Separating Active and On-Demand Behavior of Embedded Systems into Aspects [⋆]

Aleksandra Tešanović[1], Thomas Gustafsson[1], and Jörgen Hansson[2,1]

[1] Linköping University, Department of Computer Science, Linköping, Sweden
{alete,thogu,jorha}@ida.liu.se
[2] Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA
hansson@sei.cmu.edu

Abstract. Requirements on embedded systems have increased over the years leading to an increased complexity of software and, consequently, higher development and maintenance costs. One major part of the cost is the way data is engineered and maintained in the system so that resources, such as CPU, are used efficiently. There exist various ways of optimizing CPU usage by controlling how often calculations on data are performed. These include (i) algorithms that ensure updating of data on-demand, i.e., when data becomes outdated, and/or (ii) appropriate event-condition-action rules implementing active behavior in the system and, thereby ensuring that calculations, specified by actions, are carried out when an event satisfying the condition occurs. Adding active behavior ensures that time-triggered embedded systems can also respond to events that occur aperiodically.

One way of adding on-demand and active behavior to the system is by re-engineering the existing software. This is a time-consuming and costly solution that still does not provide efficient means for maintaining and evolving the system. Alternatively, if software is developed from scratch, the behavior can be integrated into the system structure during design and implementation phase, resulting in a fixed and monolithic architecture that cannot easily be changed as the system evolves. In this paper we propose a way of designing and implementing on-demand and active behavior in embedded software, where this behavior is specified and developed independently of an application, and then integrated with the software. We employ aspect-orientation as the means of realizing the independent design and development of the on-demand and active behavior. We also demonstrate that encapsulating on-demand and active behavior into aspects results in efficient development and evolution of data management in embedded software with respect to complexity and cost.

[⋆] This work is supported by the Swedish Foundation for Strategic Research (SSF), the Swedish National Graduate School in Computer Science (CUGS), the Information Systems for Industrial Control and Supervision (ISIS), and the Center for Industrial Information Technology (CENIIT) under contract 01.07.

1 Introduction

Embedded systems constitute a majority of computing systems today and can be found in a variety of applications, from domestic appliances to engine control. They are characterized by sparse memory capacity and requirements on efficient CPU utilization. Furthermore, the majority of calculations in such systems are required to be finished before a given deadline to avoid performance degradation. For example, the engine of modern cars is controlled by software that continuously monitors and controls the engine using actuators. The control computations, e.g., fuel injection, must be finished within a certain deadline, otherwise the performance of the engine decreases, and in the worst-case the engine breaks down.

Requirements, functional and non-functional, on embedded systems have increased over the years leading to a crescent complexity of software and, thereby, resulting in higher development and maintenance costs [1, 2]. One major part of the cost is the way data is maintained in the system. The requirements and ways of handling data vary depending on the target application needs. Often it is necessary to fine-tune data management of an embedded system by incorporating various methods and algorithms that ensure freshness of data, i.e., ensure that data is up to date [3–5]. The algorithms are not a part of the original functionality of the system, rather they can be added to the system when the need to optimize the overall available CPU occurs.

Typically one optimizes CPU usage by adding so-called on-demand algorithms that dynamically adjust the update frequencies of data based on how rapidly sensor values change in the environment. On-demand behavior ensures that data is updated only when it is no longer fresh [4] (this is in contrast to approaches [6–9] where sensor values are always updated when data is assumed to become obsolete at the end of a constant time period, representing the worst case). As the system evolves, more elaborate schemes for data management might be applicable [10]. Namely, adding active behavior to the system in terms of event-condition-action rules is essential to embedded applications that are time-triggered but require actions on data to be performed even when a certain event occurs, e.g., when the engine overheats appropriate updates on data items should be triggered to ensure cooling down. Each rule in the event-condition-action paradigm reacts to certain events, evaluates a condition, and based on the truth value of the condition, might carry out an action [11]. As active behavior is often dependent on on-demand behavior, it is also important to enable incremental evolution of the system. Also, since embedded systems are evolving rapidly, the number of data items is also increasing, requiring an efficient way of evolving the mentioned behavior in the system.

When the CPU usage needs to be optimized by means of adding on-demand or active behavior, one way is to re-engineer a system. This is a time-consuming and costly solution that does not provide efficient means for maintaining and evolving the system. Alternatively, if software is developed from scratch, the behavior can be integrated into the system structure during design and implementation phase, resulting in a fixed and monolithic architecture that cannot easily be changed as the system evolves. Hence, an efficient way of designing, implementing, and integrating different behaviors that enable fine-tuning of data management and consequently optimizes CPU usage is needed.

