4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER4),
30.-31. October, Heinz-Nixdorf-MuseumsForum Paderborn, Germany, 63-67.

Use Case Modeling for Embedded Software Systems - Deficiencies & Workarounds

Alexander Nyßen, Horst Lichter
Research Group Software Construction, RWTH Aachen University
{any|lichter}@cs.rwth-aachen.de

Abstract

While applying use case modeling in the domain of embedded software systems, we observed some weaknesses, related to the very special characteristics of embedded software, discriminating it from those large-scale industrial applications, use case modeling was initially developed for and where this technique still has its greatest acceptance.

In this paper we discuss some of the most severe deficiencies we observed, namely the lack of modeling capabilities to deal with timing and concurrency constraints, as well as difficulties to handle stacked interfaces, an aspect embedded software systems often have to face.

Where possible, we provide solutions to overcome the shortcomings mentioned, otherwise we try to at least propose some workarounds to handle them.

1. Introduction

Having originated in the late 1980's as a requirements engineering technique to develop large-scale industrial applications [1], use case modeling has very soon gained broad acceptance in various fields of software development. At the end of the 1990's it has finally also entered the domain of embedded software systems, as a number of object-oriented modeling approaches targeting the embedded systems domain have been established during that time (e.g. [2][3]), all employing use case modeling to capture functional requirements.

Although – as is often stated - the domain of embedded software systems does have its very own characteristics and challenges, the basic concepts of use case modeling were not adapted and customized but are indeed used in the same manner as for large-scale industrial applications. However, while the concepts remained the same, if one looks at them thoroughly, use case models for embedded software systems tend to look a bit different.

2. Observations

Very often use case models of embedded software systems contain timer actors [3] or – having the same expressiveness - cyclic use cases [4] to model that use cases are executed on a periodical basis, which is rather typical to the embedded domain. Another aspect, which is as well rather special, is the use of timing marks to model timing constraints, e.g. proposed in [2]. In general, it can be observed that capturing just the functional requirements – the sole purpose use case models are applied for in the industrial software context – is not sufficient in the embedded domain. Since especially non-functional timing constraints have a severe impact on the later system design it is necessary to capture those constraints explicitly as well.

Use case modeling, as introduced for large-scale industrial applications and defined by the UML [5], does – from a conceptual viewpoint - not offer adequate means for that. The UML Profile for Performance, Schedulability and Time [6] does also not address those issues, as it does indeed not affect the respective Use Case language unit.

Another phenomenon that can as well be observed in use case models of embedded software systems is that not only human users, but also external hardware devices or software systems, located in the environment of the embedded software system, are represented as actors. As human users most often do not directly interact with an embedded software system via standard devices like mouse or keyboard but via special hardware devices, those hardware devices are the direct communication partners of the embedded software system. Hence, those external hardware devices rather than the human users are represented as primary actors in the use case model. The human users are most often simply omitted. The reason why this cannot be modeled adequately so far is that there are no means to express that a software system has to deal with related interfaces on different levels of abstraction, and that therefore often only the most

relevant interface is chosen to be represented as an actor. Besides, this does not only hold for human users interacting with the system via special hardware devices. It does as well hold when communication to an external software system is established via a hardware interface that has to be controlled by the embedded system.

3. Deficiencies & Workarounds

We take both observations as an indication that use case modeling - at least as it is applied in practice nowadays - is not quite capable to meet the special characteristics of embedded software systems. We think that those briefly sketched deficiencies, namely the lack of expressiveness to deal with timing and concurrency concerns, as well as the modeling difficulties related to interfaces are two main reasons for this. We will elaborate this further in the following and will provide solutions – were possible – that we developed when defining the MeDUSA method for small embedded software systems [7].

3.1 Timing and Concurrency Constraints

As it originated from the domain of large scale industrial software systems, use case modeling does not deal with expressing timing or concurrency constraints. Indeed, use cases are modeled from a mere functional perspective, where statements about the timing and concurrency of their execution, points of synchronization, or other timing constraints like deadlines or latencies are not explicitly addressed.

