Towards a Component Architecture for Hard Real Time Control Applications

Wolfgang Pree and Josef Templ

Software Research Lab
University of Salzburg, A-5020 Salzburg, Austria
{pree, templ}@SoftwareResearch.net

Abstract. This paper describes a new approach towards a component architecture for hard real time control applications as found, for example, in the automotive domain. Based on the paradigm of fixed logical execution time (FLET) as introduced by Giotto [1], we develop a higher level language construct, called *module*, which allows us to organize and parallelize real time code in the large. Our module construct serves multiple purposes: (1) it introduces a namespace for program entities and supports information hiding, (2) it represents a partitioning of the set of actuators and control logic available in a system, (3) it acts as a static specification of components and dependencies, (4) it may serve as the unit of dynamic loading of system extensions and (5) it may serve as the unit of distribution of functionality over a network of electronic control units. We describe the individual usage cases of modules, introduce the syntax required to specify our needs and discuss various implementation aspects.

1 Introduction

Hard real time control applications, as found for example in the automotive domain, exhibit topologies which may be classified as (1) a single application split between multiple computation nodes or (2) a single computation node split between multiple applications. The latter case is considered to be of increasing importance for future systems because of the ever increasing computation power of microcontrollers, microprocessors and the trend towards microsystems consisting of multiple logical or physical processing units on a single chip or board. Such systems will be capable of executing multiple control applications in parallel on a single electronic control unit (ECU). They must, however, preserve all the timing properties of the applications as if they were performed independently on individual ECUs. In case of the automotive domain, the consolidation of ECUs is expected to reduce the weight and complexity of a vehicle and to save money.

This paper describes a component architecture aiming at the goal of ECU consolidation with preservation of hard real time properties. Our approach is based on the fixed logical execution time assumption introduced by Giotto, but expressed in a more convenient syntax (TDL = Timing Definition Language) [5], slightly changed seman-

tics and, most importantly, a module concept, which introduces the required abstractions for running multiple real time control applications on a single system. Our module construct serves multiple purposes: (1) it introduces a namespace for program entities and supports information hiding, (2) it represents a partitioning of the set of actuators and control logic available in a system, (3) it acts as a static specification of components, (4) it may serve as the unit of dynamic loading of system extensions and (5) in the future it may serve as the unit of distribution of functionality across a network.

2 Key Ingredients of an Embedded Control Software Model

This section summarizes what we regard as preconditions for a solid component architecture for hard real-time applications. The concepts have been invented in the realm of the Giotto project [2] at the University of California, Berkeley.

Platform-Independent Specification of Computation and Communication Activities

Figure 1 shows a simplified, visual representation of a TDL program. A TDL 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. Mode switch conditions are checked periodically with a specified frequency.

Tasks form the units of computation. They are invoked periodically with a specified frequency. They deliver results through task output ports to actuators or to other tasks, and they read input values from sensor ports or from output ports of other tasks. Thus, a TDL model specifies the real-time interaction of a set of components with the physical world, as well as the real-time interaction between the components.

A task's functionality, that is the control laws, can be implemented in any non-embedded programming language such as C.

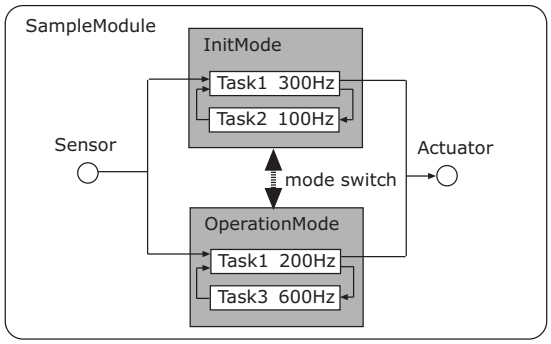

Fig. 1. Visual Representation of a TDL module

What makes TDL a good software model is the fact that the developer does not have to worry about platform details, for example: will the application be executed on a single node or on a distributed platform; which scheduling scheme ensures the timing behavior [4]; which device drivers copy the values from sensors or to actuators. Thus, the software model emphasizes application-centric transparency (simplicity), improves reliability and enables reuse, whereas the compiler that generates the code from the model emphasizes performance.

