Implementing the Timing Definition Language on Top of the TTP Tool Chain

Master Thesis

Johannes Pletzer
yohap@yohap.com
Supervisor: Univ.-Prof. Dr. Wolfgang Pree

Department of Computer Science
University of Salzburg, Austria

January 2005

Acknowledgements

This thesis would not have been possible without the help of numerous people who helped and encouraged my in the past year.

So many thanks go to:

My supervisor Prof. Dr. Wolfgang Pree for organizing the Emsys conference in Salzburg in July 2003 which brought me to the topic of embedded systems and of course for giving me the opportunity and resources to write this thesis.

Dr. Josef Templ for answering numerous questions concerning details of the TDL language and compiler and final improvement suggestions concerning the thesis.

Emilia Coste for discussing details of the work with me and a lot of critical remarks that brought me ahead.

Michael Holzmann for his continuous support and final proof-readings.

All my friends for their support and their understanding for my lack of time for them in the past months.

Thanks also to TTTech for providing the TTP Development Cluster and the corresponding software tools.

Abstract

The purpose of the thesis is to integrate the Timing Definition Language (TDL) with the Time-Triggered Protocol (TTP) for the development of fault-tolerant distributed real-time systems. TDL is a language for the definition of real-time systems that aims at a separation of the timing and the functionality of real-time applications. TTP is a communication protocol based on the ideas of the Time-Triggered Architecture and is intended for highly dependable distributed real-time systems developed by TTTech. The goal of the integration was to show that it is possible to implement an application written in TDL on the TTP platform with special focus on distribution and fault-tolerance aspects.

For the purpose of the integration of TDL and the TTP development tools a set of tools was designed and implemented. The core of it is a plugin for the existing TDL compiler that transforms TDL source code into input for TTP tools in order to finally get executable binaries suitable for the TTP hard- and software platform. This required a detailed analysis of how to map the TDL constructs to the TTP tools. In order to support fault-tolerance and distribution, additional specification was provided in a separate file.

The applicability of the work is proved by means of a simple demo application that uses the developed tool chain and incorporates distribution and fault-tolerance features. As hardware platform a TTP cluster provided by TTTech was used. The results show the feasibility of the ideas behind the TDL plugin and prove that they work at least for simple applications.

Table of Contents

List of Figures

Chapter 1

Thesis Overview

This chapter contains an overview of the thesis. It includes an introduction to the context and problem field and points out the goals and results of the work. Finally the overall structure of the thesis is outlined.

1.1 Introduction and Motivation

The purpose of the integration of the Timing Definition Language (TDL) with the Time-Triggered Protocol (TTP) tools and the case study is to explore the advantages and limitations of TDL on top of a specific time-triggered platform. Furthermore, the harnessing of platform-specific fault-tolerance in the context of the platform-independent timing and communication specification in TDL should be evaluated. The goal was to seamlessly integrate TDL and TTP by means of a TDL compiler plugin that processes TDL modules and interfaces with the tools provided by TTTech for the development of TTP applications in order to generate code for the TTP hardware platform. The TTP protocol with the hardware and software tools for its development is a suitable architecture for this purpose because the protocol already includes services such as distributed clock synchronization, membership service and fault tolerance mechanisms. The goal was to use these services and tools as much as possible.

In order to demonstrate the distribution and fault-tolerance aspects we use a simple demo application which was originally provided by TTTech to demonstrate the functioning and usage of their tools. The idea was to recreate the application as a TDL program and to use a compiler plugin to generate suitable code for the TTP platform. The plugin generates scripts for the two main tools of the TTP tool suite, namely TTPplan for cluster-level design and TTPbuild for node-level design. Via these scripts the TDL timing definition is transformed into a valid input for the TTTech tools. In order to support platform-specific distribution and fault-tolerance aspects such as replication and redundancy, a separate annotation file is used to specify these properties and some hardware specific variables for the TDL compiler.

1.2 Results

It proved to be possible to realize the timing and functionality TDL specifies on the TTP platform using the tools TTTech provides. However it turned out that not every TDL construct can be mapped to the TTP platform due to limitations of the tools or the nature of the underlying TTP communication protocol.

Another result was that the integration of distribution and fault tolerance aspects in TDL works when using a TDL module as unit of distribution and replication. A simple demo application showed the feasibility of the suggested specifications and mechanisms.

The thesis also contains a comparison of the development process that points out the differences between using the TTP tools by TTTech and the TDL language.

1.3 Thesis Structure

Chapter 2 provides an introduction to the basics that need to be known to be able to understand the work presented in the subsequent chapters. Although the first section briefly covers the field of distributed real-time systems, probably a more in-depth knowledge of it is required to fully understand all aspects presented below. The book written by Kopetz [1] is a recommended source to gain such knowledge.

Chapter 3 presents the design and implementation of the plugin for the integration of TDL and the TTP tools. The developed tool chain is explained and the relevant details of the process are covered. This includes a description of the programming interface of the TTP tools, the plugin interface of the TDL compiler and fault tolerance aspects.

Chapter 4 presents a case study that relies on the TDL compiler. The simple demo application illustrates the capabilities of the plugin especially by using fault tolerance mechanisms. The chapter covers the complete TDL-TTP tool chain to generate a working application out of TDL modules.

Finally Chapter 5 summarizes the results of the thesis and discusses the limitations and restrictions of the TDL compiler plugin and the tool chain. A comparison of the development of distributed real-time applications with the TTP tools and TDL concludes the thesis.

Chapter 2

Basics

The purpose of this chapter is to introduce the reader to a variety of terms and technologies that need to be known in order to be able to understand the work presented in the subsequent chapters. The aim is on one hand to give an overview and on the other hand to explain relevant aspects regarding the thesis with more detail.

2.1 Distributed Real-Time Systems

A brief overview of what real-time systems are all about is presented below. Typical applications are mentioned and ways to classify such systems are presented. A special focus is on distributed real-time systems regarding their additional problems and advantages they have in comparison to non-distributed ones.

2.1.1 Real-Time Systems

A computer system is called a real-time computer system when it is not only required that the system produces correct output values based on its inputs, but also to perform this calculations in a bounded time interval. The instant when a calculation must be finished and a value must be produced is called the deadline.

A distinction of real-time systems can be made based on whether the deadlines of a system are *soft* or *hard*. In a soft real-time system the occasional missing of a deadline results in degraded quality of service. An example would be a video player application. A missed deadline might result in a small playback error that may even not be noticed by the user. The more deadlines are missed the poorer the quality of the system gets. Opposed to this example a missed deadline in a hard real-time system can cause catastrophic effects in systems such as automotive engine control or aerospace applications. In the worst case scenario the motor gets damaged or the airplane gets into an unstable state and crashes.

Event-Triggered versus Time-Triggered

Another very important classification of real-time systems is whether they use an *event*-triggered or *time*-triggered approach. A trigger is a mechanism that initiates a specific activity, e.g. the execution of a computation task or the communication of a message. An event-triggered system reacts to events such as the change of a sensor value immediately and for example starts the execution of a task that processes the value. This is typically realized with the usage of an interrupt associated with the event source. In contrast to that in a time-triggered system all activities are initiated periodically by the progression of time. Activities such as sensors readings, task invocations and communication activities only happen at pre-defined periodic time instances. The only interrupt in such a system is the timer interrupt issued by the system clock.

One obvious advantage of the event-triggered approach is the potentially smaller latency between the occurrence of an event and the reaction to it. A change in the state of a sensor is detected immediately and not only when it is scheduled to be read like in a time-triggered system. But this immediate reaction also turns out to be a huge disadvantage when a lot of events (and consequently interrupts) occur almost at once. This can cause a missed deadline as the processor of the system has only finite computation capacity and therefore cannot handle all interrupts and activities that are triggered by them in parallel. The problem is that in hard real-time systems it must be proven that deadlines are never missed, even in such a worst-case scenario. It is much more straight-forward to prove that for a time-triggered system because they are more predictable as every activity is pre-planned.

2.1.2 Distribution

A distributed real-time system consists of a number of nodes and a communication system. Naturally the latter is of great importance as the communication between nodes usually is vital for the distributed system to perform the functions it is intended for. The interface between the host computer and the communication controller inside a node is called the communication-network interface (CNI). The CNI is a way of hiding from the node how the communication actually takes place. It can be designed in many different ways. An important design decision is whether the communication is controlled by the senders and receivers of messages or if the communication system handles the transmission of messages autonomously. The first is called the event message concept where the sender sends a message when an event occurs and that message is delivered via the communication system to the receiver immediately. Here the control when a message is sent is in the sphere of control of the host computer. This concept requires a one-to-one synchronization between communication partners as otherwise queues will overflow at the receiver or the sender may be blocked. In contrast, when using the so-called state message concept the communication system is in control concerning the instance of time when messages are transmitted. The sender may update the state message independently of the receiver and the receiver may read the message many times or not at all. The CNI for such a communication system typically is implemented by means of a dual-ported RAM that decouples the host computers from the communication system. This solution avoids that control signals pass the CNI, meaning that a host is not allowed to directly control what and when something is transferred on the bus. This leads to a looser coupling between the communication partners as one-to-one synchronization is not needed.

Arguments for Distribution

According to [1] there are four major reasons to choose a distributed solution for real-time systems:

- Composability

  Composability enables developers to develop and test subsystems independently and finally compose them to form a distributed system instead of a monolithic single system. It must be guaranteed that the properties of every subsystem are not invalidated by the system integration.

- Scalability

  For a scalable system it is important to avoid having a central bottleneck that limits extensibility. By adding new nodes and communication gateways additional processing power and communication bandwidth can be added almost without limit. Another argument is the cost of silicon. Because the cost of manufacturing a chip is proportional approximately to the third power of its die area, it pays off to use a larger number of smaller chips despite the fact that a distributed solution usually requires more hardware than a centralized architecture.

- Dependability

  In distributed architectures it is easier to establish so-called error-containment regions. It is possible to detect an error in a single or multiple nodes and protect the rest of the system from corruption. Furthermore in a distributed system node replication may be used in order to be able to tolerate failures of nodes.

- Physical Installation

  According to system developers it proved to be intelligent to integrate the hard and software that controls a device, in particular a sensor or an actuator with the device itself, resulting in increased reliability. A system that uses such devices can be viewed as a distributed system.

2.2 Fault Tolerance

This section is dedicated to fault tolerance in real-time systems as it represents an important aspect in the thesis. We introduce the concept of fault tolerance and the various possibilities how to achieve it. The section also covers the detection of errors and design strategies for highly dependable systems.

Fault, Error and Failure

First it is important to differentiate between the terms fault, error and failure as proposed in [3] and [1]. A failure is an event that describes the inability of a system to provide the specified or intended service. Failures are almost always consequences of an unintended or incorrect internal state of a system which is called an error. The cause of an error is called a fault. An example for a fault would be a defect memory cell. Such a fault does not necessarily lead to an error, because the system might not even use the specific cell. But when it does we have an unintended state of the system, and therefore an error, as the data element that is written to memory cannot be obtained correctly again. Of course such an error will probably lead to a behavior of the system that does not comply with its specifications, for example an incorrect calculation or output. This example shows that a fault may not necessarily cause an error and an error may not cause a failure. The reverse, however, is true. A failure always is the consequence of an error that is caused by a fault.

Fault Tolerance

The purpose of fault tolerance is to break the chain of events that lead from fault to failure. Faults cannot be completely avoided as it is not possible to build hardware

units that never fail. The idea of fault tolerance is to detect errors and mask or repair them before the service delivered by the system suffers and therefore a failure occurs. So the detection of errors is a key issue in creating a fault tolerant system, because undetected errors normally lead to failures.

There are two basic strategies for error detection:

- Detection based on a priori knowledge

  A priori knowledge can be knowledge of the code space as used with cyclic redundancy checks (CRC), activation patterns of computations or any other regularity in the temporal or value domain that can be compared to the actual behavior of the system.

- Detection based on redundant computations

  Redundant computation is possible in various ways: Time redundancy means to execute the same software multiple times on the same hardware, whereas when applying hardware redundancy it is executed on two independent hardware channels. Another possibility is design diversity where different software implementations are used on either the same or on diverse hardware.

After an error is detected, the system has to recover from it and reach an error-free state again. In addition the propagation of the error must be avoided. Next is the phase of fault treatment. If a transient fault, which is a fault that appears once and disappears by itself, occurred, no treatment is necessary. A permanent fault of a hardware device will require its repair or replacement.

Fault-Tolerant Units

A fault-tolerant unit (FTU) is formed by a collection of nodes. A node is a self-contained unit that provides some functionality. An FTU is able to mask the failure of a node. How many nodes are needed for an FTU depends on the type of failure they produce. According to [1] the following three different failure modes are distinguished:

- Fail-silent nodes

  Fail-silence means that a node either produces a correct result or produces no result at all. This is the optimal case as then an FTU consists of only two identical nodes. Both nodes get the same input and either produce two or one correct result when the FTU is operational. To guarantee fail-silence every node must be designed in a way so that wrong results are detected and the node does not output them.

- Triple-modular redundancy (TMR)

  In this configuration fail-silence is not provided by the nodes. The failure mode when a node might produce a wrong output is called fail-consistent. In order to tolerate such a node failure three nodes are necessary. In addition to that a so-called voter is required that compares the results from the three nodes and selects the one that has been computed by the majority, which in our case is two out of three nodes.

- Byzantine resilient FTU

  To be able to tolerate a Byzantine failure of a node the FTU must consist of at least four nodes. A Byzantine or malicious failure occurs when a node shows contradictory faces of a failure to each operational node. In order to ensure that four nodes are sufficient to tolerate such failures, additional requirements concerning communication paths and time synchronization have to be fulfilled. Every node needs to be connected to all other nodes of the FTU by two disjoint communication paths. Before the malicious node can be detected, at least two communication rounds need to be executed where every node sends a broadcast message. In addition, the clocks of all nodes need to be synchronized with a known precision.

Those three FTU scenarios show that it pays off to design nodes to be fail-silent or at least fail-consistent, as this will be cheaper than having such a high number of replicated nodes in most cases.

Systematic versus Application-Specific Fault Tolerance

There are two basic options in making a system fault tolerant. The systematic approach implements fault tolerance mechanisms transparent to the application software. This means that the application code does not need to be modified and the application also is not aware that any fault-tolerance mechanisms are employed. Typically this is realized by the replication of hardware units that run the original application redundantly. The advantage of this approach is that those mechanisms can be developed and tested independently from the application code and that it avoids making the application more complex and therefore more prone to errors. The major downside is that typically more hardware is needed for its implementation. Application-specific fault tolerance requires modifying the application by integrating error detection and fault tolerance functions on the application level. This results in lower hardware costs at the expense of higher design and testing efforts in application development. Since both approaches have their advantages often both are used together in practice.

2.3 Giotto and the Timing Definition Language (TDL)

In this section Giotto is introduced as a language for embedded programming. Basic Giotto concepts, in particular the fixed logical execution time (FLET) and the E machine are discussed. The Timing Definition Language (TDL) that was used in the realm of this thesis is conceptually based on Giotto and was developed to provide improved syntax and programming tools.

2.3.1 Basic Giotto Concepts

Giotto provides a programmer's abstraction for the development of hard real-time systems. It follows the time-triggered approach and is no real programming language but rather a tool that lets the programmer specify the timing and communication behavior of an application. One goal of Giotto is to contribute to a better modularization of control software by separating the timing and communication specification from the functionality implementation. Furthermore, the timing and communication specification are separated from physical realization concerns such as

hardware requirements and scheduling. This allows the developer to specify the timing and communication behavior in a platform-independent way. According to [4] the Giotto-based development of control systems is performed in three stages:

- Control design

  This step consists of typical design efforts required for real-time control systems, in particular the plant modeling and control law definition.

- Giotto program

  Based on the previous step the timing and communication behavior of the application is modeled. This most importantly includes the specification of periodic software tasks and mode switches. A mode in Giotto is a collection of concurrently executed periodic tasks that represent an operational mode or state of a real-time application. A mode switch is a condition which triggers the change of the current mode.

- Code for a specific real-time platform

  When the target platform is fixed, the application is mapped to a specific hardware and operating system. Also a computation schedule for the tasks on a node and for communication must be calculated. It might happen that the outcome of this step is that the desired target platform does not satisfy the requirements of the application.

Synchronous and Asynchronous Language Constructs

The Giotto language contains synchronous as well as asynchronous constructs. Apart from task and modes mentioned above, Giotto uses another important abstraction called drivers. Drivers contain code for sensors and actuators which interact with the physical world. The Giotto abstraction assumes that a driver is executed in logically zero time and therefore is a synchronous construct. A task however consumes a non-negligible amount of CPU time concerning the Giotto programming model, thus being an asynchronous construct. It is not allowed to set actuators or to read sensors within task code, as this would be a violation of the programming model.

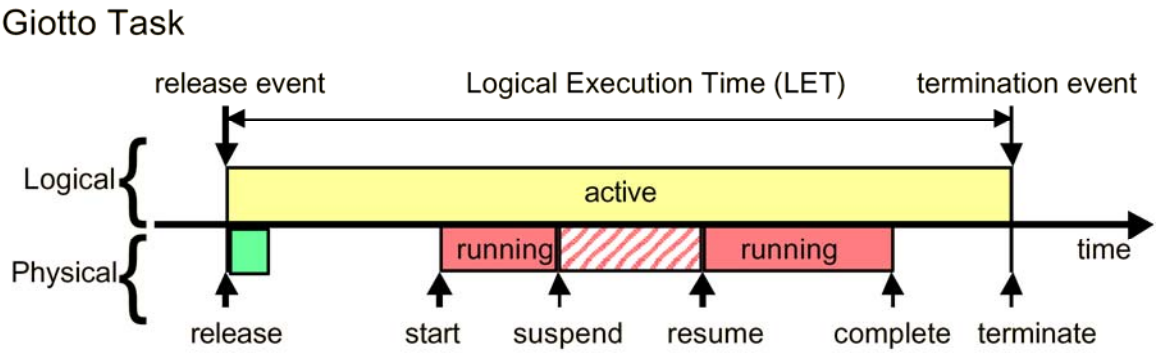

Figure 1 Giotto Task Model

Figure 1 illustrates the task model used in Giotto. Logically a task is considered active during its whole period, although typically it only takes a fraction of that time to actually execute it. Physically, tasks can be executed at any time by the scheduler of the system and also might be preempted as shown in the figure above, as long as

their execution is finished at the end of their execution interval. This interval is as long as the period of the task and is called the fixed logical execution time (FLET) or also LET like in the figure above. Drivers are executed at the beginning of FLET in logically zero time, but physically they do consume a rather small amount of CPU time, indicated by the green block in the figure.

Tasks communicate with other tasks and with sensors and actuators only via so-called ports. Therefore ports can be seen as an interface that connects all entities in the Giotto programming language. Ports are read and updated in a strictly periodic and time-triggered way only at the beginning and end of the fixed logical execution time of a task.

The consequence of the Giotto programming abstraction, especially of the FLET concept, is a platform-independent description of the timing behavior of real-time applications.

E Machine

The embedded machine implements the Giotto timing on a specific hardware platform. It executes drivers and passes the tasks to the scheduler. It is an interpreter for the so-called embedded code (E code) that is generated by the Giotto compiler and contains the timing specification of the Giotto program. Because of the simple instruction set of the E machine it is quite easy to port the E machine to different platforms. The instruction set contains among others the following three instructions [2]:

- *Call* driver instruction

  This instruction executes a driver. It is blocked which means that the E-Machine waits until it is finished before it continues to execute the next instruction.

- *Schedule* task instruction

  The schedule instruction hands a task to the scheduler of the operating system. The E machine does not schedule the task but just requests that it is scheduled when CPU time is available. The exact time of execution is not controlled by the E machine but by the scheduler of the operating system and the scheduling algorithm that is actually employed. So this may result in different patterns of task invocation on different platforms, but as long as every task finishes inside its FLET interval, the timing behavior of the application remains unchanged. It might happen that for various reasons this is not possible and a deadline violation occurs. This can be avoided by a compiler that checks for time safety on the platform concerned.

- *Future* instruction

  The future instruction marks a block of E code for later execution.

An in-depth description of Giotto and the E-Machine can be found in [2] and [4].

2.3.2 Timing Definition Language (TDL)

TDL, as described in [9], is based on the Giotto concepts, but offers a more streamlined syntax. Above all, TDL adds the module construct.

A module packages multiple modes with their tasks, sensors and actuators together. It enables the decomposition of a real-time application into smaller software components and – as we will see later – simplifies distribution and the introduction of fault tolerance.

Sample Module

The following example shows a TDL module that realizes a simple light controller that controls a light with respect to a brightness value from a sensor.

```
module lightController {

  sensor
    int brightness uses getBrightness;

  actuator
    int light uses setLight;

  public task calc [100us] {
    input
       int brightnessValue;
    output
       int lightValue := 0;
    uses calcImpl(brightnessValue, lightValue);
  }

  start mode controlLight [4000us] {
    task
      [1] calc(brightness);
    actuator
      [1] light := calc.lightValue;
  }
}
```

The code starts with a definition of all sensors and actuators that will later be used. The type `int` is an internal type of TDL, but there is also the possibility to define custom types. A sensor or actuator declaration consists of an identifier and a name of a function that implements the functionality. This function can be provided in any programming language. Next comes a task definition including a specification of the worst-case execution time (WCET), which is 100 microseconds here, the tasks input and output ports and again a function `calcImpl` that implements the functionality of the task. Finally a mode is defined with its period, which is 4000 microseconds here, and invocations for tasks and actuators. Note that the sensor value is passed as a parameter to the task as it has exactly one input port. The number in brackets indicates the frequency of a task or actuator, i.e. how often it is invoked in one period. The fixed logical execution time (FLET) of a task or an actuator is defined by the mode period divided by the task or actuator frequency.

A more detailed discussion of this example, especially concerning the integration with functionality code, can be found in 3.5.3.

2.4 Time-Triggered Protocol (TTP)

An introduction to the Time-Triggered Protocol (TTP) developed at the TU Vienna is given in this section. TTP is designed for fault-tolerant communication between nodes in a distributed real-time system and provides services such as time synchronization and membership service. Apart from explaining the protocol the section also presents the tools and tool chain that are provided by TTTech for the development of applications with TTP.

2.4.1 TTP Protocol

TTP is – as the name implies – a communication protocol that works in a time-triggered fashion. There are two different variants of TTP: TTP/A is a so-called field bus that is designed as a low-cost protocol for the connection of intelligent sensors and actuators to a node. TTP/C is more complex and provides additional services such as redundancy management and a more sophisticated membership service. It is intended for the fault-tolerant connection of nodes of a distributed real-time system. In this thesis the term TTP always refers to the TTP/C protocol.

TTP is a time-division-multiple-access (TDMA) protocol. TDMA is a commonly used access strategy for communication busses. Time-division-multiple-access means that a common media is shared by giving exclusive access to the media to one sender at a time as illustrated in Figure 2. When using the Ethernet protocol for example, it is

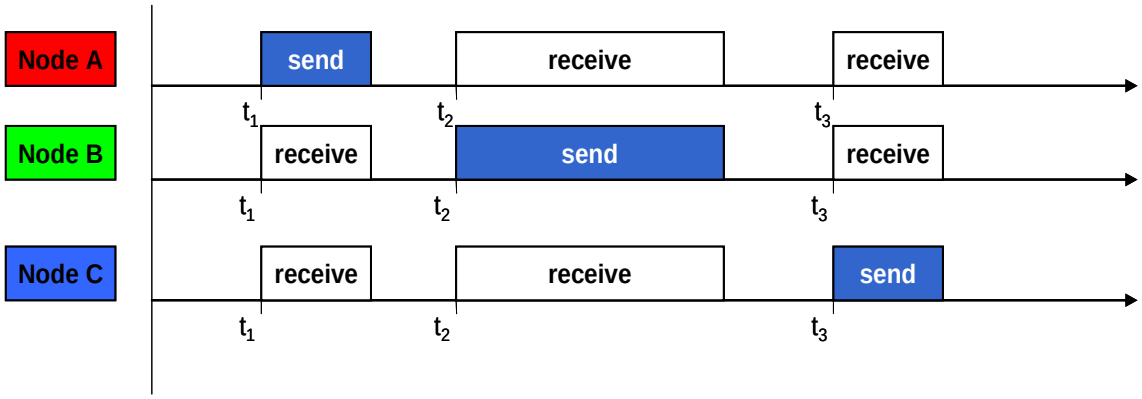

Figure 2 Time-division-multiple-access strategy

not defined when a specific sender is allowed to access the media at runtime and therefore it can happen that multiple senders send at the same time which results in distorted und unusable signals. In such a scenario these collisions must be detected and avoided, which is handled by the carrier-sense multiple access with collision detection (CSMA/CD) strategy of the Ethernet protocol. In TTP such mechanisms are not necessary since the assignment of sending slots for each node is done at design time. This seems very impractical at first and indeed limits the possibilities of the protocol, but when taking into account that typical distributed real-time systems are

not required to be very flexible and mostly perform periodic tasks, this limitation is feasible. At the expense of flexibility TTP delivers guaranteed bandwidth and features for high dependability such as bus guardians. A so-called bus guardian is an independent hardware device that allows access to the bus for the communication controller of a node only at the exact interval it is allowed to send according to the bus schedule, as illustrated in Figure 3. It protects the bus from "babbling idiot" failures where a node keeps "talking" outside its sending slot and prevents other nodes from communicating.

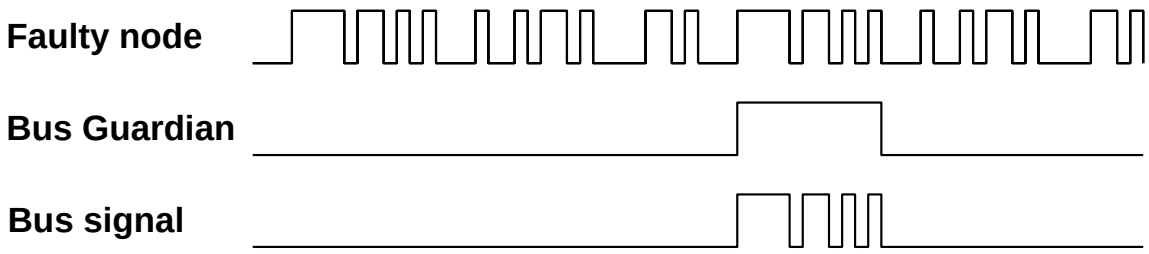

Figure 3 Bus Guardian Operation

The TTP protocol uses a number of constructs und concepts in order to realize fast and reliable real-time communication between nodes. In the following those entities are described as well as the relations between them.

Frame

A frame carries 1 to 240 bytes of user data in addition to protocol overhead such as header and CRC information. The TTP protocol is not aware of what kind of data or messages are contained in a frame. The contents of a frame have to be specified on application level consistently throughout the system. Frames are delimited by inter frame gaps that are needed by the TTP controller in order to distinguish between frames and to perform calculations within the gap duration.

Slot

A frame is transmitted within a slot. A slot is a time interval in which only the node to which the slot is assigned is allowed to send data on the TTP bus. The TTP bus always provides two independent lines for communication which are called channels. The frames sent on both channels do not need to be the same. A frame can either be transmitted on only one channel to maximize throughput or on both to maximize dependability by redundancy. Typically every slot is assigned to a single node, but there also exists the possibility to assign a slot to a group of nodes, which is called multiplexing.

Every node is required to send a frame at the beginning of its slot. The exact instance of this transmission is used to calculate the clock difference of every node. This difference is determined by every node of the cluster and transmitted to the other nodes in its sending slot. Every node continuously corrects its clock on basis of this information. The fact that every node is required to send a frame inside its slot is also used to determine whether a node is still working properly, which is information needed to provide the membership service. Again, every node sends its view of the current state of all other nodes and so a consistent view of the membership in the cluster is established. These two examples show that the strict and static nature of

the protocol has clear advantages in reducing the required overhead for services such as clock synchronization and membership service.

TDMA Round

A TDMA round is a sequence of multiple slots which might differ in their length. It is important to note that the length of the TDMA round and the slots it consists of are statically defined at design time and cannot be changed at runtime. As the TDMA round is repeated over and over again, it defines the basic communication pattern of the protocol and consequently the share of the total transmission time each node gets. The reason for this restriction with respect to flexibility is to keep the bus guardian as simple as possible.

Cluster Cycle

A TTP cluster cycle consists of multiple TDMA rounds as indicated in Figure 4. It can be seen as top-level construct that represents a cluster mode and is repeated all the time. As can be seen in the figure, the frames that are sent in a TDMA round and consequently the messages contained in them can differ throughout the cluster cycle.