Indeed, when looking at them thoroughly, the only timing information given in a use case model is the starting point of each use case, described by the trigger of the primary actor that starts the execution (either directly or indirectly by triggering the execution of another use case, which includes or is being extended by the respective use case). As there is a fundamental need to express the property of a use case to be executed periodically, it is therefore a natural approach to introduce timer actors [3], representing sources of timing events, which trigger the execution of a use case in a periodic manner. However, what remains is a lack of means to express details about the concurrent execution of use cases, as well as their temporal synchronization.

Another inelegance that we see is that in case of aperiodic events – in contrast to periodic ones – the source the event originates from, is mixed with the interface that represents the communication interface towards the embedded software system, while in case of periodic events, the event source is separately modeled in terms of a timer actor. As an example for this, consider an A/D converter, which delivers analog sensor data to the embedded software system in a digitized form. In case the A/D converter is a passive device that has to be polled by the embedded software system on a regular basis, a timer actor would be modeled to represent the source of timing events, triggering the execution of the use case, together with an interface actor representing the communication channel towards the A/D converter. If the A/D converter is an active device, notifying the embedded software system with a hardware interrupt about the availability of new raw data, the interrupt source as well as the communication channel would however be represented by just a single actor, thus mixing both aspects.

As we think that the explicit modeling of timing and concurrency constraints improves readability and understandability and raises the awareness about the non-functional requirements imposed on the application, we strongly propose to explicitly separate out all timing and concurrency concerns by introducing *eventer* actors to represent sources of aperiodic events, analogously to the already known timer actors. Together with the resulting *interface* actors, that represent mere communication interfaces (without any timing aspects) this leads to the taxonomy of actors depicted in Figure 1. Note that interface actors are further divided into *device* actors, representing external hardware devices, and *protocol* actors, representing external software systems, and that human users are not included. We will come back to this in the next section.

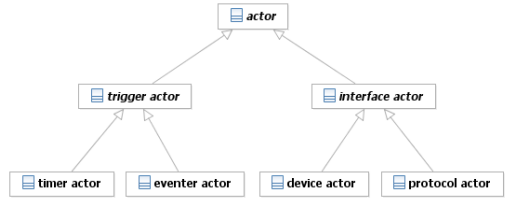

Figure 1: Actor taxonomy (compare [7])

Modeling timing constraints with the help of trigger actors, periodic and aperiodic events can be handled accordingly, thus removing the inelegance mentioned before. The A/D converter example can thus be modeled as demonstrated by Figure 2, using an internal timer actor in case of a passive A/D converter, or an external eventer actor in case of an active A/D converter.

One might argue that with the clear separation of event source and interface into two distinct actors, the information that they indeed represent one and the same real-world device (in our example the

ADC) gets lost. We partly agree to this. Of course from the structural relationships contained in the model this information cannot be inferred. However, instead of introducing e.g. an association between those related actors, we prefer to leave it out, as such relationships would most likely clutter the model and because the practical application of the presented workaround showed that the relationship between trigger and interface actors can be sufficiently expressed by appropriate naming of the actors.

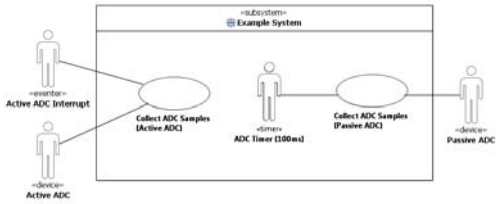

Figure 2: Use case model containing trigger actors

A better solution to this problem would probably be to introduce some sort of *composite actor*. This is however not expressible in terms of UML and from our viewpoint also not properly implementable by means of the offered UML extension mechanisms. Therefore we do not regard it to be an adequate solution, either.

One might propose that the issues of concurrently invocating a use case from another as well as synchronizing concurrently executed use cases could as well be expressed with the help of trigger actors, namely by associating use cases to (internal) eventer actors that represent the invocation or synchronization event, rather than relating the use cases to each other with the help of include and extend relationships.

As an example for this consider the one presented in Figure 3. Here, an eventer actor representing the raising of an alarm is introduced, as well as a use case associated to it, responsible of handling the alarm.

