Personal Alarm Device: A Case Study in Component-Based Design of Embedded Real-Time Software

Jimmie Wiklander*, Andrey Kruglyak*, Johan Nordlander* and Per Lindgren*

*EISLAB, Luleå University of Technology, 971 87 Luleå
Email: jimmie.wiklander@me.com, kruglyak@mac.com, johan.nordlander@ltu.se, and per.lindgren@ltu.se

Abstract—Designing software for embedded systems is complicated by such factors as the tight integration between software and hardware, scarceness of available resources, and hard real-time requirements. In our earlier work we proposed a component-based approach based on modeling both hardware and software using concurrent reactive objects and time-constrained reactions, which should allow us to overcome these difficulties. We also presented a software design methodology for embedded real-time systems.

Here we describe a system developed using this methodology and discuss its advantages. The system is a personal alarm device that should be worn at the waist of a person and that should detect his or her fall and send an alarm signal. The implementation of the system was verified using a Simulink-based simulator. The simulation demonstrated that, even though calculation of acceleration was simplified to allow for an efficient execution on a resource-constrained platform, fall detection remained satisfactory.

The case study demonstrates the advantages of the proposed software design methodology, including the fact that functional and timing properties of a system model can be preserved during implementation process by means of a seamless transition between a model and an implementation.

I. INTRODUCTION

Embedded systems possess certain qualities that turn embedded software design into a delicate task. First of all, embedded systems typically have limited resources at their disposal (CPU, memory, power, etc.) that have to be utilized efficiently in order to meet system requirements. As a consequence, embedded systems often exhibit a tight integration between software and hardware. This makes it virtually impossible to model and efficiently implement software separately from hardware. Most embedded systems can also be classified as real-time systems. The traditionally used techniques do not have an inherent support to express and model temporal behavior, which makes designing software for such systems a challenging task. Moreover, the ongoing advances in the microprocessor technology leave room for more elaborate software implementations, adding complexity to the design, which requires more mature design methods than those in use today. In [1] we proposed a component-based approach to cope with these difficulties. We presented a modeling framework based on the notions of reactive objects and time-constrained reactions, which facilitates component-based design of embedded real-time systems. Within this framework, functionality of both software and hardware components is defined in terms of reactions to discrete events, and timing requirements are specified for each reaction relative to the event that triggered it. We also presented a detailed software design methodology for embedded real-time systems based on our modeling framework.

In this work we describe design of a real-life system developed using this methodology and discuss its advantages. The system in question is a personal alarm device (PAD) and its underlying fall detection concept is based on using an acceleration sensor. In our earlier work [1], this system was used as an example to demonstrate some stages of the design process. Here we present the system design process in full, including the choice of the the fall detection algorithm, system implementation and verification, omitted in [1]. A special focus is given to specification and implementation of timing requirements.

An overview of the software design methodology is given in Section II. It is followed by a description of the PAD, including the prescribed fall detection algorithm (Section III). The design of the PAD is discussed in Section IV, with the timing requirements discussed separately in Section V. Section VI deals with the implementation of the PAD in the Timber programming language, while Section VII describes verification of the design and implementation in a Simulink-based simulator. A discussion of related work follows in Section VIII, and Section IX concludes the presentation by discussing the advantages of the methodology as demonstrated by the presented case study.

II. METHODOLOGY OVERVIEW

Our software design methodology relies on a unified, consistent modeling of both hardware and software. The modeling framework is based on the notions of concurrent reactive objects [2] and time-constrained reactions [3]. A more detailed presentation of the methodology and the modeling framework can be found in [1].

A. Modeling Framework

The modeling framework supports describing the functionality of both hardware and software. Interaction between the system and its environment, as well as between components of the system is modeled as discrete events occurring at specific times. Following the reactive approach, the functionality is...
specified in terms of reactions to such events. Embedded systems must often conform to specific timing requirements which can be specified by defining the earliest and the latest reaction time (baseline and deadline) relative to the time of the input event triggering the reaction. The time window between the reaction baseline and its deadline is called a permissible execution window for this reaction (Fig. 1), and it is denoted as after \( t_{after} \) before \( t_{before} \) doSmth. Here \( t_{after} \) is the baseline offset (period of time between the triggering event and the baseline), \( t_{before} \) is the period of time between the baseline and the deadline, and doSmth is the invoked method of the object \( obj \). A reaction with a permissible execution window defined for it will be called a time-constrained reaction.

In order to model complex systems there is a need to model the system at various levels of abstraction. The modeling framework distinguishes the following abstraction levels: system level, component level (which can include multiple sublevels to accommodate a component hierarchy), and object level, as depicted in Fig. 2.