In this paper we propose a way of handling design and implementation of on-demand and active behavior, where this behavior is designed and implemented independently of an application and then integrated with software for fine-tuning the CPU usage. As we show, the crosscutting nature of the on-demand and active behavior is a good candidate for the use of aspect-oriented programming, which allows us to encapsulate these behaviors into aspects. We also demonstrate that encapsulating on-demand and active behavior into aspects results in more efficient development and evolution of data management in embedded software.

The paper is outlined as follows. In section 2 we give background information about embedded software systems and aspect-oriented programming. Section 3 presents the design and development of on-demand and active behavior using aspects. Finally, in section 4 we give the main conclusions.

2 Background

In this section we present main characteristics of embedded software and give brief introduction to aspect-oriented programming.

2.1 Characteristics of Embedded Systems

Here we give an overview of a software controlling a vehicle, which is our target application. A vehicle control system consists of several electronic control units (ECUs) connected through a CAN communication link [12, 4]. The number of ECUs can vary depending on the way functionality is divided between ECUs for a particular type of the vehicle. One typical example of an ECU is the engine ECU (EECU). The memory of the EECU is limited to 64kb RAM and 512kb Flash. The 32-bit CPU runs at 16.67MHz. The EECU is responsible for controlling the engine. For example, the EECU controls the fuel injection and spark plug ignition in order to have an optimal air-fuel mixture for the catalyst, and to avoid knocking of the engine (knocking is a state where the air-fuel mixture is misfired and can destroy the engine). Data that the system reacts upon, e.g., temperature and air pressure, is collected from the engine environment in fixed and pre-defined periods of time and are spread out in different modules over the system. All data items in the software are known before the system start and they often have requirements on freshness. Data is considered to be fresh if a value of a data item is a correct reflection of the state of the external environment.

2.2 Aspect-Oriented Programming

Aspect-oriented programming allows separating and encapsulating system's crosscutting concerns into modules, called aspects [13]. The main constituents of aspect-oriented programs are: (i) components, written in a component language, e.g., C, C++, and Java; (ii) aspects, written in a corresponding aspect language, e.g., AspectC [14], AspectC++ [15], and AspectJ [16]; and (iii) an aspect weaver, which combines aspects with components.

An aspect in an aspect language consists of pointcuts and advices. A *pointcut* consists of one or more join points, and it is described by a pointcut expression. A *join point* refers to a point in the component code where aspects could be woven, e.g., a method, a type (struct or union). An *advice* is a declaration used to specify the code that should run when the join points, specified by a pointcut expression, are reached. Different kinds of advices can be declared, such as: (i) *before advice* code is executed before the join point, (ii) *after advice* code is executed immediately after the join point, and (iii) *around advice* code is executed in place of the join point. A special type of an advice, called *inter-type advice* or an introduction can be used for adding members to the structures or classes.

3 Designing and Implementing Active and On-demand Behavior for Evolution and Reuse

In this section, after reviewing what is needed (data structures and rules) for active and on-demand behavior, we present the way on-demand and active methods can be designed and implemented using aspects. We also discuss concrete implementation issues and identify challenges.

3.1 On-Demand and Active Behavior

In embedded systems that monitor a natural environment data items managed by the system can be divided into (i) *base data items*, which are read from sensors or communication links, and (ii) *derived data items*, which are calculated from a set consisting of base items and derived data items.

Since the values of one calculation depend on values derived by other calculations, calculations have precedence constraints that are described by a directed acyclic graph (DAG), denoted *data dependency graph* (see figure 1). Base items have zero in-degree, and nodes representing actuator values have zero out-degree. Furthermore, if a calculation derives a value of only one data item then there exists a simple mapping from precedence constraints in a DAG to calculations of data items.

The majority of embedded systems is time-triggered [17], implying that all calculations on data in the system are done in pre-defined periods of time (as elaborated in section 2.1). This in turn means that memory and CPU resources might be ineffectively used due to unnecessary updating of data items. Using the example from figure 1, periodical updates of b_1 always result in periodical updates of d_1, d_5, and d_6, regardless of the data condition, i.e., whether data is up-to date or not. To ensure that derived data items are updated only when they are stale, on-demand approaches to data updating can be used.

