Addressing Non-functional Requirements for Embedded Applications with Platform Based Aspect Design

Stefan Resmerita, Anton Poelzleitner, and Stefan Lukesch
Department of Computer Sciences, University of Salzburg, 5020 Salzburg, Austria
Email: {stefan.resmerita, anton.poelzleitner, stefan.lukesch}@cs.uni-salzburg.at

***Abstract*—Aspect Oriented Programming (AOP) and Aspect Oriented Modeling (AOM) have been proposed as development methodologies when dealing with non-functional requirements for embedded systems, which usually represent cross-cutting concerns in relation to a "functional" application. In this paper, based on the observation that non-functional aspects need to be developed from a system-level perspective, we note that application-level AOP is not sufficient to address non-functional requirements (as it usually ignores execution platform constraints). We then propose a development process that considers AOM in conjunction with two standard development approaches for embedded systems: Model Based Development and Platform Based Design. The proposed methodology is illustrated by means of an example with logical execution time requirements.**

I. Introduction

The AOP's "separation of concerns" principle provides a natural basis for dealing with non-functional requirements (NFRs) in embedded systems, since non-functional behaviors represent cross-cutting concerns in relation to the "functional" embedded software. Indeed, a number of works have proposed various aspect-oriented approaches both at the software level (AOP) and at the model level (AOM) of embedded applications (see, e.g., [3], [19], [16], [18]).

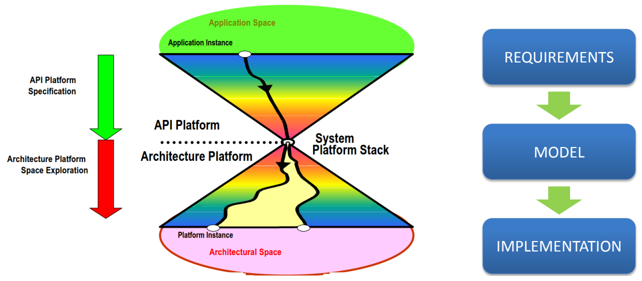
AOP has been originally proposed and mostly used as a programming and software development discipline for desktop applications [9] [8]. In general, these do not have clearly defined "non-functional" properties, due to the fact that they are usually agnostic to the computational (execution hardware and operating system) and physical environments. Thus, cross-cutting concerns added via AOP are always "functional" elements of the application. The standard join point model is defined within the application and this suffices for desktop applications.

In contrast, the *environment* concept (computational and physical) is crucial for embedded software applications [11], and NFRs are always related to the environment. The standard, application-level AOP approach deals with the environment usually via platform services used within aspects. For embedded execution platforms which are usually scarce in resources, this hides important issues such as resource availability, or the indirect effects of using system resources on the applications running on the platform. We consider that AOP employed only at the application level is not sufficient for dealing with NFRs. To illustrate this point, take the execution deadline example, where a function is required to terminate within a given time interval, otherwise a specific exception handler must be executed. This is similar to the real-time example described in [3], which is resolved by introducing an application-level aspect with three main pieces of advice: one to be executed at the start of the function - where a watchdog timer is set, an advice to be executed at the end of the function - where the timer is reset, and a third advice being the exception handler (triggered from an interrupt service routine). This approach does not seem to scale up to industrial-size systems, where such NFRs refer to several functions (say, more than 10 or 20), and employing one real-time aspect per function may lead to other problems. Thus, how does one guarantee that required platform resources (e.g., timers in this case) are available whenever needed? Furthermore, adding many interrupt service routines may increase the interrupt latency in the system to unacceptable values. Such issues are usually left for later stages of the software development process (i.e., implementation, or system integration), where the AOP-based design may be invalidated.

Thus, we argue that platform concerns related to NFRs should be taken into account in the design of aspects. In this paper, rather than addressing the question "How to use aspects to design embedded software?", we deal with the question "How to design aspects that are useful for NFRs in embedded software?". In other words, we elevate aspects from the status of tools used in software design to that of products of a development process for embedded systems. To this end, we consider AOM in the context of two main development strategies for embedded systems: Platform-Based Design (PBD) [7] and Model-Based Development (MBD) [4].

Aspects designed in an AOM approach for satisfying NFRs are likely to encapsulate pointcuts in the platform model linked with advice's in the application and vice-versa. In the above deadline example, the pointcut for the application-level exception handler could be expressed at the platform level: the deadline expiration. In the interrupts example, the advice for the application-level start/end pointcuts could be located in the platform. Thus, the same disable/enable interrupt advice may be associated with different pointcuts in distinct applications. This aspect modularization violates the PBD paradigm, which advocates a clear distinction between the platform and application models.

A non-functional aspect definition may not be able to indicate which parts of the aspect correspond to the platform and which to the application. This information must be made



(a) Platform-Based (source: [7]) (b) Model-Based

Fig. 1: Development processes for embedded systems

apparent after the aspects are weaved into the base design (which, we assume, follows the PBD discipline), possibly by altering the integrated model. Such changes should be propagated back into the aspect model, by re-defining the aspects such that the application components are separated from the platform ones and the interaction between the two is visible.

The rest of this work is organized as follows. Section II briefly presents the three development processes in which NFRs are considered in this paper. This is followed by an outline of our main proposal for aspect development (which is a combination of the three) in Section III. Section IV describes related research and Section V concludes the paper.

