when* the calculated values should be output to the environment. Note that the veracity of simulation results will ultimately depend not only on the Timber model and the way it is incorporated into a Simulink block diagram, but also on the inputs sent to it from the model of its environment.

Exactly how a Timber model can be incorporated into a Simulink block diagram (a collection of models simulated together) will be described below. Note that since we are speaking specifically about simulating a Timber model of an *embedded software system*, we assume that its environment is modeled not in Timber, but directly in Simulink or in some domain-specific simulator that can be linked to Simulink through a plugin (Figure 4). For example, if a software controller of an anti-lock braking system (ABS) is modeled in Timber, the surrounding hardware (wheel speed sensors, brake valves, etc.) should be modeled in Simulink, as should be pressure change in brake chambers and the car's behavior on the road.

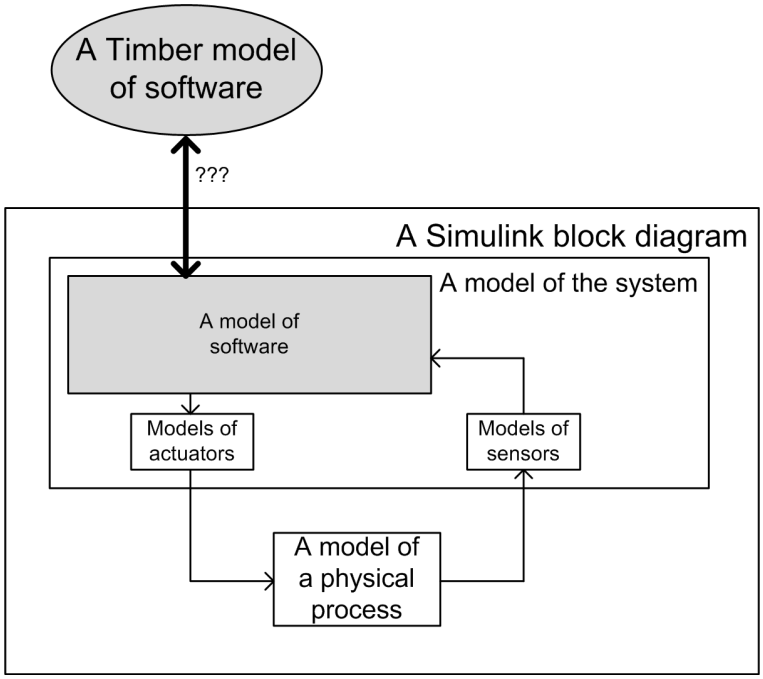

Figure 4. A Simulink block diagram that includes a model of the system (software and hardware) and a model of some physical process observed by the system's sensors and controlled by its actuators.

The environment of an embedded software system (or some part of it) often needs to be modeled as a continuous system. At the same time, it is clear that if we are to bring simulation of a model of a software system closer to its realization, its simulation should be discrete. Here Simulink's ability to combine multiple simulation domains (continuous or discrete with different simulation steps) within one block diagram comes in handy.

Interaction with the environment

Let us first consider the problems that arise when we try to transplant a Timber model of embedded software into Simulink. Not only can Timber models, as well as real software systems, read from the environment and write to it, but they can also be triggered by the environment through hardware interrupts (signals from hardware that trigger execution of special interrupt handler routines in software). An interrupt can be generated as a result of user input, or in a sensor when a value or a change in value crosses some threshold. However, the situation when an interrupt is generated in one model and should be detected in another model (e.g., when a discrete model of a software system may need to be combined with a continuous model of its environment) presents a problem since in Simulink, one model cannot trigger execution of another model. Indeed, interaction between models within a Simulink block diagram is governed by the following principles:

- A Simulink model may include (refer to) other models.
- Each Simulink model is associated with a simulation domain, and for each domain a solver and the (maximum) size of a simulation step can be chosen. There is one solver for discrete simulation and several solvers for continuous simulation.
- Simulation of each model is triggered individually as defined by the solver, i.e. the choice of simulation points for one model is not affected by the choice of such points for another model.
- Data can be sent from one model to another, but arrival of new data cannot be used to trigger simulation. In other words, input data is seen by each model as a "register" that can be read, not as an incoming signal.

Therefore we need to find a way to express *triggering through interrupts* using some constructs supported by Simulink. We can say that we need to *emulate* interrupts, which encompasses the following tasks:

- given a discretely or continuously changing value in the model of the environment, decide when to signal an interrupt;
- detect an emulated interrupt in the model of the software system.

It seems that the easiest solution is to model hardware that would generate interrupts in a real embedded system. For example, where we would have velocity in a continuous Simulink model, we would need to add Simulink blocks corresponding to speed sensors and generate a certain number of interrupts per distance unit. A special variable encoding the generated interrupts can then be used as an output from the model of the environment and an input to the model of software.

If interrupts are encoded using 0 ("no interrupt") and 1 ("interrupt", set for one simulation step), detection of interrupts is straightforward, but this leads to some problems since it is not possible to trigger execution of a model of software when an interrupt is set in the model of its environment. Firstly, if the size of a simulation step in the model generating an interrupt is *larger* than in the model of software, several interrupts may be detected instead of one (Figure 5 (a) and (b)). Secondly, if the size of a simulation step in the model generating an interrupt is *smaller* than in the model of software, some interrupts may remain undetected (Figure 5 (a) and (c)). To avoid these problems, we can encode interrupts using their timestamps instead of 0 and 1; then special triggering logic added to the model of software will detect an interrupt if the current

timestamp is different from the timestamp of the last detected interrupt (saved as a state variable) . This solution is made possible by a single clock available to all models in Simulink. However, one problem remains: in some cases, two interrupts may be coalesced, i.e. one interrupt is detected when two interrupts are generated (Figure 6). This problem can in many cases be ignored since this behavior is similar to behavior of real systems that do not buffer interrupts. Alternatively, it can be addressed by implementing buffering of interrupts or by making the size of a simulation step of the model of software sufficiently small to preclude a situation when two interrupts are generated within one simulation step of the model.

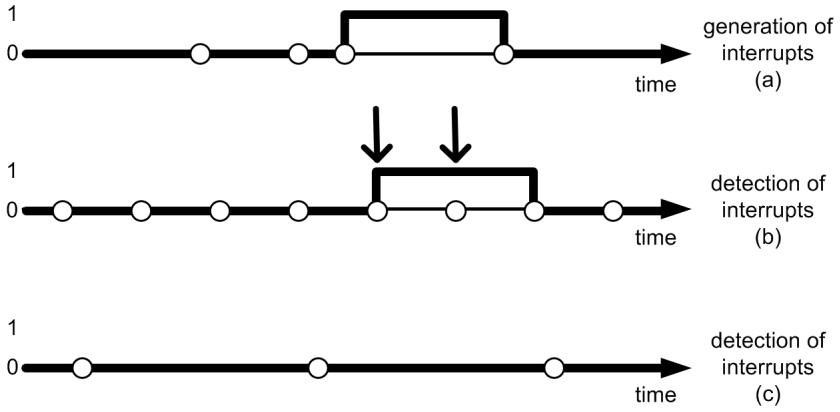

Figure 5. Problems caused by encoding interrupts using 0 and 1: one interrupt detected as two interrupts (a & b) and an undetected interrupt (a & c). The circles correspond to simulation points, and the arrows correspond to detected interrupts.