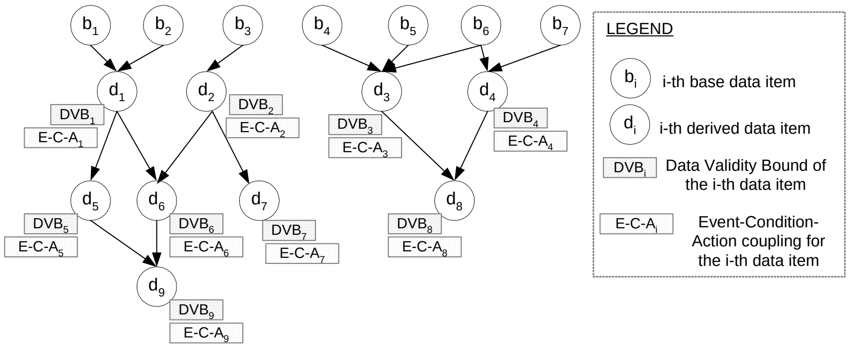


Fig. 1. Data in an embedded software system

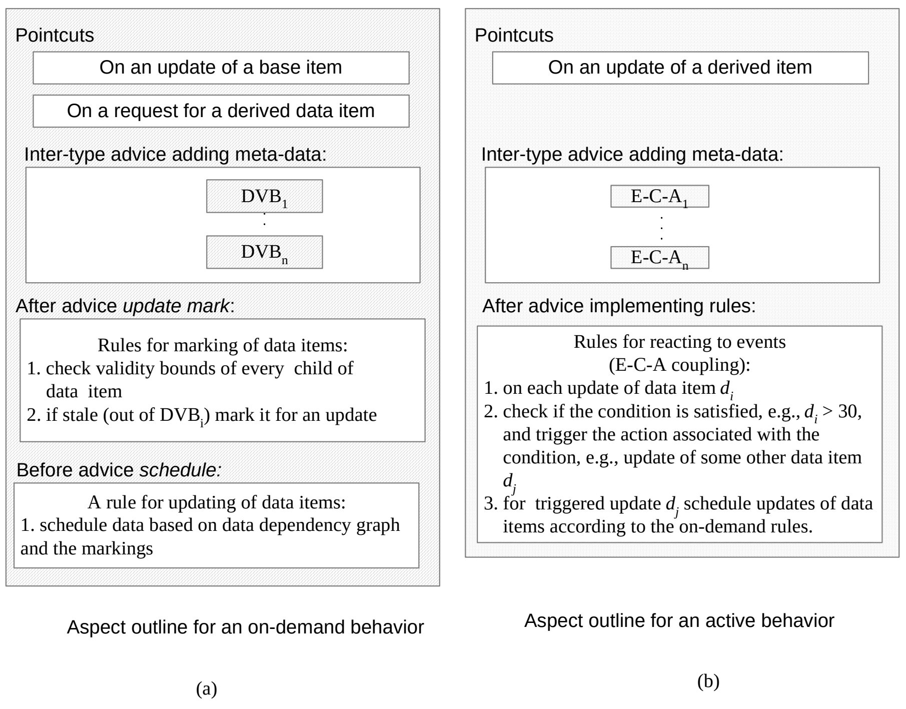
In these approaches data freshness is implemented by associating every derived data item with the *data validity bound (DVB)* (see figure 1), which specifies the allowed distance between two values of the same data. For example, associating data item d_3 with the bound DVB_3=9 implies that the data is fresh and should not be updated as long as the difference between the old and the new value is less then 9. Now base data items are updated in fixed periods, and at the time of the update they are checked for freshness [4]. If a base data item is stale, i.e., its value is outside its DVB, the base data items' immediate children are marked for an update. These markings also apply for derived data items. Thus, when a derived data item is updated its freshness is checked, and if it is stale, the immediate children are marked.

When a need for an update of a derived data item occurs, e.g., when a transaction in the system starts executing, then an algorithm (e.g., ODTB [4]) schedules data items that need to be updated based on their markings.

If the data dependency graph in the system is complex with inter-dependencies among data such that there is no path from one data item to another, then on-demand algorithms are not enough. Consider, for example, that the data item d_9 in figure 1 is the temperature of the engine, and data item d_8 contains values to be displayed on the driver's display. If the engine temperature reaches a certain threshold, then the driver should be warned. That is, an update on data item d_9 might trigger an update on d_8 if the threshold condition is satisfied. In this case we need to incorporate event-condition-action rules into data management. Namely, each data item should be associated with an event (e.g., on performed calculations when data value changes), and the condition that triggers an action, i.e., some other calculation on another data item. Hence, each data item should be augmented with meta-data information containing the condition and an action (here we assume that an event is the updating of data item).