II. Background

Platform Based Design [7] is a conceptual strategy for developing embedded systems which takes a "meet-in-the-middle" approach, where successive refinements of specifications meet with abstractions of potential implementations. This is illustrated in Figure 1(a), where the API platform is the programmer's model of the execution hardware, and the Architecture platform represents a specific micro-architecture family.

PBD is particularly applicable to System-on-Chip designs, where design and manufacturing costs depend mainly on the hardware part of the embedded system. The most critical design decisions are those which determine which part of the application is implemented in software and which part is implemented in hardware. Thus, system abstractions that are useful for design space exploration need to expose these two parts and their interaction.

Many embedded systems have been traditionally developed by creating software applications to be run on (low cost) programmable hardware. Here, production costs are mainly driven by software development. In Model Based Development (MBD), executable models are built from requirements and are used to guide the design before reaching the implementation (actual code), which in many cases can be automatically generated from the models - see also Figure 1(b). Executable models are especially useful for testing the design against requirements.

MBD meets PBD in executable models that make a clear distinction between platform components and application components.

III. Platform Based Aspect Design for Non-functional Requirements

When dealing with non-functional requirements(NFRs), Aspect Oriented Modeling (AOM) meets MBD by integrating aspect-oriented components derived from requirements into the base model. From the list of properties seen in Figure 2, one can see that NFRs usually refer to an application's environment (i.e., platform). Consequently, the corresponding non-functional aspects are naturally introduced in base models where MBD meets PBD. Thus, NFRs are best dealt with at the confluence of PBD, MBD, and AOM. To achieve this confluence, the aspectual components must be designed such that *both* MBD and PBD are observed. We propose next a process in this respect called Platform Based Aspect Design (PBAD).

A. *Development process*

Since AOM fits naturally into MBD processes, the main issue to be addressed is ensuring that the resulting aspect components maintain also the PBD discipline. This means that platform components are delineated from application components in the resulting model. This does not usually happen when aspects are first derived from NFRs (as they encapsulate both types of components by definition). Consequently, the aspect components need to be split up into *platform aspects* and *application aspects*, as shown next.

The PBAD development process has three main activities, as depicted in Figure 3. Given an application model (part A) and a platform model (part B), we are presented with a set of non-functional requirements (part C) relating the behavior of A with parameters of B. We consider here that A and B represent a working design.The new part C can be represented in various ways and it can be informal/imprecise or formal (and precise). For example, C may contain requirements such as "Function *f* of A must return as soon as possible", or "The reaction time of *f* must be less than 200 microseconds", or "The reaction time of *f* must be equal to 200 microseconds".

- Nonfunctional properties
  - Time
    - Timing
      - Deadline
      - Period
      - Release time
      - WCET
      - Logical execution time
    - Precision
      - Jitter
      - Tolerated delay
      - Drift
      - Resolution
  - Performance
    - Response time
    - Throughput
  - Memory
    - Memory allocation
    - Area
  - Energy
    - Power consumption
    - Total energy
  - Distribution
    - Task allocation
    - Communication
    - Synchronization
  - Security
    - Access
    - Encryption

Fig. 2: Overview of NFRs (adapted from [19])

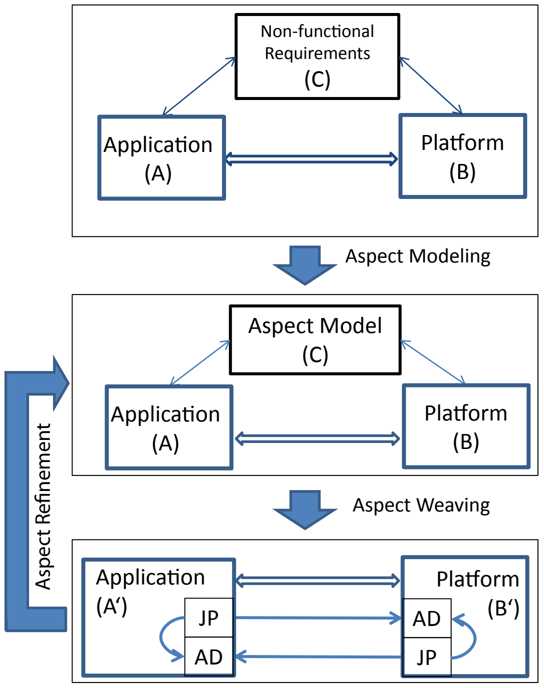


Fig. 3: The Platform Based Aspect Design process

1. Aspect Modeling: In the Aspect Modeling stage, the non-functional specifications are implemented in a model that is connected to parts A and B in an aspect-oriented manner. A non-functional aspect (NFA) may contain specifications of join points and associated actions (advices in AspectJ terms) in both A and B. In particular, an NFA may associate a join point in A with an advice involving elements of B and viceversa. In general, parts A and B may have different models of computation and consequently C may be a heterogeneous model.

2. Aspect Weaving: In this activity, the aspects modeled in part C are weaved into parts A and B. Elements for join point detection and advice components are included in the base models by using model transformation techniques such that their initial modeling language and models of computation are preserved in the final models. If part A is given as source code, then the required model transformations include code generation from components of C and modifications of the original source code. Up to this point, the development process is an instance of AOM and MBD, hand-in-hand. However, the resulting model may violate the separation of platform components from application components. Thus, the model may need to be transformed. Moreover, the platform part may need to be consolidated, as multiple aspects may refer to the same platform component. Since the resulting model does not correspond to the previous aspect model, we propose an additional activity next.