According to [1] a TDL program supervises the interaction between software processes and the physical world, but does not itself transform data. All computation is encapsulated inside the supervised software processes (tasks), which can be written in any non-embedded programming language. We refer to a TDL program as a timing program, and to the supervised processes called by the TDL program as functionality programs. A TDL program specifies only the reactivity of the functionality programs—that is, when they are invoked, and when their outputs are read—but not their scheduling.

The Fixed Logical Execution Time (FLET) Assumption and its implications

The key property of the TDL semantics is the *fixed logical execution time* (FLET) assumption, which means that the execution times associated with all computation and communication activities are fixed and determined by the model, not the platform. In TDL, the logical execution time of a task is always exactly the period of the task, and the logical execution times of all other activities (mode switching, data transfer across links, etc.) are always zero. For example, the task labelled Task1 in the OperationMode in Figure 1 logically executes for 5 micro seconds, which implies that (1) it reads its input at the beginning of its period, and (2) its output is not available to other tasks before 5 micro seconds, even if the actual execution of the task on the CPU finishes earlier.

According to [1] a TDL model is *environment determined*: for any given behavior of the physical world seen through the sensors, the model computes a *unique* trace of actuator values at periodic time instants. In other words, the only source of non-determinism in a TDL system is the physical environment. This makes the validation of the system considerably easier and forms the precondition for real-time compositional models. Thus, our component architecture relies on preserving the FLET assumption.

3 FLET-based Components

The subsequent sections present our modular architecture for control applications that rely on the FLET assumption. The related language constructs are part of TDL.

Introducing Modules

As a first step towards a modular architecture for hard real time control systems, we introduce the notion of a module as a container of a Giotto program. Thus, all code belonging to a traditional Giotto application is textually enclosed inside a module. The module construct starts with the keword 'module' followed by the name of the module and a pair of curly brackets, which represent the namespace introduced by the module. The following example shows the skeleton of a module.

```
module EngineControl {

  //Giotto/TDL code consisting of sensor, actuator,
  //task and mode declarations

}
```

As a consequence, we arrive at Giotto programs as named entities, which may be handled by an appropriate runtime system on an ECU. Such a runtime system, also called an embedded machine (E-machine), may load and execute multiple modules in parallel. It should be noted that a module must only be loaded once into an E-machine and it stays loaded until the E-machine terminates or there is some user interaction, which unloads the module explicitly. It is up to the runtime system being used on an ECU if modules are loaded dynamically or if a static configuration has to be provided.

CPU Partitioning

A module may provide a *start* mode, which is the mode the application is executing after loading the module into an ECU. Executing a module implies the reservation of a percentage of the available CPU time for execution of this module, given that the CPU is fast enough to execute this module in addition to possibly other modules loaded before. A module which needs to reserve a percentage of the CPU is called a 'partition' and splitting the CPU between multiple partitions is called 'CPU partitioning'. A module which does not provide a start mode will not be executed, which means, it will not need a CPU partition but it may still be meaningful with respect to other usage cases as explained in subsequent sections.

Assuming that an E-machine provides the means to dynamically load a module, this can be used for handling the usage case 'dynamic partitioning'. At runtime, an arbitrary module may be loaded upon request by the user, thus leading to a set of independently loaded modules. Of course, the set of modules to be loaded into a particular E-machine may be configured in some configuration file, but this is not standardized and not known to the E-machine.