3.2 On-demand and Active Aspects

The way on-demand behavior can be implemented using an aspect is shown in figure 2(a). As can be seen, the meta-data of data items (e.g., DVBs) is now added via inter-type advices that add members to structures describing data dependencies. The pointcuts identify (i) when updates on base data items occur and (ii) when the arrival of a request for calculation on a derived data item occurs, i.e., when a transaction arrives to the system. Upon reaching the first pointcut the rules from the advice of type after, which mark the data item as stale, are executed. When the request for the derived data item d_i arrives, i.e., the second pointcut is reached, the second advice of type before is executed. That is, before d_i is updated, all data upon which the value of d_i depends, and that is marked as stale, is inserted into the schedule. The schedule is then executed by the system by resuming normal operation.



Pointcuts
On an update of a base item
On a request for a derived data item
Inter-type advice adding meta-data:
DVB$_1$
DVB$_n$
After advice *update mark*:
Rules for marking of data items:
1. check validity bounds of every child of data item
2. if stale (out of DVB$_i$) mark it for an update
Before advice *schedule:*
A rule for updating of data items:
1. schedule data based on data dependency graph and the markings
Aspect outline for an on-demand behavior
(a)
Pointcuts
On an update of a derived item
Inter-type advice adding meta-data:
E-C-A$_1$
E-C-A$_n$
After advice implementing rules:
Rules for reacting to events (E-C-A coupling):
1. on each update of data item d_i
2. check if the condition is satisfied, e.g., $d_i > 30$, and trigger the action associated with the condition, e.g., update of some other data item d_j
3. for triggered update d_j schedule updates of data items according to the on-demand rules.
Aspect outline for an active behavior
(b)

Fig. 2. An outline of aspects handling on-demand and active behavior.

An aspect for active behavior has similar structure, as shown in figure 2(b). The event on which the conditions on data items are checked is identified by pointcuts, e.g., updating of a data item. The conditions that need to be fulfilled, as well as the updates that need to be triggered as actions are added to each data as the meta-data information via the inter-type advices. When an event occurs, the condition on data items is checked by executing the steps specified in the after advice.

Having the on-demand and active behavior cleanly encapsulated into aspects and separated from the underlying system offers several benefits. Namely, the task for the software developer is eased as he/she, to successfully implement on-demand and active behavior, should only know the data items in the system and possibly functions that are used for updating these items. These are needed to identify pointcuts and add meta-data information. As the system evolves and the number of data items changes, on-demand behavior can easily be updated to reflect this change by only updating the code of the aspects. For example, when new derived data items are added to the system or the old ones removed, on-demand behavior can be modified by simply adding meta-data for new data items, or removing the meta-data of the old data items. Also, when base items change, on-demand behavior can be updated to reflect this change by adding new pointcuts (if data is added) or removing the existing pointcuts (if data is removed). Moreover, if a system initially only had on-demand behavior, and due to the evolution data relationships have become more complex requiring rules for active behavior, then the aspect for active behavior could be implemented without direct access to the code of the system. Namely, the pointcuts and inter-type declarations can be reused from the on-demand aspect, resulting in a relatively straightforward step-wise evolution of the system.

3.3 Implementation Issues and Challenges

We have implemented the on-demand and active behavior in a step-wise manner into the COMET experimental embedded real-time database [18, 19, 10]. COMET database was initially developed without support for on-demand and active behavior, which were added later as the platform evolved. We used AspectC++ [15] for the implementation of aspects. The implementation confirmed the observations we stated earlier concerning efficient development and evolution of the system.

Active behavior can currently be added to COMET by shutting down the system and recompiling it. However, in application domains where availability is of utmost importance one should have an option to add active behavior on-line. Consider, for example, a wireless sensor network monitoring an environment. In such a network, due to technological advancements, environmental change, or alteration of system goals during system operation, different types of events associated with different actions might be applicable. This implies that active behavior needs to be added to the system without shutting it down. Hence, one challenge is to dynamically add, remove, or exchange event-condition-action rules in the system. Our initial work on dynamic exchange of aspects [20] could prove beneficial in this respect.

4 Summary

In this paper we proposed using aspects for designing and implementing on-demand and active behavior in embedded software. Aspects enable us to cleanly encapsulate and separate this behavior from the underlying systems. We presented the outline of the aspect-oriented design of on-demand and active behavior and showed that the development and evolution of the system, especially when it comes to data management, are eased by the proposed approach.