3. Aspect Refinement: The original aspect components are transformed to reflect the changes in the integrated model, leading to an aspect model that distinguishes between platform aspects and application aspects. This brings the model back into the PBD-MBD-AOM confluence.

The PBAD process is illustrated next by means of an example.

B. *Example: Fixing execution times*

In this section we apply PBAD to achieve a timing requirement for a given embedded application. This consists of three functions: a filter, a controller, and a monitoring function. The filter function processes a sequence of values coming from a sensor that samples some physical parameter of the plant under control, and provides the filtered data to the control function, which has a sample time of 8ms. The control output is sent to an actuator and it is also recorded by the monitoring function. These functions are deployed on top of an operating system (OS) on an embedded execution platform, being included in the same time-triggered OS task with a period of 8ms.

1) System model: A model of this embedded system in Ptolemy II [2] is shown in Figure 4. Ptolemy II is a software framework written in Java for modeling, simulation, and design of concurrent, real-time, embedded systems. A Ptolemy model consists of interconnected components, called *actors*, and its execution is controlled by a special component called *director*.

The model in Figure 4 has a discrete event director and contains, in addition to the actors pertaining to the application (the three functions), also platform actors, as follows:

- The *OSTaskPeriod* actor is a timer that generates the triggering signal for the OS task. That is, it produces a token on its output port every 8ms.
- The *SamplingRate* is a timer that generates the sampling signal for the sensor, having a period of 4ms.
- The *Sensor* produces values of some measured physical parameter of the plant under control whenever it receives a token on its trigger port.
- The *Actuator* modifies some physical parameters of the plant under control according to the value of the token received on its input port.

The application components work as follows:

- The *MovAvgF* actor represents a filter function with a worst case execution time (WCET) of 1.5ms. The actor buffers the value received on the *dataIn* input port and, whenever it receives a token on the trigger port, transfers the buffered value into an array, calculates the average value of the array, and requests to be executed after a random non-zero amount of time that is at most 1.5ms. In the follow-up execution, it sends the average value on the *dataOut* port and also sends a token on the *execEnd* port.

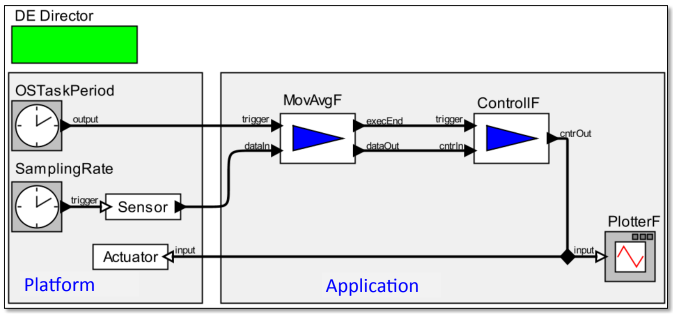


Fig. 4: Embedded system model

- The *ControlF* actor represents the controller function with a WCET of 3.5 ms. The actor buffers the value received on *cntrIn* internally and whenever it receives a token on the trigger port it takes the input from the internal buffer, calculates the output value according to the implemented control law, and requests to be executed again after a random non-zero amount of time that is at most 3.5ms. In the follow-up execution, it sends the output value to the *cntrOut* port.

- The *PlotterF* actor represents the monitoring function which simply plots the value of the input token.

A run of the base model is shown in Figure 5, where one can see that the execution time of *MovAvgF* was 1.5ms and the execution time of the *ControllF* function was 3.5ms.

2) Non-functional requirement - Logical Execution Time: Consider now that it is required to modify the application such that the controller function has a fixed execution time of 4ms. An additional requirement is to make minimal changes to the platform configuration as well as to the application. In particular, the period and the order of execution of the two functions must remain unchanged. This requirement and the corresponding specification are explained next.

In traditional real-time systems, the time instants when software tasks provide their outputs is usually influenced by platform-related factors such as scheduling, system load and memory caches. To avoid this source of unpredictability, the *Logical Execution Time* paradigm [6] abstracts from the physical execution of a task and associates a logical execution time interval (LET) to a periodic top-level software function, also called a LET task. A LET interval spans from a *release* time instant to a *termination* time instant within the task's period, as shown in Figure 6. At the release time, all the inputs to the task are read and transferred to the task's *input ports*, and the *task function* is marked as ready for execution. The actual execution of the function may be delayed and preempted as decided by the underlying scheduler. The function reads input data only from the input ports, whose values remain unchanged (until the next release time). However, the execution must end before the task's termination time, when all the function's outputs are made available to the task's environment (which may include other LET tasks) via the task's *output ports*. Thus, the LET model achieves a pre-specified, platform-independent, observable temporal behavior of a set of software functions, leading to both time and value determinism [5]. The LET specifications are given as a timing program written in a specialized language (e.g., Giotto, HTL, TDL), which is independent on the platform and the implementation of tasks.