At the highest level (system level) the system is viewed as a black box and the interaction between the system and its environment is expressed as discrete events occurring at specific times. At the next level (component level) the system is partitioned into components and the interaction between components is again expressed as discrete events. Typically, a component encapsulates a part of system state and/or hardware resources and has a clearly defined interface and functionality. We extend this definition by requiring that functionality of each component is expressed in terms of time-constrained reactions to these events, which gives us what can be described as reactive components.

A component hierarchy allows us to structure the system, but ultimately, all system functionality has to be expressed in terms of reactive objects, the smallest building blocks in our model. So at the lowest level of abstraction, each component is modeled using reactive objects. A reactive object is defined by its interface (its methods), encapsulated state, and one or more implementations. A reactive object is a model that can have one or several hardware or software implementations, differing in their non-functional (but not timing) properties. In contrast to a component which may contain both software and hardware implementations of objects, a reactive object is implemented either in hardware, or in software (written in some programming language).

B. Software Design Methodology

The different stages of the design process are presented in Fig. 3. The input to the design process is the product specification which originates from the client commissioning the system. This specification is usually written in a natural language, is often incomplete and imprecise. Hence the first step is drawing up an extended specification with a clear division into functional and (optionally) non-functional parts. In our case, functional specification is integrated with timing requirements and is used throughout system modeling and implementation. Non-functional specification lists the remaining system properties and constraints, such as system size and power consumption, and is primarily used during verification.

The second step is formulating a system-level model where the interface between the system and its environment is defined in terms of external input events triggering time-constrained reactions and output events as a part of such reactions. In the third step, this model is elaborated by identifying system components and interfaces between them. In the fourth step, the components are realized using reactive objects, and a decision

---

Fig. 1. Permissible execution window for a reaction to an event.

Fig. 2. Three abstraction levels of modeling: system level, component level (including multiple sublevels to accommodate a component hierarchy), and object level. A system is realized in terms of components, and each component is realized in terms of objects

Fig. 3. Stages in the design process: from a specification to a ready product.
is made on which reactive objects should be implemented in software and which represent models of (existing) hardware. At every step, the model of each component is matched against a repository of previously developed components (either software or hardware), which should contain reactive models of components alongside their implementation.

An object to be realized by an underlying hardware implementation should be viewed as a singleton instance (reflecting that it is indeed just an interface to a single hardware instance). Such an object typically provides an interface consisting of synchronous read/write methods (i.e., I/O port operations) and can be seen as part of the software environment. This view allows us to restrict access to hardware resources (e.g. as being in scope of just the controlling software/driver.)

The fifth step is implementation of software objects in some programming language (which might require providing the infrastructure necessary for real-time execution on a hardware platform). This step might also involve building a hardware platform from identified hardware COTS components, SoC blocks, etc. depending on whether the platform was available in the beginning of the design process or not.

The sixth and final step is system verification, which can be done by simulation of the model and/or implementation, by testing of the implementation, by formal methods, etc. Both functional and non-functional requirements can be verified, and a failed verification forces a return to an earlier development step, making the development process iterative.

C. Maximizing Component Re-use: Introducing a Resource Platform

The component-based design methodology presented here describes how a system is modeled and designed when it is developed from scratch, possibly re-using individual components from previous designs. However, in reality the hardware platform (the assembly of hardware components) can often be given from the start, which should come as no surprise since the cost of its development can be substantial. This hardware platform may, apart from a CPU and memory, contain sensors, I/O and communication hardware and the like, which in our model will be parts of different components. Such hardware often has complicated and vendor-specific interfaces, so it makes sense to add a layer of software that would encapsulate the details of software-hardware interaction and present a streamlined interface to the main application. The resulting component containing a hardware resource and the interfacing software will be called a resource component.

The collection of resource components will form a resource platform, which in essence encapsulates the hardware platform and can be viewed as providing services (such as radio communication or sensor readings) to the application (Fig. 4). Note that apart from facilitation application development, this approach facilitates component re-use in more than one way. The resource platform with its clearly defined functionality and streamlined interface can be used with different applications, decreasing the cost of hardware platform development; moreover, it can be used as a shared resource for multiple applications running simultaneously. Parts of the resource platform can be upgraded or changed to satisfy non-functional requirements of the system (e.g., power consumption, device size) without changing its interface to the applications and thus without modifying the applications themselves.

We further extend the notion of a resource platform by allowing it to include pure software components that can be shared between applications running on the platform and that can be viewed as offering services to them, such as a database component. Such components will also be classified as resource components. Inherent in the notion of a resource component is its ability for offering services to multiple applications (or components). To facilitate component composition, clients of resource components should be able to operate unaware of each other. This can be achieved if the run-time system provides an exclusion mechanism where only one request can be handled at any particular time, or if the resource component itself holds a specialized internal queue of requests. This approach is similar to what is known as “platform-based design” [4]. Note, however, that we introduce the notion of a resource platform as a part of a wider component-based approach rather than in isolation.

