Aspect-oriented hard real-time programming and tool integration

Wolfgang Pree and Josef Templ
Department of Computer Science, University of Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

Abstract

The paper sketches how aspects, integrated into a hard real-time programming language, could improve the modularization of embedded software. Based on an overview of the Timing Definition Language (TDL, as described in the TDL language report [1]), we present how aspect-oriented concepts could be woven into that language. A discussion of the advantages and disadvantages of AspectTDL rounds out the paper.

1 Motivation

Deterministic hard real-time programming systems such as Giotto [2] or TDL [1] assume to interoperate with their environment through the usage of so-called sensors and actuators exclusively. A sensor represents a value provided by the environment. It is read into the real-time system by using a getter function, i.e., a function that returns the value of the sensor by accessing appropriate hardware registers or other environment variables. An actuator represents a value to be set in the physical environment by means of a setter procedure that writes the value to an appropriate hardware register or other environment variable. Getters and setters are the only means to communicate with the physical environment of a real-time system and they are implemented outside of the source code of the real-time system, for example, in C or Java.

Getters and setters are called from the real-time system by means of internally maintained glue code, called drivers. The execution system of real-time code simply calls the appropriate driver in order to activate a getter or setter. The driver concept abstracts from the concrete implementation of getters and setters.

Now suppose that we want to equip a real-time system with an additional feature called limit monitoring. This allows a user to select a subset of sensors and actuators and to specify upper and lower bounds for the values and associated actions in case of leaving the specified range. Such a feature would affect all getters and setters insofar as code for range checking must be included for all sensors and actuators that were selected for monitoring by the user. Clearly this is an undesirable global change of program code spread across many different source files. In other words this would be a socalled crosscutting aspect, which is the subject of aspect oriented programming (AOP) as pioneered by G. Kiczales et al. [3]. Another example of a crosscutting aspect is integration, which means that the real-time system is integrated into some non real-time environment used to visualize the process state and to interact with the user. The interaction between heterogeneous real-time systems in a distributed system may also be seen as a crosscutting aspect.

Without going into the details of AOP implementation, it should be mentioned here that the getter and setter functions are a poor place for implementing crosscutting aspects of a real-time system. The drivers which are used to abstract from the getter and setter implementation are much better place. If a driver provides the means for incorporating additional functionality, aspects may be incorporated even dynamically without the need to change existing code.

The following sections describe the concepts of AOP, the relevant features of the hard real-time language TDL and refine the ideas of AOP in the context of real-time software.

2 Multi-dimensional modularization through aspects

Aspect-oriented programming (AOP, [3, 4, 5]) was pioneered by Gregor Kiczales mid of the 1990s as a means of overcoming the problems related to one static modularization as prevalent in module- and object-oriented programming languages. The premise of AOP is that there is not one perfect static modularization, but that various different modularizations are required to modularize a software system. This is why we view AOP as a means of multidimensional modularization. The additional modularizations that override the one static modularization depend on conditions that are evaluated at run-time.
Conceptually, AOP is based on the more general idea of meta-programming, which had already been refined by Gregor Kiczales. Meta-programming allows the modification and extension of the semantics of a

programming language by writing code that defines the specific semantics. For example, a method invocation in an object-oriented language such as Java can be modified so that additional processing is done, either before or after the actual method invocation. AOP applies meta-programming for improving the modularization of a software system. In the following we summarize those AOP concepts and terms that are relevant in the context of this paper.
Figure 1a schematically shows the static modularization of a software system. Each bar represents a module. The length of a bar corresponds to the number of lines of code. Those parts of the source code that would logically form a module but which had to be scattered over several modules are shown as gray bars in the modules. Source code that logically belongs together but is split over several modules has the disadvantage that changes have to be accomplished in several modules, not in one spot. Thus, it becomes difficult to maintain consistency. The AOP advocates argue that it is impossible to find one perfect modularization for a real-world software system that does not encounter the sketched problem. If one changes the static modularization to solve one detected modularization glitch, the modularization of other aspects that were originally well modularized, might be distorted. In other words, according to AOP one static modularization is not sufficient.