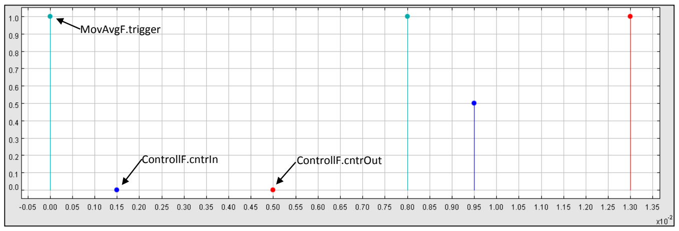


Fig. 5: Simulation output for the base model

In our example, the non-functional specification is given as a program in the Timing Definition Language (TDL) [17], as follows:

```
module MainControl {
 sensor double s1 uses getS;
 actuator double a1 uses setA;
 task MovingAverage [1.5ms]{
  input double dataIn;
  output double dataOut;
  uses MovAvgF(dataIn , dataOut);
 }
 task Controller [3.5ms]{
  input double cntrIn;
  output double cntrOut;
  uses ControllerF(cntrIn , cntrOut);
 }
 task Plotter{
  input double inputPl;
  uses PlotterF(inputPl);
 }

 start mode main [period = 8ms] {
  task
   [8,4−7] Controller(MovingAverage.dataOut);
  actuator
   [8,7] a1 := Controller.cntrOut;
 }
 asynchronous {
  [timer = 8ms] MovingAverage(s1);
  [update = Controller.cntrOut]
                 Plotter(Controller.cntrOut);
 }
}
```

The TDL program is briefly described below. See [17] for a complete TDL specification. The TDL compilation unit is called a module; this provides a namespace for the enclosed definitions. Our example module starts with declarations of sensor and actuator variables, representing the input and output of the application, respectively. Then three TDL tasks are declared, corresponding to the application functions. Each task definition specifies input ports, output ports and the name of the corresponding functional component. Moreover, the declarations of *MovingAverage* and *Controller* tasks include the WCETs (1.5ms and 3.5ms, respectively).

The *mode* declaration specifies a group of periodic time-synchronized activities with a period of 8ms. The times when activities must be carried out are specified in terms of intervals (also called slots) of a uniform partition of the mode period. This mode contains a task invocation and an actuator update.

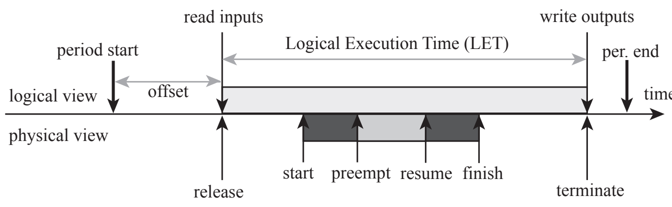


Fig. 6: The Logical Execution Time concept

Both consider a partition of the mode period with 8 slots, hence the slot size is 1ms. The task invocation specifies that the task *Controller* has a LET spanning slots 4 to 7, i.e., the LET has a size of 4ms and starts with an offset of 3ms (relative to the beginning of the period). Moreover, at the release of the task, its input port is updated with the value of the *dataOut* port of the *MovingAverage* task. The actuator update specifies that actuator *a1* must be updated with the value of the *cntrOut* port at the end of the 7th slot (which coincides with the LET end of the *Controller* task).

The *asynchronous* declaration includes activities that do not have strict timing requirements, including event-triggered computations. The first activity is the execution of the *MovingAverage* task, which is triggered by a timer with a period of 8ms and takes its input from the sensor *s1*. The second activity is the execution of the *Plotter* task, which is triggered whenever the *cntrOut* port of the *Controller* task is updated, and takes its input from that port. Note that the asynchronous declarations make no guarantee on the actual time instants when the specified activities are executed. In particular, the TDL program does not specify that the *MovingAverage* task must be executed *before* the *Controller* task.

In the aspect modeling stage, the above non-functional specifications are modeled as aspects in relation to the base model. The result, depicted in Figure 7, is an aspect model that encapsulates the aspectual functionality and interacts with the base model via parameterized ports, called *join ports*. A join port has a parameter that specifies which port(s) in the base model the port is connected to, and how the connectivity of the base port is affected by the join. The parameters of the join ports of the LET aspect model in Figure 7 are shown next to the ports. For example, the input port *period* is connected to the *output* port of the actor *OSTaskPeriod* without affecting any connection in the base model. In contrast, the parameter of the *externTrg* input port specifies that the connection to *MovAvgF.execEnd* is exclusive: all the connections of the *MovAvgF.execEnd* in the base model are removed. Similarly, the parameter of the *trigger* output port specifies that only this port must be connected to the *ControllF.trigger* input port.

The actors of the aspect model in Figure 7 work as follows. The *LETOffset* and *LETInterval* are time-delay actors, which simply delay the input by a time interval specified as parameter of the actor. The values of these parameters corresponding to the TDL program above are 3ms and 4ms, respectively. The *semaphore* actor issues a token on the output port only after receiving at least one token on each input port; input tokens may be received at different points in time. The actor resets (forgets all inputs) upon sending an output token. The *getInput* and *setOutput* actors are instances of the same Java class, providing a basic hold&sample functionality: a token received on the input port is stored internally (overwriting the previously stored token). The stored token is sent on the output port whenever a token is received on the *sampleInput* port.