Module Import

In order to allow the decomposition of large applications into smaller parts and to allow expressing dependencies between modules statically, the module concept provides an import mechanism, which allows a client module to specify that it is dependent from a service module and to access public elements of the imported module. The import relationship forms a directed acyclic graph (DAG) between client and service modules.

```
module AdvancedCar{

  import EngineControl;
  import BrakeByWire;
  import ...;

  //Giotto/TDL code consisting of sensor, actuator,
  //task and mode declarations.
  //May access public elements of imported modules

}
```

Loading a client module into an E-machine implies loading of all imported service modules unless they have been loaded before. Each of the modules may have its own start mode, thus multiple partitions may be required in order to perform loading of a client module. In this case, however, it is known statically which modules must be loaded due to the static import relationship. Thus, the usage case 'static partitioning' is dealt with by means of module imports.

While it is obvious that using imported constants, types and sensors does not pose any semantic difficulties, it is not a priori clear how to treat constructs such as tasks, modes and actuators.

Multiple applications may read the same sensors, for example, but what happens if multiple applications write to the same actuators? Note that any of the parallel running applications may be in one of several modes and it is not statically defined which actuators are under control of which application at which time. Therefore it must be prevented that multiple applications write to the same actuator. The module construct comes in handy, to solve this problem. We simply restrict actuator update to the module the actuator is declared in. Thus, the module construct acts as a partitioning of the set of actuators. In a large application, sensors could be declared in a common service module, from where they can be used in any client module. A client module declares a subset of the actuators of the complete system and provides the functionality and timing to set their values. Reading the actuator value is permissible by any client module if an actuator is made visible.

Information Hiding

According to popular programming languages we use the keyword 'public' to mark program elements as being publicly visible. There is no need (so far) for a corresponding keyword 'private', as this is the default anyway and there is no further level of visibility.

```
module EngineController {

  public const maxRpm = 6500;

  //... more code

}
```

As mentioned above, package or assembly level visibility of names is not provided by our module concept. There is, however, a simple way of mapping external functionality code to packages (Java) or name spaces (C++, C#). We allow to use structured module names, i.e. module names are allowed to contain '.'. All module name parts up to the rightmost '.' are mapped to packages in Java and namespaces in C++ or C#. Within the TDL module, structured module names are references by using the rightmost name part only. The detailed mapping rules are defined for every individual language mapping and cannot be specified in general. A mapping to ANSI C, for example, might replace the '.' by '_' in order to get unique and valid names for external functions.

Mode Extension

Mode extension is an experimental feature we are currently working on. It means to add or even override activities of a particular mode specified in a separate module. Such a feature may, for example, be useful for hot deployment of new functionality or for fixing errors of a mode without making any changes in the erroneous module. An extended mode inherits the mode period from its base mode.

```
module ExtendedEngineControl {

  import EngineControl;

  actuator int newActuator uses setNewActuator;

  task newTask ...; //provides output variable 'res'

  mode normal extends EngineControl.normal {
    task [1] newTask(...);
    actuator [1] newActuator := newTask.res;
  }

}
```

This example adds a task invocation and an actuator update to mode *normal* of module *EngineControl*. The extensions get into effect only when module *ExtendedEngineControl* is loaded into an E-machine.

A particular problem arises if a mode is extended multiple times. Since all activities of a mode (task invocation, actuator update, and mode switch) must be deterministic, i.e. there must for example be only one mode switch guard that evaluates to true, there must not be an arbitrary set of mode extensions available in a system. Therefore we limit mode extensions to a single extended mode, which may be extended itself by another single mode, thus leading to a sequence of extensions rather than a tree.

Scheduling Issues

In order to preserve the timing behavior of all concurrently executed applications, it is required to adapt the scheduling strategy to this requirement. We are currently experimenting with a simple time sharing strategy based on preemptive scheduling. The basic idea is as follows.