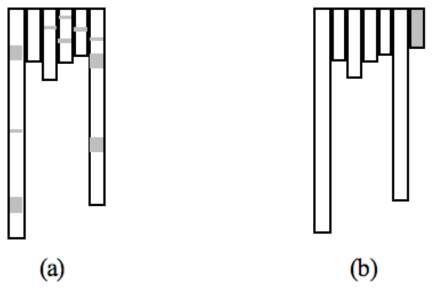

Figure 1. AOP cross-cutting

In order to solve the problem of scattered code that logically forms a unit, AOP allows us to cross cut several modules, that is, to cut the scattered code and put it into an additional module. The gray rectangle in Figure 1b represents such an additional module containing the code that was originally scattered over several modules. After presenting the Timing Definition Language we show how aspects could be integrated into that language.

3 The Timing Definition Language (TDL)

Giotto [2] with its sound formal semantics provides language constructs that allow the development of deterministic, composable control applications. Their timing and communication behavior can be specified independent of their implementation. The Timing Description Language (TDL, [1]) enhances Giotto towards a component architecture for real-time control applications. Besides some syntactical changes and minor adaptations of the semantics, however, TDL is based on the same programming model as Giotto.

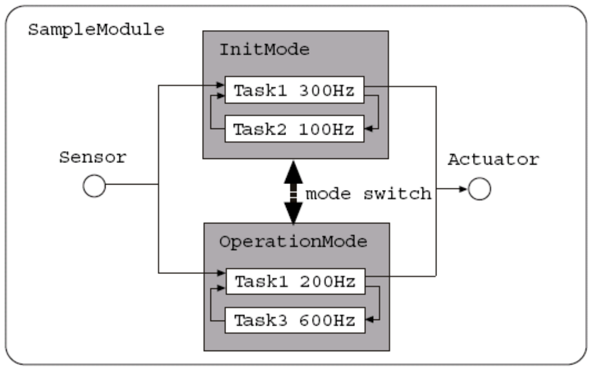

Figure 2. Visual Representation of a TDL module

Giotto programs are multi-mode and multi-rate systems that periodically execute a set of tasks and other activities. Figure 2 shows a simplified, visual representation of a Giotto program, which corresponds to a TDL module. A TDL application comprises one or more modules. A module consists of a set of modes. A mode contains a set of activities: task invocations, actuator updates, and mode switches. A TDL module is in one mode at a time. A mode defines the set of tasks that needs to be executed in parallel in a particular mode of operation of the real-time application. In addition to executing tasks, a mode also defines actuator update operations and the conditions for mode switches. All kinds of activities (task invocation, actuator updates, and mode switches) can be activated with different rates. In order to provide deterministic behavior of a control system, Giotto defines that task outputs are only available at the end of a task's logical execution period, which is also called the FLET (fixed logical execution time) assumption. The results of a task may be available earlier in internal program variables, but each result value that is visible to clients of a task is updated exactly at the end of the task's FLET.

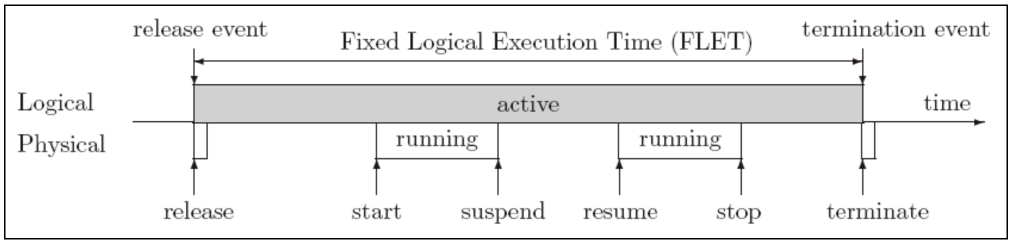

Figure 3. The Giotto/TDL Task Model

FLET provides the cornerstone to deterministic behavior, platform abstraction, transparent distribution, and well-defined interaction semantics between different activities. It is always determined which value is in use at which point in time and there are no race conditions or priority inversions involved. Figure 3 shows the FLET property of Giotto/TDL tasks.

In addition to the drivers mentioned in the motivation section, TDL uses drivers for all task events and for mode switches, where the latter are not shown above.

- the release driver is called at the beginning of the task’s FLET by the socalled E-machine, part of TDL’s run-time system,
- the start driver is called by the socalled S-machine, also part of TDL’s run-time system, in order to invoke the task’s implementation function,
- the stop driver is called by the S-machine after finishing execution of a task and
- the termination driver is called at the end of the task’s FLET by the E-machine.