Note that the LET aspect model bundles together actors that span the entire system: *getInput* and *setOutput* belong to the application, while *LETOffset*, *LETInterval*, and *Semaphore* belong to the platform. The result of weaving the LET aspect model into the base model is shown in Figure 8. This is a standard, executable Ptolemy model (with the DE director not

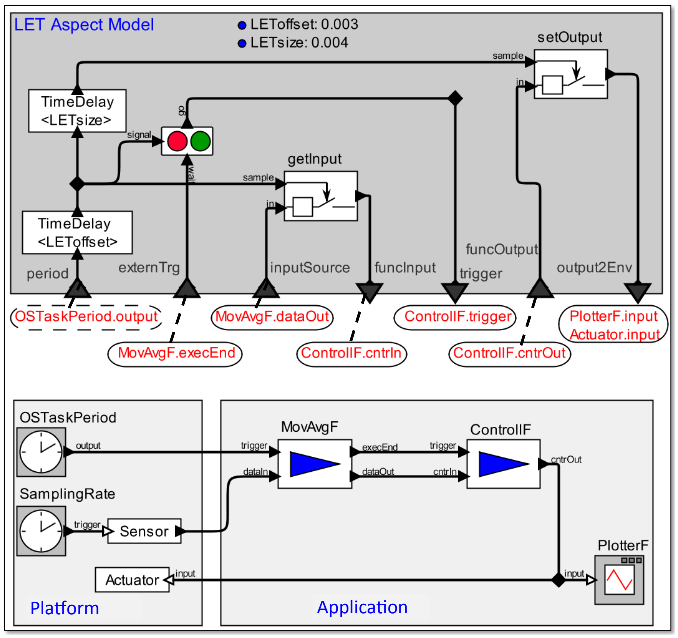


Fig. 7: Aspect model

shown in the figure). The LET-based behavior can be seen in the simulation outputs in Figure 9.

Observe that the aspect model in Figure 7 has more expressive power than the TDL specification given above: it enforces the requirement that the *ControllF* component is executed always after the *MovAvgF* component, ensuring the same execution sequence as in the legacy application. We consider that the Aspect Model is a useful representation for requirements checking and design space exploration, providing a logically coherent view of cross-cutting concerns. Moreover, the aspect model enables reusability of aspects. For instance, the LET aspect in Figure 7 forms a composite actor that can be added to an actor library and then instantiated multiple times in a model.

Consider now a modified TDL program with the activity declarations as follows:

```
start mode main [period = 8ms] {
 task
  [8,1-2] MovingAverage(s);
  [8,4-7] Controller(MovingAverage.dataOut);
 actuator
  [8,7] a := Controller.cntrOut;
}
asynchronous {
 [update = Controller.cntrOut]
              Plotter(Controller.cntrOut);
}
```

This specifies a LET of 2ms for the filter function. The corresponding aspect model is shown in Figure 10, and the integrated model is depicted in Figure 11. Note that while the compositional hierarchy helps in dealing with model complexity, it also blurs the system-level view (Platform-Application). However, by flattening the *LETAspect* actors in the model in

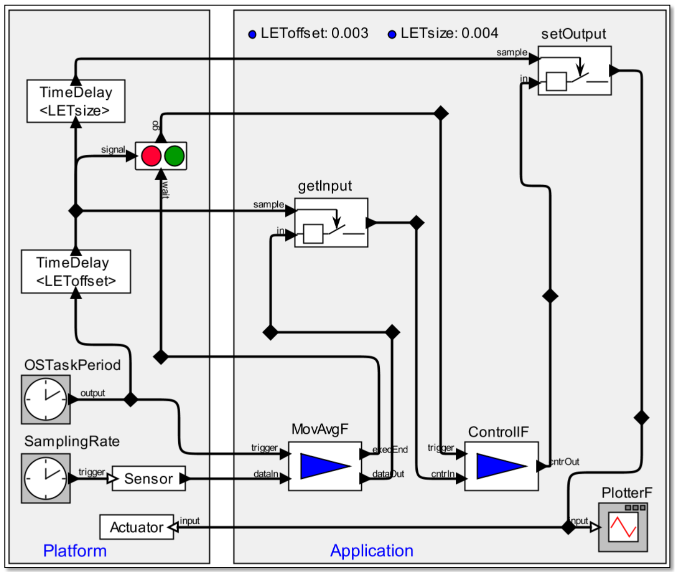


Fig. 8: Integrated model

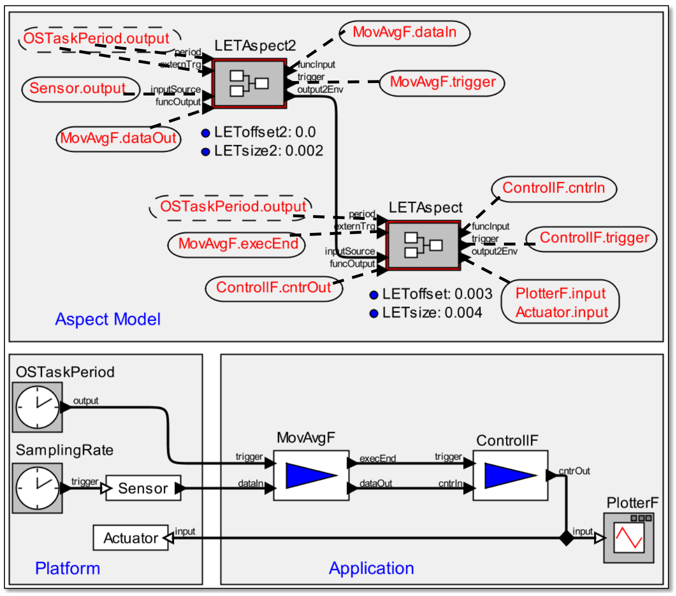


Fig. 10: Aspect model with two LET aspects

Figure 11, one obtains an integrated model where the Platform-Application distinction is again clear, as shown in Figure 12.