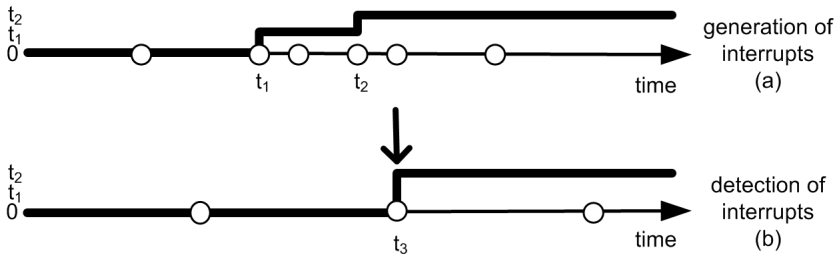

Figure 6. A problem with the suggested encoding of interrupts: coalescing of two interrupts into one. The circles correspond to simulation points, and the arrow corresponds to a detected interrupt.

We can now propose two-thronged approach to defining interaction between a discrete model of an embedded software system and a (continuous) model of its environment: reading and writing is implemented as input and output variables, whereas interrupts triggering execution of the software system are encoded using timestamps, with special triggering logic in the model of the software system.

Translation of a Timber model of software into a Simulink model

Although Simulink is not capable of running Timber code directly, it can run arbitrary C code within its simulation, using a mechanism called S-functions [15]. So we need to translate a Timber model of software into C code, which can be done with the Timber compiler or manually using tinyTimber. The result of such translation is C code for the modeled software application. However, to be able to run this code in Simulink, it is necessary to add a Simulink-specific Timber or tinyTimber kernel and I/O code (just as it is required for a realization on a hardware platform) (Figure 7).

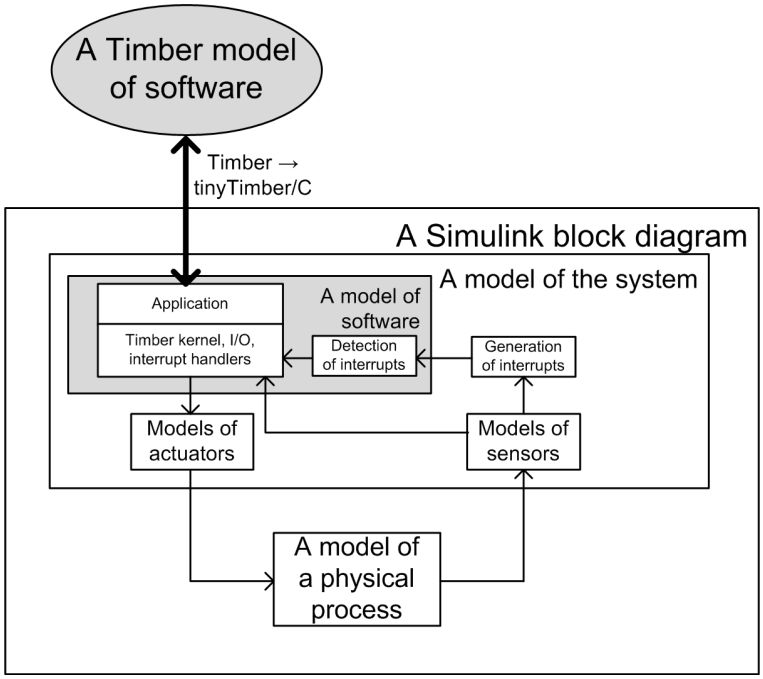

Figure 7. The Timber model becomes the application part of the model of software (cf. Figure 4).

The combined application code, kernel, and I/O code can be executed in an S-function, which is compiled to a binary form and invoked during simulation. To make simulation more effective, we put the S-function in a separate triggered subsystem [16]. Each time the S-function is executed, the kernel checks pending messages in the system and determines when the next simulation point should be; this is sent as an output. The subsystem containing the S-function will only be triggered when an incoming interrupt is detected, or when there is a message to be executed, which allows us to run simulation at a higher rate without a substantial increase in simulation time (Figure 8).

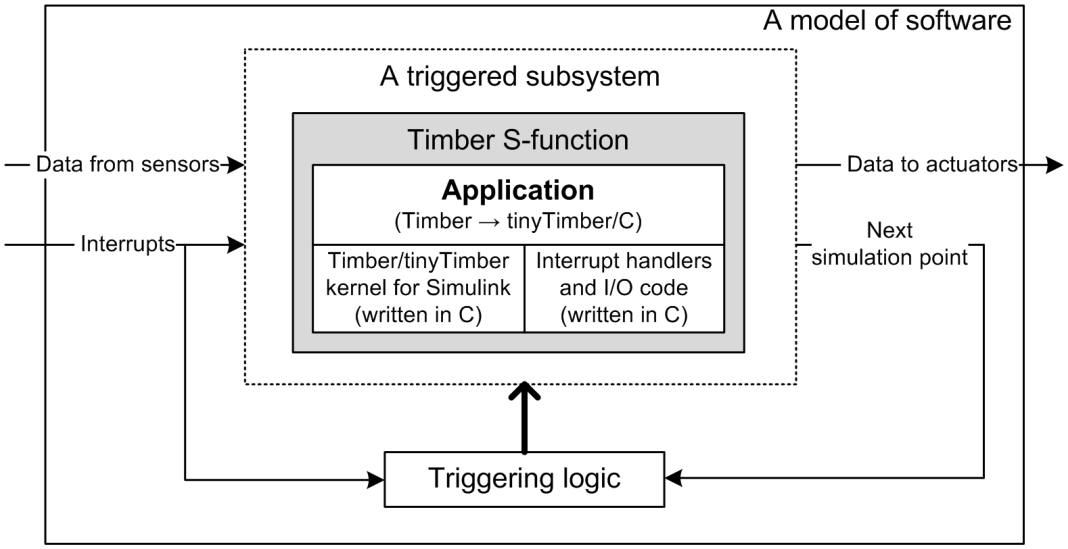

Figure 8. The structure of a model of embedded software in Simulink (corresponds to the gray rectangle in Figure 7).