III. PERSONAL ALARM DEVICE (PAD)

The Personal Alarm Device (PAD) should be worn by a person who might require assistance in the case of a fall; the aim of a PAD is to detect such falls and automatically send an alarm, as well as to enable the wearer to manually trigger an alarm by pressing a button on the device. In order to allow for the wearer’s mobility, the device should be battery-driven, and the alarms should be sent wirelessly to an alarm receiver covering a certain area.

A. Fall Detection: the Concept

Fall detection is based on using an acceleration sensor. Such a sensor would typically measure static acceleration (acceleration generated by the earth’s gravity), and dynamic acceleration (generated by sensor movements) in three perpendicular directions. The device contains an acceleration sensor, a radio, and a microprocessor. It can be fastened at the waist so that the vertical axis of the human body is aligned with a predetermined axis of the acceleration sensor. As a result, the posture of the body (standing up or lying down) can be determined by evaluating the body’s alignment relative to the gravitational field of the earth. The dynamic acceleration can be used to
analyze parameters typical for a fall, such as velocity towards the ground, fall-related impacts (when the body abruptly hits something), etc. Combining all or some of these parameters with posture evaluation makes it possible to detect falls.

B. Finding a Suitable Fall Detection Algorithm

The concept described above requires some kind of fall detection algorithm for analyzing acceleration and making decisions based on the results of analysis. However, formulating a suitable fall detection algorithm is not straightforward. In a previous study [5] different fall detection algorithms where evaluated using acceleration data recordings from intentional falls and ADL (activities of daily living). The data was recorded using a prototype device (Fig. 5) containing an internal acceleration sensor. In the study, the y-axis of the sensor was aligned with the vertical axis of the body. Besides evaluation of different fall detection algorithms, the study also had an aim to validate the data collection of the prototype device.

The algorithms were analyzed in a LabView environment using fall data collected from middle-aged test subjects. Data representing activities of daily living collected from middle-aged and older people were used as a reference. For each algorithm, sensitivity and specificity were calculated using Eqs. 1 and 2:

\[
sensitivity = \frac{TP}{TP + FN},
\]

\[
specificity = \frac{TN}{TN + FP},
\]

where TP = true positives (detected falls), FN = false negatives (undetected falls), FP = false positives (ADL samples giving false fall alarm), and TN = true negatives (ADL samples not giving fall alarm).

Threshold values for the algorithms were adjusted for optimal detection of falls with as few false alarms as possible; recordings of ADL were used as a reference. The best performing algorithm (referred to as Algorithm I in [5]) discriminated various types of falls from activities of daily living, with a sensitivity of 97.5% and a specificity of 100% using floating point simulations (presented in Table I in Section VII, where we compare the results of simulation from the previous study with the results of simulation of our implementation).

Realizing the aforementioned concept in a PAD requires implementing the fall detection algorithm on a suitable platform (such as the prototype device platform that may not support floating point operations). It is therefore important to test the effects of different representations of real values found in the mathematical definition of the algorithm (floating point, 32-bit, 16-bit, or 8-bit integers). Simulations of Algorithm I with 16-bit data format resulted in a preserved fall detection sensitivity and specificity compared to floating-point simulations. In contrast, data processing in 8-bit format resulted in a preserved fall detection sensitivity but only limited fall detection specificity. Based on the results from the study, Algorithm I can be seen as a viable choice when it comes to implementing the fall detection functionality on a 16-bit platform [5].

Description of the Algorithm: The fall detection algorithm is based on impact detection and posture evaluation. The acceleration is monitored periodically (with period \( t_{\text{period}} \)) which enables detection of impacts to the body (not necessarily fall-related). Detection of an impact triggers the posture to be evaluated after a specific time period. If the posture is categorized as lying, a fall has been detected.

Impacts are detected by calculating the sum vector

\[
SV = \sqrt{A_x^2 + A_y^2 + A_z^2},
\]

where \( A_x, A_y, \) and \( A_z \) represent the acceleration (dynamic and static) in the \( x-, y-, \) and \( z\)-direction of an acceleration sensor, respectively. If \( SV \) exceeds the experimentally established threshold, it is assumed that an impact has occurred. An impact event triggers posture evaluation to take place after a predetermined time period \( t_{\text{lag}} \). However, if an additional impact occurs within this time, posture evaluation is re-scheduled relative to the latest impact event. In posture evaluation the tilt of the vertical axis of the human body relative to the gravitational field of the earth is evaluated. A lying posture is presumed if the static acceleration of the vertical axis of the human body is less than or equal to the experimentally established threshold.