This helps in consolidating the platform model, as follows. One can see that one semaphore actor can be removed by connecting its *signal* input directly to its output. Moreover, all time delay actors can be replaced by one pair of actors as shown in Figure 13. The *ResettableTimer* actor reacts to an input token by scheduling a future event at the time equal to current time plus input value, and at that time it issues an output token. The *ArraySweep* actor has a parameter holding an array of values representing time delays and it has one output port corresponding to each cell in the array. The actor keeps track of the current position in the array. Upon receiving a token on the *input* port, the actor issues that token on the output port associated with the current position, advances the current position and sends the array value at the current position to the *current* output port. Upon receiving a token on the *reset* port, the actor sets the current position to zero and outputs the current value on the *current* port. The array parameter is set to $\{LEToffset2, LETinterval2, LEToffset - (LETinterval2 + LEToffset2), LETinterval\}$. One can check that the behavior of the application model in Figure 13 is the same as the corresponding one in Figure 12.

In the aspect refinement stage, we modify the aspect model

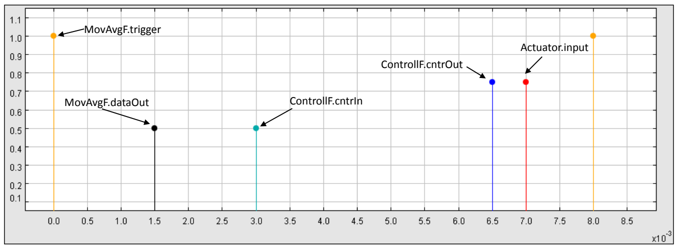


Fig. 9: Simulation output shows LET behavior

in Figure 10 to reflect the changes made in the integrated model, obtaining a refined aspect model shown in Figure 14. Notice that this model reflects more accurately the original formal specification of the non-functional requirement (the TDL program)!

IV. Related Work

Aspect oriented techniques have been employed for non-functional requirements of embedded software in [1], [13], [14], [19], [20]. The general application design process starts with control logic design, then goes to software design and then to programming. At the software design level, the control logic is considered to be the ”functional” part of the software, while real-time and platform-related properties such as triggering, scheduling, prioritization are seen as the ”non-functional” part. For instance, in the approach presented in [20], AOP is employed to support different triggering policies of control functions. An example is provided where the execution order of several control methods depends on the kind of triggering (event-triggered or time-triggered). However, only the execution order is specified by means of aspects, not the triggering itself. Thus, such an aspect must be accompanied by a specification on how the first method in the sequence is triggered, e.g., a UML sequence diagram. In this paper, we employ the same modeling framework for both application-level aspects and platform level behavior. It is also debatable if triggering of control functions belongs to the functional part or not. In many cases, sample times are hardcoded into the constants of the control functions, so they are in the functional part. For such systems, the values of control outputs depend on the triggering (even if the control logic does not). The motivational example given in [20] mentions CAN (Controller Area Network) message handling procedures, which refer to communication, not control.

Various papers describe how AOM techniques can be included in MBD processes for embedded software, mostly using