A question arises if we can make the manually written C code (the Simulink-specific kernel and I/O code) generic enough so that it does not have to be changed when we change the application. The kernel should only exist in two variants – one for C code generated from a Timber model with the Timber compiler and

another for manually written tinyTimber code, so it does not depend on the application. I/O code, namely interrupt handlers and code for writing and reading Simulink variables from C code, can be written in application-independent manner, but it depends on the configuration of the environment. Hence it need not be changed with an application, but has to be modified if the environment is changed.

Simulation of a model of software in Simulink

We have shown how a Timber model of embedded software can be translated into a Simulink model and how its interaction with the environment should be set up. Let us now describe how this model is simulated in Simulink.

Since our goal is to simulate functionality and permissible timing behavior of a software system expressed in its model (as opposed to simulating its realization on a particular hardware platform), we would like the simulation to be independent of both the particular hardware platform and the scheduling algorithm. This precludes using worst-case execution times as they are platform-specific. Note that the purpose of such simulations is to explore the design space of the system, not to analyze schedulability of a specific realization.

Recall that a Timber model (= a Timber program) is a collection of objects with state variables and methods. Objects communicate by sending synchronous and asynchronous messages to each other, each message invoking a particular method of the recipient object when it is executed. Execution of a message can be scheduled anywhere within its permissible execution window, and any two messages can be executed concurrently as long as they are not sent to the same object. Outside "regular" scheduling of messages are interrupt handlers, executed immediately upon arrival of an interrupt. An interrupt handler typically posts an asynchronous message to a method of a particular object, possibly after performing some necessary I/O that cannot be delayed.

The question is how we should schedule execution of messages inside an S-function in Simulink. During simulation, scheduling is performed by the Simulink-specific Timber or tinyTimber kernel. Below we present two possible scheduling strategies, one based on using multiple threads and the other one single-threaded. We will show that a multi-threaded simulation is necessary if all possible schedules are to be reflected in the simulation, but that system behavior can be approximated using a single-threaded simulation, which is much easier to implement. Let us start, however, by discussing how the size of a simulation step for our model of software should be set.

Choosing the size of a simulation step

In Simulink, the simulation step size is set manually for discrete models and it remains constant throughout the simulation. For continuous models, the *maximum* size of a simulation step must be set. Note that it is normally set separately for each model, independently of all other models in a Simulink block diagram.

The situation with our model of software is, however, almost reverse. Internally, we can consider an arbitrary number of points between two simulation points (posting times, baselines and deadlines, start and completion times of messages), so a model of software does not dictate a particular size of a simulation step. However, the step size determines how inputs to the model (including emulated interrupts used to trigger simulation) and outputs from the model are discretized. So the simulation step size is primarily important

for modeling the software system's interaction with the environment: it determines how often we can receive interrupts, how often input and output values can be changed (i.e. what is the maximum time lag between calculating a new value and the time when it is seen by the model of the environment). Hence the size of a simulation step of our model of software should be set based on the simulation rates of the other models it interacts with, rather than independently.

This requires that the simulation itself should be independent of the size of a simulation step; this can be achieved if the simulation step size is defined as a parameter to the model and as an input to the S-function.

Need for multi-threaded simulation

Let us demonstrate the need for a multi-threaded simulation of a Timber model of software using the following example.

Let $O_1.m$ be an asynchronous message to the object O_1 that performs two synchronous calls to the object O_2, $O_2.m_A$ and $O_2.m_B$, in that order, followed by an output Z. Let also $t_1 < t_2 < t_3$ be simulation points such that the baseline of $O_1.m$ lies between t_1 and t_2 ($t_1 < b(O_1.m) < t_2$) and the deadline of $O_1.m$ lies between t_2 and t_3 ($t_2 < d(O_1.m) < t_3$). Finally, let $O_2.irq$ be connected to an interrupt that arrives at t_2 and whose deadline lies between t_2 and the deadline of $O_1.m$ ($t_2 < d(O_2.irq) < d(O_1.m)$) (Figure 9).

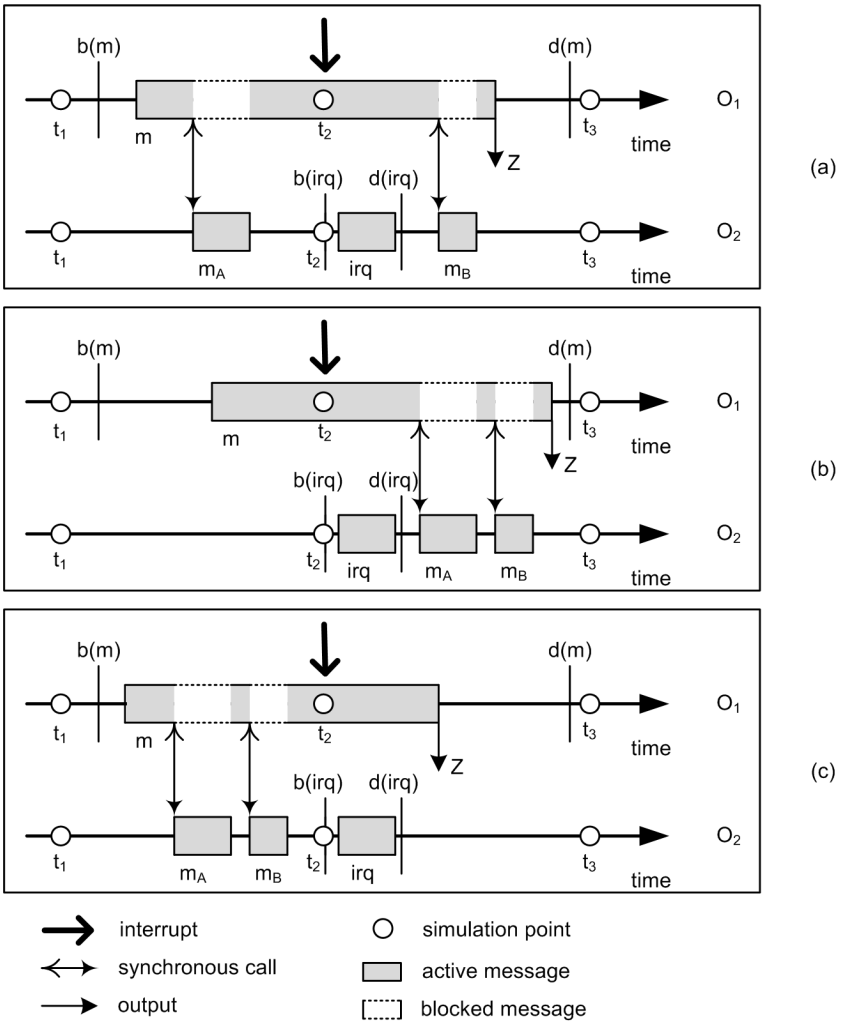

Figure 9. Some possible schedules of execution. Note that more than one message can be "active" at any given time.