References

1. Lee, E.A.: What's ahead for embedded software? Computer (2000) 18–26
2. Barry, E., Slaughter, S., Kemerer, C.F.: An empirical analysis of software evolution profiles and outcomes. In: Proceeding of the 20th International Conference on Information Systems (ICIS'99), Atlanta, GA, USA, Association for Information Systems (1999) 453–458
3. Adelberg, B., Garcia-Molina, H., Kao, B.: Applying update streams in a soft real-time database system. In: Proceedings of the ACM SIGMOD. (1995) 245–256
4. Gustafsson, T., Hansson, J.: Data management in real-time systems: a case of on-demand updates in vehicle control systems. In: Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04), IEEE Computer Society Press (2004) 182–191
5. Gustafsson, T., Hallqvist, H., Hansson, J.: A similarity-aware multiversion concurrency control and updating algorithm for up-to-date snapshots of data. In: Proceedings of the 17th IEEE Euromicro Conference on Real-Time Systems (ECRTS'05), IEEE Computer Society Press (2005)
6. Kim, Y.K., Son, S.H.: Supporting predictability in real-time database systems. In: Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96), IEEE Computer Society Press (1996) 38–48
7. Lee, C.G., Kim, Y.K., Son, S., Min, S.L., Kim, C.S.: Efficiently supporting hard/soft deadline transactions in real-time database systems. In: Proceedings of the 3rd International Workshop on Real-Time Computing Systems and Applications. (1996) 74–80
8. Xiong, M., Ramamritham, K.: Deriving deadlines and periods for real-time update transactions. In: Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99), IEEE Computer Society Press (1999) 32–43
9. Ho, S.J., Kuo, T.W., Mok, A.K.: Similarity-based load adjustment for real-time data-intensive applications. In: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97), IEEE Computer Society Press (1997) 144–154
10. Nyström, D., Tešanović, A., Nolin, M., Norström, C., Hansson, J.: COMET: A component-based real-time database for automotive systems. In: Proceedings of the IEEE Workshop on Software Engineering for Automotive Systems. (2004) 1–8
11. Atzeni, P., Ceri, S., Paraboschi, S., Torlone, R.: Database Systems: Concepts, Lanuages and Architectures. McGraw-Hill (1999) chapter Active Databases.
12. Nyström, D., Tešanović, A., Norström, C., Hansson, J., Bånkestad, N.E.: Data management issues in vehicle control systems: a case study. In: Proceedings of the 14th IEEE Euromicro International Conference on Real-Time Systems (ECRTS'02), IEEE Computer Society Press (2002) 249–256

13. Lopes, C., Hilsdale, E., Hugunin, J., Kersten, M., Kiczales, G.: Illustrations of crosscutting. In: Proceedings of the Workshop on Aspects and Dimensions of Concerns at the 14th European Conference on Object-Oriented Programming (ECOOP'00). (2000)
14. Coady, Y., Kiczales, G., Feeley, M., Smolyn, G.: Using AspectC to improve the modularity of path-specific customization in operating system code. In: Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9). (2002)
15. Spinczyk, O., Gal, A., Schröder-Preikschat, W.: AspectC++: an aspect-oriented extension to C++. In: Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'02), Sydney, Australia, Australian Computer Society (2002)
16. Xerox Corporation: The AspectJ Programming Guide. (2002) Available at: http://aspectj.org/doc/dist/progguide/index.html.
17. Buttazzo, G.C.: Rate monotonic vs. edf: Judgment day. Real-Time Systems **29** (2005) 5–26
18. Tešanović, A., Amirijoo, M., Björk, M., Hansson, J.: Empowering configurable QoS management in real-time systems. In: Proceedings of the Fourth ACM SIG International Conference on Aspect-Oriented Software Development (AOSD'05), ACM Press (2005) 39–50
19. Tešanović, A., Nyström, D., Hansson, J., Norström, C.: Aspects and components in real-time system development: Towards reconfigurable and reusable software. Journal of Embedded Computing **1** (2004)
20. Tesanovic, A., Amirijoo, M., Nilsson, D., Norin, H., Hansson, J.: Ensuring real-time perfomance guarantees in dynamically reconfigurable embedded systems. In: Proceedings of the IFIP International Conference on Embedded and Ubiquitous Computing. Volume 3824 of Lecture Notes in Computer Science., Springer-Verlag (2005) 131–141