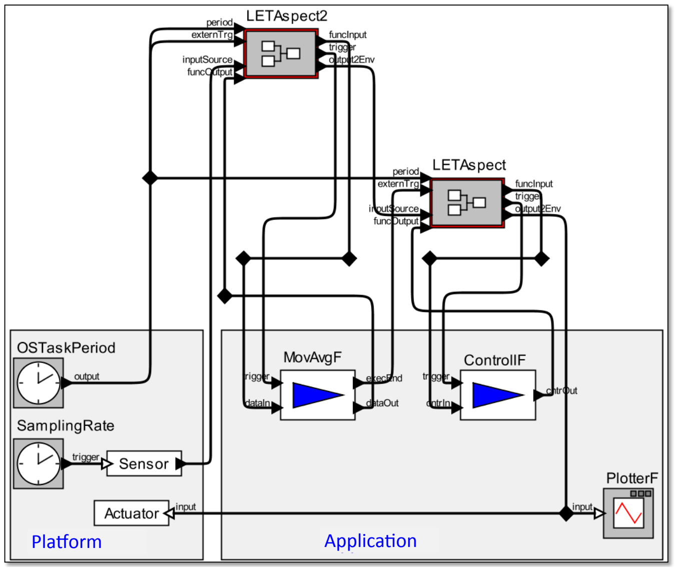


Fig. 11: Integrated model with the LET aspects weaved in

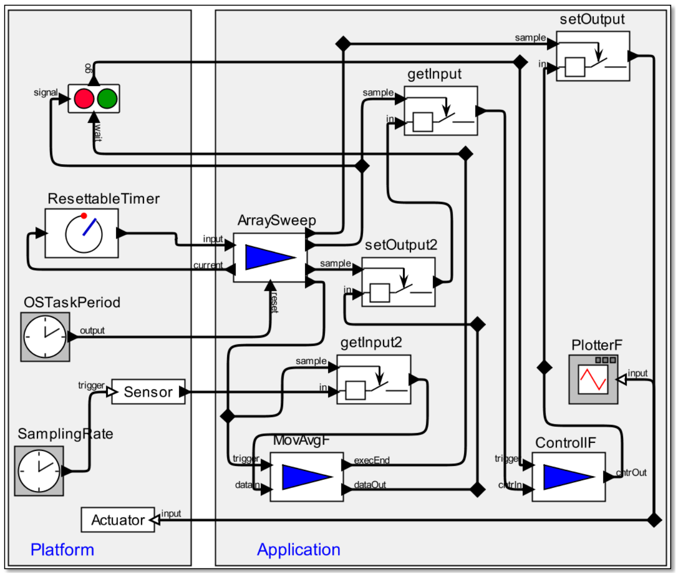


Fig. 13: Integrated model with platform part consolidated

UML models [19], [13], [16]. Here, we consider executable heterogeneous models as in the Ptolemy II framework [2], where model components can be grouped in domains with different rules for execution and interaction. This enables analysis and simulation of mixed models containing different domains for software, hardware, and physical model components. The aspect-oriented design language Theme/UML, proposed in [1], extends classic UML with a new type classifier called *theme*. A theme is a base UML construct containing the specification of a concern (base or aspect). Base themes represent functional properties, whereas an aspect theme corresponds to a non-functional property. In [19], an extensible set of aspects is introduced at the model level, dealing with NFRs for distributed embedded real-time systems. The main goal of that work is to provide an aspect framework (aspects are modeled with RT-UML), which adds behavior and structure to the system without tying the model to a particular implementation.

For this example, the implementation of the TDL specification and the aspect weaving have been done manually in Ptolemy. Work is under way to automatize such operations.

The research reported in [12] is based on the same line of thought that it is best to consider the impact of an application on the underlying platform early in the design process. The paper deals with joint validation of application and platform models at multiple abstraction levels, using also Ptolemy II. It focuses on general system design methodology rather than adaptation to NFRs. It would be interesting to investigate

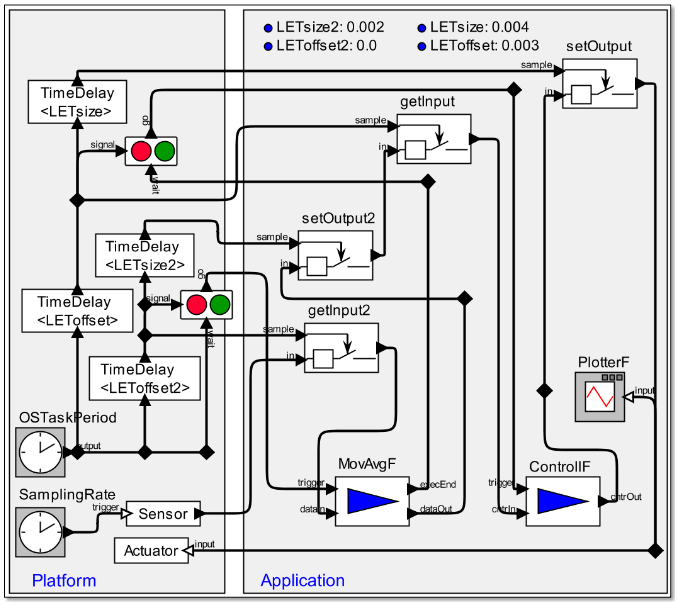


Fig. 12: Integrated model with flattened aspects

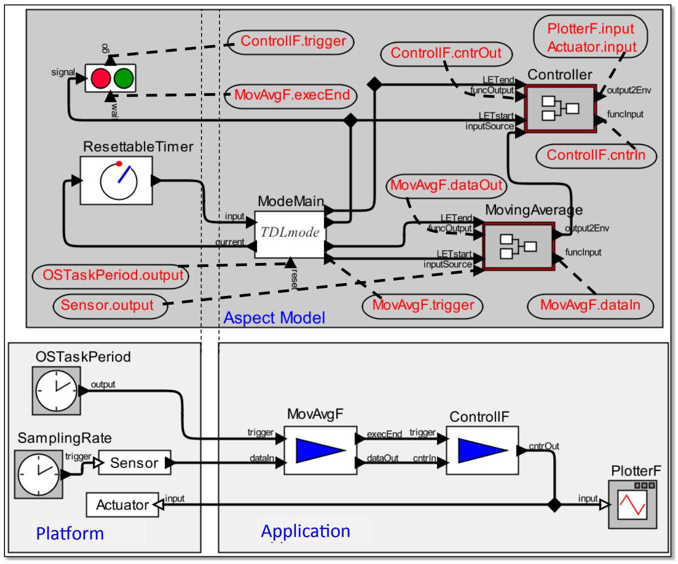


Fig. 14: Refined aspect model

if the aspect-oriented process proposed in our paper can be regarded as a method for successive refinement of application and platform models within the approach in [12].