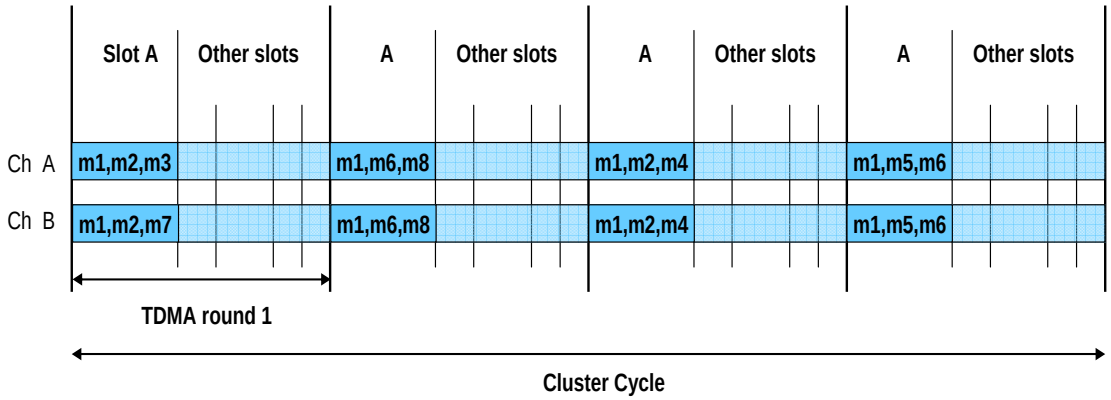

Figure 4 TTP Cluster Cycle

As the TTP protocol is not aware of messages inside frames the TDMA slots differ from each other regarding their length. When multiplexing is used, a slot is shared by a collection of nodes. In this case it is required that every slot is assigned to a node and every node sends periodically. So when for example three nodes share a slot in a cluster cycle that consists of four TDMA rounds, a valid assignment would be if one node gets slot 1 and 3, one node gets slot 2 and one node gets slot 4.

MEDL

All the information of "who sends what at what time" is stored in a data structure called the message descriptor list (MEDL). It can be distinguished between an abstract MEDL, which represents a system-wide model of the communication pattern of the bus, and a personalized MEDL derived from it that is unique for every node of the cluster. The latter contains node-local information, such as the serial number of the corresponding node, in addition to the bus schedule including information on the TDMA round, cluster cycle and different cluster modes. The personalized MEDL is stored in the memory of the TTP communication controller of each node which

handles the transmission of messages on the bus independently from the host computer.

2.4.2 TTP Tool Chain by TTTech

TTTech provides two main tools for application development for their hardware in order to realize the so-called two-level design approach which consists of the cluster and the node level design. The idea is that the system integrator knows about all functions and therefore about the bus messages that are needed for them. The cluster design specifies the interfaces between the nodes and the cluster in both the value and time domain. After this is done, the outcome can be passed on to various sub-manufacturers which design specific nodes of the cluster. The composability of the nodes is guaranteed because the bus schedule is already generated in the first design level. The reason for having those two levels in the design of the system is the development process and requirements that are found in the automotive industry, especially between car manufacturers and their suppliers. Typically, a car manufacturer plays the role of a system integrator by hiring different component suppliers to deliver certain subsystems. The clear separation of concerns inherent to the two-level design approach leads to defined responsibilities of all parties involved while reducing the risk of integration. Another important benefit is the possibility for the subsystem manufacturers to hide the information of how the components are actually implemented, as the system integrator does not need to know this for a successful integration and operation of the whole system.

According to TTTech the following distinct steps have to be taken in order to design a distributed and fault tolerant real-time application:

- Application design (including control algorithms)
- Communication and fault tolerance requirements
- TTP/C cluster schedule design
- Implementation
- Test/verification

In the following the main tools of the TTP tool suite for the development of applications based on the TTP protocol are described.

TTPplan

TTPplan is the cluster-level development tool for designing the bus schedule of the cluster. Every message which is sent from every node has to be specified and an automatic scheduler then generates a bus schedule for these requirements. Also fault tolerance properties such as replication, reintegration, and redundancy must be specified at this development stage. The outcome of TTPplan is a cluster database with a cluster schedule and a MEDL (message descriptor list) for the TTP chip of each node which contains the cluster schedule. The MEDL specifies exactly when a node is allowed to send which message and when messages from the other nodes can be received, so every MEDL contains the cluster schedule from the view of a specific node.

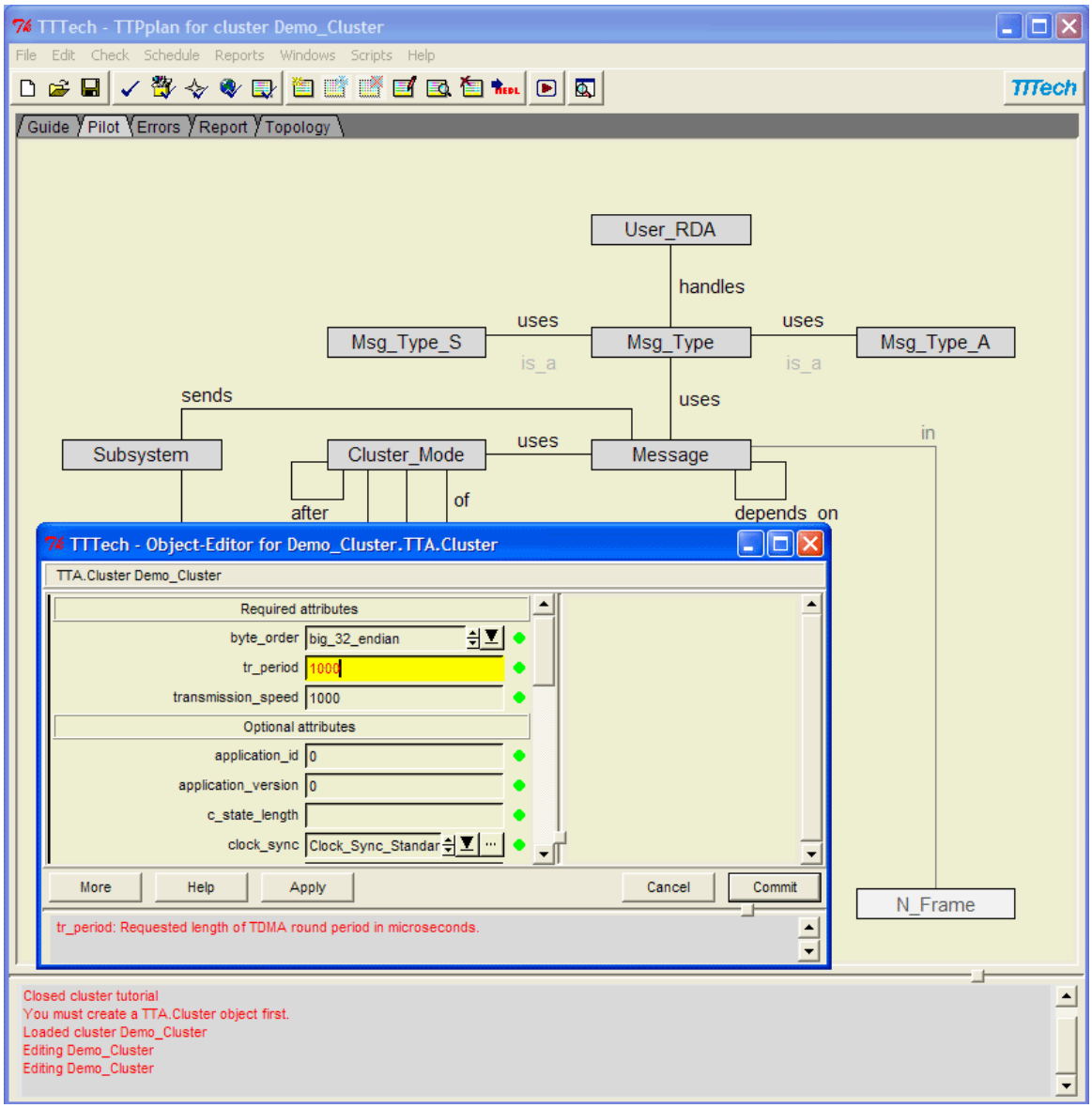

Figure 5 TTPplan Screenshot

TTPplan uses an object model to represent the TTP cluster and all objects it consists of such as hosts and messages. The graphical user interface lets the user create and modify this model by either using a "Step-by-Step Guide" that guides the user through the process to create an application in about ten steps. Another possibility is to directly modify the object model in the "Pilot" view. Figure 5 illustrates this view with a screenshot of the user interface with the object model in the background and a dialog for modifying objects in the foreground. Below the menu and the toolbar there are a number of tabs that allow switching between various views, including the guide mentioned above, the pilot view and a view where errors are displayed. The window in the foreground displays the required and optional attributes for the "Cluster" object. The current value of the attributes is displayed and can be changed. The bottom of the window contains help on the selected attribute, whereas on the right part error information is displayed in case improper input is detected. At the

very bottom of the main window status information is displayed on the current operation the tool executes.

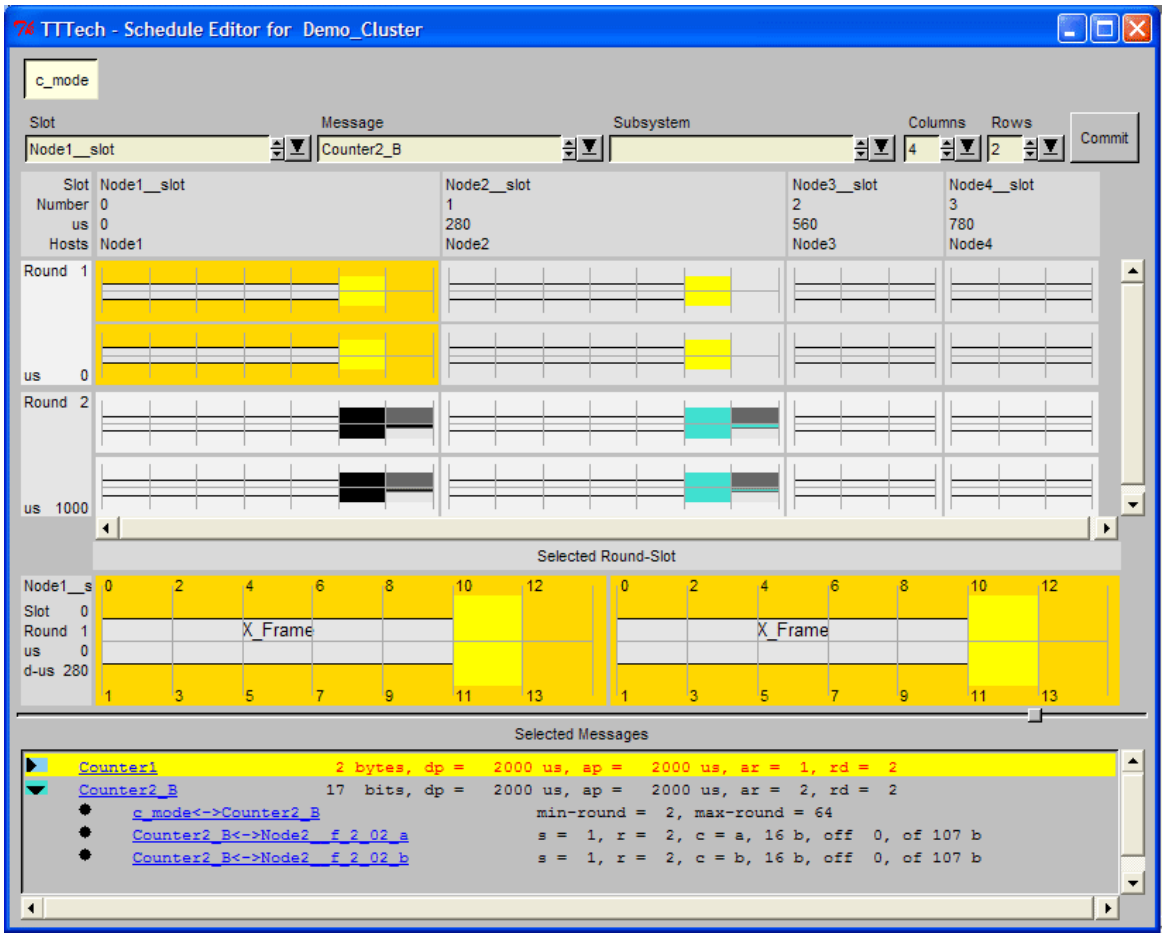

Figure 6 TTPplan Schedule Editor Screenshot

After the object model is checked for validity and consistency, the cluster schedule generation can be invoked. As Figure 6 illustrates the schedule and all messages it contains can be viewed with a lot of detailed information. The cluster schedule is visualized by means of colored blocks that symbolize messages. The area marked with the darker yellow (`Node1_Slot` in Round 1) illustrates what a host sends in its slot. The selected slot is magnified below in the area titled "Selected Round-Slot". It is possible to select a single message and view detailed information on it such as its name and period at the bottom of the window. The interface also offers the user the ability to alter the schedule by drag and drop. As a final step the MEDLs for every host of the cluster is generated and written to file.

TTPbuild

TTPbuild requires a cluster database created by TTPplan that contains the object model and also the cluster schedule before it can be used to design a single node of the cluster. With TTPbuild the user can specify every periodic task that should run on a node and the messages it consumes and produces. The messages that are sent by a node on the bus were already specified and cannot be changed in this development stage following the two-level design approach. For every task a so-called time budget must be given, which consists of the worst-case execution time (WCET) of a task plus some additional overhead needed by the operating system. Just like in TTPplan a "Step-by-Step Guide" is available to be guided through the creation of a valid host object model. Figure 7 shows the user interface of TTPbuild, viewing the object model of a node. The interface is very similar to that of TTPplan, which was described above. The object model with the attributes of the objects can be viewed and edited in the exact same way.

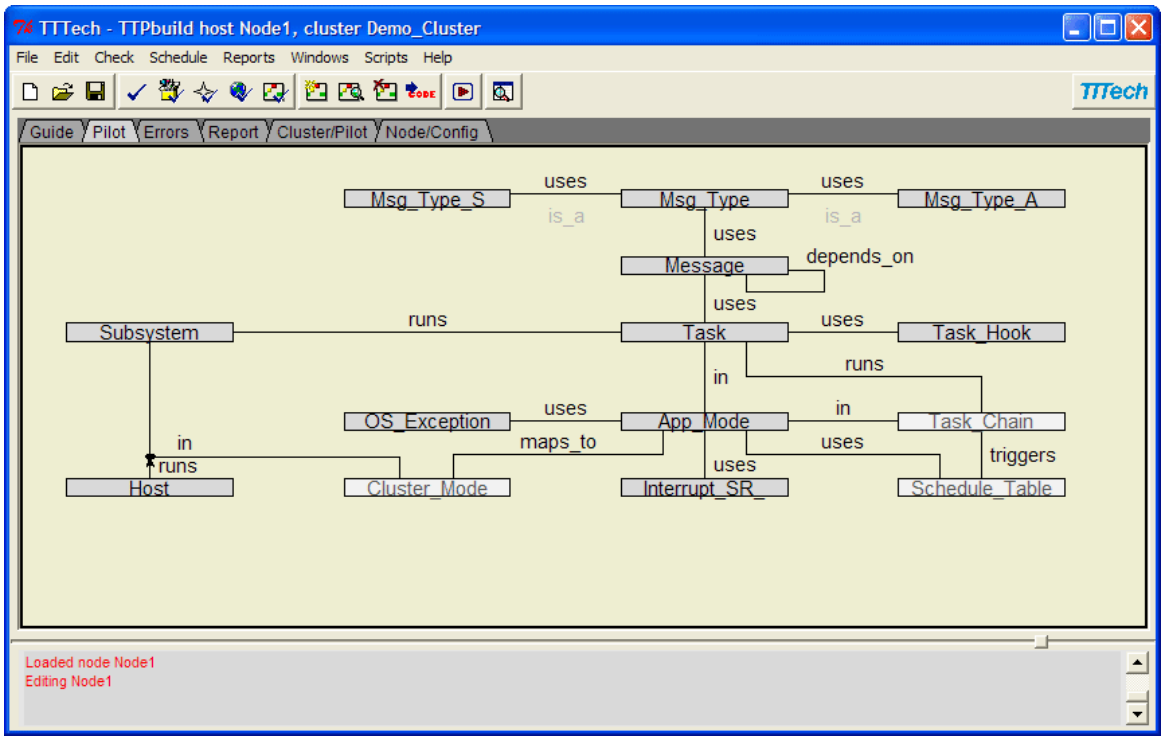

Figure 7 TTPbuild Screenshot

After a valid and consistent object model was created a task schedule can be generated. On basis of the cluster schedule, the time budget of the tasks and information on the target hardware TTPbuild tries to create a feasible task schedule for the node. As illustrated in Figure 8 the task schedule can be viewed to see at what exact time every task is executed. The numbers before the name of the task chains indicate the time of invocation in micro seconds relative to the beginning of the application mode period. TTPbuild combines a number of tasks to a task chain in order to reduce the overhead caused by switching between tasks. When expanding a task chain by clicking on the black triangle, all tasks of which the chain consist are displayed together with information on their time budget and deadline. By further expanding all messages a task sends on cluster and node level are shown. TTPbuild

displays all tasks that will actually be executed on a node, including the ones the tool adds automatically to execute functions that perform the transmission of messages on the bus and clock synchronization.

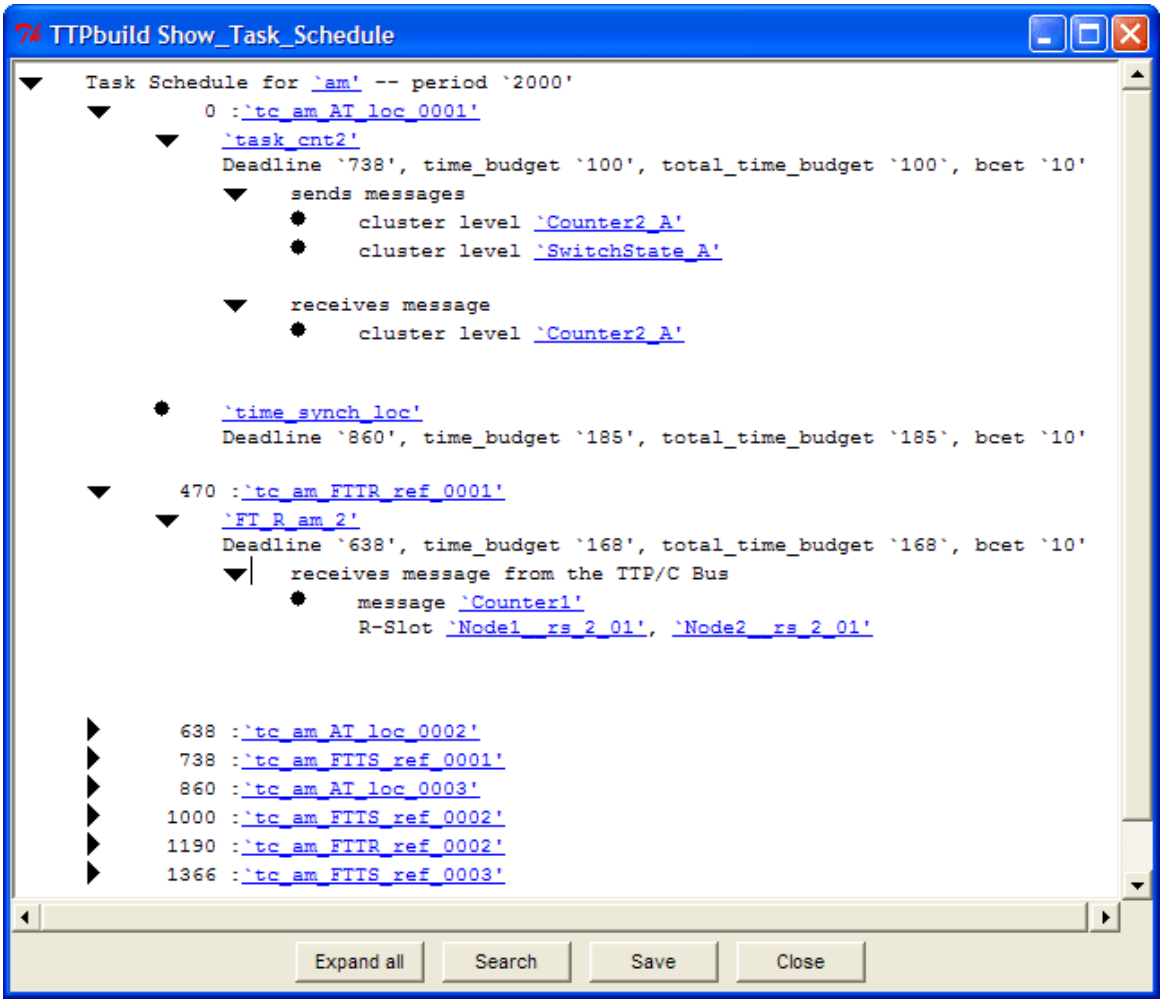

Figure 8 TTPbuild Node Schedule Viewer

TTPbuild generates a static schedule table with all user defined tasks and with system tasks used for time synchronization and the FT-Com layer. The fault-tolerant communication layer (FT-Com) is automatically generated C code for handling the reception and transmission of typically redundant or replicated messages. Another outcome of the tool is an OS configuration file for the TTTech operating system TTPos in which the schedule is specified. All the developer has to do then to get a working binary for the target platform is to provide the task functionality code in C.

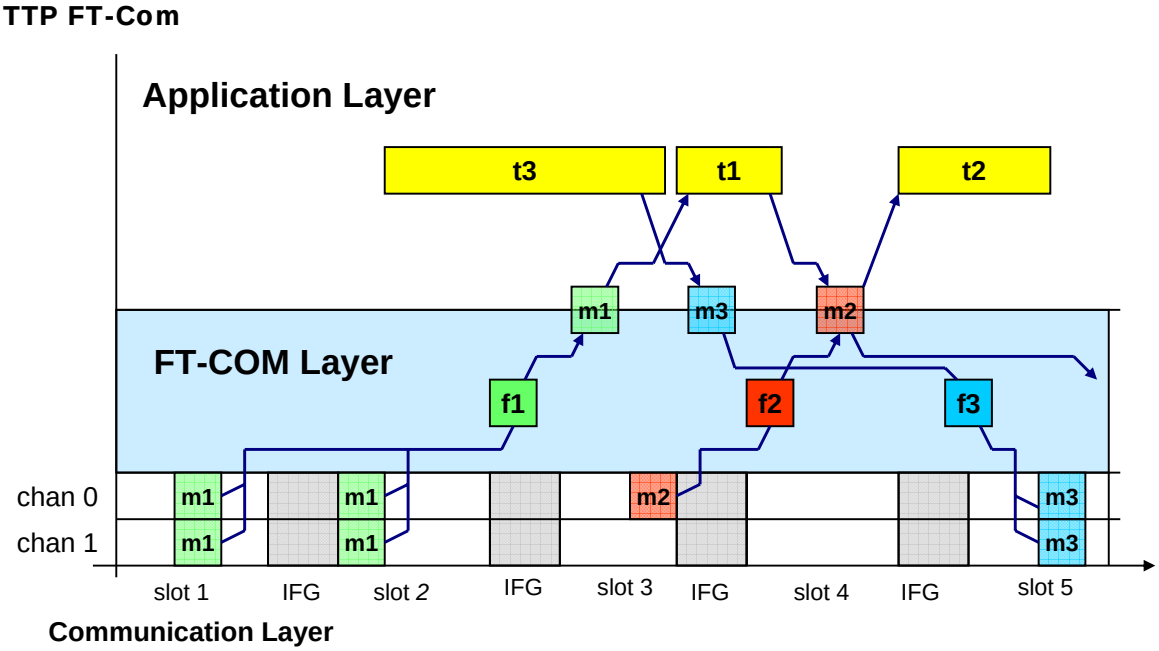

Figure 9 TTP FT-Com Layer

An important component in the TTTech tool chain is the fault-tolerant communication layer FT-Com. As Figure 9 illustrates it acts as a layer and interface between the communication layer and the application layer. The communication layer consists of frames and messages on the physical TTP bus and is determined by the MEDLs of the participating hosts with their TTP controllers. It can be seen that for example message `m1` that is consumed by the task `t1` is transmitted four times on the bus. This is due to redundant transmissions on both TTP channels and because it is also replicated, which means it is produced by two hosts. The FT-Com layer merges those four versions of the message into one that is made available to the task that consumes the message. This merging is determined by an algorithm called the replica-deterministic agreement (RDA) algorithm. Depending on the application the FT-Com layer might calculate the average of the two replicated copies or just take any valid instance of it. Furthermore the FT-Com layer enables the application to detect errors by providing a message status for every message that is an integer value that indicates how many instances of the message were received correctly. A detailed description of how application code interfaces with the FT-Com layer can be found in 3.5.3.

The FT-Com layer is implemented as automatically generated C code that is created by TTPbuild. It therefore is specifically tailored to the actual application and the messages, tasks and RDA algorithms it uses.

TTPos

TTPos is an operating system for real-time systems that is provided by TTTech for their tool chain [5]. It is compliant with the OSEKtime operating system specification and is strictly time-triggered. A notable aspect is that the TTPos operating system has no mechanism to provide scheduling at runtime. The task activation table is planned and scheduled offline by TTPbuild and TTPos just executes that table periodically. The table must contain a schedule that ensures that no resource sharing conflicts or deadline violations occur.

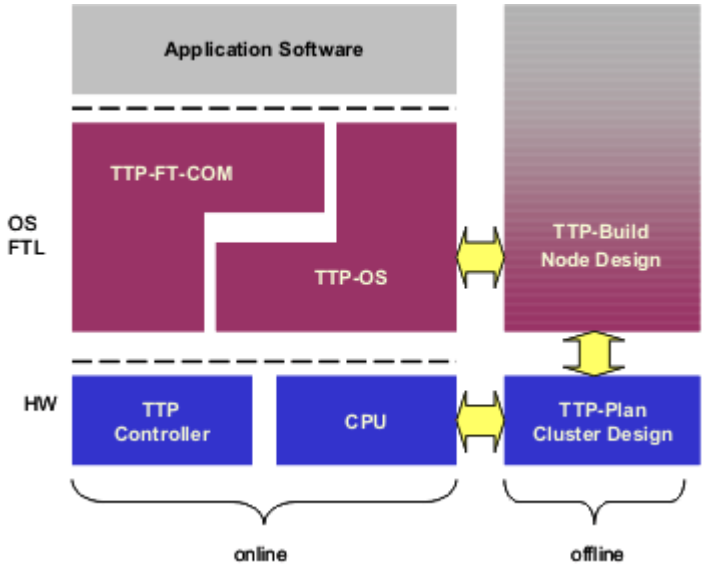

Figure 10 Position of TTPos in TTTech's TTP Tool chain

TTPos is tightly integrated in the TTP tool chain, as Figure 10 illustrates. It depends on TTPbuild that outputs a configuration file that contains the task schedule. This schedule not only contains the application tasks but also a task for time synchronization and tasks that execute the FT-Com layer. TTPos is also tightly coupled with the FT-Com layer that uses TTPos to access the hardware and provides the application with an interface to access the TTP bus.

TTPload

TTPload is the download tool in the TTTech TTP tool suite. It is used to download the MEDLs to the communication controller and the application binary files to the program flash memory of each host computer. All this is done through a special hardware box called the TTP monitoring node that acts as a gateway between standard Ethernet and the TTP bus and protocol. The monitoring node is connected to the TTP bus as a passive device that listens on the bus but does not send anything. All MEDLs and application files can be downloaded conveniently in one step to all nodes without the need to plug a cable into each of them.