Looking at the object O_2, we can see that the order of execution of its methods can vary:

(a) $t(O_2.m_A) < t(O_2.irq) < t(O_2.m_B)$;

(b) $t(O_2.irq) < t(O_2.m_A) < t(O_2.m_B)$;

(c) $t(O_2.m_A) < t(O_2.m_B) < t(O_2.irq)$.

We write $t(msg)$ to denote the *start time* of the message *msg*, i.e. the time when its execution begins.

Clearly, the order of execution may affect the output Z from $O_1.m$ if the three methods in O_2 modify the same state variables and the output Z is dependent on them (e.g., if m_A resets a counter, *irq* increases it by one, m_B reads its value, and this value is then used for the output Z). As all these orders of execution are allowed by the Timber model, there is non-determinism present in it. Since this non-determinism may affect the observable results of execution, it can be argued that it should be reflected in the simulation. However, this presents a problem, since we then need to be able to suspend execution of $O_1.m$, execute $O_2.irq$, and then resume execution of $O_1.m$. The only reasonable way to achieve that without modifying the application code is to use two *different threads*: one to execute $O_1.m$ (together with $O_2.m_A$ and $O_2.m_B$), and another to execute $O_2.irq$.

Scheduling based on a multi-threaded simulation

In order to capture all possible schedules we propose the following simulation scheme. Let us randomly choose the *start time* of each message between its baseline and deadline. Further, let us split execution of a message into parts so that a new part starts whenever a message (synchronous or asynchronous) is posted or when a synchronous call returns (which is also the completion time of the synchronous message). The start time of each part is chosen randomly between the start time of the previous part and the deadline of the asynchronous message. For the last part, the completion time of the asynchronous message is also chosen randomly. Any output is scheduled at the start time of the part that contains it.

Thus defined start times may be distributed over several simulation steps. For each simulation step, we have a number of parts of (possibly different) asynchronous and synchronous messages that are executed in the order of their start times, provided that the object to which the message is posted is not locked by another message[2] (otherwise, the execution is delayed until the object is unlocked).

Each asynchronous message is assigned a new thread that is also used for execution of synchronous messages posted from it. The *start time* of the thread is set to the start time of the message. Only one thread is runnable at any given time, as switching between threads is always performed explicitly by blocking the thread after execution of each part of the message. When a thread is blocked, its new start time is set to the start time of the next part of the message. The thread is unblocked when it has the earliest start time of all the threads, provided that (a) the object that the thread will run on is not locked, and (b) the start time of the thread precedes the current simulation point.

At each simulation point, we may also have one or more pending interrupts. The order of simulation is the following: processing interrupts (i.e. executing interrupt handlers, which normally includes posting an

[2] A message locks the recipient object for the whole time of execution, even if it is split into several parts.

asynchronous message); processing messages with passed baselines; and executing threads in *start time order* until there are no threads left or until the earliest start time follows the current simulation point. Note that executing threads may involve posting new messages, with synchronous messages processed at once and asynchronous messages processed if their baseline comes before the current simulation point.

It can be shown that the presented scheduling strategy covers all interesting schedules of message execution. However, implementation of multiple threads inside an S-function is rather difficult. We therefore suggest an alternative single-threaded implementation that disallows interleaving of execution of different messages and thus excludes some of the schedules that are allowed by the model (this approach was used in the case study described in the next chapter).

Scheduling based on a single-threaded simulation

To be able to simulate execution without using multi-threading, we set execution time of each method to zero. Then we do not have any interleaving of execution of messages, and each message can be executed from the start to the end, including synchronous calls to other objects; as only one message will be executed at any given time, the object to which a synchronous message is posted will not be locked (unless we have a circular dependency, which results in a detectable deadlock both during simulation and at runtime). Note that it is still possible to express some of the non-determinism of the model described in the example above by randomly choosing the start time of each asynchronous message between its baseline and deadline.

Going back to the example above, we can see that of the three possible orders of execution only two (b and c) can be simulated with this strategy; interleaving execution (a), which requires suspending and resuming execution of a message (i.e. multi-threading), cannot be achieved.

Clearly, implementing this scheduling strategy is easier than implementing a multi-threaded version. However, since some orders of execution allowed by the model will never be observed in this kind of simulation, it is important to understand for which systems such simulation will be sufficient. In principle, interleaving should only be significant if *two synchronous calls* are made from one message; for systems that do not include such behavior, both implementation strategies should be equivalent.

Significance and limitations of simulation results

In conclusion, let us summarize what a simulation of a Timber model of embedded software in Simulink, performed as described in this chapter, can tell us.

Firstly, such simulations reflect the behavior of a model (closely related to specification), not of a specific realization. Specifically, the timing behavior exhibited by a model during simulation represents the timing behavior *allowed by the model*, not *achievable* or *realistic on any particular hardware platform*. The question of schedulability has to be addressed separately for each realization and for each hardware platform.

Secondly, although each simulation represents the result of only one (randomly chosen) possible schedule of execution that conforms to the model, a number of simulations can be used to create a general picture of system behavior. However, if a single-threaded implementation is used, some potentially important variants of system behavior will be omitted.

Finally, it is important to remember that the quality of the simulation also depends on the quality of input sent to the model of software, i.e. the quality of the other models co-simulated with it.

A Case Study: Anti-Lock Braking System Controller

An ABS (anti-lock braking system) controller is a typical example of an embedded software system that has to operate under hard real-time requirements. In the case study presented in this chapter, ABS software is modeled in Timber, translated to tinyTimber, and simulated in Simulink. The vehicle dynamics is simulated in CarSim [14], which can be connected to a Simulink model using a special plugin. The case study aims to demonstrate that our approach to modeling and simulation works in reality.

System description

ABS [W12] is a safety system developed to allow the driver to maintain steering control of a motor vehicle under heavy braking by preventing skidding. This is typically achieved by continuously monitoring the speed of each wheel and reducing hydraulic pressure to the brake when the wheel's speed becomes less than 80% of the vehicle's speed. Here we present the hardware configuration and the control algorithm of a basic ABS.

Hardware configuration

ABS consists of several hardware components:

- a *master cylinder* with brake fluid; displacement of the brake pedal is amplified and transformed into hydraulic pressure in the master cylinder;
- a *low pressure accumulator* for brake fluid; brake fluid is pumped from the accumulator to the master cylinder;
- a *hydraulic control unit* (HCU), controlling the flow of brake fluid between the master cylinder, the low pressure accumulator, and the brake chambers; the HCU contains two valves for each wheel:
    - an *isolate valve* controls the flow of brake fluid between the master cylinder and the brake chamber: when the valve is open, the pressure in the brake chamber increases until it reaches the pressure in the master cylinder;
    - a *relief valve* controls the flow of brake liquid between the brake chamber and the low pressure accumulator: when the valve is open, the pressure in the brake chamber decreases;
- magnetic *speed sensors* at the wheels, generating a certain number of interrupts per rotation;
- an *electronic control unit* (ECU) that opens and closes the valves based on the incoming interrupts.