The work presented in this paper is also related to the concept of Aspect-Oriented Multi-View Modeling [10], which enables descriptions of the same software system from multiple points of view (e.g., structural and behavioral). More general work in AOM has been done regarding join point definition languages. In [15], join points are described in UML models with so called Join Point Designation Diagrams in numerous ways, capturing control-flow, data-flow and state models.

V. Conclusions

This paper discusses the AOM approach to dealing with non-functional requirements for embedded applications in the context of two main development strategies for embedded systems: Platform Based Design and Model Based Development. It describes a first attempt to harness the advantages of the three development models under a sub-process called Platform-Based Aspect Design. This is illustrated by means of an example where logical execution times are added to functional components of a base model.

Our motivation for addressing non-functional requirements with models that can be simultaneously viewed from the three perspectives, is based on three basic observations: (1) AOM is a natural approach to NFRs due to its ability to address cross-cutting concerns, (2) AOM follows the principles of MBD, and (3) NFRs refer to platform properties, which are best expressed in models following the PBD paradigm. Most of the existing research on non-functional requirements for embedded systems is based on observations (1) and (2). We add here the third and end up with a combined approach, where aspects developed with AOM in an MBD way may need to be refined in order to reflect PBD characteristics. In some cases, such a refinement can be done only after weaving the MBD-aspects into the (PBD-compliant) base model, and performing model transformations in order to maintain the PBD discipline. Then the refinement means transforming the aspect model such that the AOM discipline is kept (i.e., the integrated model represents the result of aspect weaving into the base model).

In future work, we intend to employ the process described here to use cases involving different types of NFRs and various platform-application models. We also plan to improve the join point specification mechanism for actor-based models which is briefly (and informally) touched upon in this paper.

References

[1] Cormac Driver, Sean Reilly, Éamonn Linehan, Vinny Cahill, and Siobhán Clarke. Managing embedded systems complexity with aspect-oriented model-driven engineering. *ACM Trans. Embed. Comput. Syst.*, 10(2):21:1–21:26, January 2011.

[2] Johan Eker, Jorn W Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[3] Andreas Gal, Olaf Spinczyk, and Wolfgang Schroeder-Preikschat. On aspect-orientation in distributed real-time dependable systems. In *In Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 7–9. IEEE Computer Society, 2002.

[4] Holger Giese, Gabor Karsai, Edward A. Lee, Bernhard Rumpe, and Bernhard Schtz. *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*. Springer, 2010.

[5] Thomas A Henzinger. Two challenges in embedded systems design: predictability and robustness. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3727–3736, 2008.

[6] Tom Henzinger, Ben Horowitz, and Christoph Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, January 2003.

[7] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, November 2000.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, pages 327–353, 2001.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[10] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modeling. In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, pages 87–98, New York, NY, USA, 2009. ACM.

[11] Edward A. Lee. Computing needs time. *Communications of the ACM*, 52(5):70–79, May 2009.

[12] Sanna Määttä, Leandro Möller, Leandro Soares Indrusiak, Luciano Ost, Manfred Glesner, Jari Nurmi, and Fernando Moraes. Joint validation of application models and multi-abstraction network-on-chip platforms. *IJERTCS*, 1(1):86–101, 2010.

[13] Takahiro Soeda, Yuta Yanagidate, and Takanori Yokoyama. Embedded control software design with aspect patterns. In Dominik lzak, Taihoon Kim, Akingbehin Kiumi, Tao Jiang, June Verner, and Silvia Abraho, editors, *Advances in Software Engineering*, volume 59 of *Communications in Computer and Information Science*, pages 34–41. Springer Berlin Heidelberg, 2009.

[14] John A. Stankovic, Ruiqing Zhu, Ram Poornalingam, Chenyang Lu, Zhendong Yu, Marty Humphrey, and Brian Ellis. Vest: An aspect-based composition tool for real-time systems. In *Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '03, pages 58–, Washington, DC, USA, 2003. IEEE Computer Society.

[15] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Expressing different conceptual models of join point selections in aspect-oriented design. In *Proceedings of the 5th International Conference on Aspect-oriented Software Development*, AOSD '06, pages 15–26, New York, NY, USA, 2006. ACM.

[16] Masayoshi Tamura, Tatsuya Kamiyama, Takahiro Soeda, Myungryun Yoo, and Takanori Yokoyama. A model transformation environment for embedded control software design with simulink models and uml models. *Proceedings of the International MultiConference of Engineers and Computer Scientists*, 1, July 2012.

[17] Josef Templ. Timing Definition Language (TDL) 1.5 specification. Technical Report T024, University of Salzburg, July 2009.

[18] Armin Wasicek, Patricia Derler, and Edward A. Lee. Aspect-oriented modeling of attacks in automotive cyber-physical systems. In *Proceedings of the 51st Design Automation Conference (DAC)*, June 2014.

[19] M.A. Wehrmeister, E.P. Freitas, C.E. Pereira, and F.R. Wagner. An aspect-oriented approach for dealing with non-functional requirements in a model-driven development of distributed embedded real-time systems. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC '07. 10th IEEE International Symposium on*, pages 428–432, May 2007.

[20] Takanori Yokoyama. An aspect-oriented development method for embedded control systems with time-triggered and event-triggered processing. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 302–311. IEEE Computer Society, 2005.