The screenshot in Figure 11 shows the main screen of TTPload. All nodes are listed and it can be selected if the MEDL and application should be downloaded. The monitoring node is listed here as well, as it also needs a correct MEDL for the

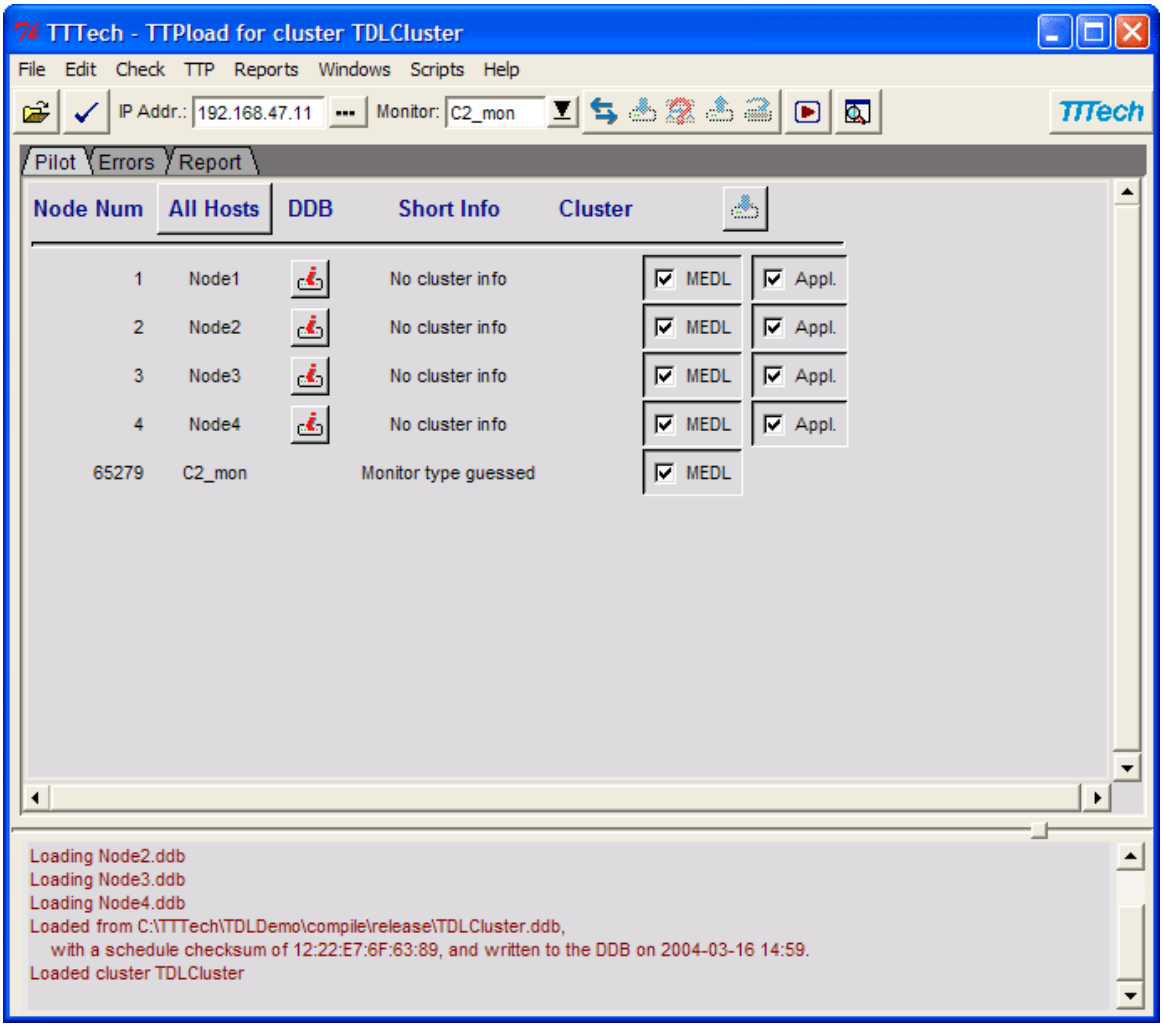

Figure 11 TTPload Screenshot

application it should monitor to know when which messages are sent by the nodes. TTPload also provides a function for querying the nodes to check if they already have a current version of the MEDL and application.

TTPview

TTPview enables the user to display all data that is transferred over the TTP bus. It also uses the monitoring node to access the TTP bus via Ethernet like TTPload. TTPview needs a cluster database created by TTPplan because it needs the information of what messages are transferred at what time. Figure 12 shows its graphical user interface. On the left all entities of the loaded cluster database that should be monitored can be selected and dragged to the right window. Among the selectable entities the actual values of the bus messages along with status information on them and global status values such as the current clock or the membership vector. There are various different ways to view the values, from simply displaying the numerical value to a gauge view and graphs to see the change of a value over time. After this step online monitoring can be started with the first button from the left at the bottom of the window and all values will be updated in real time. There is also the possibility to record everything that is transferred on the bus for a later offline analysis by pressing the record button indicated by a red dot on it. The

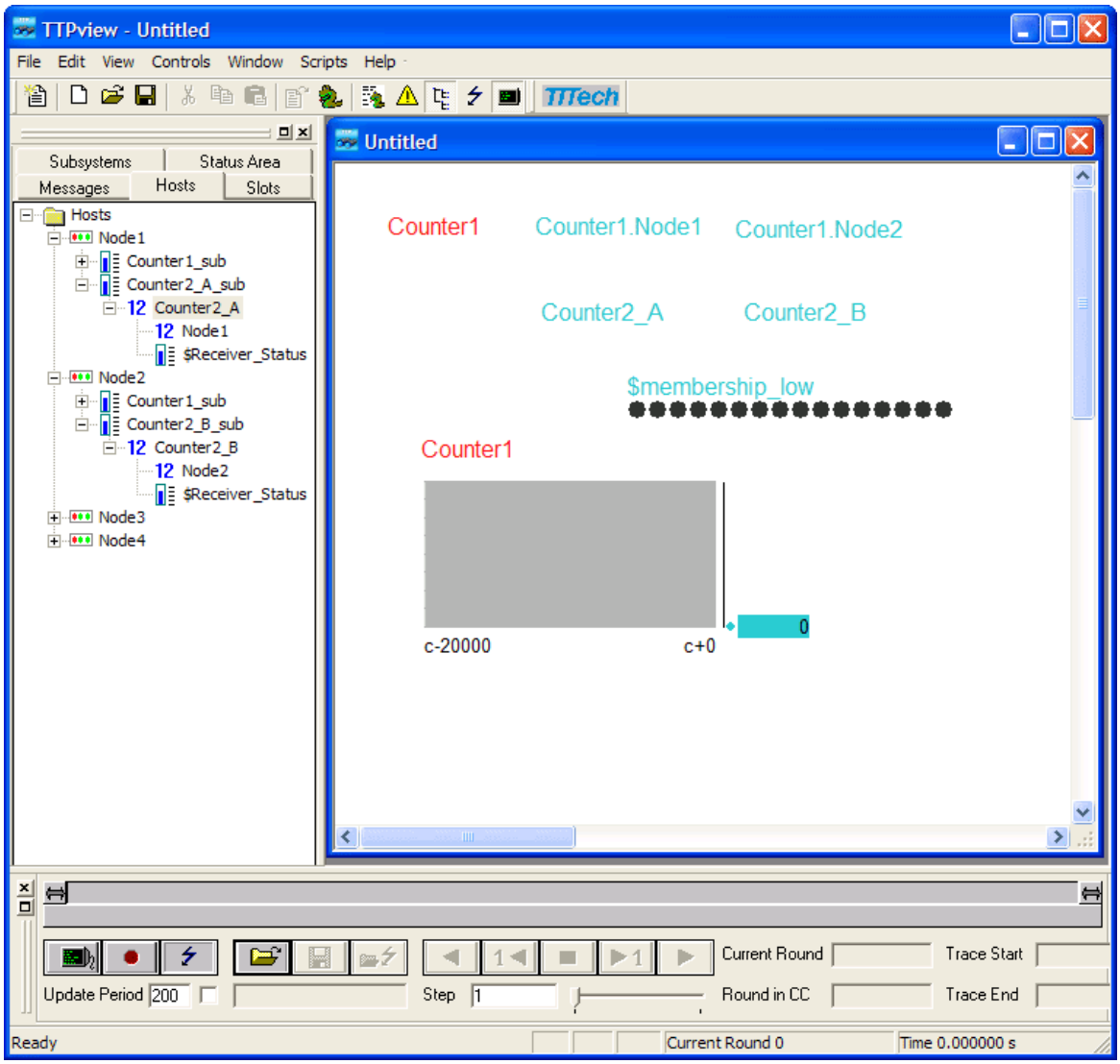

Figure 12 TTPview Screenshot

other buttons at the bottom are used to store and load recorded data and to playback them forwards, backwards or step-by-step.

Although TTPview is limited to viewing bus traffic and cannot be used to access any values or variables inside the node's CPU, it is a valuable tool for debugging and development purposes.

Chapter 3
Tool Chain for the Integration of TDL and the TTP Tools

3.1 Tool Chain Overview

An overview of the complete tool chain for developing TDL programs for the TTP hardware is presented below. All entities in the tool chain are described together with their order and connections between each other.

Figure 13 gives an overview of the tool chain. In the following all entities of it are summarized. A more detailed description of the key items can be found in the next sections.

TDL Code

This is a collection of files that contain the TDL program of the application. In the current implementation of the TDL compiler a separate file for each module must be used. Typically the modules are connected by public statements on one hand to indicate tasks and sensors that are available to other modules and import statements on the other hand that make these public construct accessible inside another modules. The additional platform specification has to be provided in a separate property file.

Property File

Our TDL compiler plugin needs specifications in addition to the TDL program. In order to be able to provide these to the plugin an additional file is used. It is a standard Java property file with name-value pairs. The information that must be specified consists of information on how the TDL program should be distributed among nodes, fault-tolerance specification and platform-specific properties that are specific to the TTTech hard- and software. In addition to that the file also contains a number of paths to the programs that the plugin needs to call such as the binaries of the TTP tools.

TDL Compiler

The TDL code is compiled using the existing TDL compiler. The main function of the compiler is to parse the syntax of the modules and to generate E code for each module. The TTP plugin only uses the former and does not need the generated E code. It only uses the compiler to get access to the modules in an appropriate data structure via the compiler's plugin interface. The reason for bypassing the use of E code is the static nature of the TTP platform. With the TTP tools every task is statically scheduled at design time and this schedule cannot be changed at runtime. Thus, the interpretation of E code at runtime would not make sense. The execution of TDL drivers for reading sensors and updating actuators is managed by the

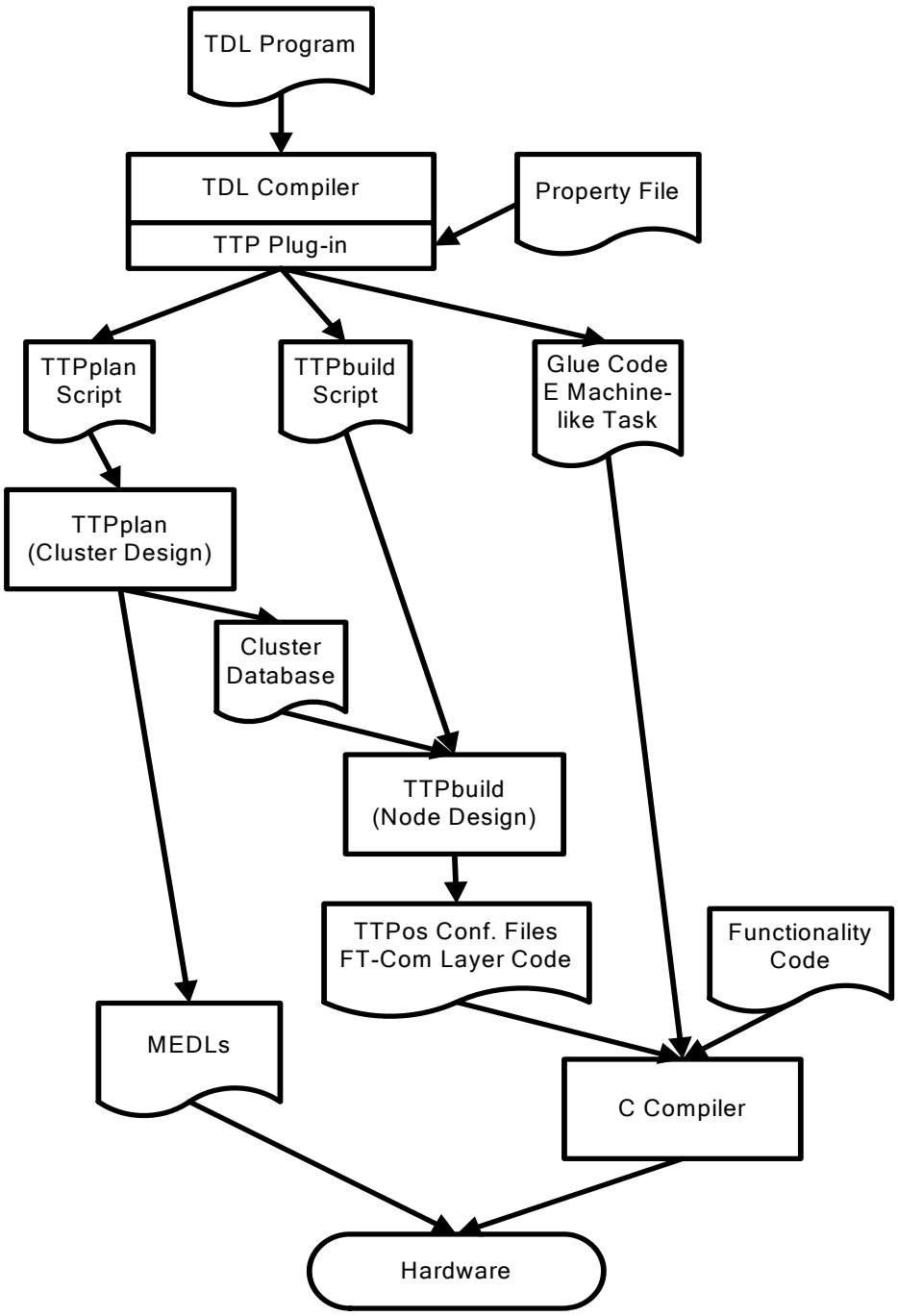

Figure 13 Tool Chain Overview

introduction of a periodically scheduled E machine-like task that is generated by the plugin.

TDL Compiler Plugin

The core element of the tool chain is the TTP plugin for the TDL compiler. It accesses the parsed modules from the compiler and controls the transformation of them into a TTP application. To accomplish that it has to generate scripts for the TTP tools and execute them. Furthermore it has to generate glue code to link the functionality code of the tasks and drivers to the operating system TTPos and the FT-Com layer provided by TTTech. This glue code does also periodically execute an E machine-like

task that handles the execution of drivers for reading sensors and updating actuators.

TTPplan Script

The plugin generates a file that contains a script for the TTP tool TTPplan. It contains commands that realize the timing and functionality that the TDL modules specify on cluster level. This includes communication between nodes of the cluster, the distribution of modules among nodes and fault-tolerance properties.

TTPplan

TTPplan is the cluster design tool of the TTP tool suite and the first tool that is employed by the plugin. It is used to design a global schedule for communication between nodes. It is executed with a script as input that controls the generation of a suitable schedule that corresponds to the TDL modules and their distribution among the individual nodes. The communication schedule is stored in a cluster database file and for each node a separate message descriptor list (MEDL) is generated that contains the bus schedule.

MEDLs

For each node a message descriptor list (MEDL) is generated by TTPplan. It contains the relevant information for each node which is when it is allowed to send what data and when it can receive data from other nodes. The MEDL file must be downloaded to the TTP communication controller of each node which completely controls the communication behavior of it. For the downloading process the TTP tool TTPload is used.

Cluster Database

The cluster database file created by TTPplan contains all information that was specified in TTPbuild by the script for it and in addition the generated cluster schedule.

TTPbuild Script

This script is a file created by the plugin for every node. It contains commands that realize the timing and functionality of the TDL modules on node level. This includes the implementation of the E machine-like task and the execution of sensors, actuators and tasks.

TTPbuild

TTPbuild is used for node design. It relies on the cluster database with the generated communication schedule. The plugin creates one script per node and then invokes TTPbuild for each of them. The script contains commands that most importantly create all tasks that are supposed to run on every node. These tasks consist of tasks that are defined in TDL, the E machine-like tasks that are generated by the plugin and tasks that TTPbuild adds to perform time synchronization and execute the FT-Com layer that handles the sending and receiving of messages on the bus. TTPbuild creates a task schedule and the FT-Com layer as output.

TTPos Configuration Files and FT-Com layer

These are a collection of files that TTPbuild creates for each node. The TTPos configuration files are C code that contain a table that represents the complete task schedule of the node. It is intended to be used with TTTech's operating system TTPos. The FT-com layer is also a collection of C files generated by TTPbuild. It provides the application with the required functions to send and receive messages on the bus. It handles the fault-tolerant transmission and reception of messages according to the properties specified in TTPplan and TTPbuild.

Glue Code and E Machine-like task

These files are all created by the TDL compiler plugin for every node. The glue code maps the functionality code to the operating system and the FT-Com layer. It also contains the E machine-like task that has two purposes: One is the calling of drivers for sensors and actuators and the other is the relaying of messages in order to maintain the FLET property of TDL.

Functionality Code

The functionality code must be provided for each module and contains the implementations of all tasks and drivers that are used in the TDL code. It is not specific to the TTP plugin and is intended to work with other platforms using the C language as well. The naming convention for the functionality code for an example module `myModule.tdl` is `myModule.c` for the C code containing the task and driver code and `myModule.h` for the corresponding header file. Inside the functionality code other header files can be referenced with an `include` statement in order to be able to use for example one file pair `drivers.h` and `drivers.c` with driver code for sensors and actuators by multiple modules throughout the whole application.

C Compiler

The C compiler compiles and links the TTPos code with its configuration files, the FT-Com layer, the glue code and the functionality code and eventually creates a binary file that is ready to be downloaded to the target platform. As compiler the Diab C/C++ Compiler for PowerPC by Windriver is used, as it is recommended by TTTech and also shipped with their development cluster hardware.

Hardware

As target hardware everything that is supported by the TTP tools can be used. A typical hardware setup consists of a collection of TTP nodes together with a special monitoring node that acts as a gateway between standard Ethernet and the TTP protocol. This node can be used to monitor all data that is transferred on the bus and to download MEDL and application files to the individual nodes. Monitoring can be accomplished by TTPview, whereas downloading of code and MEDLs can be done by TTPload.

3.2 TDL Compiler

This section is dedicated to the current implementation of the TDL Compiler that has been used for this thesis. Its functionality is described together with its plugin interface.

The version of the TDL Compiler used for the thesis supports multiple modules to be combined with the use of the `import` and `public` keywords. Functionality such as the output ports of tasks and sensors can be made available to other modules by using the `public` keyword. This is the only distribution aspect the compiler supports to date. It deliberately does not handle the platform specific assignment of modules to nodes or any kind of fault tolerance functionality.

3.2.1 Calling the Compiler

The compiler is invoked at the command line as follows:

```
java emcore.tools.tdlc.Compiler [options] [TDL files]
```

The following options are available:

`-d <destination directory>`

This option specifies the directory to store the generated files. Typically those are E code files for every module. This directory is also passed on to a plugin which typically also places generated files in this directory.

`-java`

This option specifies to use Java as target platform.

`-ansic`

This option specifies to use ansi-C as target platform.

`-cpp`

This option specifies to use C++ as target platform.

`-platform <class name>`

This option lets the user specify any Java class as a platform plugin. This is how the TTP plugin presented in this thesis is integrated.

The compiler always produces one E code file for every TDL module no matter what target platform is specified. For every platform an individual collection of additional files is generated that typically is glue code for mapping the functionality code to the specific properties of the target platform. In the case of the TTP plugin additional software – the TTP tool chain – is employed to generate those files and to eventually produce a working system running TDL code.

3.2.2 Plugin interface

A platform plugin is realized by a Java class that implements the `Platform` interface. The only method in this interface is the `emitCode` method which must be implemented like this:

```
public void emitCode(String destDir, Module module) throws Exception
```

This method is called for every module the compiler processes. It has the destination directory that was provided at the command line for the compiler and the module as parameters. The module object contains the abstract syntax tree of the TDL module that was parsed by the compiler. A plugin typically analyzes the module, generates files and puts them in the destination directory.

3.3 TTP Tools

This section explains the relevant TTP tools for the integration with TDL in detail, focusing on their programming interface and the object model that the main tools TTPplan and TTPbuild use. All basic steps a user has to go through to get a correct model and therefore a working TTP application are explained briefly. Furthermore the script language of the tools is demonstrated with simple examples.

Both TTPplan and TTPbuild have two modes of operation: They can either be started in an interactive mode with a graphical user interface or in batch mode without any interactive interface and textual output only. With a few exceptions all functions are available in both modes.

Both applications also share their basic architecture of using an object model for the internal representation of the data and also as part of their user interface. A TTP cluster in TTPplan and a TTP node in TTPbuild are composed of a number of objects with relations between them. Every object also has a number of mandatory and optional attributes.

3.3.1 TTPplan

Figure 14 shows the object model of TTPplan. The main object of the model is the cluster object. It represents the whole system and has attributes that specify system-wide properties. A cluster consists of multiple hosts that all must be given a name. Here the number of hosts in the cluster is defined and for every host the developer later has to create a host object model with TTPbuild. A host in the TTP tools is a node of the network that has computational and communicating capabilities. Another key object is the subsystem. It is the unit of distribution and replication and is used for the packaging of messages. Consequently a subsystem is linked to a number of messages. A message represents a message on the TTP bus. In this development stage it is only defined which messages a subsystem sends, not which messages it receives. As the reception of messages does not influence the cluster schedule, because of the broadcast nature of the bus, it is sufficient to specify reception later on node-level. Subsystems are also linked to hosts and by this it is defined what functionality is executed on which host and if and how subsystems are replicated, as it is possible to assign a subsystem to multiple hosts. All this associations are later

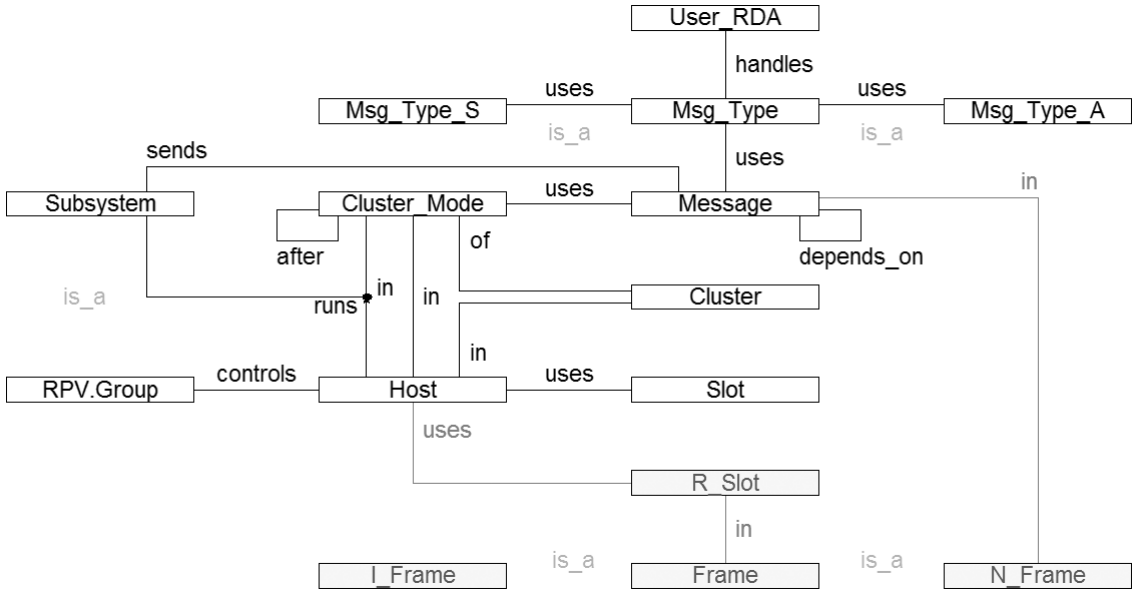

Figure 14 TTPplan Object Model

also available when designing a single host with TTPbuild and cannot be changed then.

A detailed description of all relevant objects in TTPplan for the integration with TDL can be found in 3.5.1, whereas a complete specification and definition is contained in [6].

Sample Script

The scripting language of TTPplan and also TTPbuild consist of a number of commands that (1) control the tool itself, typically with the `TTA.Application_Command.run` command that for example performs file handling commands or initiates the generation of schedules, or (2) are used to create and modify the object model of the tools. The commands are followed by a number of parameters in parentheses. Strings need to be enclosed in single quotes. Normally the parameters need to be formatted in the required type for the parameter, i.e. strings have to be quoted but integer values are unquoted. However, there is a so-called raw mode, indicated by the parameter `raw=1`, that requires every parameter to be formatted as a string. This mode is especially needed for attributes of objects that only have a defined set of valid values.

The following lines of code illustrate what a script for TTPplan that creates an object model looks like. It is only an excerpt that shows the beginning and end of the script and as an example the creation and linking of a subsystem and a message.

```
TTA.Application_Command.run ('File.New', 'myCluster')

TTA.Subsystem.define ('mySubsystem', reintegration_type =
'Reinit_Reintegration', raw=1)
```

```
TTA.Message.define ('myMessage', agreement = 'RD_1_valid', init_value =
'1', raw=1)

TTA.Subsystem_sends_Message.add (TTA.Subsystem.instance ('mySubsystem'),
TTA.Message.instance ('myMessage'))

[...]

TTA.Application_Command.run('Schedule.Make new schedule')

TTA.Application_Command.run('Schedule.Make MEDLs')

TTA.Application_Command.run('File.Save cluster as ...', 'myCluster.cdb')
```

The first line creates a new cluster database with the name `myCluster`. Then a number of statements follow that define every object and link in the object model including their required and optional attributes. As an example the above statements create a subsystem `mySubsystem` and a message `myMessage`, together with a link that defines that the message is sent by the subsystem. A complete list of commands used by the plugin can be found in 3.5.1. The last lines tell TTPplan to create a cluster schedule and to make the message descriptor lists (MEDLs) for each node. Finally all data is saved in a cluster database file that is indicated by the ending `.cdb`.

Script Execution

A script is executed by calling the batch version of TTPplan with the `–script` option, providing the name of the script as parameter. So a script called `sample.cmd` would be executed by calling

```
ttpplan_batch -script=sample.cmd
```

in the directory where `sample.cmd` resides. The program outputs information on what it does to the console. This output may also contain error information in case of a syntax error or when trying to generate a schedule for an incorrect or inconsistent object model. In the latter case it is advisable to load the faulty object model with the interactive version of TTPplan, invoke the integrated check for object model errors and review the output. This is possible because TTPplan also writes the object model to the given file when the script execution fails.

3.3.2 TTPbuild

Figure 15 illustrates the object model of TTPbuild. Node design with TTPbuild relies on the cluster design and consequently a cluster database file from TTPplan is required before the user can begin using the tool. There are some objects in the object model that exist in both tools. These are the cluster mode, host, subsystem, message and message type objects. The instances of those objects are copied to the node database and cannot be deleted or changed as such an operation would contradict the cluster database that is shared by all nodes of the cluster. The creation of additional instances of the mentioned objects is allowed and often necessary, though. An example would be a message that locally connects two tasks with each other and is not visible to other nodes as it is not transferred over the TTP bus. Apart from the mentioned objects the key object in this development stage is the task. A subsystem runs one or more tasks to realize certain functionality. A task uses messages to communicate with other tasks either locally or remotely via the TTP bus. Exactly one task of a subsystem is required to send the message that was already

assigned to be sent by the subsystem in TTPplan. Every task runs in an application mode that represents a state of operation of the host. This object is not that important due to the limitation of only being able to use a single application mode in the current version of TTPbuild. A detailed coverage of all relevant objects and their relations with regard to the integration with TDL is presented in 3.5.2. A complete description of the object model of TTPbuild can be found in [8].

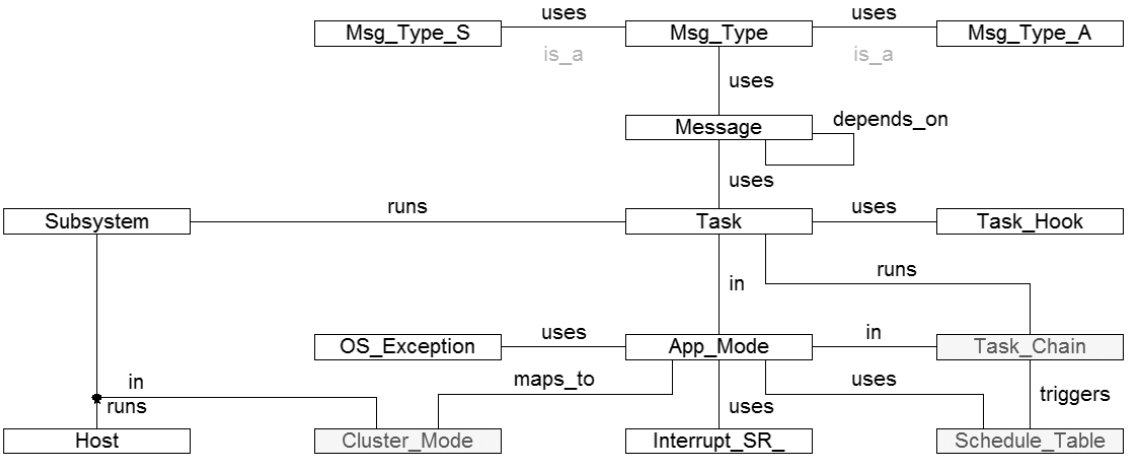

Figure 15 TTPbuild Object Model

Sample Script

The script language basically uses the same constructs as that of TTPplan which was explained above. The following is a sample script that creates an object model using the batch version of TTPbuild. It includes the complete beginning and ending of a typical script, but contains only a fraction of the actual code that would create a complete object model.

```
TTA.Application_Command.run('File.New node ...', 'myNode',
'myCluster.cdb')

TTA.Node.App_Task.define ('myTask', time_budget = 100, period = 4000,
deadline = 150, phase = 0)

TTA.Node.Task_uses_Message.add (TTA.Node.App_Task.instance ('myTask'),
TTA.Message.instance ('myMessage'), access_type = 'agreed', raw=1)

TTA.Node.Task_uses_Message.link ('myTask', 'myMessage').set (sends =
'yes', receives = 'no', raw=1)

[...]

TTA.Application_Command.run('Schedule.Make new schedule')

TTA.Application_Command.run('Schedule.Generate code')

TTA.Application_Command.run('File.Save node as ...', 'myNode.ndb')
```

This first line creates a new node database by assigning a name and providing the cluster database file `myCluster.cdb` on which it relies. The next three statements define the task `myTask` and link it to a message `myMessage`. It can be seen that the message is sent, but not received by the task. After a number of other lines that

create a complete and consistent node object model, the next command tells TTPbuild to create a task schedule for the node. After successful scheduling, C code files are generated that consist of a file that contains the node schedule in form of a configuration file for the operating system TTPos and files that implement the FT-Com layer. Finally, the object model is saved to a node database file that is indicated by the ending `.ndb`.