The GCD of all activity periods of all modes of all partitions must be calculated. This is called the 'hyper period' and has the obvious property that no event (task invocation, actuator update, mode switch) happens during this period. The hyper period defines the period of time, which will be shared by all partitions according to their needs. A CPU intensive partition will get a higher percentage of the hyper period than a less CPU intensive partition. The percentage needed for a partition is determined by the most CPU intensive mode of the partition and may be regarded as a fixed slot inside the hyper period. Within the slot of the hyper period assigned to a particular partition, this partition may perform any calculation and it may execute in any mode. Due to the calculated size of the slot, there is litte waste of CPU time. When all partitions execute their most CPU intensive mode, the CPU will be allocated up to 100%.

When loading and scheduling a new partition, it must be checked if the former hyper period needs to be changed because of the activity periods of the new partition will not be a multiple of the old hyper period. If there is a change, all partitions must be

rescheduled for the new hyper period, otherwise it suffices to schedule the new partition only and check if there is a slot within the hyper period available for it which is large enough to execute the most CPU intensive mode of the new partition.

In practice, there are only a few periods commonly in use (e.g. 500, 1000, 10000 usecs) and these tend not to be prime numbers. In case of primes, the hyper period would get as small as 1 time unit, which is 1 microsecond in our implementation. This would produce large scheduling tables and would not allow splitting the hyper period into several slots.

Since scheduling must not be considered to be done in logical zero time, there must be a mechanism to perform scheduling during real time execution of the previously started partitions. This can be done easily by using a thread which is run whenever the ECU would be idle anyway. Only after finishing the scheduling (or rescheduling) there may be an update of the runtime data structures, but this is a very simple step which can be regarded as executing in logical zero time.

Implementation Status

We have implemented a variant of Giotto called TDL by using the compiler generator tool Coco [6]. An experimental runtime system based on Java threads, which are not strictly real time but serve well as a test bed for our architecture, has been implemented. Execution of parallel partitions is possible and scheduling works as described above. In addition we are working on an implementation of our architecture based on industry standard operating systems such as OSEK and OSEK/Time to get a detailed knowledge about what is really possible under these platforms and what is not. We are currently considering to port our Java based E-machine to 'realtime' Java.

4 Distributed TDL Components—Outlook and Related Work

During the course of the development of our modular architecture for control systems, it became clear that modules may serve another purpose, namely as unit of distribution. In an analogy to general-purpose programming languages, a TDL task corresponds to a function and a TDL module corresponds to a module in languages such as Oberon or Ada. A TDL module is supposed to encapsulate the functionality that is put on one ECU in current automotive system designs, such as the all-wheel-drive control system or the engine control system. This implies that TDL modules exhibit weak dependencies on each other, corresponding to weak coupling between modules, whereas one TDL module has a narrow interface and strong cohesion. This means that TDL modules are a perfect choice as units of distribution of functionality across a network of ECUs in case that the CPU or I/O of a single node is not capable of handling all the control tasks. In the following we sketch the development scenario that illustrates how the complexity of distributed system implementation is significantly reduced compared to state-of-the-art approaches.