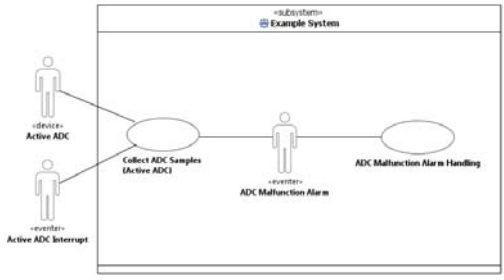

Figure 3: Use case model with an internal eventer actor

However, this is not an adequate solution, as the concept of actors in this scenario would be misused to represent a relationship that is indeed established conceptually between two use cases and not between use cases and actors. We therefore regard synchronizing of concurrent use cases to be an open issue that has to be investigated further.

3.2 Interfaces on different levels of abstraction

Another situation that one faces in the context of embedded software systems is that the software has to deal with related interfaces on different levels of abstractions.

Besides the already mentioned example of a human user interacting with the application via a non standard hardware device, one could think as well of an external software system, connected to the application via a hardware communication interface, such as a UART or a SPI, which is controled not by an underlying operating system, but by the embedded software system itself. In such a case, the embedded software system has to deal with interfaces on different levels of abstraction, namely the UART or SPI hardware communication interface, as well as the software protocol that is exchanged over that interface. As we already pointed out, use case modeling – as specified by the UML - does not offer adequate means to represent such a situation.

There are several reasons for that. The most obvious one is that the relationships between actors are inspired from a kind of *access rights perspective*, so that only generalization relationships are modeled between them. Actors representing related interfaces on different levels of abstraction can thereby not be adequately related to each other. Further, the defined relationships between use cases are motivated from a kind of *reuse perspective*, so that there is no adequate possibility to specify that two use cases are related to each other, other than that the functionality of one use case is included within the functionality of the other (include and extend), or that it is a specialization (generalization).

A possiblity that might however be investigated, is the use of dependency relationships (respectively use depencency relationships), which are not directly meant to be used in use case diagrams, but which might be used as the dependency relationship is defined between classifiers, and use cases and actors are both classifiers. Figure 4 shows how the above described scenario could thereby be represented.

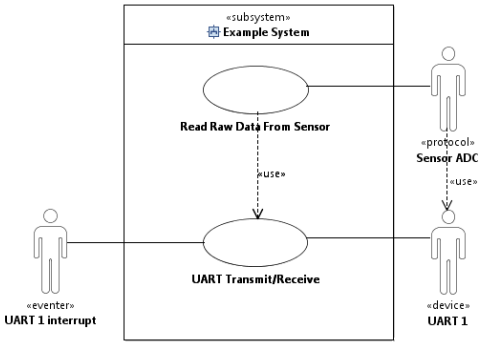

Figure 4: Modeling stacked interfaces by means of dependencies

The only problem of this approach is that no precise statement about the actual relationship between the two use cases can be inferred. Indeed, the dependency relationship can be seen as the most informal kind of relationship that is defined by the UML.

Further, statements about concurrency of the respective use case executions or about their synchronization needs cannot be inferred. To denote that, means to specify concurrent execution of use cases, as well as the synchronization of concurrently executed use cases would be needed. As already stated in the preceding section, use case modeling does not offer any means for that.

For reasons we have discussed at the end of the preceding section the use of an internal *eventer* actor to represent the synchronization event, which would lead to a model similar to the one denoted in Figure 5, is no valid solution, because the actor is misused to represent a relationship that should actually be established directly between the use cases.

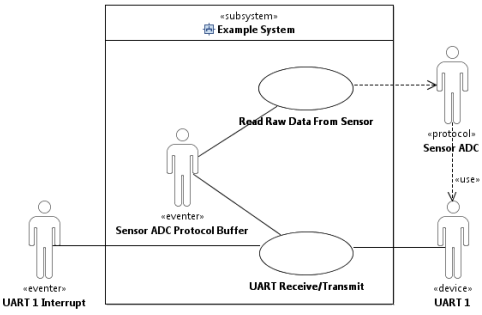

Figure 5: Using an eventer actor to synchronize concurrently executing use cases

As there is no concise solution to deal with such a situation, we propose to apply the one sketched in Figure 4, namely the use of dependency relationships, or – where this is not appropriate, e.g. because the modeling tool does not support such relationships in use case diagrams - to only concentrate on the interface on the lowest relevant level of abstraction.