Script Execution

The execution of a script works identical to TTPplan and is invoked by calling the batch version of TTPbuild with the `–script` option and providing the script file name as parameter:

```
ttpbuild_batch -script=sample.cmd
```

Just like TTPplan, the tool then outputs information on the progress and possible errors to the console. In case of an error it is an advisable strategy to load the faulty node database to the interactive version of TTPbuild and analyze it there with the provided error checks.

3.4 Fault Tolerance Aspects

The fault tolerance aspects of the toolchain are explained in detail in this section. The functions that are provided by the TTP tools are analyzed and ways for achieving fault tolerance and error detection in TDL are suggested.

One of the goals of the integration of TDL with the TTP tools was to gain experience in how platform-dependent fault tolerance can be harnessed in a TDL program. The TTP protocol and corresponding tools provide various fault tolerance mechanisms in hard- and software. Most of these mechanisms realize systematic fault tolerance that do not require application awareness. The use of the TTP bus alone does not make the whole application fault tolerant, but it transparently takes care of the transmission of messages in a safe, fault-tolerant and consistent way. This is achieved by the bus guardian, the redundant communication channels and the membership service that all are hard-coded on the TTP communication controllers and verified to work reliably. Those features are covered in more detail in 2.4.1 above and [7].

On top of the TTP bus protocol layers, the TTPos operating system and the fault-tolerant communication layer (FT-Com) provide additional fault tolerance features. Those are controlled by the two design tools TTPplan and TTPbuild, that are used to configure them and eventually also generate the code that implements them. TTPplan creates MEDLs that determine the behavior of the TTP communication controller chips and TTPbuild generates the code for the FT-Com layer.

The idea in the integration of TDL with the TTP tools is to come up with a way of describing what fault tolerance mechanism to apply as an annotation to the TDL language.

Replicated Modules

A module is the top level construct of TDL that combines multiple modes and their tasks to form a functional unit. A TDL module represents the unit of distribution. This means that a single module cannot be split up by distributing parts of it among multiple nodes of a distributed real-time system. The fact that a module provides a service with well-defined input and output also makes it suitable as a unit of replication. A module that is replicated and gets exactly the same input as the original module will consequently also produce the same output. This is guaranteed by the exact specification of the timing behavior that is implemented by TDL and reliable time synchronization between the nodes of the cluster provided by the TTP protocol. Indeed there is a quite similar construct to the module used in the TTP tools called the subsystem. A subsystem is "the basic unit of packaging software components; as such, it is the unit of distribution, replication, and composability" [6]. All this leads to the conclusion that it is feasible to associate a TDL module with a subsystem in the context of the TTTech TTP tools. The TTP tools provide the ability to replicate subsystems and distribute them among the nodes of the TTP cluster. What must be provided to support replicated TDL modules is a facility to specify their distribution among nodes together with the ability to assign a module to multiple nodes, as it is possible for a subsystem with the TTP tools. The replication of modules also raises the issue of how to decide what the actual output of a module is when the replicas produce outputs that differ from each other, which might be due to a failure of a node. So what also must be specified is a so-called replica-deterministic agreement (RDA) algorithm that determines the actual value consistently for all modules that access the output of a replicated module.

Details of the mapping of modules to subsystems are discussed in 3.5 below, whereas the specification of the distribution of modules is described in 3.6.

Redundant Messages

When mapping TDL modules to the objects in the TTP tools for every public port of a task a message on the TTP bus is generated, as we will see below in section 3.5. This is necessary as the module that uses the port might be located on a remote node. The TTP bus always is equipped with two independent communication lines, called channels. The TTP tools offer the possibility of transferring messages redundantly on both channels in order to tolerate a fault of one of them. Of course it may also be desirable to use both channels independently for maximized data throughput. It makes sense to let the user decide what level of safety should be applied to every public output port of a TDL task. How this specification works is discussed in 3.6 below.

Application-Specific Fault Tolerance

The two fault tolerance aspects above are a good example of systematic fault tolerance as described in 2.2 in the previous chapter. The TDL application is not aware that the values of ports are transmitted redundantly or that modules are replicated and executed on multiple nodes. But for some applications it is beneficial to make them aware of the status of the fault tolerance mechanisms applied. The FT-Com layer of the TTP development tools provides a message status that indicates the current status of replication by providing the number of online replicas. It is implemented as an integer value that is 0 when no replica is operating correctly and

equals to the number of replicas that produce valid output otherwise. Making this value accessible inside TDL modules enables them to react to the failure of a module and for example trigger an emergency shutdown or inform the operator of the system that a node has to be repaired or replaced.

The integration of the message status function in TDL is realized as a special kind of sensor. To avoid changing the TDL syntax and consequently change the implementation of the compiler, a special driver name with the prefix `REPL_` is used. It is followed by the name of a public output port of a task in the same or a different module. Consider the following example:

```
sensor
  int lightValue_messageStatus uses REPL_lightValue;
```

This TDL sensor declaration assigns the message status of the public task output port `lightValue` to the sensor named `lightValue_messageStatus`. The sensor value can be used in the TDL program as a normal sensor as input ports for a task or directly as input for updating an actuator.

For a detailed description of the implementation of this feature read on in section 3.5.3 below.

3.5 Mapping of TDL to TTP

This section contains various aspects of the mapping of TDL modules to the TTP tools in order to execute them on the TTP platform. In the beginning the realization of the E machine-like task is explained, with special focus on the maintenance of the FLET property of TDL. Also an overview is given on the possibilities of mapping TDL constructs to the TTP tools. This is followed by subsections containing an in-depth description of how to map TDL modules to the object models used in the TTP tools. All objects in the model are described together with a procedure of how to generate them out of a TDL program and corresponding property file. Furthermore the glue code that links the TDL functionality code with the TTP operating system TTPos and the handling of the type mapping between TDL and the TTP tools is explained.

E Machine Implementation

In [2] an interesting statement concerning the implementation of the E machine can be found:

"The E machine is a virtual machine. In an actual implementation of the E machine, E code need not be interpreted, but may be compiled into, say, C code, or even silicon."

Indeed, when considering the nature of the TTP tools it does not make sense to have an E machine implementation that interprets E code. This is due to the lack of a scheduler in the operating system TTPos and the lack of a file system that would allow to load the E code file. TTPos only gets a pre-planned schedule by TTPbuild and dispatches the tasks according to this scheduling table. The schedule is static and planned at design time by TTPbuild and it is not possible to make any changes to it at runtime. So this fact makes the "schedule" instruction of the E code, which hands a task over to the scheduler, quite pointless. However we do need the "call" instruction

of the E code to execute drivers for sensors and actuators. The solution is to realize some parts of the E machine offline and create a periodic E machine-like task that performs the other tasks that have to be done at runtime. The offline part is the scheduling of tasks and bus messages and the implementation of the FLET semantics. This is done by generating a bus schedule with TTPplan and a suitable task schedule with TTPbuild as described below. The runtime part handles the execution of drivers for reading sensors and updating actuators and also contributes to the realization of the FLET property by relaying messages. The interpretation of the MEDLs by the TTP controllers can actually also be seen as a runtime task that originally is meant to be carried out by a standard TDL E machine.

Maintaining FLET

The main challenge of this plugin part was to maintain the FLET property of TDL, as it was explained in 2.3 above. Sensors have to be read at the beginning of FLET and actuators have to be updated at the end. To ensure this, a periodic E-machine-like task was introduced that is scheduled at the beginning of each FLET period and that executes the driver code for the actuators of the previous FLET period and the sensors of the next one. Local messages are generated for interfacing between the tasks and the E machine-like task. But still there might occur a violation of the FLET property when a task sends a message over the bus and it arrives before a task that uses this message is executed. This might happen especially if the actual CPU time consumption is short in comparison to the FLET length. In this case the receiving task has access to a value that should not be available to it at this point of time.

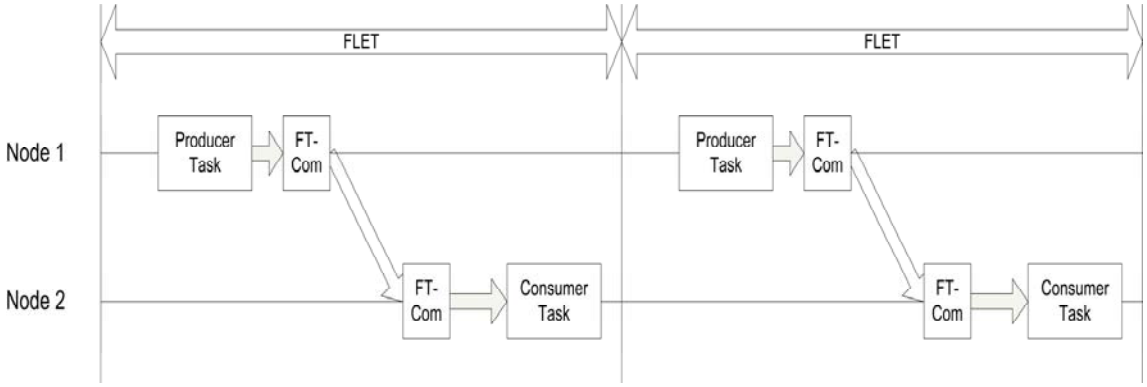

Figure 16 FLET is violated

Figure 16 illustrates such a scenario with a producer task that produces some value that is consumed by the consumer task. The FLET indicated at the top of the drawing applies to both tasks and therefore a value produced in one FLET should not already be available before the end of the FLET. Such a violation of FLET is not unlikely as TTPbuild actually tries to schedule tasks in a way that minimizes the time between message arrival and task execution to minimize the delay time. This is of course not correct for a TDL implementation and so the only way of maintaining the FLET is to pass messages through the E machine-like task on sending, receiving or both. Because messages can only be sent by tasks that are linked to the subsystem that is specified to send the messages in TTPplan and tasks can only be linked to one subsystem, it would require one E machine-like task per subsystem to pass messages that are sent through it. This would get quite complicated and would

produce some overhead because of the fact that every task takes at least 75 microseconds due to context switching time. A better solution, and also the one that the plugin uses, is to pass only received messages to the E machine-like task and forward them to the appropriate tasks.

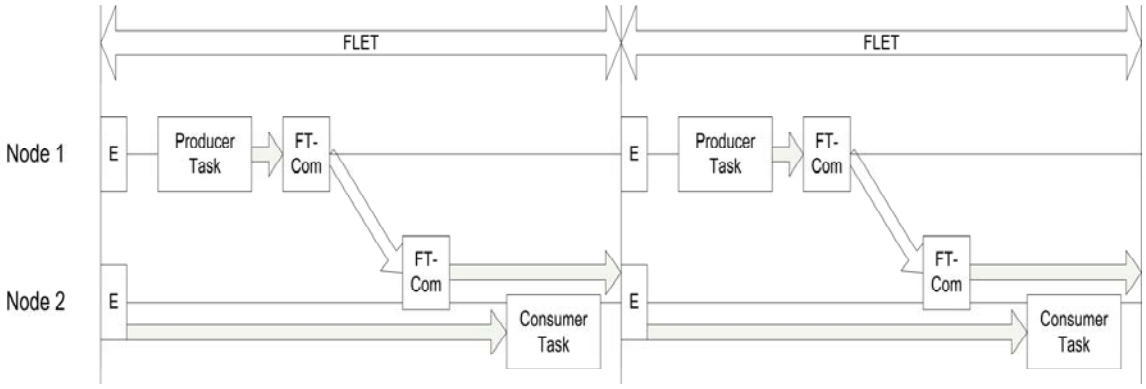

Figure 17 FLET is maintained with an E machine-like task

Figure 17 shows that same consumer and producer example as above with an E machine-like task that retransmits the message that contains the value produced by the producer task. Note that all tasks including the FT-Com layer are invoked at the exact same instance of time. Only the messages that link them were changed to ensure that the FLET property of TDL is maintained.

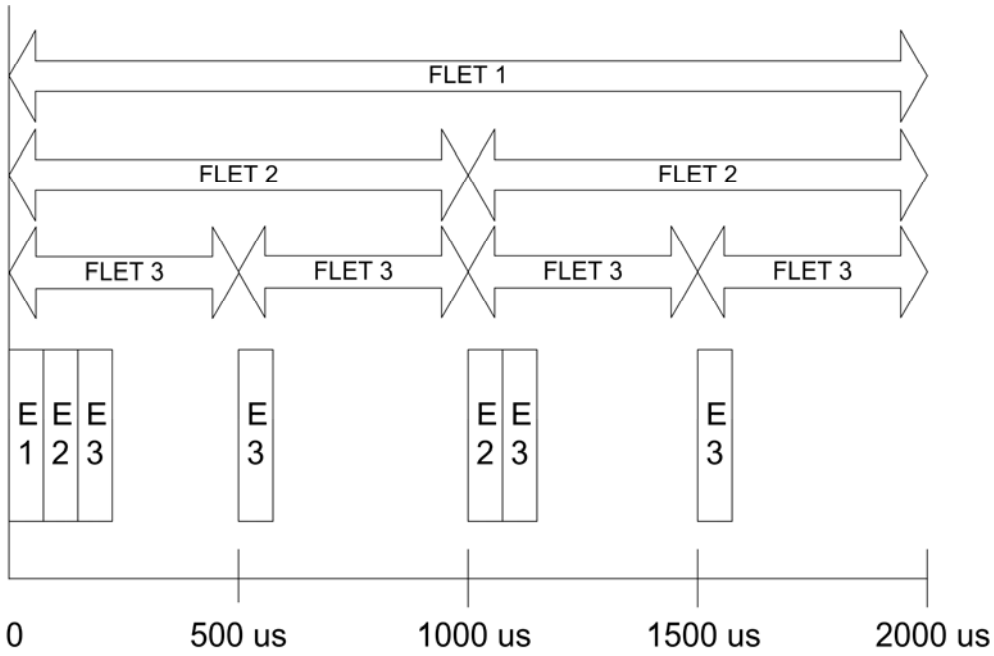

Figure 18 Example with 3 E machine-like tasks

Passing received messages through the E machine-like task solves the problem, but relies on the sending task to be executed before its output message gets transferred over the bus, which might in spite of sufficient CPU time still be impossible due to the global message schedule. An efficient and simple solution for maintaining FLET that

even works if there is not much free CPU time, a lot of messages on the bus and a lot of different FLET periods, which all require separate E machine-like tasks, is not possible with the TTPbuild tool. Consider the example in Figure 18 with three different FLET periods FLET 1 with 2000, FLET 2 with 1000 and FLET 3 with 500 microseconds. For every different FLET period a separate E machine-like task is required, which are labeled E1, E2 and E3 in the figure. As the minimum time a task consumes in TTPbuild is 75 micro seconds, this means that we loose already 525 micro seconds (7 times 75) of the 2000 micro seconds period for the execution of the E machine-like tasks. The fact that in the first 500 micro seconds three E machine-like tasks have to be executed makes the situation even worse. When considering that a typical run of the FT-Com layer, which is needed to transfer messages via the TTP bus, also takes at least about 100 micro seconds, it is clear that this rather simple example is already tough to implement, as it only allows less than 200 micro seconds for the execution of an application task for the 500 micro seconds FLET. This value is anyway only reached in the best case, assuming that the global message schedule is designed in a way such that the value the task produces can be transmitted exactly before the end of its FLET.

A solution that can handle scenarios such as the one described above would require modifying TTPbuild and the operating system TTPos so that every message is handled correctly according to the FLET property and it is not necessary to have one E machine-like task for every FLET period.

Modules

As already mentioned in 3.4, TDL modules can be mapped to subsystems in the context of the TTP tools. Both concepts have in common that they are the unit of distribution and are used to form a functional unit with well-defined input and output. In 3.4 arguments were presented in favor of using TDL modules also as unit of replication in the same way as subsystems are used in the TTP tools.

A notable difference between TDL modules and TTP subsystems concerns the possibility of the occurrence of cycles due to dependencies between modules or subsystems. In TDL cycles can occur when a module imports ports from another module that in turn import ports from the first one. TTP subsystems are linked solely via messages and so problems with cycles are ruled out because messages have an initialization value. So even if a message is not already produced by a subsystem it does have a well-defined value.

The tasks inside a TDL module are linked to the corresponding subsystem in TTPbuild. It is not possible in the tool to have tasks that are not associated to a subsystem. Every task must be linked to exactly one subsystem. This corresponds to the semantics in TDL, as a TDL task also is contained in exactly one module.

For public task output ports of TDL modules a message on the TTP bus must be created. This is required as such a port must be accessible from all nodes of the system, as modules that import the public port might be located on a remote node. The TDL TTP plugin does not check if the port is actually used on any another node, but simply creates a TTP message for every public output port. It is implemented this way to keep it as simple as possible, but it might result in wasted bus bandwidth. On the other hand, it is in some cases beneficial to reserve the bus bandwidth for later extensions or modifications. This way it is for example possible to shift around modules between nodes without worrying if the change prohibits the communication

scheduler from finding a suitable schedule when additional messages have to be transferred via the bus.

Tasks

TDL tasks are basically mapped one-to-one to tasks in TTPbuild, as there is a similar task object there. However it is important to take care that the FLET semantics of TDL are preserved. How this is done is described above in the discussion of the E machine implementation. The ports of TDL tasks are realized by messages in the TTP tools. There are two main types of messages in the TTP tools, which are cluster messages, which are transferred over the TTP bus, and local messages, that are used to transfer values between tasks of the same node. It is transparent to the task which message is actually used. To ensure that the FLET property is not violated, it is not allowed for tasks to be linked directly to other tasks by a local or cluster message. Therefore all ports of a TDL task result in messages from and to the corresponding E machine-like task, where sensors are read, actuators are updated and messages are forwarded to other tasks. Initialization values for ports can be used as message initialization values. However it is not possible to use initialization functions, as message initialization values cannot be set at runtime.

The WCET of a task can be mapped to the time budget of a task in the TTP tools. The time budget is the WCET plus some overhead and may also include some room for later extensions of a task.

Modes

Currently the TTP tools are restricted to only one application mode on cluster and node level. Only some special modes like a predefined startup mode for initialization purposes are supported. This restriction also requires that the TDL modules only use one mode, as it is not possible to implement more than this when using the TTP tools. The single TDL mode is mapped to a cluster mode in TTPplan and an application mode in TTPbuild.

A mode also contains information on the invocation period for tasks and update frequencies for actuators. The period of a TDL task can be mapped one-to-one to the period of a task in the TTP tools. However for every different task period also a separate E machine-like task on the node with the same period of the task is required to handle its ports. The same applies to actuator updates, which also require an E machine-like task for execution.

Sensors

The reading of sensors must be performed inside the E machine-like task of the task that uses the sensor to maintain the FLET property. The value read is then either used directly by an actuator or delivered to the task that uses it using a local message. Public sensors are not supported, because their handling is rather difficult as the update frequency is determined by all consumers of the sensor value, requiring an analysis of the whole application to find out a suitable communication pattern.

Actuators

Like sensors, actuators are required to be updated in the E machine-like tasks that were created for the FLET period that corresponds to the update frequency of the actuator. Actuators may be updated by task output ports within the same module or by imported public ports of other modules that can be located on a remote node. Consequently, this results in local or bus messages that are received by the E machine-like task that performs the actuator update.

3.5.1 Mapping TDL to TTPplan Objects

Before going into detail by listing every relevant object and link and how the plugin has to generate it, we give a brief overview of what has to be done. Figure 14 on page 29 illustrates the object model that TTPplan uses with all objects and their relations. The most relevant objects are hosts, subsystems and messages. A host is a node of the distributed system in the context of the TTP tools. A subsystem should comprise a well-defined functional abstraction and is the unit of distribution, composability and replication. This definition suggests that TDL modules and TTPplan subsystems can be mapped one-to-one, which is exactly what the plugin does. Every subsystem then needs to be assigned to a host according to the property file. Cluster messages are created for every public output port of a task and are linked to the corresponding subsystem. The period of those messages can be determined by the period of the tasks specified in the TDL code that produce them. In TTPplan it only has to be defined which subsystems send which messages. Reception does not have to be specified because the broadcast nature of the bus allows every subsystem on every node to receive any message. The message type and length is derived from a standard TDL type mapping table that maps TDL types to types in C as described in 3.5.4 below.

In the following all objects and links of the TTPplan object model are listed together with relevant attributes. For every entity it is described how the TDL plugin generates values for it, based on the TDL modules and the property file that contains additional specification, especially regarding distribution and fault tolerance aspects. In addition, the script command that is used to generate the object is shown. Attribute values which act as place holders for real values are enclosed in square brackets.

Object: Message

Description:

Represents a message that is sent on the bus. It is important to note that only the sending of messages is relevant in TTPplan, because due to the broadcast nature of the TTP bus every message can be received by every host and therefore the MEDLs created by TTPplan do not need to contain this information.

Generation:

The generation is done by iteration over all modules and all tasks invoked within the modules. If the task is tagged with the `public` keyword, for all output ports of the task a message is created. Public sensors are not supported by the plugin as their handling is rather complicated. The problem here is that all tasks or actuators that use the sensor would have to be analyzed in order to determine the update

frequency of the sensors and consequently the period of the message that transports the sensor value.

Important attributes:

- `init_value`:

  This value determines the initialization value for the message. As TDL allows the specification of initialization values for output ports this value can be used directly, with the limitation that currently only integer values are handled correctly.

- `agreement`:

  Specifies the replica-deterministic agreement algorithm. This needs only to be set for messages of replicated subsystems or TDL modules respectively. As TDL does not provide such a specification, the value has to be read from the property file for replicated modules and left blank if replication is not used.

Script command:

```
TTA.Message.define ('[messageName]', agreement = '[agreement]',
  init_value = '[initValue]', raw=1)
```

Object: Msg_Type

Description:

This object represents the type of a message. There are pre-defined types and it is also possible to create custom types.

Generation:

The type handling mechanism of the TDL plugin are described in the separate section 3.5.4 below.

Script command:

```
TTA.Msg_Type_P.define ('[typeName]', length = '[typeLength]',
  type_cat = '[typeCategory]', typedef = '[cTypeDef]',
  type_length = '[cTypeLength]', raw=1)
```

Link: Msg_Type uses Message

Description:

This link associates a message with a message type.

Generation:

Every message has to be linked to the message type that was created for it before.

Script Command:

```
TTA.Message_uses_Msg_Type.add (TTA.Message.instance ('[messageName]'),
  TTA.Msg_Type_P.instance ('[messageType]')
```

Object: Cluster

Description:

Represents the whole TTP cluster which is a collection of hosts. It contains a number of global attributes.

Generation:

The name for the cluster can be obtained by reading the property file. For most attributes of the cluster object the default values can be used.

Important attributes:

- `tr_period`:

  This attribute determines the length of a TDMA round of the TTP communication bus of the cluster. This value must not be longer than the shortest period of a message, because otherwise it would not be possible to transfer it in the required time. The shortest message period is figured out by the plugin by analyzing the list of all messages that need to be transferred. `tr_period` is set to the shortest message period divided by 2. The reason for this is that a cluster cycle anyway has to consist of at least of two TDMA rounds and the period length of a TTPbuild application cycle equals the cluster cycle length. Therefore it makes more sense to double the number of TDMA rounds to run the task that produces the message with the shortest period once per cluster cycle. This way also the schedule TTPbuild generates is more readable when viewing it with the tool, as otherwise the application cycle has the double length of the shortest message period and every task invocation would be contained twice in the schedule, as the application cycle length is always based on the cluster cycle length.

- `transmission_speed`:

  This value specifies how fast data is transferred on the TTP bus. The value can be 5000 kilobits per second at the maximum, resulting in a bandwidth of 10000 kilobits per second when using both channels independently. Of course this setting is a trade-off between being able to transmit more data versus a less-fault tolerant transmission with probably more transmission errors. The value is read from the property file. The recommended value is the maximum of 5000 bits per second, as it gives the scheduler more room and therefore increases the probability to find a valid schedule for a given TDL program. It would also be possible to let this value be determined automatically with good knowledge of how the schedule is generated by TTPplan and information of how much bandwidth is actually available to the application, which is the net bandwidth after the subtraction of protocol overhead. The size of the messages, which is needed for such a calculation as well, is available to the plugin through the type mapping mechanisms, as described in section 3.5.4 below.

- `controller_type`, `physical_interface`:

  These are hardware-specific parameters that need to be customized to match the target hardware. The plugin currently leaves them at their default values `TTTech_C2` and `MFM`.

- `fixed_round_number`, `max_tdma_rounds`, `min_tdma_rounds`:

  With these values the cluster scheduler can be influenced concerning the number of TDMA rounds a cluster cycle consists of. The current version of the plugin does not use them and leaves them at their default values.

Script command:

```
TTA.Cluster.define ('[clusterName]', byte_order = 'big_32_endian',
  tr_period = '[lengthOfTDMARound]', transmission_speed =
  '[transmissionSpeed]', raw=1)
```

Object: Host

Description:

This object represents a host of the cluster. A host in context of the TTTech TTP tools is a node of the distributed system.

Generation:

The specification of hosts has to be obtained from the property file. It contains a list of hosts for every module that specifies on which hosts they should be assigned. Out of this list a list of all hosts can be obtained and then for every host an object can be generated named with the name of the host.

Important attributes:

- `serial_number`:

  The serial number is a number used to clearly identify a host. It must be unique and can be automatically generated by numbering every host starting at 1.

- `mux_period`, `mux_round`:

  These two attributes are needed when using multiplexing of TDMA slots. The period and round of the host can be set to share a slot with a different host. The plugin does not use multiplexing and so it sets both values to 1, resulting in having exactly one slot per host.

- `controller_type`:

  This is the same hardware-specific attribute as described above for the cluster object and is set to `TTTech_C2`, which specifies the hardware configuration that matches the available development cluster.

Script command:

```
TTA.Host.define ('[hostName]', mux_round = '1', mux_period = '1',
  serial_number = '[serialNumber]', controller_type =
  'TTTech_C2', raw=1)
```

Link: Host in Cluster

Description:

This attribute specifies the hosts a cluster consists of. Every host needs to be linked to a cluster.

Generation:

As we have only a single cluster all the plugin has to do is to link every generated host object with the cluster.

Script command:

```
TTA.Host_in_Cluster.add (TTA.Host.instance ('[hostName]'),
  TTA.Cluster.instance ('[clusterName]'))
```

Object: Slot

Description:

A slot represents a single slot of the TDMA round of cluster communication. Every slot must be assigned to a host.

Generation:

Every host needs to have a slot in the TDMA round and so for every host a slot object needs to be generated. Its name consists of the name of the node and the ending `_slot`.

Important attributes:

- `sort_key`:

  This attribute can be used to influence the arrangement of slots. By default the scheduler of TTPplan determines the order of the TDMA slots. When this value is set, the order of slots will follow the order of the values of this attribute. The plugin does not use this option and lets the scheduler determine the order automatically.

Script command:

```
TTA.Slot.define ('[slotName]')
```

Link: Host uses Slot

Description:

This link assigns every host a sending slot in the TDMA round of the cluster. When multiplexing is not used, then every host has to be linked to exactly one slot.

Generation:

Because the plugin does not use multiplexing, it simply links every host to the slot it created for it.

Script command:

```
TTA.Host_uses_Slot.add (TTA.Host.instance ('[hostName]'),
  TTA.Slot.instance ('[slotName]'))
```

Object: Cluster_Mode

Description:

Represents a cluster mode. The version of TTPplan that was used for the thesis has a limitation of only being able to handle a single cluster mode. This limitation does not include the following three pre-defined modes that TTPplan supports by default: `Startup_Mode`, `Sleep_Mode` and `Download_Mode`. The `Startup_Mode` is a special mode in which every cluster starts to do some initialization routines for every host and in which the start-up of the communication system is performed.