The timing constraints form a critical part of the algorithm, and our methodology allows to express and preserve them at all steps of the design and implementation process, as will be demonstrated below. This distinguishes our methodology from numerous other approaches to designing real-time systems.

IV. PAD DESIGN

Design of the PAD was performed in accordance with the methodology presented in [1] and summarized in Section II. The first stage is to define an extended system specification on basis of the product specification. For this particular system, the focus was on the functional requirements, which had to be expressed in terms of time-constrained reactions. For the application, two such reactions were identified: sending an assistance alarm within \( t_{\text{alarm}} \) msec after the push button has been pressed, and sending a fall alarm within \( t_{\text{alarm}} \) msec after a fall has been detected. Fall detection is based on the fall detection algorithm described in Section III, which brings with it additional (internal) timing requirements \( - t_{\text{period}} \) msec (the time between acceleration samplings) and \( t_{\text{lag}} \) msec (the time between impact detection and posture evaluation). This illustrates that timing requirements can either stem from the desired (observable) behavior or from the selected solution approach.
System-Level Model: The next stage in the design process is to define a model of the system. Modeling is performed successively at three different levels of abstraction. At the highest level (system level) the interaction between the system and its environment is defined in terms of input events and output events. Drawing from the intended functionality of the PAD (as described in Section III) it is possible to identify the different forms of interaction that the system should support. The overall PAD interaction was formulated in terms of two input events (reset and pressing of the button) and two output events (alarm transmission and acceleration reading). Note that an acceleration reading is initiated by the system, not its environment, thus it is considered an output event, even though the dataflow is from the environment to the system. The reaction to reset consists of initializing the system and starting up periodic reading of acceleration (with period time $t_{period}$); a reading may result in detection of a fall and then transmission of an alarm, which has its own deadline $t_{alarm}$. The reaction of pressing the button consists of sending another type of alarm with the same deadline $t_{alarm}$. The system-level model of the PAD is presented in Fig. 6.

Component-Level Model: The next modeling level is the component level. Analyzing the specification and the system-level model we conclude that the PAD application will need the following independent resources: an acceleration sensor, a message sender (containing a radio transceiver), and a push button. Their independence warrants creating three separate components, each of them including both hardware and software objects (Fig. 7). These components can be seen as resource components, and together they form the resource platform for the PAD application.

It now remains to partition the rest of the system – the application – into components. Here two independent activities can be identified: fall detection and assistance call handling, resulting in at least two separate components. At the same time, it is appropriate to decouple the fall detection algorithm from how the system should react to a detected fall. For our application, this involves creating a message and forwarding it to the message sender, which can be done by a separate component. The resulting component structure is presented in Fig. 7, showing the observable timing requirements of the application.

The fall detector component is activated periodically (with period time $t_{period}$) and it triggers sampling of acceleration by acceleration sensor component. The sampling itself is a complicated interaction with hardware conducted in several steps, each with its own timing requirements; it will be discussed in detail in the next section. This is hardware-specific and is encapsulated in the acceleration sensor component, that only presents one method in its interface ($sampleAcc$), one that can be used to initiate acceleration sampling procedure. Once the sampling has been completed, the acceleration sensor component triggers analysis of the collected data in the fall detector component, using a method provided to it during initialization ($consumeAcc$).

Detection of a fall in the fall detector component constitutes a new (internal) event, which leads to invocation of fall alarm sender ($sendAlarm$) and message sender ($sendMsg$) components. Both reactions have a common deadline of $t_{alarm} \text{ msec}$. The push button component is activated on the arrival of an interrupt from the button. The component takes care of filtering interrupts to eliminate the effect of “bouncing”, when a single press results in multiple interrupts delivered to the microcontroller. Reaction to a button pressing event involves the three components (push button, assistance alarm sender, and message sender) with a common deadline of $t_{alarm} \text{ msec}$.

Note that the message sender represents a clear example of a shared resource – it can be used by any of the independent tasks of (a) fall detection, and (b) handling an assistance call. As such, it will have to include either message queuing or some kind of arbitration to synchronize access to the resource transparently to the components that may want to use it simultaneously. It is also worth noting that the message sender is defined in such a way that its timing behavior is specified by the “client” components rather than locally.

Object-Level Model: The lowest level of abstraction in the modeling framework is the object level. At this level the internals of each component are modeled in terms of reactive
objects. Similarly to partitioning into subcomponents, the process of defining the reactive object model is performed on each component independently of its context. For each component, it is necessary to identify: hardware resources apart from CPU and memory; object state in terms of state variables; and object functionality in terms of methods. The object-level model should also contain additional information on which objects are implemented in software and which in hardware. The object-level model of the PAD is presented in Fig. 8.