It is beyond the scope of this paper to describe the details of the driver implementations, however, it should be clear that the drivers are the key of an AOP implementation for TDL. They allow for extending the semantics of various runtime activities without the need to change the E-machine or the functionality code.

Differences between Giotto and TDL

The most visible syntactical differences between TDL and Giotto are:

- the introduction of a top level language construct (module) and the reorganization of mode declarations, where ’start’ is a modifier of a mode declaration in *TDL*.
- the elimination of global output ports, which are replaced by task output ports in *TDL*,
- the elimination of explicit task and mode drivers, which are merged into mode declarations in *TDL*,
- the addition of constants, which may also be used to initialize ports in *TDL*,
- the introduction of units for timing values in *TDL*.

The following list explains differences between TDL and Giotto semantics.

- *program start*. A *TDL* program is started by switching to the start mode. This means that at time zero, there are neither actuator updates nor mode switches. In Giotto, the actuator updates and mode switches of the start mode take place at time zero. There are, however, no further actuator updates or mode switches of the target mode at time zero.
- *non-harmonic mode switch*. Giotto allows to switch a mode even if there are running tasks as long as those tasks exist with the same task period in the target mode. However, there may be delays involved when switching to the target mode. Furthermore, the task will deliver output values to the target mode, which do not correspond to inputs specified there. *TDL* does not allow non-harmonic mode switches. We

are thinking about alternative ways of performing even faster mode switches without the need to continue running tasks in the target mode, with simpler semantics and, last but not least, without any delays.

- *deterministic mode switch*. Giotto requests that among all mode switch guards of a mode only one may return true at a particular point of time. In contrast, *TDL* evaluates mode switch guards in textual order from top to bottom and performs the first mode switch whose guard returns true. This definition allows a more efficient implementation and preserves determinism.
- *actuator update*. A guarded actuator update in Giotto means that the actuator setter is called independently of the guard's result. In TDL, actuator update *and* actuator setter are both guarded and performed only if the guard returns true.

The following list describes tool related differences between TDL and Giotto.

- *E-code file format*. TDL defines a binary, platform independent E-code file format and uses statically typed APIs for connecting programs with external functionality code. Platform specific output may be generated by a platform plugin for the TDL compiler.
- *E-code instructions*. The structure and semantics of Giotto E-code instructions has not been changed in TDL but a SWITCH instruction has been added. It is used to perform mode switches. In Giotto, mode switches are performed by the JUMP instruction by jumping to code of a different mode. The SWITCH instruction makes this special usage of JUMP explicit and thereby simplifies the detection of mode switches in the E-machine.
- *Time Resolution***.** TDL uses microseconds internally for all timing values, whereas Giotto is based on milliseconds. This means, that TDL programs may use mode periods below 1 millisecond, given that the underlying E-machine is fast enough.

4 AspectTDL

The following example illustrates our ideas regarding an aspect-oriented extension of TDL. Note that this is an experimental syntax and that we have not yet implemented the aspect-oriented extensions of TDL. The presentation should lead to discussions regarding, for example, the pros and cons of aspects in the context of TDL or which level of genericity such an extension should provide. The syntax will significantly be shaped by the implementation of these language features.

The basic concept of AspectTDL is that the various entities described in TDL, associated with various TDL modules, are subject to cross-cutting. For example, the developer could define an extra module as aspect that deals with a subset of tasks that are defined in the already specified modules.

We use the TDL module construct, which is the primary unit of program decomposition in TDL, as a container for aspect-oriented declarations. Following the terminology introduced for *aspect-oriented Java* [4], we declare aspects by using the keyword *aspect*. A module may declare any number of aspects.

```
module SampleAspectTDLModule {
  aspect limo { //assume: limit monitoring on task outputs
    pointcut limoTasks (task): uses limoPointcutImpl;
    after terminate [wcet=1us] uses limoImpl;
  }
  aspect connect { //assume: output und actuator ports will be connected to the environment
    pointcut connectedPorts (output || actuator): uses connectPointcutImpl;
    before set [wcet=2us] uses connectImpl;
  }
  aspect watchdog { //assume: mode switches will be monitored
    pointcut watchSwitches (mode): uses watchPointcutImpl;
    after switch uses watchImpl;
  }
}
```

An aspect has a name, which serves to describe the basic idea associated with the aspect. The details of the aspect are defined within the aspect's body, which is surrounded by a pair of curly brackets.