Generation:

In addition to the startup mode, an application mode needs to be created that is the mode the system runs in when operating normally. The name of the mode is generated by taking the name of the TDL mode inside a TDL module and adding the ending `_clustermode`. It does not matter from what module the name is taken as the

plugin anyway only allows the usage of a single mode throughout the whole set of TDL modules.

Important attributes:

- `i_frame_factor`:

  This is a mandatory attribute that sets the minimum number of so-called I frames per TDMA round. Initialization frames contain no application data and therefore are protocol overhead. They most yimportantly contain the global time and membership information and are required for the integration of hosts into the cluster. The plugin uses a standard value of 2 here.

Script command:

```
TTA.Cluster_Mode.define ('[clusterName]', i_frame_factor = 2)
```

Link: Cluster_Mode after Cluster_Mode

Description:

This link lets the user specify which modes follow each other by specifying a successor and a predecessor mode. Typically there exists a link to define which mode should follow the `Startup_Mode`.

Generation:

Since there is only a single application mode, there simply needs to be created a link to let the application mode be the next mode after the `Startup_Mode`.

Important attributes:

- `request_mode_change`:

  This mandatory attribute determines in which TDMA round a mode change can be requested by a host. The plugin uses a value of 1 here, so a mode change can be requested in the first TDMA round of the cluster cycle.

Script commands:

```
TTA.Cluster_Mode_after_Cluster_Mode.add
  (TTA.Cluster_Mode.instance ('Startup_Mode'),
  TTA.Cluster_Mode.instance ('[clusterMode]'))
TTA.Cluster_Mode_after_Cluster_Mode.link ('Startup_Mode',
  '[clusterMode]').set (request_mode_change = '1', raw=1)
```

Link: Cluster_Mode of Cluster

Description:

This link assigns a cluster mode to the cluster. Every mode that is used by the application has to be linked to the cluster.

Generation:

The created application mode and the `Startup_Mode` are the only cluster modes we need, and so it is sufficient to create two links to the cluster for them.

Script command:

```
TTA.Cluster_Mode_of_Cluster.add (TTA.Cluster_Mode.instance
  ('[clusterMode]'), TTA.Cluster.instance ('[clusterName]'))
```

Link: Cluster_Mode uses Message

Description:

This link specifies all messages that are used in a cluster mode.

Generation:

Since we have only one cluster mode besides the dedicated `Startup_Mode`, the plugin has to link all messages with it.

Important attributes:

- `d_period`:

  This attribute stands for "design period" and specifies the period of the transmission of a message on the TTP bus. The plugin needs to set this value to the period of the TDL task that belongs to the output port for which the message was created.

- `max_round`, `min_round`:

  These two attributes can be used to influence the scheduling of messages. A maximum and minimum TDMA round number of the cluster cycle can be specified, resulting in the message being scheduled between those two values. It might be necessary for more complicated applications to modify those values. Basically it is not desirable to transfer messages to early in the cluster cycle, as the mechanism to maintain FLET described above relies on the task that produces the message to run before it is transferred. On the other hand, transferring messages too late in the cycle is also not good, as then there might be not enough room for the FT-Com layer to run which has to provide the correct values to the E machine-like task before the beginning of the next FLET period. The plugin does not use the values, which works for simple schedules.

- `redundancy_degree`

  This attribute specifies whether to transfer the message via one or two channels of the TTP bus. The only values allowed are therefore 1 and 2. It is read from the property file so that the user can decide what level of redundancy should be applied. The property and its handling is described in detail in 3.4 and 3.6 respectively.

Script commands:

```
TTA.Cluster_Mode_uses_Message.add (TTA.Cluster_Mode.instance
  ('[clusterMode]'), TTA.Message.instance ('[messageName]'))
TTA.Cluster_Mode_uses_Message.link ('[clusterMode]',
  '[messageName]').set (d_period = [messagePeriod],
  redundancy_degree = [redundancyDegree])
```

Link: Host in Cluster_Mode

Description:

This link connects a host to a cluster mode.

Generation:

Due to the limitation of TTP build there is only one cluster mode. All the plugin has to do is to link every host to this cluster mode and the `Startup_mode`.

Important attribute:

- may_request_mode_change

  This attribute specifies to which mode a host is allowed to change to when it is in the cluster mode linked to it. For the link to the Startup_Mode the only regular cluster mode must be specified here.

Script commands:

```
TTA.Host_in_Cluster_Mode.add (TTA.Host.instance ('[hostName]'),
  TTA.Cluster_Mode.instance ('[clusterMode]'))
TTA.Host_in_Cluster_Mode.link ('[hostName]', '[clusterMode]).set
  (may_request_mode_changes = '[modeAllowedToChangeTo]', raw=1)
```

Object: Subsystem

Description:

This object represents a subsystem, which is the unit of distribution, replication and composability in the TTP tool chain.

Generation:

We already argued above in 3.4 and 3.5 that a subsystem can be mapped one-to-one to a TDL module. In TTPplan the plugin needs to create a subsystem for each module that sends a message on the bus, i.e. for every module that contains tasks with public output ports.

Important attributes:

- reintegration_type:

  Specifies if and how a repaired host hosting that subsystem should be reintegrated. This attribute is read from the property file for each module. Details can be found in section 3.6.

Script command:

```
TTA.Subsystem.define ('[subsystemName]', reintegration_type =
  '[reintegrationType]', raw=1)
```

Link: Subsystem sends Message

Description:

Specifies the messages that are sent by a subsystem.

Generation:

As TDL modules are mapped to TTP subsystems, all messages created for the public task ports of TDL modules need to be linked to the corresponding subsystem.

Script command:

```
TTA.Subsystem_sends_Message.add (TTA.Subsystem.instance
  ('[subsystemName]'), TTA.Message.instance ('[messageName]'))
```

Link: Host runs Subsystem in Cluster_Mode

Description:

This link defines the assignment of subsystems to hosts and cluster modes.

Generation:

The assignment of modules to hosts can be obtained from the property file as can be seen in 3.6 below. As TDL modules are mapped one-to-one to subsystems the plugin has to generate one link for every subsystem and link it to the corresponding host from the property file and to the only cluster mode.

Script command:

```
TTA.Host_runs_Subsystem_in_Cluster_Mode.add (TTA.Host.instance
  ('[hostName]'), TTA.Subsystem.instance ('[subsystemName]'),
  TTA.Cluster_Mode.instance ('[clusterMode]'))
```

3.5.2 Mapping TDL to TTPbuild Objects

After the generation of the bus schedule by TTPplan, system configuration is continued on node-level with TTPbuild. Figure 15 on page 31 illustrates the object model used by TTPbuild. On node-level tasks, messages and subsystems are the key objects. Every task in TDL can be mapped to a TTPbuild task. A task has to be linked to exactly one subsystem. A subsystem can be mapped one-to-one to a TDL module as explained above and therefore every task in a module has to be linked to the according subsystem. In the TTPbuild object model, a task can only interact with other tasks by sending and receiving messages in TTPbuild, which is very similar to TDL where task communicate via input and output ports. An important aspect of the creation of the TTPbuild object models for the hosts is the maintaining of the FLET property and other TDL semantics like the execution of drivers for reading sensors and updating actuators at the begin and end of FLET as described at the beginning of this chapter.

On the following pages all relevant objects and links of the TTPbuild object model are explained together with the information of how the plugin generates them out of TDL modules and the property file.

Object: App_Mode

Description:

This object represents an application mode.

Generation:

An application mode in TTPbuild can be identified with a TDL application mode. Thus, TDL modes can be mapped onto the App_Mode object one-to-one. The current version of TTPbuild only supports a single application mode and therefore all TDL modules must follow this restriction as well, i.e. only TDL modules having one mode are supported. Consequently, only one mode exists and the name of the mode can be used for the App_Mode object.

Important attributes:

For the required attributes maximum_interrupt_latency, neg_correction_limit, pos_correction_limit, neg_synch_limit and pos_synch_limit default values suggested by TTPbuild are used.

Script command:

```
TTA.Node.App_Mode.define ('[applicationMode]',
  maximum_interrupt_latency = '150 us', pos_synch_limit = 'max (
  TTA.Cluster.clock_sync.macro_tick_length / 1000 * 2,
  TTA.Cluster.tc_period * 0.0015)', neg_synch_limit = 'max (
  TTA.Cluster.clock_sync.macro_tick_length / 1000 * 2,
  TTA.Cluster.tc_period * 0.0015)', neg_correction_limit = 'max (
  TTA.Cluster.clock_sync.macro_tick_length / 1000 * 3,
  TTA.Cluster.tc_period * 0.002)', pos_correction_limit = 'max (
  TTA.Cluster.clock_sync.macro_tick_length / 1000 * 3,
  TTA.Cluster.tc_period * 0.002)', raw=1)
```

Link: App_Mode maps_to Cluster_Mode

Description:

This link associates an application mode of a node to a mode on cluster level.

Generation:

Since there is only a single application mode and a single cluster mode those two have to be linked.

Script command:

```
TTA.Node.App_Mode_maps_to_Cluster_Mode.add (TTA.Node.App_Mode.instance
  ('[applicationMode]'), TTA.Cluster_Mode.instance ('[clusterMode]'),)
```

Object: Host

Description:

Represents a host or node of the cluster.

Generation:

The host object was already generated by TTPplan.

Important attributes:

- `node_config`:

  Specifies the type of hardware used. It is set to "TTPpowernode_C2", which is used in the TTTech TTP development cluster.

Script command:

```
TTA.Host.customize ('[hostName]', node_config = 'TTPpowernode_C2',
  raw=1)
```

Object: Task

Description:

This represents a task executed by the TTPos operating system. There are user-defined tasks that implement the actual application and also system tasks that are automatically generated in order to handle the transmission and reception of messages and to perform time synchronization with the other nodes of the cluster.

Generation:

Basically every TDL task is mapped onto a corresponding TTPbuild task with its name and period. But also additional tasks have to be created for maintaining the FLET property and the execution of TDL drivers, which are functions that are performed by the E machine-like task as described at the beginning of this chapter.

Important attributes:

- `time_budget`:

  TTPbuild uses the concept of a time budget. It is calculated by taking the worst case execution time (WCET) of a task and adding some overhead needed by the operating system to actually switch between task (context switching time). The plugin neglects this time and sets the time budget to the TDL WCET value. For the E machine-like tasks the minimum value for the time budget of 75 micro seconds-like is used because the execution of drivers in the E machine-like task should not take more time as they are logically executed in zero time. Also the retransmission of messages that are received on behalf of a task does only take very little CPU time.

- `period`:

  This attribute is used to specify the period of the task in micro seconds. It is set to the period of the TDL task. For the E machine-like tasks also the period of the corresponding TDL task is used.

- `deadline`:

  Used to set a deadline for the task to influence the scheduling of TTPbuild to schedule the task in a way that it will finish before the deadline. This value is relative to the task period and in microseconds. For normal tasks this value is left blank because TTPbuild will schedule them with respect to the time when the messages a task consumes are received and the time messages a task produces have to be available for other tasks. For the periodic E machine-like task this value is set to ensure that it is scheduled at the beginning of every period.

- `phase`:

  The phase is the time interval between the beginning of the cycle and the execution of the task. Again this is not set for normal tasks but set for E machine-like tasks to ensure that they are scheduled correctly at the beginning of the FLET.

- `time_source`:

  This attribute defines the time source for the task. There are two options here: `local_time`, which refers to the local clock of the host, and `reference_time`, which refers to the global clock obtained by the synchronization mechanisms of the TTP protocol. In most applications this choice is not that important, because the local clock is synchronized to the reference time anyway. The default is `local_time`, which has the advantage that it is available regardless of a working connection to the bus.

Script command:

```
TTA.Node.App_Task.define ('[taskName]', time_source =
  'local_time', time_budget = '[timeBudget]', period =
  '[period]', deadline = '[deadline]', phase = '[phase]', raw=1)
```

Link: Task in App_Mode

Description:

This links a task to an application mode.

Generation:

Since there is only one application mode, every task is linked to it. Also every E machine-like task has to be linked to it.

Script command:

```
TTA.Node.Task_in_App_Mode.add (TTA.Node.App_Task.instance
  ('[taskName]'), TTA.Node.App_Mode.instance ('[applicationMode]')
```

Object: Subsystem

Description:

The subsystem object is the same as in TTPplan. Subsystems that were already created in TTPplan are available in TTPbuild as well and cannot be edited or deleted. In addition there is also the possibility to create node-local subsystems.

Generation:

For normal TDL application tasks the subsystems that were already created with TTPplan are used. For the E machine-like tasks a separate subsystem named "emachine" is created. It would also be possible to assign the E machine-like tasks to one of the already created subsystems, but as those tasks serve multiple modules and consequently subsystems created for them, it would be misleading to do so.

Script command:

```
TTA.Subsystem.define ('emachine')
```

Link: Subsystem runs Task

Description:

Specifies which task is run by which subsystem. Every task is required to be run by exactly one subsystem that was specified to send the message produced by the task in TTPplan.

Generation:

Since TDL modules are mapped one-to-one to subsystems, every task is linked to the according subsystem that was created for the module in TTPplan. All E machine-like tasks are linked to the "emachine" subsystem.

Script command:

```
TTA.Node.Subsystem_runs_Task.add (TTA.Subsystem.instance
  ('[subsystemName]'), TTA.Node.App_Task.instance ('[taskName]'))
```

Link: Host runs Subsystem in Cluster_Mode

Description:

This link specifies which subsystem is executed on which host in which cluster mode.

Generation:

This link is already present in the host object model, as it was created by TTPplan before.

Object: Message

Description:

This object represents a message on node level. It can either be a global message that is transferred over the TTP bus and was already defined with TTPplan, or a local message that is used to transfer values between tasks running on the same host.

Generation:

Global messages were already defined with TTPplan and cannot be altered in TTPbuild. Local messages are generated for transferring values from the E machine-like tasks to application tasks in order to transfer sensor readings and messages received from the bus. For actuator updates local messages from the tasks to the E machine-like tasks are created.

Important attributes:

- `d_period`:

  Specifies the design period for local messages. The message is transferred once per period. It is set to the period of the E machine-like task and the sending or receiving task

- `init_value`:

  This attribute sets an initialization value for the message. TDL supports the specification of initialization values for task output ports and actuators. The plugin supports the usage of constants as initialization values in TDL modules and applies them to the `init_value` attribute.

Script command:

```
TTA.Message.define ('[messageName]', d_period = [period],
  init_value = [initValue])
```

Link: Message uses Msg_type

Description:

This link assigns a message type to a message. The message types were already created in TTPplan.

Generation:

The type of a message is derived from the type of the task port the message is created for.

Script command:

```
TTA.Message_uses_Msg_Type.add (TTA.Message.instance ('[messageName]'),
  TTA.Msg_Type_P.instance ('[messageType]'))
```

Link: Task uses Message

Description:

This link specifies which messages a task uses. Every message that was specified to be sent by the host in TTPplan before must be linked to a task and selected to be sent by it.

Generation:

For every TDL task it is necessary to analyze the input and output ports and link messages according to them. For sensor input ports the local messages from the E machine-like task where the sensor code is executed to the task that uses the sensor must be linked. Messages from other tasks also have to be passed via the E machine-like tasks to ensure the FLET property of TDL. For output ports it depends on whether a task is tagged with the public keyword or not. The output ports of public tasks are transferred over the bus and therefore have to be linked to the cluster messages which where already generated with TTPplan. For non-public tasks local messages that contain their output port values have to be created and linked with the tasks.

Important attributes:

- `received`:

  This attribute selects whether or not the linked message is received by the task. Valid values are yes and no. Every message on the bus can be received by every task without changing the bus schedule, as every message is broadcasted and can be received by every host. If a task has to receive messages from the bus, creation of an additional task that implements the fault tolerance communication (FT-Com) is initiated. This task will be scheduled before the consuming task to prepare the value for it. Therefore every such message sent over the bus also consumes CPU time on the sending and the receiving host and complicates the task schedule.
  Basically every input port of a task results in a received message. For messages from the bus this means that the E machine-like task has to receive the message and pass it on to the task by means of a local message. In addition, TDL provides for every task that the output port value of the last round is available to it in the current round as well, regardless if the task actually uses it or not. For public tasks this means to receive the output port message from the bus with an FT-Com layer task and for local tasks it means to create a message that is passed through the E machine-like task.

- `sent`:

  This attribute selects if a message is sent by a task. This also triggers the creation of FT-Com layer tasks when the messages are broadcasted on the TTP bus. Basically for every task output port one message has to be sent that either is a cluster message or a node-local one to other tasks or to the E machine-like task for actuator updates. The E machine-like tasks itself send messages containing sensor readings and received messages from the bus that are forwarded via local messages to the appropriate tasks.

- access_type:

  The access type can be raw, agreed or both agreed and raw. The plugin sets this to the default value of agreed, which is suitable for most applications.

Script commands:

```
TTA.Node.Task_uses_Message.add (TTA.Node.App_Task.instance
  ('[taskName]'), TTA.Message.instance ('[messageName]'),
  access_type = 'agreed', raw=1)
TTA.Node.Task_uses_Message.link ('[taskName]', '[messageName]').set
  (sends = '[isSent]', receives = '[isReceived]', raw=1)
```

3.5.3 Glue Code Generation

The so-called glue code for the integration of TDL and the TTTech TTP tools consists of two parts. One part of it is wrapper code that integrates the C functionality code that is provided for all sensors, actuators and tasks of every TDL module with the TTP tools. The specification of TDL includes language binding rules that outline how the functionality code for the C language should look like. The advantage of such a standard is that the code is reusable even in the case of different target platforms. The goal for the design and implementation of the glue code was to follow those standards as closely as possible and to achieve a certain degree of platform independence for the functionality code. The other part of the glue code is code that implements the E machine-like tasks that have two purposes: The execution of drivers, which is sensor and actuator code, and the handling of retransmission of messages in order to maintain the FLET property of TDL. Those two parts will be explained below in detail together with strategies of automatic code generation for them.

Note that for the implementation of the plugin only a basic set of language binding rules were used due to the fact that the rules for the C language were still in developmental state. The rules are most probably not identical in a later version as rules for the type mapping and the naming of functions to support qualified names might be created or changed.

TDL Compiler Language Bindings

We reconsider the simple TDL module lightController.tdl as introduced in 2.3.2:

```
module lightController {

  sensor
    int brightness uses getBrightness;

  actuator
    int light uses setLight;

  public task calc [100us] {

    input
       int brightnessValue;
```

```
    output
       int lightValue := 0;
    uses calcImpl(brightnessValue, lightValue);
  }

  start mode controlLight [4000us] {
    task
      [1] calc(brightness);
    actuator
      [1] light:=calc.lightValue;
  }
}
```

In this module there are three calls to external functionality code, indicated by the keyword `uses`. The following functionality code header file `lightController.h` must be provided:

```
int getBrightness();

void setLight(int lightValue);

void calcImpl(int brightness, int *lightValue);
```

Note that sensor getter functions are parameter-less functions with a single return value and actuator setter functions have no return value and a single parameter. Functions that implement tasks have no return value and their parameters are passed according to the order in the TDL code, where input ports are passed by value and output and state ports are passed by reference. The reference to the output port variable initially contains the value from the last execution of the task. In addition to the header file, a file `lightController.c` must exist that contains the actual implementation of the functions.

TTP tools task implementation

In TTTech's TTP toolchain as a last step the code for the task implementation must be provided. The tasks are specified with their period and the messages they send and receive. It does not matter whether the message is transmitted over the TTP bus or whether it is a local message as both are handled in the same way by TTPos and the FT-Com layer respectively.

Let's assume a task `increment` that consumes the message `inputValue` and sends the message `outputValue`. An implementation of this task would look like the following:

```
tt_task (increment)
{
  tt_Raw_Value (outputValue) = inputValue + 1;
}
```

`tt_task` and `tt_Raw_Value` are both macros that are either defined in a TTPos library file or a file that is created by TTPbuild containing the FT-Com layer code. The

programmer can assume that at the beginning of the execution of the code all messages a task receives are available as variables with the current value of the message. The name of the variable equals to the name of the message that was given in TTPplan and TTPbuild respectively. The tools handle the generation and execution of the FT-Com layer code that receives messages from the bus as needed. Arbitrary C code can be used in the task implementation. In contrast to TDL semantics, also sensor getter and actuator setter functions are typically included in the task code. The output messages can be passed by calling the macro `tt_Raw_Value (message)` for every message. TTPos and the FT-Com layer then handle the transmission of messages either locally or via the TTP bus.

Let us take a look at the resulting glue code from the example TDL module `lightController.tdl` above. Before the glue code can be generated, a task in TTPbuild must be specified that has the appropriate period of 4000 micro seconds and a time budget of 100 micro seconds. Furthermore the two messages `brightnessValue` for the input and `lightValue` for the output ports must be created. Because TDL requires that the output port of a task contains the output value produced in the last round, we need an additional message from the E machine-like task that we name `lightValue_in`. So the glue code that maps the TDL functionality code to the TTP task code looks like this:

```
#include lightController.h

tt_task (calc)
{
  calcImpl (brightnessValue, &lightValue_in);
  tt_Raw_Value (lightValue) = lightValue_in;
}
```

Note that `brightnessValue` is passed by value and `lightValue_in` is passed by reference. The function `calcImpl` modifies the value `lightValue_in`, which is then passed on to the macro `tt_Raw_Value` in order to set the value of the message `lightValue`.

E Machine-like task Generation

The E machine part of the glue code consists of the execution of sensor readings and actuator updates and the retransmission of messages. A retransmission is necessary when a task receives a message from the TTP bus. As described in 3.5 in order to ensure that the FLET property of TDL is not violated, the E machine-like task receives the message and then passes it on to the task with the help of a local message. To perform a retransmission, an E machine-like task must be created with TTPbuild with the corresponding received and sent messages. The actual retransmission is done with one line in the task code that copies the value of the received message to the sent one:

```
tt_Raw_Value (sentMessage) = receivedMessage;
```

The execution of drivers for sensor readings and actuator updates is also done by making the E machine-like task send and receive the appropriate messages. For sensor readings only local messages have to be created as public sensors are not supported. Actuator updates might also require the creation of messages on the bus, depending on if the task that sets the value for it is located on a remote node or not.

The following part of the glue code implements the E machine-like task for the example module `lightController.tdl` above.

```
#include lightController.h

tt_task (emachine)
{
  setLight(lightValue);
  tt_Raw_Value (brightnessValue) = getBrightness();
}
```

A special case occurs when a sensor is connected directly to an actuator. An example for such a situation is the following modified version of the example above:

```
module lightController {

  sensor
    int brightness uses getBrightness;

  actuator
    int light uses setLight;

  start mode controlLight [4000us] {

    actuator
      [1] light := brightness;

  }

}
```

In this case the value of the sensor is read and within the same instance of the E machine-like task the actuator must be updated with the value. This results in a different handling of the sensor inside the E machine-like task, whereas the code generated for the actuator remains the same:

```
#include lightController.h

tt_task (emachine)
{
  int brightness = getBrightness();
  setLight(brightness);
}
```

For the sensor a local variable with the name of the sensor is generated. Compared to the solution of directly calling `setLight(getBrightness())` this has the advantage of unchanged code generation for the actuators and it also prohibits that the sensor is called more than once when it is used by multiple actuators or if it is broadcasted in a message on the TTP bus.

Message Status Sensor

As described in 3.4 above, the plugin supports a special sensor that gives TDL access to the message status value that is provided by the FT-Com layer. The message status of an example message `lightValue` can be obtained by calling the macro

`tt_Message_Status (lightValue)`

So for a special sensor declaration like

```
sensor
  int lightValue_messageStatus uses REPL_lightValue;
```

we need to replace the call of the function `REPL_lightValue()`, that normally would be used in the glue code of the E machine-like task, with the call to obtain the message status `tt_Message_Status (lightValue)`.

3.5.4 Type Mapping

For a successful generation of code for the TTP platform out of TDL modules and functionality code, the types that TDL uses must be mapped to the types of the TTP tools. The TTP tool chain itself contains two different type systems. As applications for the TTP platform are developed in C it is necessary to use the types of this language. But there is a different notion of types for the TTP communication bus which uses its own type classification. These types must be mapped to each other as well, which is done by the TTP tools by means of the `Msg_Type` object, which contains a number of attributes for type specification. We need to have a way to specify the mapping of a TDL type to those two different type systems.

The goal for the type handling of the plugin was to provide a standard type mapping that is suitable for most applications, but also to give the user the ability to customize it in case he or she has specific requirements. The solution was to use an external file that specifies the mapping and to provide a standard version of the file that contains a default mapping. This gives the user the ability to conveniently modify the mapping when needed. As file format the Java property file format was used, same as for the main property file that the plugin uses (see 3.6). The file containing the type mapping is called `types.properties` and must be located in the `resource` directory of the plugin that is specified in the `TTPPLatform.properties` file. All properties of a type are directly used as attributes for the `Msg_Type` object in the TTP tools. For every type the following properties must be specified in the file:

`type.Length=`

This property specifies the length of the type for TTPplan, i.e. the amount of bits and bytes that the type uses for message transmission. The syntax is [bytes][:bits], so for example 2 means 2 bytes, :10 means 10 bits and 1:2 means one byte plus 2 bits. The reason for this exact specification down to single bits is to enable the user to avoid any waste of bandwidth. For example a boolean value this way only takes one bit and for a 12 bit value from an A/D converter only those 12 bits are allocated within a message to transfer the value on the TTP bus.

`type.Category=`

This property specifies the category of the type for TTPplan. Valid categories are:

- INT: Used for signed integer values.
- UINT: Used for unsigned integer values.
- REAL: Used for floating point values.

`type.CTypeLength`

This property specifies the length of the type for the C programming language, i.e. it is equal to the value returned by `sizeof()`.

`type.CTypeDef`

This property lets the user specify the C type definition.

The following is the type mapping that the file `types.properties` contains by default:

```
short.Length=2
short.Category=INT
short.CTypeLength=2
short.CTypeDef=short int

boolean.Length=1
boolean.Category=UINT
boolean.CTypeLength=1
boolean.CTypeDef=unsigned char

byte.Length=1
byte.Category=UINT
byte.CTypeLength=1
byte.CTypeDef=unsigned char

int.Length=4
int.Category=INT
int.CTypeLength=4
int.CTypeDef=long int

long.Length=4
long.Category=INT
long.CTypeLength=4
long.CTypeDef=long int

float.Length=4
float.Category=REAL
float.CTypeLength=4
float.CTypeDef=float

double.Length=8
double.Category=REAL
double.CTypeLength=8
double.CTypeDef=double

char.Length=1
char.Category=INT
char.CTypeLength=1
char.CTypeDef=unsigned char
```

3.6 Property File for Specification beyond TDL

This section contains a list of all the various properties that need to be specified in a separate file in addition to the TDL modules. For every item a description is provided together with an explanation why it is needed. An example property file for the demo application that is filled with proper values can be found in section 4.2.2 below.

The format of the property file is a standard Java property file as used by the class `java.util.Properties`. It is a text file with name-value pairs separated by a = symbol. The file is required to be named `TTPPlatform.properties` and has to be placed in the directory that is specified as destination directory when calling the TDL compiler.

Program and File Location Parameters

The plugin needs a number of files and executables for proper operation. The following properties tell the plugin where to find those resources.

`TTPPlanLocation=`

The value of this property must contain the path to the executable of the batch version of TTPplan. Typically the file is located at `C:\TTTech\TTPplan\4.4\TTPplan_batch.exe`.

`TTPBuildLocation=`

This property specifies the location where the plugin can find the batch version of TTPbuild. By default this file is installed at `C:\TTTech\TTPbuild\4.4\TTPbuild_batch.exe`.

`CMDLocation=`

The value of this property must contain the path to the executable `cmd.exe`, which is the command line interpreter of the Windows operating system. It is needed by the plugin for the execution of batch files such as the one that handles the compilation of a node. Typically this file is located at `C:\windows\system32\cmd.exe`.

`ResourceDirectory=`

Unlike the last properties this is a path to a directory and not to a file. The following files come with the plugin in the directory `resource` and must be placed somewhere on the local system:

- `main.c` is a C code file that contains some standard initialization routines for nodes of the TTP cluster and is originally provided by TTTech as part of the demo application for the TTP tools.
- `make.bat` is a batch script that controls the invocation of the Windriver Diab C compiler that compiles and links the source code for every node.
- `prj_setup.bat` is a helper file to `make.bat`.
- `types.properties` is a Java property file that specifies the type mapping of TDL types to types in C and the TTP tools and contains default values for all standard TDL types.

Plugin Control Parameters

`LastModule=`

This property is needed to tell the plugin the name of the last module that is passed to the TDL compiler as a command line parameter. All modules of the distributed application have to be passed at once when calling the compiler at the command line. The plugin needs to have them all compiled and available before it can start

processing them and as this data is not available through the plugin interface of the compiler, the plugin needs to cache all of them until the last module is compiled. To know what the last module is it compares its name to the value of this property.