Interaction between objects is described in terms of internal events, just as interaction between components. Interaction between software objects can be implemented using messages, whereas interaction between software and hardware objects is implemented as writing to hardware registers, invoking interrupt handlers, etc.

Let us now take a deeper look at some of the internal of the components that constitute the PAD resource platform. The function of the acceleration sensor component is to deliver acceleration readings in three perpendicular directions. The component is passive in the sense that it reacts to sampling commands issued by the application. Specific hardware is needed in order to acquire acceleration readings from the environment, such as an analog accelerometer and an A/D converter. In Fig. 8 the hardware objects are shown as shaded boxes (the parts of their interfaces only used during system initialization have been left out). We choose an A/D converter that operates sequentially, i.e. the converter can only sample one channel (or acceleration direction) at a time. Hence, before triggering a conversion the appropriate input channel must be selected. All A/D converter interaction, such as channel selection, is handled by the A/D controller software object. It is also responsible for delivering the results (in our case – to the fall detector component) after acquiring the three samples. As a result, all hardware-dependent timing requirements associated with A/D conversion must be handled by the A/D controller, which is described in the next section.

The function of the message sender component is to send messages over some media. Since the media in our case is radio it includes hardware in the form of a radio transceiver. The radio transceiver is controlled by a transceiver controller software object which encodes the message following a specific network protocol. In order to support transparent sharing of the message sender between multiple components the message sender component also includes two software objects that enable message queuing. The rather complicated timing constraints on the individual objects’ reactions within this component are not discussed here.

In the push button component, the single reaction deadline $t_{alarm}$ msec is defined. It is inherited by the assistance alarm sender and later on by the message sender components.

Let us not turn to the components comprising the PAD application. All of them are pure software components. The fall detector is the only one of these three components that consists of more than one reactive object. The objects are acceleration sampler, periodically triggering sampling of acceleration; acceleration analyzer, triggered by the acceleration sensor; and fall detection controller, which controls the fall detection procedure.

The fall detection procedure starts with the fall detection controller being invoked upon detection of an impact. Once invoked, the fall detection controller schedules posture evaluation to be carried out after $t_{tag}$ msec. When this time period has elapsed the fall detection controller reads the person’s posture from the acceleration analyzer. If he or she is lying down, the fall detection controller triggers the fall alarm sender component with the deadline $t_{alarm}$ msec (relative to the time when the fall has been detected). This deadline is inherited by the message sender component. The assistance alarm sender is triggered in the same manner by the push button component.
V. TIMING REQUIREMENTS FOR THE ACCELERATION SENSOR COMPONENT

Sampling of acceleration is initiated by invoking the sampleAcc method of the acceleration sensor component. Sampling acceleration involves multiple steps that are performed by the A/D controller object (implemented in software), which interacts with the A/D converter object implemented in hardware. The A/D converter is only capable of sampling acceleration on one channel at a time, so each conversion must be preceded by setting the channel (setChan) with a certain minimum delay between that and initiating conversion (convert). A completion of the conversion is signaled by raising an interrupt which is handled by the adIRQHandler method of the A/D controller object. In the model, this behavior can be encoded as follows:

```plaintext
adController setChan convert read accAnalyzer = class
  chan := 1 { -number of the channel to read from -}
  values := [] { -a list of acceleration values -}
  sampleAcc = before t_{setChan} action
    after (t_{setChan} + t_{wait}) initConversion
    setChan chan { -set the channel -}
    chan := chan + 1 { -update channel state -}
  initConversion = before t_{convert} action
    convert { -initiate conversion -}
  to be continued...
```

Here we are using the notation from the Timber language that reflects the concepts of time-constrained reactions and reactive objects that our model is built upon. In Timber, communication between software objects ("internal event" in our model) is accomplished using messages that can be sent synchronously (with the caller waiting for the callee to return) and asynchronously (allowing the callee to execute concurrently with the caller). An asynchronous message can be postponed by offsetting its baseline relative to the baseline of the caller, which corresponds to the semantics of our model.

In the code above, a "class" is a definition of an object constructor; an "action" is a method that is to be invoked asynchronously, the "before"-notation defines the deadline (relative to the baseline), and the "after"-notation shifts the baseline of the invoked method relative to the baseline of the method being executed. Note that posting the message to the method convert with a delay of \((t_{setChan} + t_{wait})\) ensures that at least \(t_{wait} \text{ msec}\) elapses between setting the channel and initiating conversion, as required by the hardware.