Control algorithm

ABS controller continuously monitors the speed of each wheel and compares it to the speed of the vehicle to detect skidding. When skidding is detected, ABS is activated and remains active until all the wheels have stopped skidding. The inputs to the control system are interrupts from the four speed sensors (40 interrupts per rotation in our case study), and the outputs are control signals to respective valves.

The system should asynchronously record timestamps of incoming interrupts and use them to calculate speed and acceleration of each wheel. The speed of the vehicle as a whole is then determined based on a

previous value of the vehicle's speed and current values of wheel speed. A simplified algorithm for determining the vehicle's speed is to take the speed of the fastest wheel (which takes care of skidding by up to three wheels) but also to check whether all four wheels are skidding by comparing the current value of the vehicle's speed with its previous value. If the drop in the vehicle's speed exceeds a certain threshold, all four wheels are assumed to be skidding, and the vehicle's deceleration is assumed to be constant (in our case study it is kept at $0.5\times G$, where G is the gravitational constant). A new value of the vehicle's speed is calculated at regular intervals (4±0.1ms in our case study).

Once the vehicle's speed has been established, the speed of each wheel is compared to it. Skidding is detected if the wheel's speed is less than 80% of the vehicle's speed. The current status of the wheel (skidding/not skidding) is not propagated further, but it is compared it to its previous status, and two kinds of events can then be detected: a wheel may have *started skidding* or it may have *stopped skidding*.

These events are then used as input to valve control algorithms:

- by default (no skidding), ABS is inactive: all isolation valves are open and all relief valves are closed; the hydraulic pressure in the wheels' brake chambers is kept equal to the pressure in the master cylinder;
- if a wheel has started skidding, the pressure in the corresponding brake chamber should be reduced; this is done by immediately closing the isolation valve and repeatedly opening the relief valve for certain periods of time dependent on the wheel's acceleration a (in our case study it is opened for $t=|a\times 10|\pm 0.1$ms and closed for 5±0.1ms); the relief valve is not opened if the wheel is accelerating ($a > 0$);
- if a wheel has stopped skidding, the pressure in the corresponding brake chamber should be increased; this is done by immediately closing the relief valve (if it is open) and repeatedly opening the isolate valve for certain periods of time dependent on the wheel's acceleration a (in our case study, it is opened once for 100±0.1ms and closed for 20±0.1ms; it is then opened 10 times for $t=|a\times 50|\pm 0.1$ms and closed for 20±0.1ms); after that, the system reverts to the default state and the isolate valve is left open.

Timber model

In this section, we will discuss a possible Timber model of the ABS controller described above. Complete Timber code for the model can be found in Appendix.

Since timestamps of incoming interrupts from each speed sensor need to be recorded, it makes sense to have four objects (one for each wheel) that hold this information and provide methods for calculation of speed and acceleration. These objects also need an asynchronous method *irq* that is connected to the corresponding interrupt.

A separate object is needed to hold the previous value of the vehicle's speed. It has two methods: the method *fetch* that is invoked periodically to collect the values of speed from the four wheels, determine the vehicle's speed, and detect *start skidding* / *stop skidding* events for each wheel; and the method *reset* that is connected to the reset interrupt and invokes *fetch* for the first time when ABS is activated. Subsequent periodic invocation is achieved by the method *fetch* posting an asynchronous message to itself with a delayed baseline (in our case study the delay is 4ms).

Finally, there should be four controller objects (one for each wheel), each implementing a finite state machine [W13] controlling the valves (Figure 10). The methods in these objects are *resetController*, which returns the system to the default state; *startedSkidding* and *stoppedSkidding*, which are invoked from the method *fetch* in the central controller object when corresponding events are detected; and *step*, which is invoked internally inside the object to stepwise increase or decrease the pressure in the brake chambers.

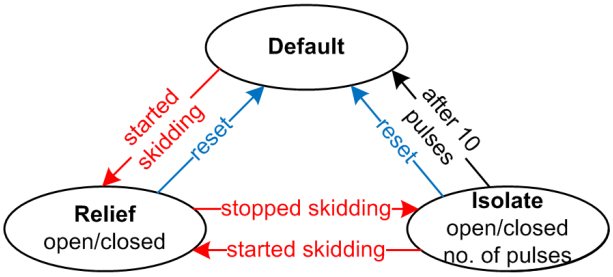

Figure 10. The state machine of a wheel controller object.

The object structure of the model is presented in Figure 11. Note that the environment (whose interface has to be defined in Timber even though it is not modeled in Timber) includes eight valves (two for each wheel) that can be opened and closed. The interrupts include the reset interrupt, invoked when ABS is activated, and four interrupts from the speed sensors at the wheels.

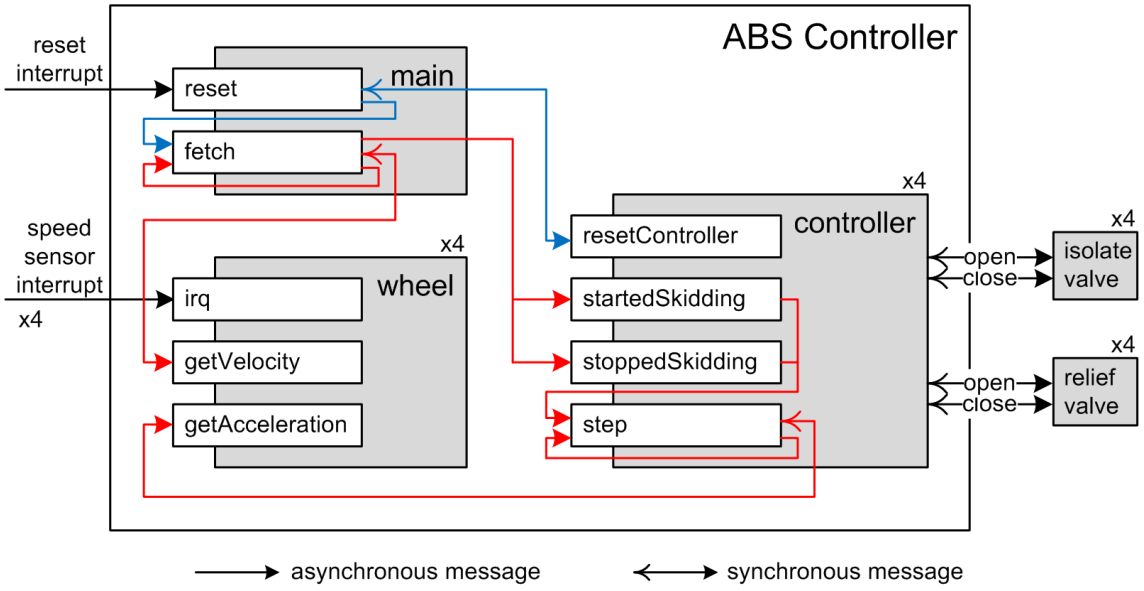

Figure 11. A Timber model of an ABS controller. The system reacts to five interrupts and controls eight valves. Apart from the central controller object (*main*), there are four *wheel* objects and four *controller* objects.