`FilesToCopyForEachNode=`

With this property a number of files can be specified that will be copied from the destination directory, which was passed to the TDL compiler, to the directory of every node that contains all files that will later be compiled for it. An example for such files would be driver code for sensor readings and actuator updates that can be used identically for each node and so only have to be created and maintained in a single file. It is optional whether to use this parameter or not as driver code can as well be included in the functionality code file for every module.

TTP Tools Parameters

`ClusterName=`

This value defines the name of the cluster that will subsequently be used in the TTP tools. This also influences the file name of the cluster database created by TTPplan.

`TransmissionSpeed=`

This entry specifies the transmission speed on the TTP bus in kilobits per second. The value may be up to 5000. The recommended value is the maximum of 5000 as this gives the cluster schedule algorithm the highest degree of freedom and makes it easier to schedule the messages. Consequently it also simplifies the finding of a task schedule for single nodes and increases the probability that a schedule is found that conforms to the FLET property of TDL.

Distribution Parameters

```
TDLModule1.Node=Node1:Node2
TDLModule2.Node=Node3
```

This property consists of the name of a TDL module followed by the ending `.Node`. The value indicates on which node the module should be executed. The value can contain the name of a single node or multiple nodes separated by a colon. Specifying multiple nodes here results in replicated execution of the module on these nodes. The assignment of modules to nodes also serves for getting a list of all nodes the cluster consist of, as this is specified nowhere else. So every usage of a new name of a node implicitly creates one.

Fault Tolerance Parameters

`TDLModule1.RDA=`

This property is only needed for modules that are replicated by an assignment to more than one node. When a module is executed on more than one node, the values it produces might be different due to a failure in for example either the node's hardware, operating system or the communication subsystem. Therefore a so-called replica-deterministic agreement algorithm must be specified, that integrates the possibly different values of a message to a single one that is consistent throughout the whole system. The value of this property is directly passed to TTPplan and only the following strings are valid as they represent different RDA algorithms [6]:

RD_1_valid, RD_Add, RD_Average, RD_M_Vote and RD_Piecewise. The RD_1_valid algorithm picks any valid value. RD_Add adds the value of all valid values of the replicated modules. RD_Average calculates the average of all valid values. RD_M_Vote stands for a majority vote algorithm that picks the value that the majority of the replicas produce. RD_Piecewise is used for structured data types and enables the application of different RDA algorithm to every element. As structured data types are not supported by the plugin tool chain this algorithm cannot be used.

TDLModule.ReintegrationType=

Reintegration occurs after the failure of a node. In case of a transient failure the node might be fully functioning after a short amount of time and ready for operation again. A permanent failure might require the repair or replacement of the node. In both cases the node must be reintegrated into the running cluster. This can be done by resetting the whole cluster by restarting all nodes. In safety critical application this is often not an option and it is required that a node performs reintegration while the cluster is running. Reintegration is specified on TTP subsystem level. With this property the user can decide whether a module should perform reintegration or not. This setting is necessary for all modules. There are two valid strings for this setting: Reinit_Reintegration and No_Reintegration. Reinit_Reintegration tells the plugin to perform reintegration into the running cluster whereas when specifying No_Reintegration this does not happen and the TDL module only continues working when the whole cluster is reset.

TDLPublicOutputPort.ChannelRedundancy=

This property must be set for every output port of a public task. These are exactly the ports for which messages on the TTP bus are generated. The TTP bus has two channels that can be used redundantly for improved safety and fault tolerance or independently for maximized data throughput. The value of this property can be either 1 or 2. 1 causes the message to be transmitted only on one channel of the TTP bus and 2 triggers the redundant transmission of the message.

3.7 Implementation of the TTP TDL Plugin

This section explains step-by-step what the TDL TTP plugin for creating code for the TTP platform does. It describes how the generation of the scripts for TTPplan and TTPbuild is done, how the glue code is generated and how all files are compiled and linked together to get a working application binary.

Execution Environment

The TDL plugin implementation requires a number of tools for proper operation. This section lists those tools and also contains information on the exact version that was used for the development of the plugin. Other versions might work as well, but as features and functions might change this cannot be guaranteed.

The plugin is implemented as a set of Java classes under the Java Development Kit (JDK) version 1.4.2. Therefore it needs a Java Runtime Environment (JRE) that is able to execute code developed with this version of the JDK.

The version of the TDL compiler used is a development version from April 2004. It contains only basic support for distribution and will certainly be developed further.

The version of the TTP tool suite by TTTech used for the plugin development is release R6.3 from December 2003. It contains TTPplan version 4.4, TTPbuild version 4.4, TTPload version 5.4, TTPview version 5.10 and a version of TTPos for the MPC555 platform with TTPChip AS8202 in version 4.4. The tools must be installed with valid installation keys. It is important to properly customize the file `mysetup.bat` that is typically located in `C:\TTTech\BSP`. Most importantly this file contains the path to the C compiler and if not set properly the plugin is unable to invoke the compilation of files.

As C compiler the Diab C/C++ Compiler for PowerPC in version 5.0 by Windriver was used. This compiler is recommended by TTTech and also shipped together with their development hardware. It requires a valid license. During installation a lot of questions concerning the hardware target are asked. It is not necessary to answer them correctly as the batch files for compilation overrule those settings anyway.

The hardware platform used for development was a TTP development cluster by TTTech that consisted of four TTP PowerNodes PN212 and a TTP MonitoringNode.

The plugin only works under Microsoft Windows despite the fact that it is a Java program. The reason is that the TTP tools and also the batch files for compilation that are included only work on Windows and to date there is no version for other operating systems available.

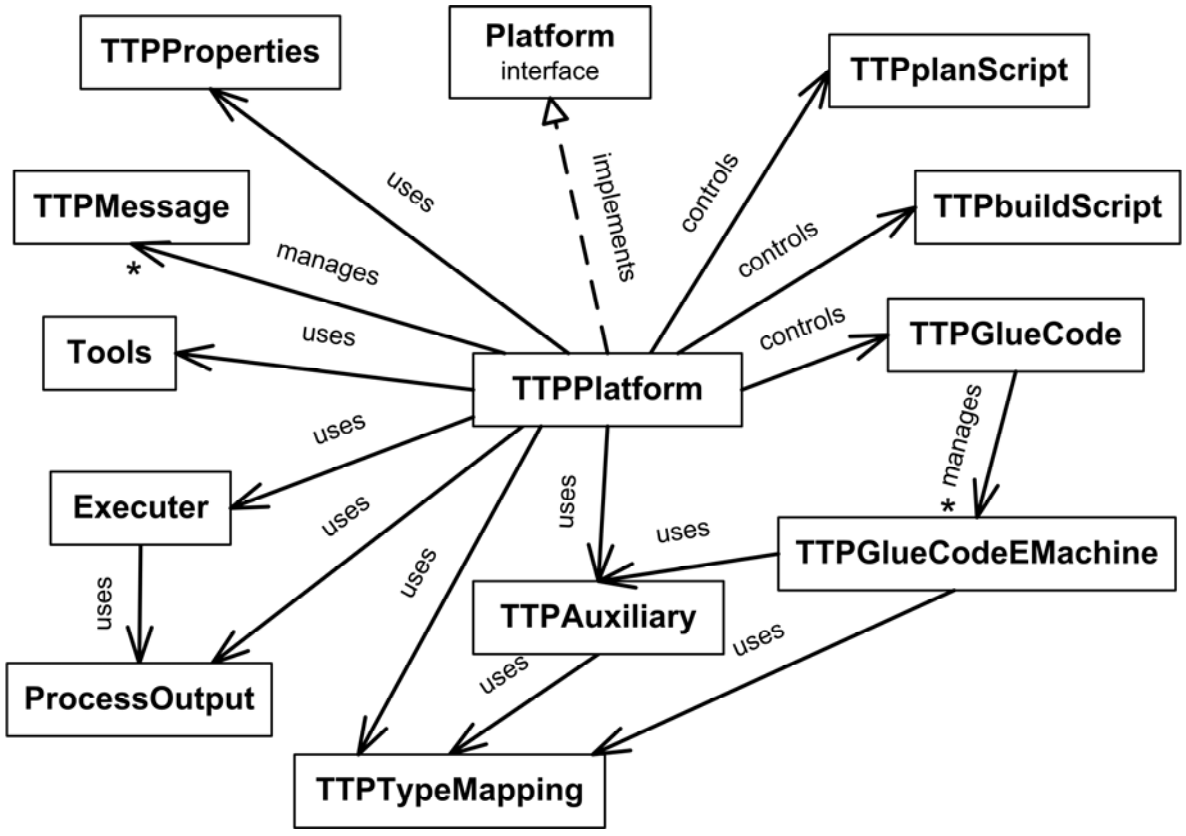

Figure 19 Plugin Class Diagram

3.7.1 Classes

All classes the plugin consists of are contained in the Java package `emcore.tools.tdlc.platform.ttp`. Figure 19 presents all classes and their relations to each other. It illustrates that `TTPPlatform` is the main class that coordinates and controls the generation process. The decomposition of the plugin is mainly driven by the structure and type of the output and resulted in three classes `TTPplanScript`, `TTPbuildScript` and `TTPGlueCode` for creating scripts for TTPplan and TTPbuild and generating the glue code that lets the functionality code interface with the TTP platform.

In the following all Java classes of which the plugin consists of are listed and their purpose and function is described.

TTPPlatform

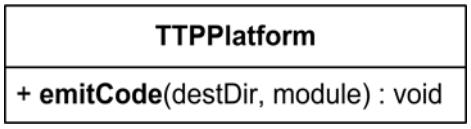

Figure 20 TTPPlatform Class Diagram

This class implements interface `Platform` and represents the core of the TDL compiler plugin for integration with TTP. All internal and external activities for generating binaries for the TTP platform are coordinated by the implementation of method `emitCode`. The name of this class must be specified with option `–platform` of the TDL compiler command line interface in order to activate this plugin.

TTPMessage

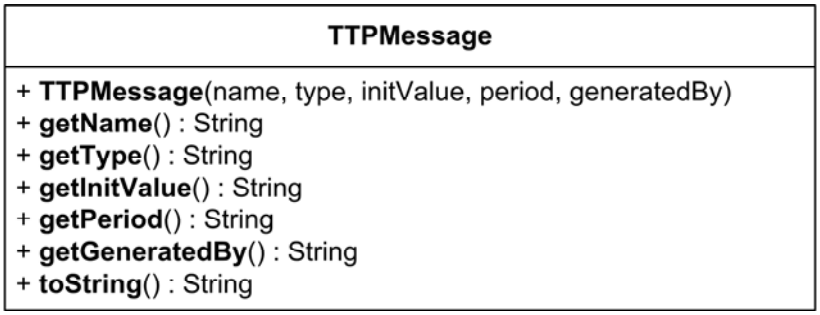

Figure 21 TTPMessage Class Diagram

This class represents a message on the TTP bus and is instantiated and used by class `TTPPlatform`. A `TTPMessage` object is a container for message attributes. It provides getter methods to access the attributes and does not do further calculations.

TTPplanScript

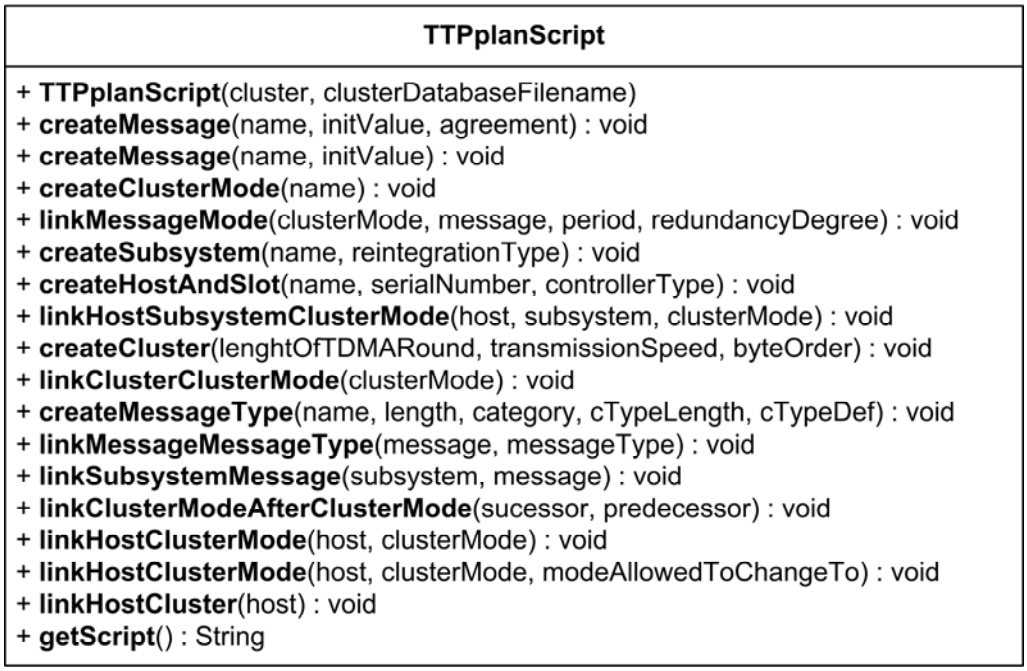

Figure 22 TTPplanScript Class Diagram

This class is a wrapper for generating TTPplan scripts that can be passed as argument to TTPplan in order to generate a valid object model and eventually the cluster communication schedule. The class provides a number of methods for defining the TTPplan object model by creating objects and links between them and specifying their attributes. A description of all relevant entities can be found in 3.5.1. Furthermore it has a method for obtaining the generated script.

TTPbuildScript

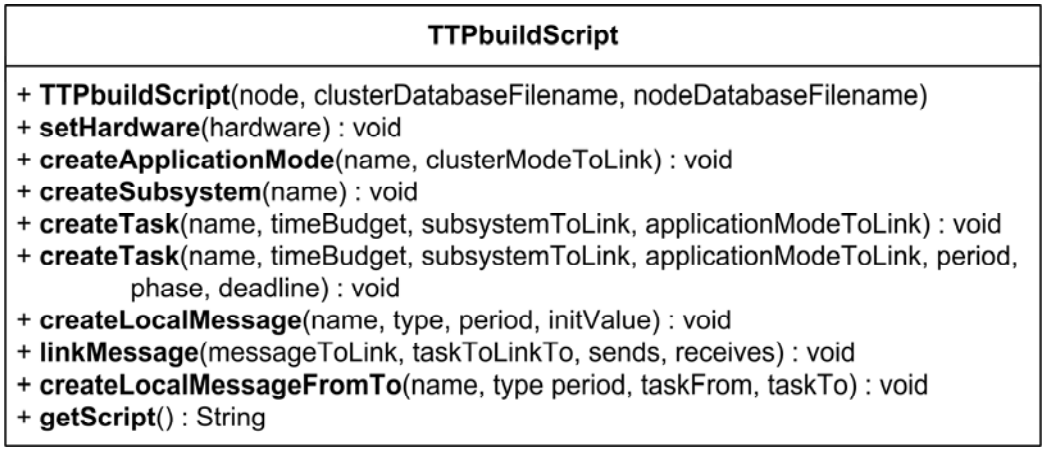

Figure 23 TTPbuildScript Class Diagram

This class is a wrapper for generating TTPbuild scripts. The two TTP tools are very similar regarding their programming interface and so this class is very much like class `TTPplanScript`, with the difference that it is designed to create an object model for TTPbuild. A list of the relevant entities can be found in 3.5.2.

The wrapper classes for TTPbuild and TTPplan do not only map a method call to a single instruction in the script for the tools, but also do some processing. For example when the method `createTask` is invoked, also links are generated to link tasks to a subsystem and an application mode. The idea is to provide a convenient and powerful interface to simplify the usage for class `TTPPlatform`.

TTPGlueCode

This class generates the C glue code for every node. The glue code acts as a middleware layer between the TTP platform, which mainly consists of the operating

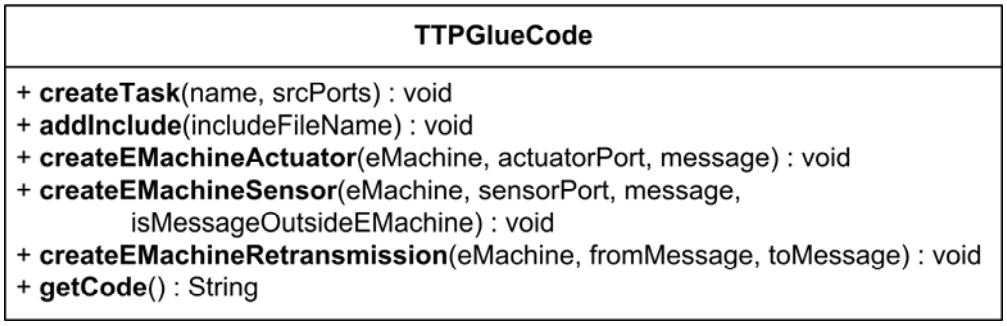

Figure 24 TTPGlueCode Class Diagram

system TTPos and the FT-Com layer, and the functionality code for tasks and drivers that are provided by the user in a standardized form. It also contains automatically generated code for the E machine-like tasks that handle the reception of bus messages and the execution of drivers. The class has methods for adding tasks, sensors, actuators and messages the E machine-like tasks have to retransmit. It also provides a method to obtain the generated C code.

TTPGlueCodeEMachine

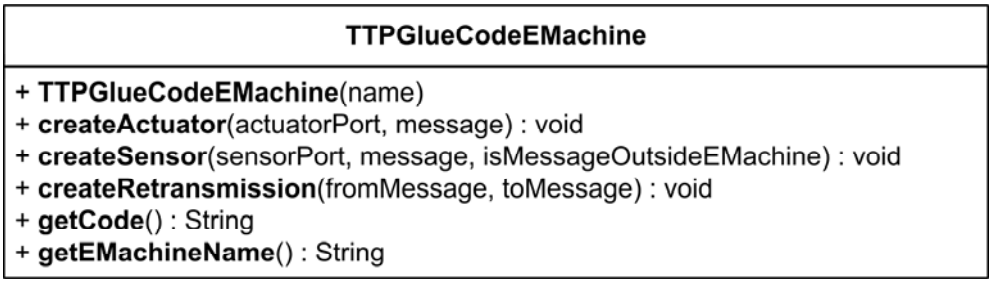

Figure 25 TTPGlueCodeEMachine Class Diagram

This is a helper class for `TTPGlueCode` and is instantiated and maintained by it. Its purpose is to collect lines of code that belong to the code of the E machine-like tasks. This is necessary because these tasks typically contain lines of code for different modules and there may also be multiple E machine-like tasks for different

frequencies. Every instance of this class represents a single E machine-like task and has a method to return the appropriate code to class `TTPGlueCode`.

TTPProperties

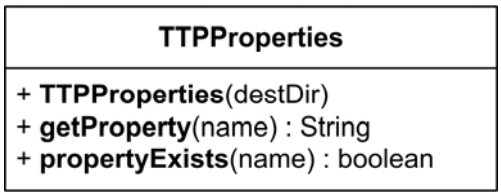

Figure 26 TTPProperties Class Diagram

This class handles the access to the property file that contains specifications in addition to the TDL modules. It is instantiated by an instance of the `TTPPlatform` class with the name of the property file and then provides access to it including error handling for non-existent properties.

TTPTypeMapping

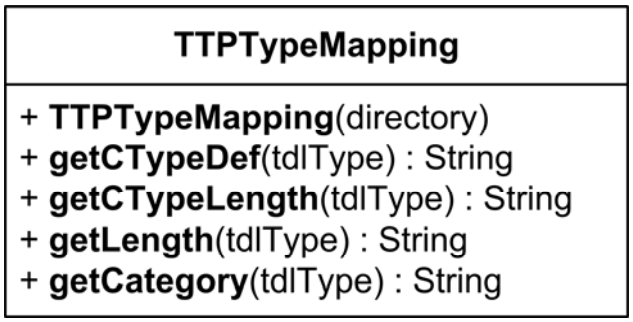

Figure 27 TTPTypeMapping Class Diagram

This class handles the mapping of TDL types to types in C and types specific to the TTP tools. It is instantiated with the directory where the external file `types.properties` is located, which contains a mapping for every standard TDL type and also gives the user the ability to alter the mapping or to define custom types. The class provides four methods for getting the properties of a TDL type.

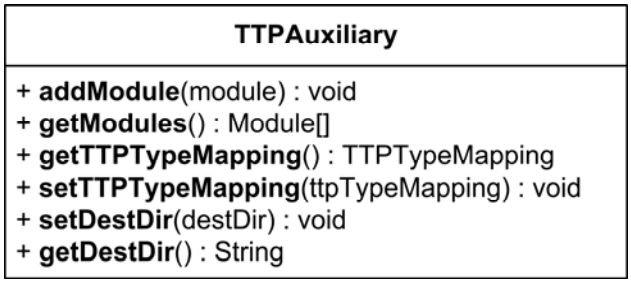

Figure 28 TTPAuxiliary Class Diagram

TTPAuxiliary

This is an auxiliary class which serves other classes with objects that are static throughout the whole plugin lifetime. It provides access to the destination directory and the type mapping class. Furthermore it is used to cache all modules that are processed by the TDL compiler, as the plugin needs to have all compiled modules available before it can start generating scripts and code.

Executer

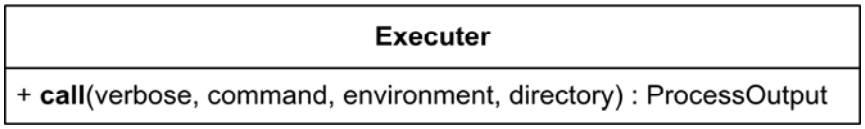

Figure 29 Executer Class Diagram

This class provides the ability to execute external programs from within Java. It also takes care of the standard and error output and the return value of a command. The call method is called with the path and name of the executable, optional environment variables and the working directory. It returns an instance of the class `ProcessOutput` described below.

ProcessOutput

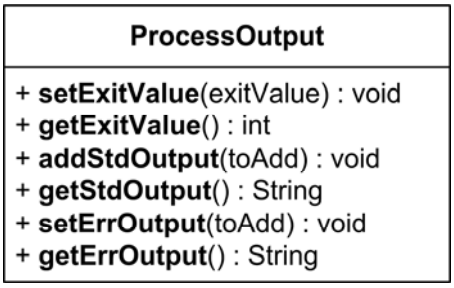

Figure 30 ProcessOutput Class Diagram

This is a helper class for class `Executer`. It is used to collect the standard and error output of an external program execution as well as the exit value of it and provides methods to read this data in a convenient way.

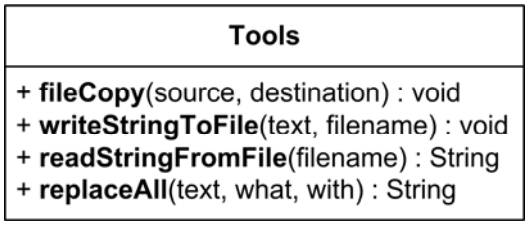

Figure 31 Tools Class Diagram

Tools

This is a helper class that provides static methods for copying files, reading and writing a String to a file and a method to replace all occurrences of a string in another string.

3.7.2 Program Flow

This section describes step-by-step what the plugin does. For better readability the flow of actions is divided in smaller parts that follow each other.

Initialization

After the plugin is called by the TDL compiler via method `emitCode`, the property file is read and the value of `LastModule` is compared to the name of the current module. The plugin has to wait until the last module is compiled by the compiler before it can start its work. All module objects are stored by the `TTPAuxiliary` class in order to be available for every instance of the plugin invoked during the compilation process.

When the last module is reached, the plugin starts with reading the type mapping property file and instantiating the corresponding class `TTPTypeMapping`. It also processes the destination directory and sets the appropriate value in the `TTPAuxiliary` class. All files generated by the plugin are stored in the subdirectory `TTP` which is created in the destination directory that is passed by the compiler as parameter of the `emitCode` method.

TTPplan Script Generation

In order to generate the script for TTPplan, the plugin first needs to determine a list of all messages that need to be transferred via the TTP bus and creates subsystems for them. As mentioned earlier, TTP subsystems can be mapped to TDL modules and so for every module that sends a message a subsystem is created. Then every module is analyzed with respect to public ports for which a message has to be created with the corresponding TTP type that can be found through class `TTPTypeMapping` with the TDL type name. Furthermore the period of each message is computed. Also other properties such as the type of RDA algorithm to be used and channel redundancy are processed. During those actions also the shortest message period is determined that is needed to specify the length of a TDMA round of the cluster.

The next important step is to create hosts and to link them to the subsystem. For this purpose the information which module runs on which host is necessary. This mapping is obtained by reading the property file. All modules are iterated and the `[module].Node` entry in the property file is analyzed. This way also the list of nodes in the cluster is determined. The information gained is used to create hosts, link them to slots and the cluster object and assign subsystems to them. Furthermore during this iteration the TTPbuild script generation is invoked with a list of modules each node has to execute. Finally the script is written to a file for later processing by TTPplan.

A detailed description of how to map the TDL modules to the TTPplan object model can be found in 3.5.1.

TTPbuild Script and Glue Code Generation

For each host a dedicated script for TTPbuild and glue code is generated. The data needed for this step consist of the name of the host, a list of modules the host should execute, the destination directory and the filename of the cluster database.

First all necessary classes, namely `TTPbuildScript` and `TTPGlueCode`, are initialized. Next an iteration over all modules of the host is performed. The first action in this iteration is copying the files `<module>.h` and `<module>.c` to the directory of the host. Those files must exist and must contain the header and body file of the functionality code for the TDL module. The name of the header file is also passed on to the glue code generator class to add an appropriate `include` statement to the glue code. What happens next is to carry out all steps to realize the mapping of TDL modules to the TTPbuild object model and the generation of the glue code that are described in 3.5.2 and 3.5.3 respectively. The plugin does this by calling the appropriate functions of the classes `TTPbuildScript` and `TTPGlueCode`. Finally the glue code and the script for later execution with TTPbuild are stored in the destination directory of the host.

Script Execution and Compilation

After the generation of the scripts for TTPplan and TTPbuild is completed, first the batch version of TTPplan is called with the appropriate script as parameter. The tool outputs the MEDLs for each host and a cluster database file containing the cluster schedule. Successful execution of the tool is checked by analyzing the exit value, which should be zero, and additionally by checking if the output contains the string "`Schedule successfully made`". Otherwise the plugin stops with an error message. For debugging reasons the output of the execution of the tools is always outputted to the console.

Next TTPbuild is called once for every node with the appropriate script generated by the plugin as parameter. The success of the execution is again checked by analyzing the exit value and checking the output for strings that indicate a successful run of the tool. TTPbuild creates three files for every host that are written to the corresponding host directory: `ttpc_ftl.c`, `ttpc_msg.h` and `ttpos_conf.c`. Those files contain the static schedule table, in which all task invocations are specified, and the fault tolerant communication layer that acts as an interface between tasks and messages on the TTP bus.

Before the Diab C compiler can be invoked, some additional files have to be copied from the `ResourceDirectory` specified in the property file. Two of them are the files `make.bat` and `prj_setup.bat` which were provided by TTTech for the compilation of applications developed with their tools. Furthermore the file `main.c`, which contains some initialization routines, needs to be copied to every node directory. `main.c` needs to be patched in order to contain the name of the application mode that is used throughout the system. To work properly the compiler script needs a valid `mysetup.bat` file that is typically located in the directory `C:\TTTech\BSP`, which most importantly contains the path to the compiler binaries. Finally the plugin invokes the compiler for every host, checks if the compiled and linked binary was created successfully and copies the file to the download database directory created by TTPplan before. At this point all what is left to do for the user is to start TTPload and download the application to the TTP cluster hardware.

Chapter 4
Demo Application

4.1 Experimental Setup

The demo application that is used to demonstrate the application of the developed TDL-TTP tool chain is described in this section. Although it is a rather simple example it is sophisticated enough to show basic fault tolerance behavior such as replication and redundancy. Furthermore the hardware and software environment used for the demo is described.