Once conversion has been completed, an interrupt is handled by the A/D controller object (and the time-stamp of the interrupt becomes the natural baseline for its adIRQHandler method):

```plaintext
...continued from above
adIRQHandler = before t_{adIRQ} action
  val ← read { -read the value -}
  values := val:values
  { -deliver the values when all three have been collected -}
  if chan == 4 then
    chan := 1
    { -send acceleration values for processing -}
    accAnalyzer.consumeAcc values
```

Here the special construct "after 0" resets the baseline of the invoked method to the time when the message was posted. The permissible execution windows for the methods of the A/D controller object are illustrated in Fig. 9.

VI. PAD IMPLEMENTATION

The aim of the implementation process is to realize the objects defined in the model (assuming that it is complete) in software and hardware, as defined by the object-level model. This involves implementing the software reactive objects in some programming language (which might require providing the infrastructure necessary for real-time execution on a hardware platform, e.g., an operating system supporting scheduling), and building a hardware platform from the identified hardware components. So far, only software part of the implementation has been completed, but we have checked that there exist hardware COTS (components-off-the-shelf) that correspond to each reactive object in the model that should be implemented in hardware.

The software must be implemented using a suitable programming language. We choose Timber [6]–[8] as programming language for the PAD software implementation which gives us many advantages compared to using standard programming languages. Timber fully supports the underlying modeling framework and therefore the reactive objects along with their timing constraints do not have to be translated into a separate programming model but can be described directly as Timber models, avoiding re-writing the model using different constructs and then verifying that the translation has preserved the essential properties of the system, such as timing properties.

A. Implementation in Timber

Timber allows to express the model in terms of Timber objects, communicating through asynchronous and synchronous messages. Permissible execution windows are inherited by default, but can in the asynchronous case be set explicitly.

Each reactive object in the model of the PAD was encoded as such in Timber. To illustrate this we now present Timber code for the reactive objects that comprise the fall detector component.

![Fig. 9. Timing requirements for A/D conversion described as permissible execution windows for the methods of the A/D controller object.](image-url)
In the acceleration sampler object, the method `sample` triggers acceleration sampling by invoking the method `sampleAcc` of the acceleration sensor component. It also posts an asynchronous message to itself with a baseline delayed by \( t_{\text{period}} \), which results in a periodic sampling.

```python
accSampler adController = class
  sample = before \( t_{\text{sample}(D)} \) action
    adController.sampleAcc
  after \( t_{\text{period}} \) Sample
```

Once the acceleration values have been collected, the acceleration sensor component invokes the `consumeAcc` method of the fall detector component, which is realized as the `analyze` method of the acceleration analyzer object. This object interacts with the fall detection controller by invoking its method `impact` on detection of an impact:

```python
accAnalyzer fallDetectionController = class
  \{ - state variable storing low-pass filtered acc. data - \}
  a := []
  analyze xyz = before \( t_{\text{analyze}(D)} \) action
    \{ - low-pass filter the y input (vertical acceleration) - \}
    if \{ - look for impact in xyz input - \} then
      fallDetectionController.impact
  getPosture = request
    if \{ - determine posture using filtered data - \} then
      result LyingDown
    else
      result Upright
```

The synchronous method (“request” in Timber) `getPosture` returns the current posture of the person determined from the vertical acceleration. The result returned is encoded as one of two values, either `LyingDown` or `Upright`. The partial Timber code of the fall detection controller is presented next:

```python
fallDetectionController accAnalyzer alarmSender = class
  impact = action
    after \( t_{\lag} \) evalPosture
  evalPosture = before \( t_{\text{eval}(D)} \) action
    posture \leftarrow accAnalyzer.getPosture
    if posture == LyingDown then
      alarmSender.sendAlarm
```

According to the algorithm there should be a time delay between detection of an impact and checking the posture. In the model this is encoded as posting an asynchronous message to the method `getPosture` with the baseline delayed by \( t_{\lag} \) relative to the current baseline. The algorithm also states that posture evaluation should be re-scheduled relative to the latest impact event if additional impacts are detected.

### B. Building an Executable

Before running the Timber model of the PAD as an executable on a target platform it needs to be compiled using the Timber compiler [8] and linked with the Timber run-time system for the target platform written in C [9]. The Timber run-time system provides scheduling of messages directly based on the timing constraints from the model preserved throughout implementation and encodes interrupts coming from hardware as messages to handler objects (in our case, the push button and the A/D controller objects). E.g., the timing requirements from the push button (originating at the component level), will be translated down to actual scheduling of the reaction chain triggered by the `handlebuttonIRQ`.