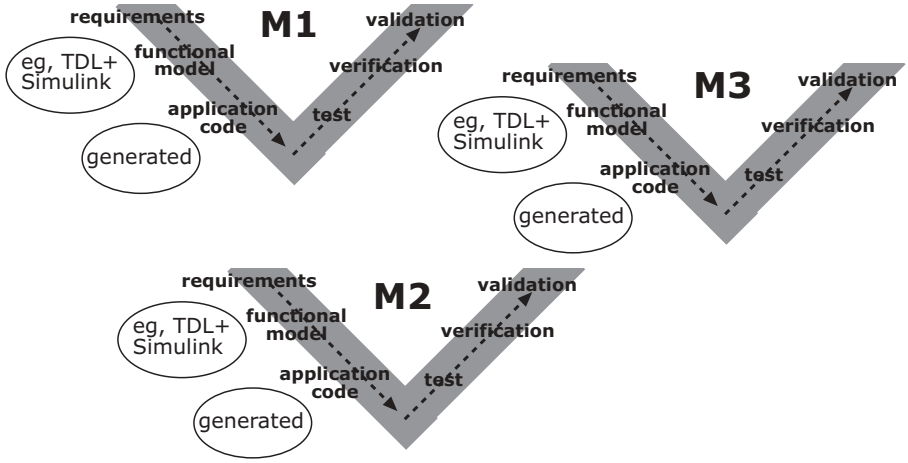

Fig. 2. V-Cluster-Life-Cycle

Application-centric Development

The key benefit of the development methodology that results from the TDL component architecture is that a developer does not have to worry from the beginning whether the overall system is going to be executed on a single ECU or a distributed platform. The distribution of the TDL modules is either generated automatically or specified later in the development process (see below). Figure 2 shows what we call the V-Cluster-Life-Cycle: Modules are developed independently of each other. Each V-Life-Cycle delivers a module. The TDL modules are separate units of compilation. The behavior (timing and functionality) of the modules is unchanged no matter how they are distributed on a specific platform. A time-safety check ensures that the timing requirements can be met.

If the modules should be executed on a distributed platform, the modules have to be assigned to ECUs on the particular platform. We might find heuristics that allow an automatic assignment of modules to ECUs. For example, one aspect that needs to be considered in the distribution is that the network traffic between the ECUs is minimized. Thus modules should be close to their sensors and actuators if possible. The idea is that a tool proposes a distribution of the modules, if one can be found that satisfies the timing requirements. The proposed distribution can then be changed manually if necessary. Basically, the distribution is described in a table like that:

module	@
M1	ECU1
M2	ECU2
M3	ECU1

A visual/interactive editor could more conveniently support the editing of the assignments. The developer could also view the current CPU usage that would be updated according to the modules assigned to one ECU.

Model-based Development

Besides the TDL program the developer has to come up with the functionality code. Figure 3 shows a sample model-based tool chain that assumes that Simulink is used for modeling the functionality (control laws). The Simulink simulation environment can be used to validate the behavior of the modules and their interaction before the code is generated from the models.

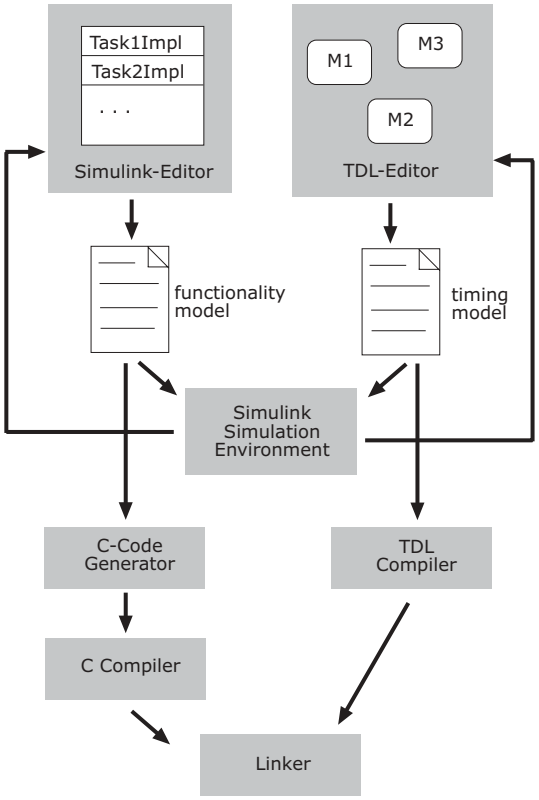

Fig. 3. Model-based development with the Simulink and TDL tool chain.

Software-Bus Abstraction

Besides the TDL program and the functionality code the developer has to come up with getter and setter functions, which copy values from the environment to sensors and from actuators back to the environment. To simplify the implementation of getter and setter functions in the realm of a distributed system, we aim at providing the abstraction of a set of globally available sensors and actuators, which we call a *software bus*. The implementation of the software bus for common networks in the automotive domain, such as (TT-)CAN, FlexRay and the TTA, along with the distribution of modules as sketched above will be our next major steps towards a full-fledged component architecture for control applications.