Hardware Setup

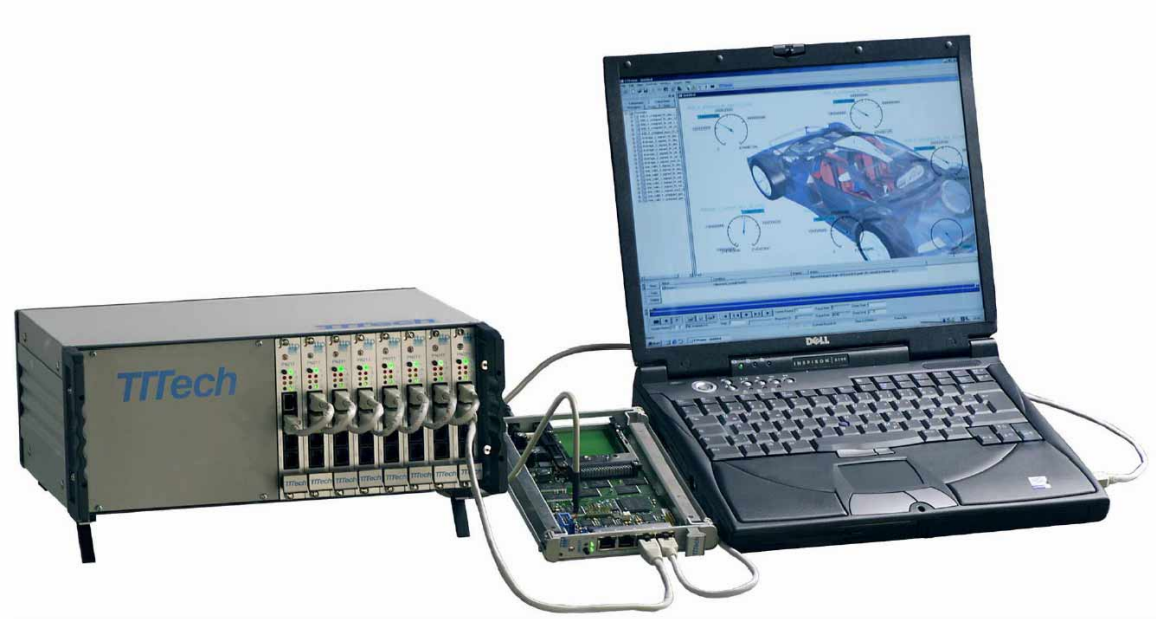

Figure 32 TTP Development Cluster by TTTech

The hardware platform used for the demo application is a TTP development cluster provided by TTTech. Figure 32 shows the setup with the actual cluster on the left, the monitoring node in the middle a standard laptop PC on the right. The cluster consists of a power supply and a number of TTP Powernodes. For the demo application four of these nodes with the model number PN212 were used. The monitoring node acts as a gateway between the TTP bus and standard Ethernet and is used for programming the host CPUs and the TTP chips and for monitoring all data on the TTP bus at runtime.

The TTP-Powernode is a board that integrates a powerful Motorola MPC555 CPU with a TTP-Chip C2 communication controller, which is an implementation of the TTP/C protocol in silicon. [12] contains details of the board layout and functioning.

Software Setup

The software setup consists of the tools listed in 3.7 above, that are required for the TTP plugin for the TDL compiler to run. All tools were installed on a Windows XP system.

Demo Application

The demo application is meant to be a simple demonstration that the TDL plugin for the TTP platform described in Chapter 3 actually works as intended. It uses a number of modules that are distributed on four nodes and that use tasks, sensors and actuators. The goal was to use all fault tolerance mechanisms described in 3.4.

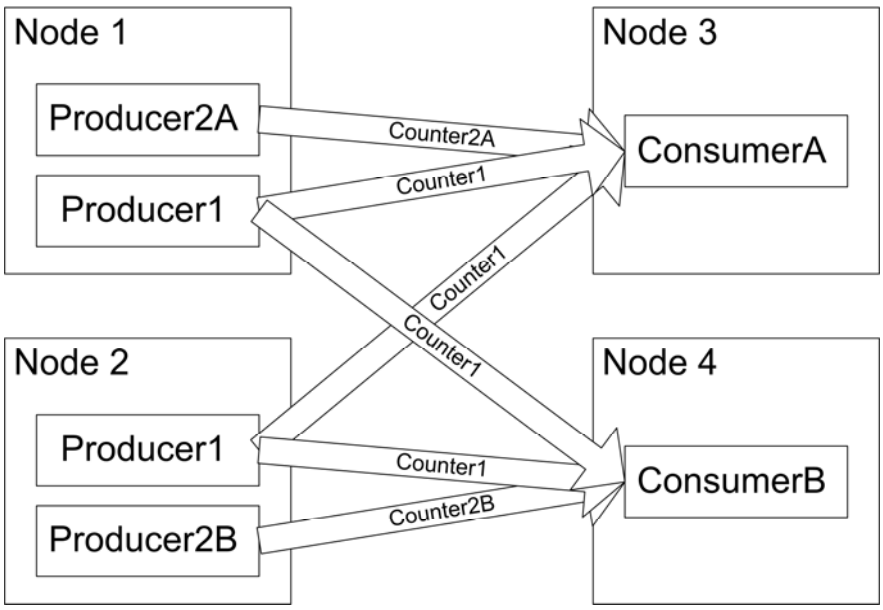

Figure 33 Demo Application Data Flow Diagram

The functionality of the demo application consists of three independent counters that simply count upwards. The four nodes are divided into two producer and two consumer nodes. Figure 33 shows the data flow between the nodes and the associated modules. Each producer - denoted by `Node1` and `Node2` in the figure - generates a counter value and an additional counter called `Counter1` is replicated on both producer nodes. The consumer nodes `Node3` and `Node4` receive the counters. `Node3` receives the counter value Counter2A produced by `Node1` and `Node4` receives the counter from the second producer node denoted by `Node2`. The replicated counter `Counter1` is received on both consumer nodes and in addition both are aware of the number of operational replicas. The values of the counters are indicated by LEDs on each node and the actual values can also be monitored on a PC.

Figure 34 is a symbolic picture of the TTP Powernode with a description of the LEDs. As shown in the figure, five LEDs are available to the application on the host CPU, whereas the other two green LEDs are reserved to display the status of the TTP communication controller. The demo application uses the two yellow HOST4 and HOST5 LEDs to display the current value of the counter. The LEDs blink with a fixed period and the phase of the blinking corresponds to the current value. The idea is to

see that the values on different nodes are the same when the blinking of the LEDs is in sync. The left LED of every node indicates the value of the replicated counter value whereas the right LED indicates the other counter value that is produced or consumed by a node with a different blinking period. So when all nodes of the cluster are started at the same time initially all left LEDs of the nodes and all right LEDs are in sync and will stay in sync unless one of the producer nodes is reset, which also results in a reset of the counter value. On the consumer nodes also the red LEDs HOST2 and HOST3 are used to indicate the status of the replicas of the replicated counter. The two LEDs are used as alarm or warning LEDs that indicate the failure of one replica with one red LED and the failure of both with two red LEDs switched on.

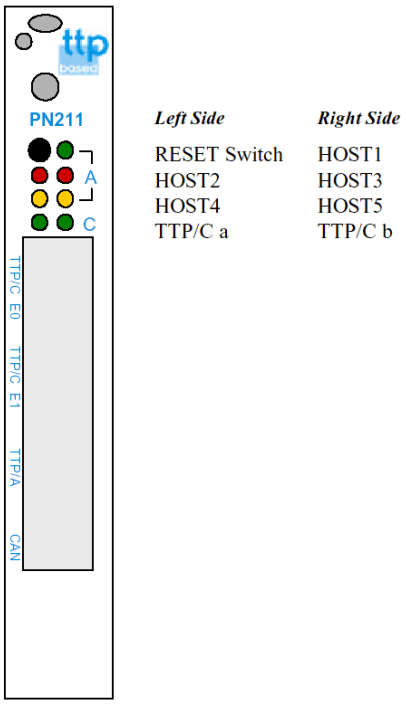

Figure 34 TTP Powernode LEDs

4.2 Implementation

This section is dedicated to the implementation details of the demo application. Everything from the TDL modules and property file entries, the generated object model and bus and task schedules to the generated glue code is presented and discussed.

4.2.1 TDL Code

In the following the TDL code and the corresponding functionality code of the demo application is presented and explained. It consists of five TDL modules that are split in three producer and two consumer modules.

Producer Modules

The producer modules generate a counter value and indicate that value by means of an actuator which is a blinking LED as described above. The three modules are very similar. There are only differences in the name of the module, the counter value and the functionality code calls. As example here is the code of module `Producer1.tdl`:

```
module Producer1 {

  actuator
    short YellowLED1 uses setYellowLED1;

  public task Produce1 [100us] {
    output
      short Counter1 := 0;
    uses produce1Impl(Counter1);
  }

  start mode DemoMode [4000us] {
    task
      [1] Produce1();
    actuator
      [1] YellowLED1:=Produce1.Counter1;
  }
}
```

This module consists of an actuator for driving the LED, a task that actually produces that value and increments it and a mode that invokes the task and the actuator update. The keyword `public` indicates that the task output ports can be accessed by other modules and therefore have to be transferred over the TTP bus.

Drivers like `setYellowLED1` and task code like `produce1Impl` have to be provided in the functionality code file. The file that contains them must be called `Producer1.c` with a corresponding header file `Producer1.h`. `Producer1.c` looks like this:

```
#include "drivers.h"

void produce1Impl(short int *value) {
  value = value + 1;
}
```

As can be seen the task simply adds 1 to the counter that is passed as a reference. The functionality code for the actuator is contained in the referenced file `drivers.h`, which contains all drivers for all modules. The following is an excerpt of the file `drivers.c` that illustrates the relevant functions for the producer module:

```
#include "TTPos.h"

void setYellowLED1(short int value) {
```

```
  LED (LED_YELLOW_1, value & 0x100);
}

void setYellowLED2(short int value) {
  LED (LED_YELLOW_2, value & 0x200);
}
```

This code realizes the blinking of the LEDs according to the value of the counter. The header file `TTPos.h` provides access to the LED functions of the operating system.

Consumer Module

The consumer module receives the counter values from the bus by accessing output ports of modules located on a remote node by using the `import` statement with the corresponding modules. In the following is the TDL code for one of the two consumer modules `ConsumerA.tdl` is presented:

```
module ConsumerA {

  import Producer1;
  import Producer2A;

  sensor
    short Counter1MessageStatus uses REPL_Counter1;

  actuator
    short YellowLED1 uses setYellowLED1;
    short YellowLED2 uses setYellowLED2;
    short RedLEDs uses setRedLEDs;

  start mode DemoMode [4000us] {
    actuator
      [1] YellowLED1 := Producer1.Produce1.Counter1;
      [1] YellowLED2 := Producer2A.Produce2A.Counter2A;
      [1] RedLEDs := Counter1MessageStatus;
    }
}
```

As can be seen the consumer modules do not contain any task. The sensor value and values from other imported modules are directly used as input for actuator updates. The message status of Counter1 indicates how many replicas producing this message are present. This value is accessed by the special driver `REPL_Counter1` that does not require functionality code as it is processed in a special way by the plugin as described in 3.5.3.

The drivers for the actuators are contained in the file `drivers.c`. In addition to the two drivers whose code is denoted above, a driver for the red LEDs is provided:

```
void setRedLEDs(short int value) {
```

```
  if (value > 1)
    {
      LED_OFF (LED_RED_1);
      LED_OFF (LED_RED_2);
    }
  else if (value)
    {
      LED_ON (LED_RED_1);
      LED_OFF (LED_RED_2);
    }
  else
    {
      LED_ON (LED_RED_1);
      LED_ON (LED_RED_2);
    }
}
```

This function implements the red warning LEDs that indicate the failure of one or two replicas of the `Producer1` module.

4.2.2 Property File

A Java property is used for specifying distribution and fault-tolerance aspects and platform specific details. Basically it contains all properties that need to be known in order to generate a distributed, fault-tolerant application for the TTP platform but cannot be determined from the TDL modules. All properties and the syntax of the file are explained in 3.6. The filename has to be `TTPPlatform.properties` and the file must be in the destination directory specified when calling the TDL compiler.

```
TTPPlanLocation=C:\\TTTech\\TTPplan\\4.4\\TTPplan_batch.exe
TTPBuildLocation=C:\\TTTech\\TTPbuild\\4.4\\TTPbuild_batch.exe
CMDLocation=C:\\windows\\system32\\cmd.exe
ResourceDirectory=C:\\Demo\\resource
```

These properties tell the plugin where to find required files and executables.

```
LastModule=ConsumerB
```

The last module can be any module that is later passed to the compiler as the last one in the list of modules to be compiled at the command line.

```
FilesToCopyForEachNode=drivers.h:drivers.c
```

As mentioned above, the demo application uses the same driver code in the files `drivers.h` and `drivers.c` for all modules. These files must be specified to make the plugin copy them to each node directory for compilation and linking.

```
ClusterName=DemoCluster
```

It is required to specify a name for the cluster.

```
TransmissionSpeed=5000
```

For the transmission speed on the TTP bus the maximum value is used. This gives the bus scheduler more freedom to find a correct schedule for the application.

```
Producer1.Node=Node1:Node2
Producer1.RDA=RD_1_valid
Producer1.ReintegrationType=Reinit_Reintegration
```

The module Producer1 is specified to be executed redundantly on two nodes. This also makes it necessary to choose an RDA algorithm. RD_1_valid just picks the first valid message that contains a value produced by the module. For the reintegration type we request that the module tries to reintegrate by reinitialization.

```
Producer2A.Node=Node1
Producer2A.ReintegrationType=Reinit_Reintegration

Producer2B.Node=Node2
Producer2B.ReintegrationType=Reinit_Reintegration

ConsumerA.Node=Node3
ConsumerA.ReintegrationType=Reinit_Reintegration

ConsumerB.Node=Node4
ConsumerB.ReintegrationType=Reinit_Reintegration
```

Also all other modules must be distributed among the nodes. Note that for example on Node1 two modules are executed.

```
Counter1.ChannelRedundancy=2
Counter2A.ChannelRedundancy=2
Counter2B.ChannelRedundancy=2
```

For all public output ports of tasks the channel redundancy is set to 2, so that both TTP channels are used for transmission.

4.3 Execution

In the following the output of the execution of the TDL compiler and the plugin for the TTP platform including the output of invoked TTP tools is presented.

4.3.1 Compiler Invocation

The TDL compiler is invoked for the demo application with the following command:

```
java emcore.tools.tdlc.Compiler –d . –platform
  emcore.tools.tdlc.platform.ttp.TTPPlatform Producer1.tdl
  Producer2A.tdl Producer2B.tdl ConsumerA.tdl ConsumerB.tdl
```

As destination directory the current directory is used. This is also the directory where the property file named TTPPlatform.properties must be located. As usual when calling java programs the file Compiler.class is located in the directory \emcore\tools\tdlc relative to the current directory or a location listed in the CLASSPATH environment variable. This also analogously applies to the file TTPPlatform.class. The TDL module files must be located in the current directory too. The order in which modules are passed at the command line does not matter except for the requirement that modules that provide functionality to other modules need to precede modules that use it. The only requirement is that the last module is the one that is specified as LastModule in the property file.

4.3.2 TTPplan Script

TTPplan is the first tool the plugin creates a script for and executes it. Below the method calls of the wrapper class `TTPplanScript` and the generated script for the demo application together with some remarks are presented. It is the complete script without any modifications.

```
TTPplanScript ttpPlan = new TTPplanScript("DemoCluster",
"C:\Demo\.\TTP\DemoCluster.cdb");
```

```
TTA.Application_Command.run ('File.New', 'DemoCluster')
```

The beginning of the script creates a new cluster database file.

```
ttpPlan.createClusterMode("DemoMode_clustermode");
```

```
TTA.Cluster_Mode.define ('DemoMode_clustermode', i_frame_factor = 2)
TTA.Cluster_Mode_after_Cluster_Mode.add (TTA.Cluster_Mode.instance ('DemoMode_clustermode'),
   TTA.Cluster_Mode.instance ('Startup_Mode'))
TTA.Cluster_Mode_after_Cluster_Mode.link ('DemoMode_clustermode', 'Startup_Mode').set
   (request_mode_change = '1', raw=1)
TTA.Cluster_Mode_of_Cluster.add (TTA.Cluster_Mode.instance ('Startup_Mode'),
   TTA.Cluster.instance ('DemoCluster'))
```

```
ttpPlan.linkClusterClusterMode("DemoMode_clustermode");
```

```
TTA.Cluster_Mode_of_Cluster.add (TTA.Cluster_Mode.instance ('DemoMode_clustermode'),
   TTA.Cluster.instance ('DemoCluster'))
```

This first block deals with the creation of the cluster mode for the TDL mode in the modules. In addition TTPplan requires creating a startup mode as described in 3.5.1.

```
ttpPlan.createSubsystem("Producer1", "Reinit_Reintegration");
```

```
TTA.Subsystem.define ('Producer1', reintegration_type = 'Reinit_Reintegration', raw=1)
```

```
ttpPlan.createMessage("Counter1", 0, "RD_1_valid");
```

```
TTA.Message.define ('Counter1', agreement = 'RD_1_valid', init_value = '0', raw=1)
```

```
ttpPlan.linkMessageMode("DemoMode_clustermode", "Counter1", 4000, 2);
```

```
TTA.Cluster_Mode_uses_Message.add (TTA.Cluster_Mode.instance ('DemoMode_clustermode'),
   TTA.Message.instance ('Counter1'))
TTA.Cluster_Mode_uses_Message.link ('DemoMode_clustermode', 'Counter1').set ( d_period = 4000,
   redundancy_degree = 2)
```

```
ttpPlan.linkSubsystemMessage("Producer1", "Counter1");
```

```
TTA.Subsystem_sends_Message.add (TTA.Subsystem.instance ('Producer1'), TTA.Message.instance
   ('Counter1'))
```

```
ttpPlan.createMessageType("short", "2", "INT", "short int", "2");
```

```
TTA.Msg_Type_P.define ('short', length = '2', type_cat = 'INT', typedef = 'short int',
   type_length = '2', raw=1)
```

```
ttpPlan.linkMessageMessageType("Counter1", "short");
```

```
TTA.Message_uses_Msg_Type.add (TTA.Message.instance ('Counter1'), TTA.Msg_Type_P.instance
   ('short'))
```

This block defines the subsystem `Producer1` for the corresponding module. The messages for the module are created and linked to the appropriate message type and subsystem. Note that the message type `short` is also created on first usage. With the link between a message and the cluster mode also the attributes for the message period and its redundancy degree are determined.

```
ttpPlan.createSubsystem("Producer2A", "Reinit_Reintegration");
```

```
TTA.Subsystem.define ('Producer2A', reintegration_type = 'Reinit_Reintegration', raw=1)
```

```
ttpPlan.createMessage("Counter2A", 0);
```

```
TTA.Message.define ('Counter2A', init_value = 0)
```

```
ttpPlan.linkMessageMode("DemoMode_clustermode", "Counter2A", 4000, 2);
```

```
TTA.Cluster_Mode_uses_Message.add (TTA.Cluster_Mode.instance ('DemoMode_clustermode'),
   TTA.Message.instance ('Counter2A'))
TTA.Cluster_Mode_uses_Message.link ('DemoMode_clustermode', 'Counter2A').set ( d_period = 4000,
   redundancy_degree = 2)
```

```
ttpPlan.linkSubsystemMessage("Producer2A", "Counter2A");
```

```
TTA.Subsystem_sends_Message.add (TTA.Subsystem.instance ('Producer2A'), TTA.Message.instance
   ('Counter2A'))
```

```
ttpPlan.linkMessageMessageType("Counter2A", "short");
```

```
TTA.Message_uses_Msg_Type.add (TTA.Message.instance ('Counter2A'), TTA.Msg_Type_P.instance
   ('short'))
```

This block creates the subsystem, message and appropriate links for the Producer2A module.

```
ttpPlan.createSubsystem("Producer2B", "Reinit_Reintegration");
```

```
TTA.Subsystem.define ('Producer2B', reintegration_type = 'Reinit_Reintegration', raw=1)
```

```
ttpPlan.createMessage("Counter2B", 0);
```

```
TTA.Message.define ('Counter2B', init_value = 0)
```

```
ttpPlan.linkMessageMode("DemoMode_clustermode", "Counter2B", 4000, 2);
```

```
TTA.Cluster_Mode_uses_Message.add (TTA.Cluster_Mode.instance ('DemoMode_clustermode'),
   TTA.Message.instance ('Counter2B'))
TTA.Cluster_Mode_uses_Message.link ('DemoMode_clustermode', 'Counter2B').set ( d_period = 4000,
   redundancy_degree = 2)
```

```
ttpPlan.linkSubsystemMessage("Producer2B", "Counter2B");
```

```
TTA.Subsystem_sends_Message.add (TTA.Subsystem.instance ('Producer2B'), TTA.Message.instance
   ('Counter2B'))
```

```
ttpPlan.linkMessageMessageType("Counter2B", "short");
```

```
TTA.Message_uses_Msg_Type.add (TTA.Message.instance ('Counter2B'), TTA.Msg_Type_P.instance
   ('short'))
```

This block creates the subsystem, message and appropriate links for the Producer2B module.

```
ttpPlan.createCluster(2000, "5000", "big_32_endian");
```

```
TTA.Cluster.define ('DemoCluster', byte_order = 'big_32_endian', tr_period = '2000',
   transmission_speed = '5000', raw=1)
```

This command creates the cluster object with transmission speed according to the property file and sets the length of the TDMA round (tr_period) according to the shortest message period.

```
ttpPlan.createHostAndSlot("Node1", 1, "TTTech_C2");
```

```
TTA.Host.define ('Node1', mux_round = '1', mux_period = '1', serial_number = '1',
   controller_type = 'TTTech_C2', raw=1)
TTA.Slot.define ('Node1_slot')
TTA.Host_uses_Slot.add (TTA.Host.instance ('Node1'), TTA.Slot.instance ('Node1_slot'))
```

```
ttpPlan.linkHostClusterMode("Node1", "DemoMode_clustermode");
```

```
TTA.Host_in_Cluster_Mode.add (TTA.Host.instance ('Node1'), TTA.Cluster_Mode.instance
   ('DemoMode_clustermode'))
TTA.Host_in_Cluster_Mode.add (TTA.Host.instance ('Node1'), TTA.Cluster_Mode.instance
   ('Startup_Mode'))
TTA.Host_in_Cluster_Mode.link ('Node1', 'Startup_Mode').set (may_request_mode_changes =
   'DemoMode_clustermode', raw=1)
```

```
ttpPlan.linkHostCluster("Node1");
```

```
TTA.Host_in_Cluster.add (TTA.Host.instance ('Node1'), TTA.Cluster.instance ('DemoCluster'))
```

```
ttpPlan.linkHostSubsystemClusterMode("Node1", "Producer1",
"DemoMode_clustermode");
```

```
TTA.Host_runs_Subsystem_in_Cluster_Mode.add (TTA.Host.instance ('Node1'), TTA.Subsystem.instance
   ('Producer1'), TTA.Cluster_Mode.instance ('DemoMode_clustermode'))
```

```
ttpPlan.linkHostSubsystemClusterMode("Node1", "Producer2A",
"DemoMode_clustermode");
```

```
TTA.Host_runs_Subsystem_in_Cluster_Mode.add (TTA.Host.instance ('Node1'), TTA.Subsystem.instance
   ('Producer2A'), TTA.Cluster_Mode.instance ('DemoMode_clustermode'))
```

This block of code creates a host object for Node1 together with a transmission slot for the node. Links have to be created between the host and the slot, the cluster, the cluster mode and the subsystems the host runs.

```
ttpPlan.createHostAndSlot("Node2", 2, "TTTech_C2");
```

```
TTA.Host.define ('Node2', mux_round = '1', mux_period = '1', serial_number = '2',
   controller_type = 'TTTech_C2', raw=1)
TTA.Slot.define ('Node2_slot')
TTA.Host_uses_Slot.add (TTA.Host.instance ('Node2'), TTA.Slot.instance ('Node2_slot'))
```

```
ttpPlan.linkHostClusterMode("Node2", "DemoMode_clustermode");
```

```
TTA.Host_in_Cluster_Mode.add (TTA.Host.instance ('Node2'), TTA.Cluster_Mode.instance
   ('DemoMode_clustermode'))
TTA.Host_in_Cluster_Mode.add (TTA.Host.instance ('Node2'), TTA.Cluster_Mode.instance
   ('Startup_Mode'))
TTA.Host_in_Cluster_Mode.link ('Node2', 'Startup_Mode').set (may_request_mode_changes =
   'DemoMode_clustermode', raw=1)
```

```
ttpPlan.linkHostCluster("Node2");
```

```
TTA.Host_in_Cluster.add (TTA.Host.instance ('Node2'), TTA.Cluster.instance ('DemoCluster'))
```

```
ttpPlan.linkHostSubsystemClusterMode("Node2", "Producer1",
"DemoMode_clustermode");
```

```
TTA.Host_runs_Subsystem_in_Cluster_Mode.add (TTA.Host.instance ('Node2'), TTA.Subsystem.instance
   ('Producer1'), TTA.Cluster_Mode.instance ('DemoMode_clustermode'))
```

```
ttpPlan.linkHostSubsystemClusterMode("Node2", "Producer2B",
"DemoMode_clustermode");
```

```
TTA.Host_runs_Subsystem_in_Cluster_Mode.add (TTA.Host.instance ('Node2'), TTA.Subsystem.instance
   ('Producer2B'), TTA.Cluster_Mode.instance ('DemoMode_clustermode'))
```

This block creates all necessary objects and links for Node2.

```
ttpPlan.createHostAndSlot("Node3", 3, "TTTech_C2");
```

```
TTA.Host.define ('Node3', mux_round = '1', mux_period = '1', serial_number = '3',
   controller_type = 'TTTech_C2', raw=1)
TTA.Slot.define ('Node3_slot')
TTA.Host_uses_Slot.add (TTA.Host.instance ('Node3'), TTA.Slot.instance ('Node3_slot'))
```

```
ttpPlan.linkHostClusterMode("Node3", "DemoMode_clustermode");
```

```
TTA.Host_in_Cluster_Mode.add (TTA.Host.instance ('Node3'), TTA.Cluster_Mode.instance
   ('DemoMode_clustermode'))
TTA.Host_in_Cluster_Mode.add (TTA.Host.instance ('Node3'), TTA.Cluster_Mode.instance
   ('Startup_Mode'))
TTA.Host_in_Cluster_Mode.link ('Node3', 'Startup_Mode').set (may_request_mode_changes =
   'DemoMode_clustermode', raw=1)
```

```
ttpPlan.linkHostCluster("Node3");
```

```
TTA.Host_in_Cluster.add (TTA.Host.instance ('Node3'), TTA.Cluster.instance ('DemoCluster'))
```

```
ttpPlan.linkHostSubsystemClusterMode("Node3", "ConsumerA",
"DemoMode_clustermode");
```

```
TTA.Host_runs_Subsystem_in_Cluster_Mode.add (TTA.Host.instance ('Node3'), TTA.Subsystem.instance
   ('ConsumerA'), TTA.Cluster_Mode.instance ('DemoMode_clustermode'))
```

This block creates all necessary objects and links for Node3. Note that here only one subsystem is linked according to the mapping in the property file.

```
ttpPlan.createHostAndSlot("Node4", 4, "TTTech_C2");
```

```
TTA.Host.define ('Node4', mux_round = '1', mux_period = '1', serial_number = '4',
   controller_type = 'TTTech_C2', raw=1)
```

```
TTA.Slot.define ('Node4_slot')
TTA.Host_uses_Slot.add (TTA.Host.instance ('Node4'), TTA.Slot.instance ('Node4_slot'))
```

```
ttpPlan.linkHostClusterMode("Node4", "DemoMode_clustermode");
```

```
TTA.Host_in_Cluster_Mode.add (TTA.Host.instance ('Node4'), TTA.Cluster_Mode.instance
   ('DemoMode_clustermode'))
TTA.Host_in_Cluster_Mode.add (TTA.Host.instance ('Node4'), TTA.Cluster_Mode.instance
   ('Startup_Mode'))
TTA.Host_in_Cluster_Mode.link ('Node4', 'Startup_Mode').set (may_request_mode_changes =
   'DemoMode_clustermode', raw=1)
```

```
ttpPlan.linkHostCluster("Node4");
```

```
TTA.Host_in_Cluster.add (TTA.Host.instance ('Node4'), TTA.Cluster.instance ('DemoCluster'))
```

```
ttpPlan.linkHostSubsystemClusterMode("Node4", "ConsumerB",
"DemoMode_clustermode");
```

```
TTA.Host_runs_Subsystem_in_Cluster_Mode.add (TTA.Host.instance ('Node4'), TTA.Subsystem.instance
   ('ConsumerB'), TTA.Cluster_Mode.instance ('DemoMode_clustermode'))
```

This block creates all necessary objects and links for Node4.

```
ttpPlan.getScript()
```

```
TTA.Application_Command.run('Schedule.Make new schedule')
TTA.Application_Command.run('Schedule.Make MEDLs')
TTA.Application_Command.run('File.Save cluster as ...', 'C:\Demo\.\TTP\DemoCluster.cdb')
```

The last block of code initiates the creation of the cluster schedule and the MEDLs. Finally the cluster database is written to file.

TTPplan Results

The complete generated object model is not easy to present here. It consists of the objects, links and attributes created by the script presented above and all default attributes of the objects and links. For this reason only key features of the object model will be presented here, most importantly the generated cluster schedule.