### C. Optimizing for a lightweight microcontroller

Fall detection was implemented in conformance with the algorithm described in Section III. The algorithm uses mathematical functions defined on the whole set of real numbers, so when implemented on a computer, especially a light-weight microcontroller with limited processing capabilities, it becomes an approximation. In fact, we have a trade-off between precision of calculations and hardware requirements, where a more advanced microprocessor would be more expensive and presumably consume more power. In the implementation integer-only 16-bit data and operations were used, and multiplication was replaced by using a look-up table. The result of such approximation has to be verified.

### VII. PAD Verification

A traditional verification method is to use simulation. Unlike formal verification methods, simulation cannot produce any guarantees on system behavior. At the same time, simulation as opposed to testing allows to significantly lower design times and costs, by allowing to verify certain aspects of the design at an early stage of development. However, the main question that has to be asked is whether results of a simulation of a model are still valid for the final implementation of the system. Another key question is how it is possible to simulate timing properties (essential properties of any real-time system) when there is no specific hardware platform to measure WCETs on. We will try to answer these questions before we describe the simulations of the PAD that have been conducted.

The first problem is addressed in our approach by using the same formalism for modeling and for implementation. Indeed, as the model of the software only contains reactive objects, and an implementation in the Timber programming language is also based on them, the transition from a model to an implementation is no longer a translation. It should instead be viewed as filling in the model with more details, such as the actual code of the methods that have already been defined. Thus the properties of the model verified in simulation become the properties of the system, unless simulation relies on some assumptions not expressed in the model itself.

In general, the same is true about the timing properties, as permissible execution windows are defined in the model and are preserved in the implementation to be used at run-time (by the Timber kernel) to guide scheduling. However, there is an important point to be made here. The permissible execution windows, defined for each reaction at system, component, and finally object level, are in fact timing specifications rather than timing properties; they define the multitude of allowed timing behaviors but do not in themselves guarantee that they will be adhered to at run-time. In conjunction with the Timber kernel, however, these timing specifications are guaranteed to be followed as long as the underlying hardware platform is fast enough to allow the software to meet all its deadlines. Thus simulation of the model (with timing specifications) aims
to verify the validity of such specifications with respect to intended system behavior, and the question whether a particular platform allows for all deadlines to be met has to be studied separately based on WCETs and the chosen scheduling algorithm.

Simulink-Based Timber Simulator: In the case of our system, the model is implemented in Timber, and the resulting code is compiled to C by the Timber compiler. This code can then be executed on a real hardware platform with an appropriate version of the Timber kernel; simulated on a PC under POSIX with Timber’s POSIX run-time system; or simulated in Simulink [10] together with a specialized version of the Timber kernel. The significance of being able to run literally the same code both in simulations and on the real hardware platform cannot be overestimated; it allows us to verify the system’s behavior and eliminate software bugs already in simulation.

Simulink is widely used in industry for modeling and simulation of various control systems, and numerous subject-specific simulators come with the ability to interface Simulink models. Being able to execute Timber code in Simulink allows us to easily create models of the environment for our embedded software. The specialized version of the Timber kernel provides communication between the Simulink world and the Timber world by emulating interrupts as input to the Timber system and output signals as output from it. It also schedules the execution of Timber objects in accordance with the defined permissible execution windows. Since the model does not represent the system running on any particular hardware platform, there are no execution times available to us and the virtual execution time of each reaction is set to zero. It is possible to schedule the execution of the reactions at any point within the permissible execution window; in our case, we chose to execute all reactions as early as possible, i.e. at their respective baselines. Note that simulation thus becomes an approximation of system behavior as the effect execution times on the actual scheduling is ignored, but the simulated behavior is one of those conforming to the system specification.

Simulations of the PAD: The goal of Simulink simulations of the PAD was to eliminate possible design flaws and software bugs in the model and its Timber implementation as well as to verify that the system will operate as intended under realistic conditions. To this end, data recordings from simulated falls as well as ADL (activities of daily living) were used as input to the simulations. Apart from the actual execution times and the resulting scheduling, we also ignore memory consumption.

It was not possible to utilize the whole set of recorded data because some of these recordings did not provide enough data for posture evaluation after an impact. In order to utilize as many recordings as possible, the parameter $t_{lag}$ (which determines the time interval between a detected impact and posture evaluation) was set to 0.9 sec. It was also necessary to exclude four recordings from the set of recorded ADL when the prototype device was accidentally misaligned.

The results of the simulations are shown in Table I, with fall detection sensitivity and specificity calculated using Eqs. 1 and 2. These results can be compared to floating-point simulations of the algorithm in LabView (presented in a previous study [5] and in Table I), conducted with the same recordings as input data. Note, however, that in the previous study absence of sufficient data after impact in certain recordings of falls was compensated for by using supplementary evaluation methods, and the recordings of ADL with misaligned prototype device were used after correcting the values of acceleration to counter the effect of misalignment. The necessary exclusion of these recordings from our simulations accounts for the differences in the number of simulations.