Simulation

Traditionally, an ABS controller is developed and simulated in Simulink purely at the algorithm level, as a continuous function, and simulated without regard to the effects of scheduling on a real hardware platform. In contrast, a Timber model of an ABS controller, while being sufficiently abstract to facilitate model exploration and modification, is an executable model, so it can be both simulated in Simulink and executed on a real hardware platform. Thus functionality as well as timing specification of the resulting software can be verified in simulation of the model. Below we will describe how this simulation is set up.

Let us start with a description of the environment. Simulation of vehicle dynamics is performed in a separate simulator CarSim for a specified behavior of the driver (e.g. braking 2 sec after the start of simulation), a specified vehicle model, and a specified road surface (e.g. ice on the left side of the road and asphalt on the right side). There is a special plugin that allows to connect a model in CarSim to a model in Simulink and to simulate them in parallel. The outputs from CarSim are master cylinder pressure and each wheel's speed, and the inputs to CarSim are the calculated values of pressure in the brake chambers. Communication between the simulations is discrete, i.e. the inputs and outputs are updated at predetermined intervals.

So the Simulink block diagram for the simulation includes (Figure 12):

- the CarSim plugin, i.e. the S-function implementing communication between Simulink and CarSim;
- a continuous model of brake chambers that calculates pressure in individual brake chambers from master cylinder pressure and valve control signals;
- a discrete model that translates each wheel's speed into interrupts, effectively modeling speed sensors at the wheels;
- a discrete model of software (ABS controller).

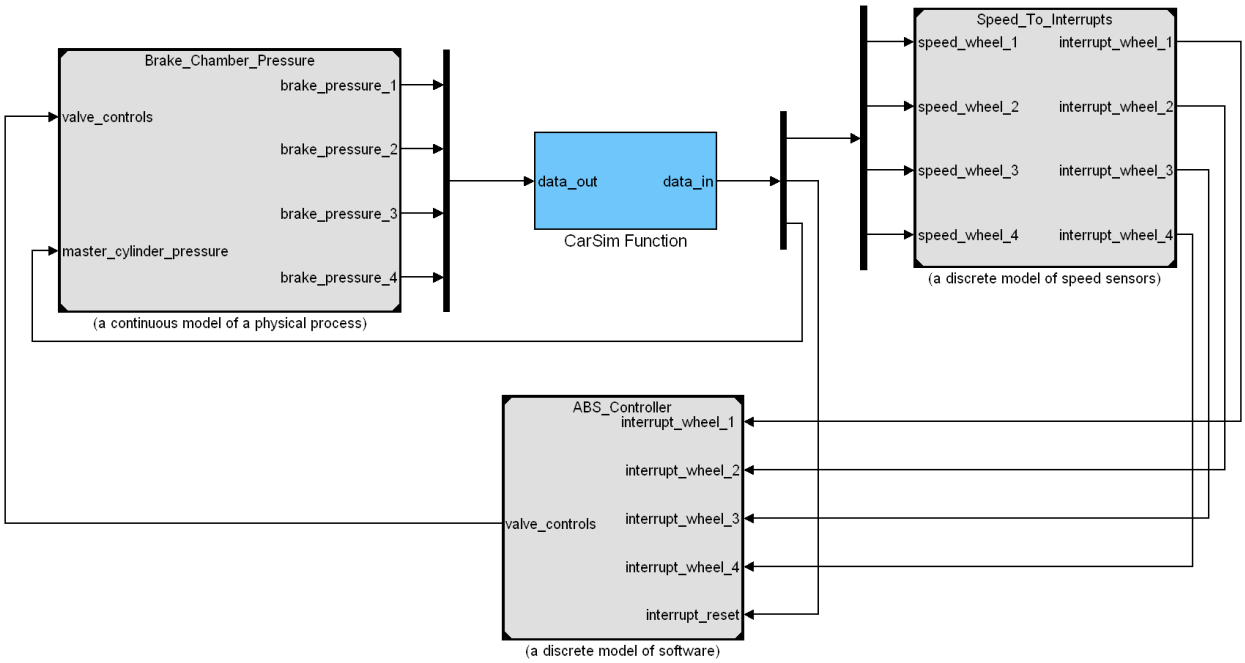

Figure 12. The Simulink block diagram for simulation of ABS.

The model of software contains, in accordance with the structure proposed in the previous chapter (Figure 8), triggering logic and a triggered subsystem with an S-function (Figure 13). The former analyzes input variables used to emulate interrupts and the "next simulation point" output from the S-function in order to determine whether the triggered subsystem should be executed at the current simulation point. The Timber S-function contains C code for the application (ABS controller), a tinyTimber kernel (the variant used in the simulation is single-threaded, with execution time of each method set to zero), interrupt handlers for the five interrupts (the reset interrupt and four interrupts from the speed sensors), and some I/O code that writes

outputs (control signals to the valves) to Simulink variables. The application code has been obtained by translating the Timber model of an ABS controller into tinyTimber.