Related work

Various methods and tools aim at the development of distributed control applications. For example, DaVinci (Vector Informatik) and SysDesign (Cadence) represent the state-of-the-art method and tool support. Both have in common that they help the developer to simulate the behavior of the control system(s) on a distributed platform. The important difference to the TDL component architecture is that these methods do not abstract from the distributed platform so that the developer still has to perform the activities in a platform-centric manner. That is, the developer has to build the application with the selected distributed platform in mind.

Nevertheless, some of the tools could be used to simplify the implementation of the TDL component architecture. For example, DaVinci could provide a suitable implementation of the software bus abstraction. This has to be evaluated. SysDesign could be used to test the finally distributed TDL modules and to validate that the behavior of the simulated TDL program(s) is equivalent to the executables. SysDesign would have to be checked whether it can indeed provide the necessary granularity for such virtual prototypes.

Figure 4 shows the abstraction levels of the various approaches. Besides the platform-centric approach state-of-the-art methods and tools imply a non-deterministic behavior of (composed) control applications. For example, both DaVinci and SysDesign deal with task priorities. The composition of task sets could, for example, result in race conditions. In other words, with such tools the developer has the chance to detect and fix anomalies, hopefully before the system is delivered to the real platform. But the methods and tools do not provide the appropriate abstractions so that a straight-forward, error-free composition is guaranteed.

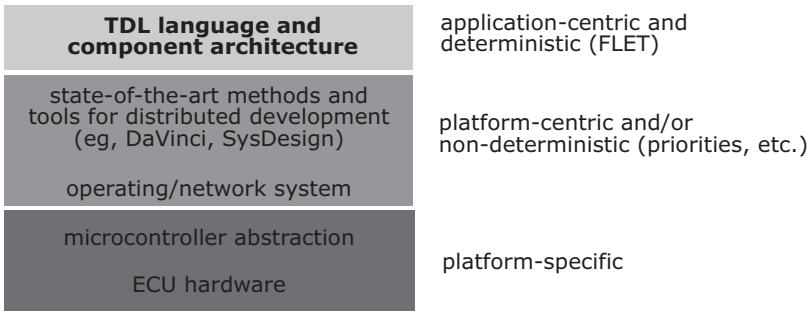

Fig. 4. Abstraction levels for control application development.

5 Acknowledgements

We thank Christoph Kirsch for supporting us in understanding and extending the Giotto ideas. Emilia Coste and Claudiu Farcas provided valuable feedback on the TDL compiler while they worked on a plugin for generating output for an ANSI-C based E-machine. They also helped to shape the scheduling algorithm currently in use in the Java based E-machine. Michael Holzmann, Sebastian Fischmeister, Guido Menkhaus and Gerald Stieglbauer provided valuable input during informal discussions and group meetings.

This research was supported in part by the FIT-IT Embedded Systems grant 807144 provided by the Austrian government through the Bundesminsterium für Verkehr, Innovation und Technologie.

6 References

1. T. Henzinger, C. Kirsch, W. Pree, M. Sanvido: From Control Models to Real-Time Code Using Giotto; IEEE Control Systems Journal, February 2003, Vol. 23 No.1, Special Issue on Software-Enabled Control
2. Web reference: http://www-cad.eecs.berkeley.edu/~fresco/giotto/
3. N. Wirth: Tasks versus threads: An alternative multiprocessing paradigm, *Software-Concepts and Tools*, vol. 17, pp. 6-12, 1996.
4. T.A. Henzinger: Masaccio: A formal model for embedded components, in *Proc. 1st IFIP Int. Conf. Theoretical Computer Science*, LNCS 1872, Springer Verlag, 2000, pp. 549-563.
5. Josef Templ. TDL Specification and Report. Technical report, Software Research Lab, University of Salzburg, Austria, October 2003. http://www.SoftwareResearch.net/site/publications/C055.pdf
6. Mössenböck, H.: Coco/R for Java. http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/Java/