According to the results of the simulation, the PAD implementation preserves the sensitivity and specificity exhibited by the fall detection algorithm in previous simulations.

VIII. RELATED WORK

The Rubus component model [11] shares many of the goals and implementation approaches with our methodology. It provides constructs for encapsulating software functions (called software circuits) and can be used to express interaction between them in single- and multi-node systems in terms of control flow as well as data flow. However, the software model has to be translated into executable threads, and the execution is controlled by a specialized run-time system such as RubusRTOS [12].

Another approach is Time-Triggered Architecture [13] which requires that each component is fully specified, including in the time domain, and can thus be verified separately from the rest of the system. Originally targeting distributed systems, this approach can easily be applied to componentization of a single-node system provided that components either do not share any resources, or utilize them according to a statically pre-defined schedule (including a shared CPU). This approach is very robust and can be used for safety-critical systems, but robustness comes at the cost of flexibility of the design and leads in most cases to a below-optimal utilization of resources.

Apart from well-developed approaches with well-defined semantics such as Rubus and TTA, there exist a number of other

<table>
<thead>
<tr>
<th>No. of rec.</th>
<th>Specificity (S) / Sensitivity (S)</th>
<th>No. of rec.</th>
<th>Specificity (S) / Sensitivity (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
<td>100.0% *</td>
<td>164</td>
<td>100.0% *</td>
</tr>
<tr>
<td>38</td>
<td>100.0% **</td>
<td>40</td>
<td>100.0% **</td>
</tr>
<tr>
<td>37</td>
<td>100.0% **</td>
<td>40</td>
<td>100.0% **</td>
</tr>
<tr>
<td>39</td>
<td>97.4% **</td>
<td>40</td>
<td>95.0% **</td>
</tr>
<tr>
<td>39</td>
<td>100.0% **</td>
<td>40</td>
<td>100.0% **</td>
</tr>
<tr>
<td>40</td>
<td>100.0% **</td>
<td>40</td>
<td>100.0% **</td>
</tr>
<tr>
<td>233</td>
<td>99.1% **</td>
<td>240</td>
<td>97.5% **</td>
</tr>
</tbody>
</table>

TABLE I SIMULATION RESULTS.
modeling frameworks and design tools that can be used for component-based development of real-time systems. Here we will only mention real-time synchronous languages (Esterel, SCADe, Lustre, etc. [14]); time-triggered languages such as Giotto [15]; the Ptolemy framework for assembly of concurrent components [16], which is particularly suitable for modeling distributed systems; and tools such as Rhapsody [17], Artisan Studio [18], and Rose-RT [19].

We should also mention work on specification of real-time systems, such as RT-UML [20] and MARTE [21]. In contrast to our work, these approaches offer numerous, often highly specialized ways to define timing properties, which are difficult to preserve (with consistent semantics) throughout the design process.

IX. Conclusion

In this work we have presented a personal alarm device developed using the component-based design methodology presented in [1] and summarized in Section 2 of this paper. The Timber implementation of the system was verified using a Simulink-based simulator. During simulation, the system operated according to its specification. The simulation also demonstrated that, even though the calculations were simplified in order to execute efficiently on a 16-bit platform, the ability to detect falls remained satisfactory. This makes it possible to utilize a lightweight microcontroller which in turn implies a lower power consumption, a smaller physical size of the device, and a lower price. These qualities are very important for portable systems and therefore they are also important for the future deployment of the system.

The case study demonstrates how our methodology allows us to model complex interaction between hardware and software, facilitating design of embedded systems. We have also demonstrated that complex timing behavior can be not only modeled but also preserved in the implementation.

Our component-based approach, used to partition this particular real-time system into reactive components, allowed us to specify and model system functionality as well as timing behavior at different abstraction levels. Thus the case study demonstrates the potential of our methodology to bring the benefits of classical component-based design (re-use of components, ease of system maintenance and modification, etc.) to the realm of embedded real-time systems.

Using Timber for implementation of the system allowed us to preserve the properties of the model by means of a seamless transition from model-based design to implementation. It also allowed us to simulate the actual implementation, when the same code can be used for simulation in a Simulink environment and on a target platform.

Future work includes an IDE supporting the suggested software design methodology, with repository management and ties to code synthesis and validation tools.

Acknowledgment

This work was supported in part by the Knowledge Foundation in Sweden under a research grant for the SAVE-IT project, the EU Interreg IV A North Programme (grant no. 304-15591-08), the ESIS project (European Regional Development Fund, grant no. 41732), and the EU AESOP project.

References