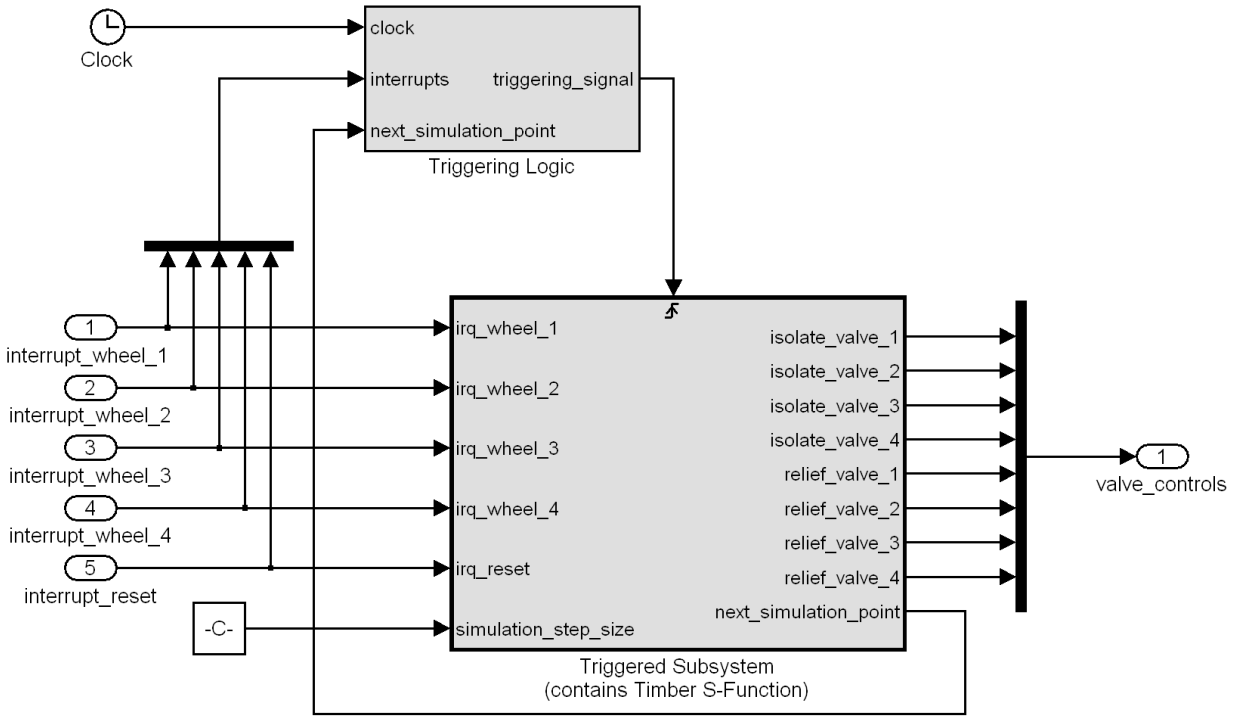

Figure 13. The Simulink model of an ABS controller.

The size of a simulation step for the model of software is set so that the emulated interrupts are not coalesced. This can easily be achieved if the model of software is simulated at the same rate as the discrete model of speed sensors that generates emulated interrupt signals.

The simulation has been conducted successfully, which demonstrates that a Timber model of embedded software can indeed be simulated in Simulink as described in the previous chapter.

Conclusion

In this thesis, we have identified the requirements on a model of embedded real-time software and showed that Timber models satisfy these requirements. The requirements discussed above are: reactive approach to modeling; independent specification of independent reactions; consistent modeling of hardware and software; incorporating timing requirements into a functional model; a high level of abstraction; and executability of models.

In addition, we have presented a method for translating a Timber model of embedded software into a discrete Simulink model that can be incorporated into a traditional Simulink model of the environment. Specifically, we suggested a solution to the problem of emulation of interrupts (which arises when the model of software and the model of its environment need to be simulated in different simulation domains) and suggested two implementation strategies for scheduling of execution of messages in a simulation of a Timber model.

Finally, we have demonstrated applicability of this approach to modeling and simulation in a case study of an ABS controller. We have shown that a Timber model of embedded software can be simulated in Simulink, a widely used tool for modeling, simulating and analyzing multi-domain dynamic systems.

The results presented in this thesis open for simulation of Timber models together with a wide range of simulators of discrete and continuous systems that can be co-simulated with a Simulink model, and even together with real hardware (so-called hardware-in-the-loop simulation). In the proposed simulation setting, the specification of timing behavior of a software system as expressed in its Timber model can be observed, allowing to adjust the specification to achieve intended system behavior. This introduction of verifiable timing properties into simulation, together with the true identity between the code used for simulation and the code used for realization (both can be derived from the same executable Timber model) should greatly simplify the design process and make the resulting software more reliable.

References

[1] Timber homepage.
http://www.timber-lang.org

[2] Programming with time-constrained reactions. / Nordlander, Johan; Carlsson, Magnus; Jones, Mark P.; Jonsson, Jan. 2005. 20 p.
http://pure.ltu.se/ws/fbspretrieve/441200

[3] The semantic layers of Timber. / Carlsson, Magnus; Nordlander, Johan; Kieburtz, Dick. In: Programming Languages and Systems, First Asian Symposium, Proceedings: APLAS 2003. Springer, 2003. p. 339-356 (Lecture Notes in Computer Science; 2895).
http://pure.ltu.se/ws/fbspretrieve/398223

[4] Time for Timber. / Lindgren, Per; Nordlander, Johan; Svensson, Linus; Eriksson, Joakim. Luleå: Luleå tekniska universitet, 2005. 20 p. (Research report / Luleå University of Technology; 2005:01).
http://pure.ltu.se/ws/fbspretrieve/299960

[5] Reactive objects. / Nordlander, Johan; Jones, Mark P.; Carlsson, Magnus; Kieburtz, Richard B.; Black, Andrew P. In: Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. IEEE, 2002. p. 155-158.
http://pure.ltu.se/ws/fbspretrieve/398236

[6] Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms. / Stankovic, John A. (editor). Kluwer, 1998.

[7] Types and Programming Languages./ Pierce, Benjamin C. MIT Press, 2002.

[8] TinyTimber, reactive objects in C for real-time embedded systems. / Lindgren, Per; Nordlander, Johan; Aittamaa, Simon; Eriksson, Johan. In: Proceedings, Design, Automation and Test in Europe: Munich, Germany, March 10-14, 2008. European Design and Automation Association, 2008. p. 1382-1385.
http://pure.ltu.se/ws/fbspretrieve/1942030

[9] Programming with the TinyTimber kernel. / Nordlander, Johan. Luleå: Luleå tekniska universitet, 2007. 18 p. (Technical report / Luleå University of Technology; 2007:14).
http://pure.ltu.se/ws/fbspretrieve/1249466

[10] A comprehensive approach to design of embedded real-time software for controlling mechanical systems. / Eriksson, Johan; Lindgren, Per. In: Asia Pacific Automotive Engineering Conference Technical Papers. SAE, 2007. (SAE Technical Papers; 2007-01-3744).

[11] Using Timber in a multi-body design environment to develop reliable embedded software. / Eriksson, Johan; Nybacka, Mikael; Larsson, Tobias; Lindgren, Per. In: Intelligent vehicle iniative (IVI) technology controls and navigation systems, 2008. Warrendale, PA: SAE, 2008. 8 s. (SP / Society of

Automotive Engineers, SAE; 2193).
http://pure.ltu.se/ws/fbspretrieve/2154795

[12] Mutual Exclusion with Locks – an Introduction.
http://www.thinkingparallel.com/2006/09/09/mutual-exclusion-with-locks-an-introduction/

[13] Simulink - Simulation and Model-Based Design..
http://www.mathworks.com/products/simulink/

[14] Mechanical Simulation - CarSim.
http://www.carsim.com/products/carsim/index.php

[15] Overview of S-Functions.
http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/sfg/f6-633.html

[16] Triggered Subsystems.
http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/ug/f4-84283.html

[W1] Wikipedia: Real-time computing.
http://en.wikipedia.org/wiki/Real-time_computing

[W2] Wikipedia: Race condition.
http://en.wikipedia.org/wiki/Race_condition#Computing

[W3] Wikipedia: Microkernel.
http://en.wikipedia.org/wiki/Kernel_(computing)#Microkernels