Every aspect needs a definition of TDL program entities that are the subject of the aspect. We use the keyword *pointcut* for defining the selection of those subjects. In addition, an advice (before or after) must be specified which will be executed whenever the selected program entity performs a specific action.

A pointcut always refers to one of the various TDL program entities such as task, sensor, actuator, output or mode. The filter specified in the parentheses may be an elaborate expression or simply an appropriate keyword. In addition to selecting the program entities in the aspect declaration the user may specify a filter using an external function (e.g., limoPointcutImpl). This function must be implemented as external functionality code.

An advice is either performed *before* or *after* the selected program entity performs a specific action, which depends on the kind of entity. A task, for example, may be released, started, stopped, and terminated. An output or actuator port may be set or a mode may be switched to. An advice may need a non-negligible amount of CPU time, which can be specified by an optional wcet-annotation (worst case execution time). In any case the implementation of an advice will be done in external functionality code.

Implementation strategy

The basic idea of implementing aspects in TDL is to augment the driver architecture of the TDL E-machine. Every action performed on a program entity is invoked by using a so-called driver, which serves as the glue code between the E-machine and the external functionality code. By applying the Observer pattern [6] to the notion of a driver, we arrive at a framework that allows the registering of listeners for driver events. In addition to executing a driver's code, we prepare the hooks needed for pre- and postprocessing as sketched in the following pseudo code:

```
void terminateDriver() {
 for each b in beforeTerminateListeners: b.call();
  driver code;
 for each a in afterTerminateListeners: a.call();
}
```

A listener object is created for each task that belongs to a particular aspect. The developer has to provide the corresponding pointcut implementation function which would look like the following pseudo code:

```
boolean limoPointcastImpl(Task t) {
  return (aspect limo applies to t);
}
```

Returning true means that a listener is to be created for the particular task and the listener has to be registered. Registering a listener means that in case of a before advice the listener is added to the corresponding 'before...' list and in case of an after advice the listener is added to the corresponding 'after...' list.

5 Conclusions

From the evidences outlined in the paper we believe that weaving aspects into a hard real-time programming language such as TDL would help to improve the modularization of embedded software. As mentioned in the motivation, it also facilitates the integration with legacy systems or with non-real-time environments.

We have integrated TDL with the Simulink tool chain [7] so that it becomes useful for our cooperating industry partners. In this context, the aspect-oriented extension was inspired as we had to think of requests such as the one how limit monitoring could be specified. The aspect-oriented extension of TDL seems to be the adequate means for that and for analogous future requirements. In other words, an AspectTDL will significantly improve the already accomplished integration of TDL within the Simulink environment.

A potential disadvantage is the additional complexity of TDL. Furthermore, the processing of aspects leads to a run-time overhead that interfers with the Giotto/TDL paradigm of synchronous input and output.

A first implementation of AspectTDL might imply changes to the syntax of AspectTDL. Several details, such as the syntax of filters, still need to be defined.

Bibliography

[1] J. Templ. TDL Specification and Report, Technical Report, University of Salzburg; http://www.SoftwareResearch.net/site/publications/C059.pdf

[2] T.A. Henzinger, C.M. Kirsch, and B. Horowitz. Giotto: A time-triggered language for embedded programming. Proceedings of the IEEE, 91(1):84–99, January 2003.

[3] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. In Proc. of ECOOP, Springer-Verlag (1997). http://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf

[4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. An Overview of AspectJ. In Proc. of ECOOP, Springer-Verlag (2001). http://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP2001-AspectJ.pdf

[5] Hidehiko Masuhara and Gregor Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In Proceedings of ECOOP 2003, Springer-Verlag (2003). http://www.cs.ubc.ca/~gregor/papers/masuhara-ECOOP2003.pdf

[6] E. Gamma, R. R. Helm, R. Johnson, J. Vlissides: Design Patterns—Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[7] W. Pree, G.Stieglbauer: Visual and interactive development of hard real-time code; Automotive Software Workshop San Diego, CA, January 2004 (Springer LNCS)