In our example we would therefore just represent the underlying hardware interface as an actor and drop the representation of the related software protocol interface, as there the embedded software system has to control the underlying hardware communication interface by itself.

We may have to point out that the embedded systems we have analyzed are rather small devices, built-up from special purpose hardware and having no real-time operating system. Therefore the embedded software is directly responsible of driving the hardware, so that the hardware interface can be regarded as the interface on the lowest relevant level of abstraction.

In case of embedded systems having other characteristics it might be very reasonable to concentrate on higher levels of abstraction, if for example standard hardware devices are used or if a real-time operating system takes the responsibility of driving the hardware. In such a case a protocol actor representing the interface to the operating system or even a human user actor could be concentrated on instead.

We indeed tend to not represent human users as actors. The reason for this is that they normally play a subordinate role in case of an embedded software system, as they nearly never directly communicate with the system, but most often via software or non-standard hardware interfaces residing in the environment of the embedded system. Therefore, even if not concentrating on the interface on the lowest relevant level of abstraction, but when applying a solution using dependency relationships, we would propose to normally not model human user actors.

In case a human user communicates with an embedded system via standard hardware as sketched above, and those hardware is controlled by an underlying operation system, it might however be reasonable to represent the human actor rather than the standard hardware or the operating system as the human user is in this situation the interface on the lowest relevant level of abstraction.

Anyhow, such a software system does probably show - even if being embedded - more characteristics of an industrial information system than of a typical embedded software system (at least as we sketch it here), so that use case modeling as currently offered by the UML may be quite appropriate do deal with it.

For the type of embedded systems that we face, we propose to omit human actors, as the decrease in clarity of the resulting diagram would outweigh the

benefit of representing them. This is also the reason why the actor taxonomy denoted in Figure 1 does not represent any human user actors.

Nevertheless, if a concise solution to deal with modeling interfaces on all levels of abstraction appropriately could be found, human user actors should of course be represented.

4. Conclusion & Outlook

Although use case modeling has gained broad aceptance in the field of embedded systems in the last years, quite a number of deficiencies can be observed concerning the special characteristics of embedded software systems.

As described, timing constraints - something that is up to now not regarded thoroughly - can be quite well expressed by separating them out by means of *trigger* actors. Whereas *timer* actors are a concept already introduced by former approaches [3], we propose to introduce *eventer* actors, which we define to represent asynchronous event sources accordingly.

What remains an open issue in this context is that there are no adequate modeling means to deal with synchronization of concurrently executing use cases. As we pointed out, this is as well a problem, when trying to model interfaces on different levels of abstractions, as that almost always leads to concurrent behavior inside the system and thereby creates the need for synchronization as well.

Having those deficiencies and open issues in mind, we proposed some workarounds that are in line with the UML specification, but which all have – as we pointed out - their respective advantages and disadvantages. Therefore we see the need for further research on this topic in the future.

References

[1] Ivar Jacobson: Object Oriented Development in an Industrial Environment. In OOPSLA ’87 Proceedings. pp 183-191. 1987.

[2] Bruce Powel Douglass: Doing Hard Time – Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. Addison Wesley - Object Technology Series. 1999.

[3] Hassan Gomaa: Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison Wesley - Object Technology Series. 2000.

[4] Tim Weilkiens: Systems Engineering mit SysML/UML – Modellierung, Analyse, Design. dpunkt.verlag. 2006.

[5] OMG: UML Superstructure v2.1.1, OMG Document 07-02-05, 2007.

[6] OMG: UML Profile for Schedulability, Performance, and Time v1.1, OMG Document 05-01-02, 2005.

[7] Alexander Nyßen, Horst Lichter: MeDUSA- MethoD for UML2-based Design of Embedded Software Applications. Aachener Informatik Berichte. AIB-2007-07. ISSN 0935–3232. May 2007.