[W4] Wikipedia: Type safety.
http://en.wikipedia.org/wiki/Type_safety

[W5] Wikipedia: Type Systems.
http://en.wikipedia.org/wiki/Type_system

[W6] Wikipedia: Type inference.
http://en.wikipedia.org/wiki/Type_inference

[W7] Wikipedia: Mutual exclusion.
http://en.wikipedia.org/wiki/Mutual_exclusion

[W8] Wikipedia: Thread.
http://en.wikipedia.org/wiki/Thread_(computer_science)

[W9] Wikipedia: Thread pool pattern.
http://en.wikipedia.org/wiki/Thread_pool_pattern

[W10] Wikipedia: Memory-mapped I/O.
http://en.wikipedia.org/wiki/Memory-mapped_I/O

[W11] Wikipedia: Interrupt handler.
http://en.wikipedia.org/wiki/Interrupt_handler

[W12] Wikipedia: Anti-lock braking system.
http://en.wikipedia.org/wiki/Anti-lock_braking_system

[W13] Wikipedia: Finite-state machine.
http://en.wikipedia.org/wiki/Finite-state_machine

Appendix

Below you can find the Timber code for the model of an ABS controller used in the case study.

```
---------- file Env.t ----------
module Env where

struct Valve where
  open:: Request ()
  close:: Request ()

struct Env where
  installSpeedSensorHandler:: Action -> Int -> Request ()
  installResetHandler:: Action -> Request ()
  isolate_valves:: Array Valve
  relief_valves:: Array Valve

extern absEnv:: World -> Class Env

---------- file Wheel.t ----------
module Wheel where

struct Wheel where
  irq:: Action
  getSpeed:: Request Float
  getAcc:: Request Float

wheel = class
  t1:= (sec 0)
  t2:= (sec 0)
  t3:= (sec 0)
  tmr = new timer
  irq = action
    t1:= t2
    t2:= t3
    tmr.reset
    temp <- tmr.sample
    t3:= temp
  getSpeed = request
    result distance/(deltaT t3 t2)
  getAcc = request
    result (distance/(deltaT t3 t2) - distance/(deltaT t2 t1))/(deltaT t3 t2)
  result Wheel{..}

wheelRadius          = 0.3
interruptsPerRotation = 40.0
distance             = wheelRadius*2.0*3.14/interruptsPerRotation
toFloat t            = (fromInt (secOf t)) + (fromInt (microsecOf t))/1000000.0
deltaT t_last t_prev = (toFloat t_last) - (toFloat t_prev)
```

```
---------- file ABS.t ----------
module ABS where

import Env
import Controller
import Wheel

root:: World -> Cmd () ()
root world = do
  env = new absEnv world
  wheelList = forall i<-[0..3] new wheel
  wheels = array wheelList
  forall i <- [0..3] do
    env.installSpeedSensorHandler (wheels!i).irq i
  contrList = forall i<-[0..3] new controller (wheels!i) (env.isolate_valves!i) (env.relief_valves!i)
  contrs = array contrList
  reset = new main wheels contrs
  env.installResetHandler reset

main:: Array Wheel -> Array Controller -> Class Action
main wheels contrs = class
  vehicleSpeed:= 0.0
  wheelSpeeds:= array [0.0,0.0,0.0,0.0]
  skidding:= array [False,False,False,False]
  pendingMsg:= Nothing

  reset = before (microsec 100) action
    if (isJust pendingMsg) then
      abort (fromJust pendingMsg)
    forall i<-[0..3] do
      (contrs!i).resetController
    skidding:= array [False,False,False,False]
    after (millisec 4) fetch

  fetch = before (microsec 100) action
    forall i<-[0..3] do
      wheelSpeed <- (wheels!i).getSpeed
      wheelSpeeds!i:= wheelSpeed
    vehicleSpeed:= let
                     v_new  = max (max (wheelSpeeds!0) (wheelSpeeds!1)) (max (wheelSpeeds!2) (wheelSpeeds!3))
                     decel  = (vehicleSpeed - v_new)/0.004
                     v_proj = vehicleSpeed - 5*0.004 in
                   if (v_new < vehicleSpeed && decel > 5) then v_proj else v_new
    forall i<-[0..3] do
      if (wheelSpeeds!i < 0.8*vehicleSpeed) then
        if (not (skidding!i)) then
          after (sec 0) (contrs!i).startedSkidding
      else
        if (skidding!i) then
          after (sec 0) (contrs!i).stoppedSkidding
```

```
      msg <- after (millisec 4) fetch
      pendingMsg:= Just msg

   result reset

---------- file Controller.t ----------
module Controller where

import Env
import Wheel

data Mode    = Open | Closed
data State = Default | Relief Mode Msg | Isolate Mode Msg Int

default eqMode:: Eq Mode

struct Controller where
   resetController:: Request ()
   startedSkidding:: Action
   stoppedSkidding:: Action

controller wheelObj i_valve r_valve = class
   state:= Default

   resetController = request
      r_valve.close
      i_valve.open
      case state of
         Default ->
         Isolate _ msg _ ->
            abort msg
            state:= Default
         Relief _ msg ->
            abort msg
            state:= Default
      result ()

   startedSkidding = before (microsec 100) action
      case state of
         Default ->
         Isolate mode msg _ ->
            abort msg
            if (mode == Open) then
               i_valve.close
         Relief _ _ -> raise 1
      a <- wheelObj.getAcc
      if (a<0) then
         r_valve.open
         msg <- after (millisec (round (abs a)*10)) step
         state:= Relief Open msg
```

```
    else
      msg <- after (millisec 5) step
      state:= Relief Closed msg

stoppedSkidding = before (microsec 100) action
  case state of
    Default -> raise 1
    Isolate _ _ _ -> raise 1
    Relief mode msg ->
      abort msg
      if (mode == Open) then
        r_valve.close
      a <- wheelObj.getAcc
      i_valve.open
      msg <- after (millisec 100) step
      state:= Isolate Open msg 0

step = before (microsec 100) action
  case state of
    Default -> raise 1
    Isolate mode _ counter ->
        if (mode == Open) then
          i_valve.close
          msg <- after (millisec 20) step
          state:= Isolate Closed msg counter
        elsif (counter < 10) then
          i_valve.open
          a <- wheelObj.getAcc
          msg <- after (millisec (round (abs a)*50.0)) step
          state:= Isolate Open msg (counter+1)
        else
          i_valve.open
          state:= Default
    Relief mode _ ->
        if (mode == Open) then
          r_valve.close
          msg <- after (millisec 5) step
          state:= Relief Closed msg
        else
          a <- wheelObj.getAcc
          if (a<0) then
            r_valve.open
            msg <- after (millisec (round (abs a)*10)) step
            state:= Relief Open msg
          else
            msg <- after (millisec 5) step
            state:= Relief Closed msg

result Controller{..}
```