Figure 35 illustrates the instances of the link that associates subsystems to hosts in a

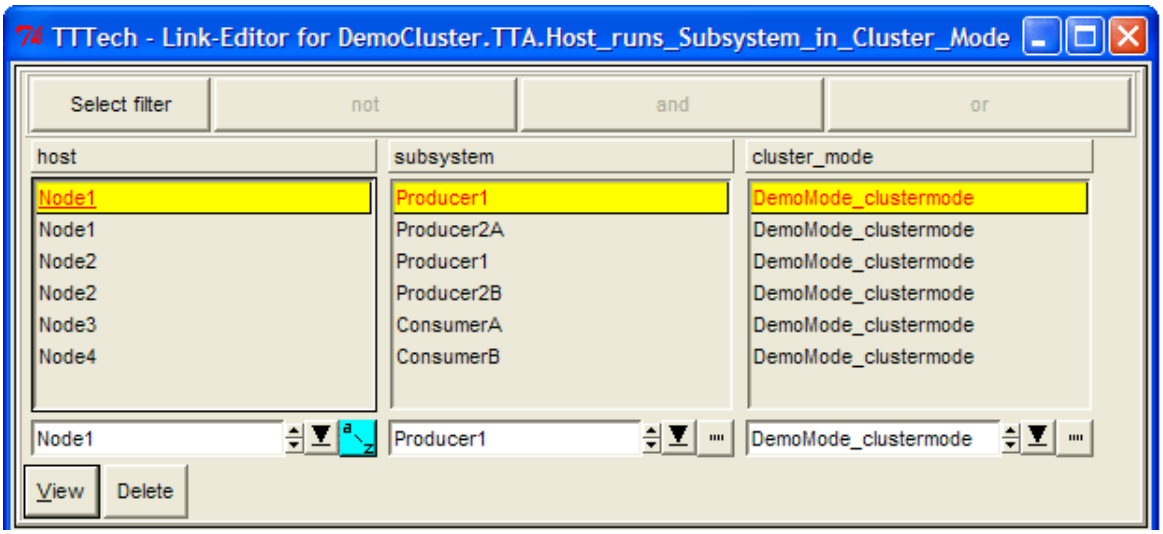

Figure 35 Mapping of Subsystem to Hosts in TTPplan

cluster mode. Hosts in context of the TTP tools are nodes of the distributed system. This is a good example to see that the specification in the property file, in this case the mapping of modules to nodes, is realized in the TTPplan object model.

Figure 36 is a screenshot of the schedule editor of TTPplan showing the generated schedule for the demo application. It shows the slots of all hosts, each slot having a length of 500 micro seconds. The complete cluster cycle consist of two rounds, which

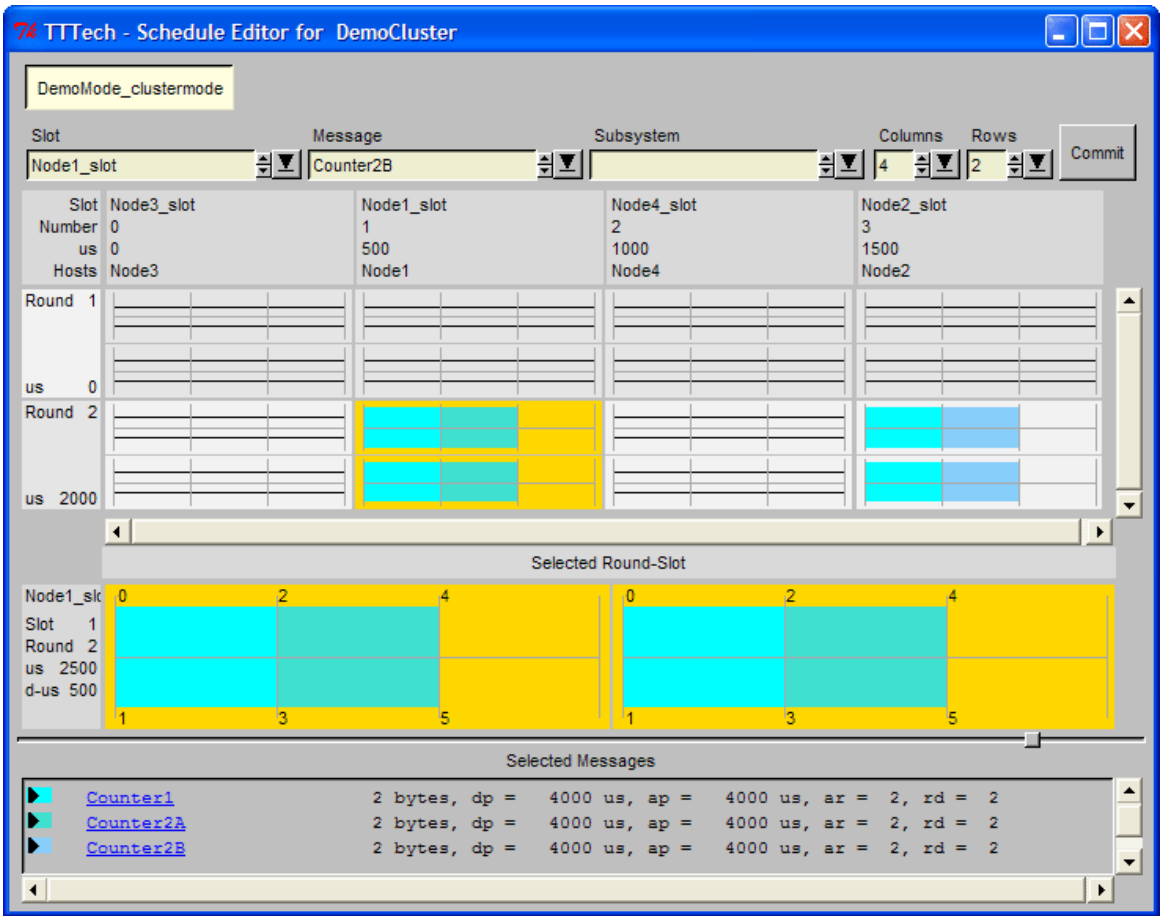

Figure 36 Demo Application Cluster Schedule

results in eight slots per cluster cycle and exactly fills up the 4000 micro seconds mode period of the TDL mode. The messages `Counter1`, `Counter2A` and `Counter2B` are indicated by the various blue colored blocks. Note that the message `Counter1` is sent by `Node1` and `Node2` because it is produced by a replicated module that is executed on both nodes. The upper part of the blocks symbolize channel A of the TTP bus, whereas the lower part symbolizes channel B. All blocks span over both parts, as the messages were all specified to be transferred on both channels of the bus.

4.3.3 TTPbuild Script

The plugin creates a script for each node that is intended for the node design tool TTPbuild. Unlike TTPplan, here one script for every node is required. As a first example we will take a look at the complete script the plugin generates for `Node1` of the demo application.

```
TTPbuildScript ttpBuild = new TTPbuildScript("Node1",
"C:\Demo\.\TTP\DemoCluster.cdb", "Node1.ndb ");
      TTA_Application_Command.run('File.New node ...', 'Node1', 'C:\Demo\.\TTP\DemoCluster.cdb')
```

The first command creates a new node database file. It is needed to specify the cluster database that was created by TTPplan before here and to select which node of the cluster should be designed.

```
ttpBuild.setHardware("TTPpowernode_C2");
```

```
TTA.Host.customize ('Node1', node_config = 'TTPpowernode_C2', raw=1)
```

```
ttpBuild.createApplicationMode("DemoMode", "DemoMode_clustermode");
```

```
TTA.Node.App_Mode.define ('DemoMode', maximum_interrupt_latency = '150 us', pos_synch_limit =
   'max ( TTA.Cluster.clock_sync.macro_tick_length / 1000 * 2, TTA.Cluster.tc_period * 0.0015)',
   neg_synch_limit = 'max ( TTA.Cluster.clock_sync.macro_tick_length / 1000 * 2,
   TTA.Cluster.tc_period * 0.0015)', neg_correction_limit = 'max (
   TTA.Cluster.clock_sync.macro_tick_length / 1000 * 3, TTA.Cluster.tc_period * 0.002)',
   pos_correction_limit = 'max ( TTA.Cluster.clock_sync.macro_tick_length / 1000 * 3,
   TTA.Cluster.tc_period * 0.002)', raw=1)
TTA.Node.App_Mode_maps_to_Cluster_Mode.add (TTA.Node.App_Mode.instance ('DemoMode'),
   TTA.Cluster_Mode.instance ('DemoMode_clustermode'))
```

This block of code sets the hardware configuration, creates an application mode and links it to the cluster mode that is already present in the object model.

```
ttpBuild.createSubsystem("emachine");
```

```
TTA.Subsystem.define ('emachine')
```

```
ttpBuild.createTask("emachine1", 75, "emachine", "DemoMode", 4000, 0,
80);
```

```
TTA.Node.App_Task.define ('emachine1', time_source = 'local_time', time_budget = '75', period =
   '4000', deadline = '80', phase = '0', raw=1)
TTA.Node.Task_in_App_Mode.add (TTA.Node.App_Task.instance ('emachine1'),
   TTA.Node.App_Mode.instance ('DemoMode'))
TTA.Node.Subsystem_runs_Task.add (TTA.Subsystem.instance ('emachine'),
   TTA.Node.App_Task.instance ('emachine1'))
```

This block deals with the creation of the E machine-like task that needs to have a subsystem and must be linked to the application mode.

```
ttpBuild.createTask("Produce1", 100, "Producer1", "DemoMode");
```

```
TTA.Node.App_Task.define ('Produce1', time_source = 'local_time', time_budget = '100', raw=1)
TTA.Node.Task_in_App_Mode.add (TTA.Node.App_Task.instance ('Produce1'),
   TTA.Node.App_Mode.instance ('DemoMode'))
TTA.Node.Subsystem_runs_Task.add (TTA.Subsystem.instance ('Producer1'),
   TTA.Node.App_Task.instance ('Produce1'))
```

```
ttpBuild.linkMessage("Counter1", "Produce1", true, false);
```

```
TTA.Node.Task_uses_Message.add (TTA.Node.App_Task.instance ('Produce1'), TTA.Message.instance
   ('Counter1'), access_type = 'agreed', raw=1)
TTA.Node.Task_uses_Message.link ('Produce1', 'Counter1').set (sends = 'yes', receives = 'no',
   raw=1)
```

```
ttpBuild.linkMessage("Counter1", "emachine1", false, true);
```

```
TTA.Node.Task_uses_Message.add (TTA.Node.App_Task.instance ('emachine1'), TTA.Message.instance
   ('Counter1'), access_type = 'agreed', raw=1)
TTA.Node.Task_uses_Message.link ('emachine1', 'Counter1').set (sends = 'no', receives = 'yes',
   raw=1)
```

```
ttpBuild.createLocalMessageFromTo("Counter1_in", "short", 4000,
"emachine1", "Produce1");
```

```
TTA.Message.define ('Counter1_in', d_period = 4000, init_value = 0)
TTA.Message_uses_Msg_Type.add (TTA.Message.instance ('Counter1_in'), TTA.Msg_Type_P.instance
   ('short'))
TTA.Node.Task_uses_Message.add (TTA.Node.App_Task.instance ('emachine1'), TTA.Message.instance
   ('Counter1_in'), access_type = 'agreed', raw=1)
TTA.Node.Task_uses_Message.link ('emachine1', 'Counter1_in').set (sends = 'yes', receives =
   'no', raw=1)
TTA.Node.Task_uses_Message.add (TTA.Node.App_Task.instance ('Produce1'), TTA.Message.instance
   ('Counter1_in'), access_type = 'agreed', raw=1)
TTA.Node.Task_uses_Message.link ('Produce1', 'Counter1_in').set (sends = 'no', receives = 'yes',
   raw=1)
```

This block of code handles the creation of the task Produce1. In order to maintain the FLET property of TDL, as discussed in 3.5, it is required that received messages from the bus pass the E machine-like task. This is realized by creating a local message Counter1_in from the E machine-like task and the task. The E machine-like task also has to receive the sent message of the task Produce1 because the actuator update code inside the E machine-like task needs access to the output port of the task.

```
ttpBuild.createTask("Produce2A", 100, "Producer2A", "DemoMode");
```

```
TTA.Node.App_Task.define ('Produce2A', time_source = 'local_time', time_budget = '100', raw=1)
TTA.Node.Task_in_App_Mode.add (TTA.Node.App_Task.instance ('Produce2A'),
   TTA.Node.App_Mode.instance ('DemoMode'))
TTA.Node.Subsystem_runs_Task.add (TTA.Subsystem.instance ('Producer2A'),
   TTA.Node.App_Task.instance ('Produce2A'))
```

```
ttpBuild.linkMessage("Counter2A", "Produce2A", true, false);
```

```
TTA.Node.Task_uses_Message.add (TTA.Node.App_Task.instance ('Produce2A'), TTA.Message.instance
   ('Counter2A'), access_type = 'agreed', raw=1)
TTA.Node.Task_uses_Message.link ('Produce2A', 'Counter2A').set (sends = 'yes', receives = 'no',
   raw=1)
```

```
ttpBuild.linkMessage("Counter2A", "emachine1", false, true);
```

```
TTA.Node.Task_uses_Message.add (TTA.Node.App_Task.instance ('emachine1'), TTA.Message.instance
   ('Counter2A'), access_type = 'agreed', raw=1)
TTA.Node.Task_uses_Message.link ('emachine1', 'Counter2A').set (sends = 'no', receives = 'yes',
   raw=1)
```

```
ttpBuild.createLocalMessageFromTo("Counter2A_in", "short", 4000,
"emachine1", "Produce2A");
```

```
TTA.Message.define ('Counter2A_in', d_period = 4000, init_value = 0)
TTA.Message_uses_Msg_Type.add (TTA.Message.instance ('Counter2A_in'), TTA.Msg_Type_P.instance
   ('short'))
TTA.Node.Task_uses_Message.add (TTA.Node.App_Task.instance ('emachine1'), TTA.Message.instance
   ('Counter2A_in'), access_type = 'agreed', raw=1)
TTA.Node.Task_uses_Message.link ('emachine1', 'Counter2A_in').set (sends = 'yes', receives =
   'no', raw=1)
TTA.Node.Task_uses_Message.add (TTA.Node.App_Task.instance ('Produce2A'), TTA.Message.instance
   ('Counter2A_in'), access_type = 'agreed', raw=1)
TTA.Node.Task_uses_Message.link ('Produce2A', 'Counter2A_in').set (sends = 'no', receives =
   'yes', raw=1)
```

This block of realizes the task Produce2A of the module Producer2A in an identical way as for the previous task.

```
ttpBuild.getScript();
```

```
TTA.Application_Command.run('Schedule.Make new schedule')
TTA.Application_Command.run('Schedule.Generate code')
TTA.Application_Command.run('File.Save node as ...', 'Node1.ndb')
```

The last commands triggers the generation of the schedule for Node1 and the generation of the code that contains the schedule and the FT-Com layer. Finally the node database is saved to disk.

The second script we will take a look at is that for Node3, which only executes the module ConsumerA. The beginning and end of the script is identical to that presented above for Node1. The only difference is the missing task code since the module does not contain any tasks. It is sufficient to create an E machine-like task and link the two messages Counter1 and Counter2A to it to perform actuator updates:

```
TTPbuildScript ttpBuild = new TTPbuildScript("Node3",
"C:\Demo\.\TTP\DemoCluster.cdb", "Node3.ndb ");
```

```
TTA_Application_Command.run('File.New node ...', 'Node3', 'C:\Demo\.\TTP\DemoCluster.cdb')
```

```
ttpBuild.setHardware("TTPpowernode_C2");
```

```
TTA.Host.customize ('Node3', node_config = 'TTPpowernode_C2', raw=1)
```

```
ttpBuild.createApplicationMode("DemoMode", "DemoMode_clustermode");
```

```
TTA.Node.App_Mode.define ('DemoMode', maximum_interrupt_latency = '150 us', pos_synch_limit =
   'max ( TTA.Cluster.clock_sync.macro_tick_length / 1000 * 2, TTA.Cluster.tc_period * 0.0015)',
   neg_synch_limit = 'max ( TTA.Cluster.clock_sync.macro_tick_length / 1000 * 2,
   TTA.Cluster.tc_period * 0.0015)', neg_correction_limit = 'max (
   TTA.Cluster.clock_sync.macro_tick_length / 1000 * 3, TTA.Cluster.tc_period * 0.002)',
   pos_correction_limit = 'max ( TTA.Cluster.clock_sync.macro_tick_length / 1000 * 3,
   TTA.Cluster.tc_period * 0.002)', raw=1)
TTA.Node.App_Mode_maps_to_Cluster_Mode.add (TTA.Node.App_Mode.instance ('DemoMode'),
   TTA.Cluster_Mode.instance ('DemoMode_clustermode'))
```

```
ttpBuild.createSubsystem("emachine");
```

```
TTA.Subsystem.define ('emachine')
```

```
ttpBuild.createTask("emachine1", 75, "emachine", "DemoMode", 4000, 0,
80);
```

```
TTA.Node.App_Task.define ('emachine1', time_source = 'local_time', time_budget = '75', period =
   '4000', deadline = '80', phase = '0', raw=1)
TTA.Node.Task_in_App_Mode.add (TTA.Node.App_Task.instance ('emachine1'),
   TTA.Node.App_Mode.instance ('DemoMode'))
TTA.Node.Subsystem_runs_Task.add (TTA.Subsystem.instance ('emachine'),
   TTA.Node.App_Task.instance ('emachine1'))
```

```
ttpBuild.linkMessage("Counter1", "emachine1", false, true);
```

```
TTA.Node.Task_uses_Message.add (TTA.Node.App_Task.instance ('emachine1'), TTA.Message.instance
   ('Counter1'), access_type = 'agreed', raw=1)
TTA.Node.Task_uses_Message.link ('emachine1', 'Counter1').set (sends = 'no', receives = 'yes',
   raw=1)
```

```
ttpBuild.linkMessage("Counter2A", "emachine1", false, true);
```

```
TTA.Node.Task_uses_Message.add (TTA.Node.App_Task.instance ('emachine1'), TTA.Message.instance
   ('Counter2A'), access_type = 'agreed', raw=1)
TTA.Node.Task_uses_Message.link ('emachine1', 'Counter2A').set (sends = 'no', receives = 'yes',
   raw=1)
```

```
ttpBuild.getScript();
```

```
TTA.Application_Command.run('Schedule.Make new schedule')
TTA.Application_Command.run('Schedule.Generate code')
TTA.Application_Command.run('File.Save node as ...', 'Node3.ndb')
```

TTPbuild Results

Again it is not easily possible to present the complete object model of a node here. Basically the script is already a sufficient representation of the generated object model. However what is missing is the schedule that TTPbuild generates out of the model.

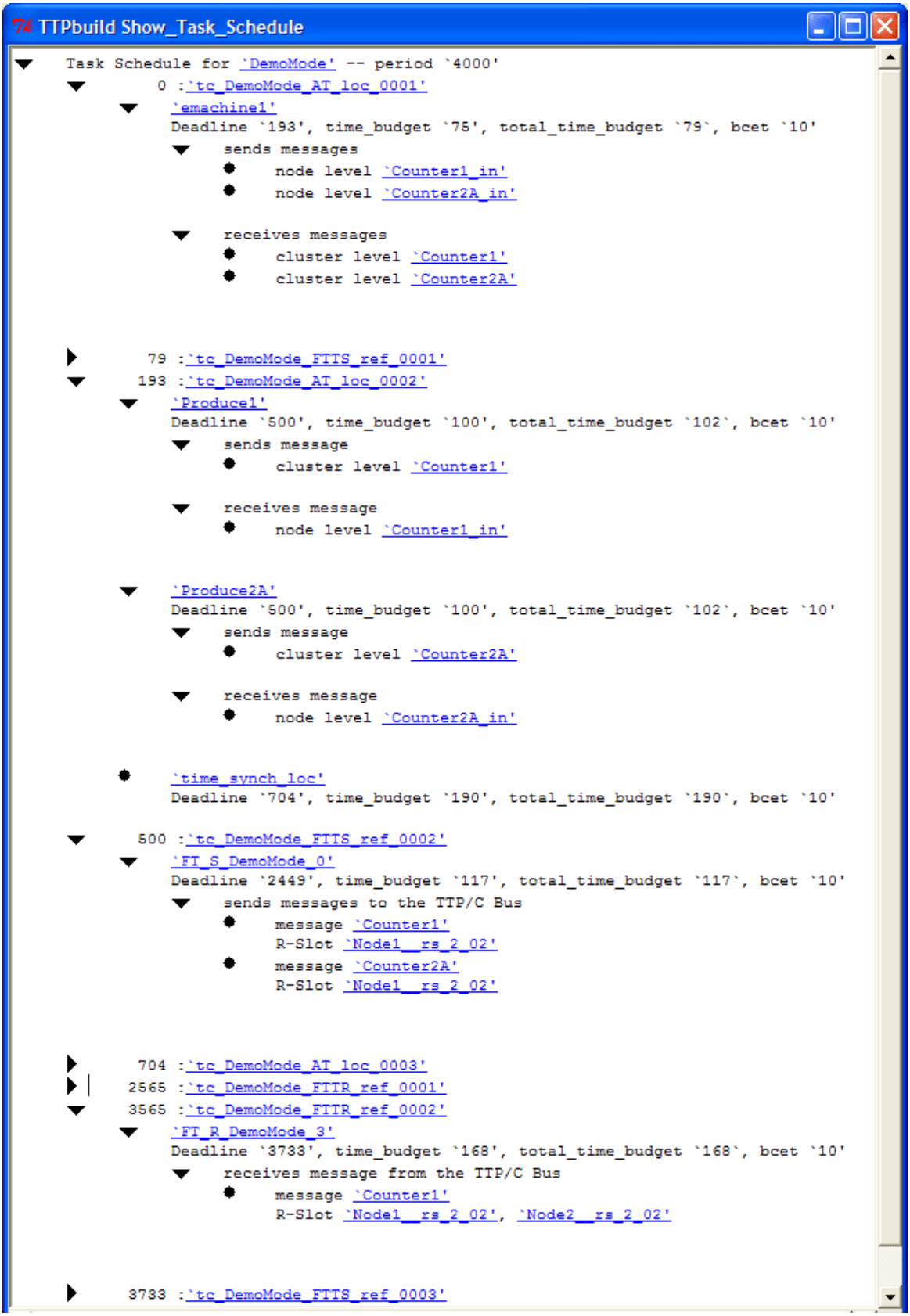

Figure 37 Task Schedule of Node1 of the Demo Application

Figure 37 is a screenshot of TTPbuild illustration the node schedule of Node1 of the demo application. How to read to node schedule view is explained in 2.4.2. All relevant tasks including the FT-Com layer tasks have been expanded to see all messages that are received and sent by them. It can be seen that the E machine-like task actually is executed at time instance zero and that the FLET property of TDL is realized by the plugin.

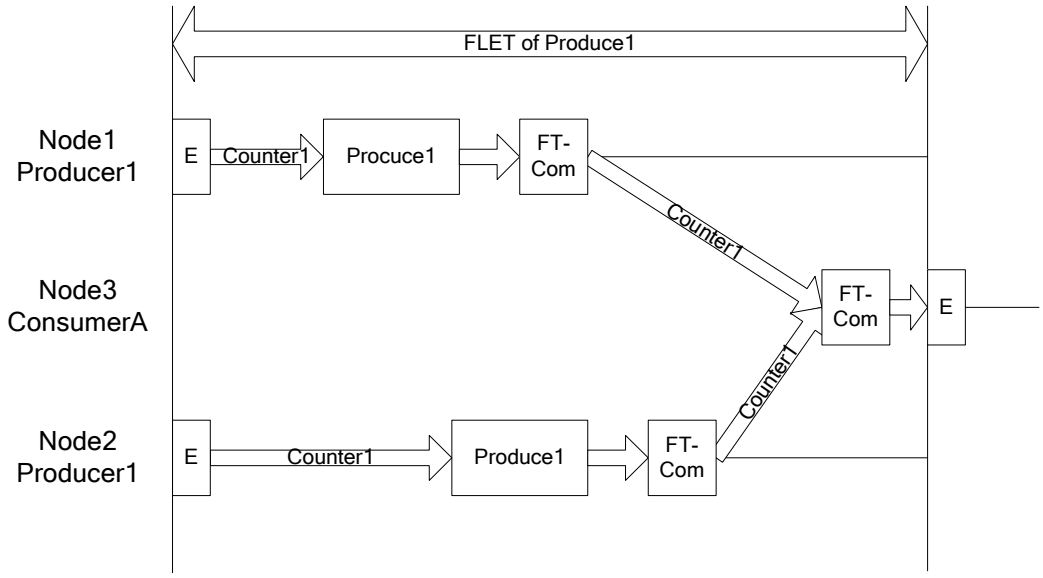

Figure 38 Demo Application Task Invocation Diagram for Counter1

Figure 38 illustrates one period of the processing of the message Counter1. It starts with the E machine-like tasks on Node1 and Node2 that receive the message containing Counter1 from the bus and pass it on to the task that uses it, which is Produce1 on both nodes. Note that the task invocation time is different on both nodes. This is drawn this way to show that due to FLET the actual instance of task invocation does not matter. It does not represent that actual schedule for the demo application. On both nodes the task Produce1 increments the counter value by one. Then the FT-Com layer task takes care of the transmission of the messages via the TTP bus. Node3 receives both messages with its own FT-Com task that has the additional function of applying the specified RDA algorithm to combine both messages containing a value of Counter1 into one consistent value. This value is passed on to the E machine-like task on Node3 where finally the actuator is updated.

4.3.4 Generated Glue Code

In the following the generated glue code for the demo application is presented. First we will take a look at the glue code generated for Node1 where the modules Producer1 and Producer2A are executed:

```
#include "TTPos.h"
#include "hal.h"
#include "ttpc_msg.h"
#include "Producer1.h"
```

```
#include "Producer2A.h"

tt_task (Produce1)
{
  produce1Impl(&Counter1_in);
  tt_Raw_Value (Counter1) = Counter1_in;
}

tt_task (Produce2A)
{
  produce2AImpl(&Counter2A_in);
  tt_Raw_Value (Counter2A) = Counter2A_in;
}

tt_task (emachine1)
{
  setYellowLED1(Counter1);
  setYellowLED2(Counter2A);
  tt_Raw_Value (Counter1_in) = Counter1;
  tt_Raw_Value (Counter2A_in) = Counter2A;
}
```

In the two blocks for the tasks Produce1 and Produce2A the functionality code of the task is called and the produced value is passed on to the FT-Com layer for transmission. The E machine-like task calls the two functions of the functionality code for actuator updates and forwards two messages in order to maintain the FLET property.

The glue code for Node3 looks quite different, as there are no tasks except the E machine-like task:

```
#include "TTPos.h"
#include "hal.h"
#include "ttpc_msg.h"
#include "ConsumerA.h"

tt_task (emachine1)
{
  short int Counter1MessageStatus = tt_Message_Status (Counter1);
  setYellowLED1(Counter1);
  setYellowLED2(Counter2A);
  setRedLEDs(Counter1MessageStatus);
}
```

The E machine-like task retrieves the message status of Counter1 and stores it in a local variable. Then the appropriate actuators are updated according to the TDL code.

Chapter 5
Evaluation

5.1 Summary

In this section the results and findings of the work are summarized. The main purpose of the thesis was the integration of TDL with the TTP tools. In the following various aspects of it are discussed.

Distribution

The current version of the TDL language and compiler has basic support for distribution. This support consists of constructs to link modules with the public and import statements. The platform specific specification of how modules should be distributed among multiple nodes must be provided in addition to the TDL program. The tool chain presented in this thesis performs all the necessary steps to generate code for the distributed TTP platform out of TDL modules plus an additional property file. It uses TTP protocol services such as time synchronization by means of the corresponding TTP tools and introduces the necessary properties to specify the distribution of modules among nodes. Some of those integration mechanisms may serve as generic solutions for the integration of other distributed platforms with TDL.

Fault Tolerance

The TDL language does not provide ways to specify or handle fault tolerance. The challenge was to identify how it is possible to realize features like replication, redundancy and error detection for TDL modules. The module was chosen as unit of replication and the feasibility of this choice was demonstrated. Ways to specify fault tolerance properties for TDL modules have been developed and a solution for providing information about the status of replicated modules by means of dedicated sensors has been found. This may as well serve as an example for the integration of fault tolerance features on other platforms and as contribution to the development of a generic way to specify fault tolerance properties for TDL programs, including the introduction of a generic error detection interface.

Reusability

A glue code generator was implemented to provide easier integration of C code with the operating system and its services that are provided for the TTP platform. The goal was to have no platform-specific code in the functionality code and to be able to use the generic language bindings for C that are specified for TDL. This proofs that the approach of TDL regarding platform-independence is feasible with the TTP platform.

TDL Plugin Interface

The development of the plugin and the corresponding tool chain demonstrated the flexibility of the plugin interface of the TDL compiler. The suitability for a plugin that

realizes distribution was shown but the experience can as well be used for future improvements to the interface. One specific suggestion for an improvement would be to enable plugins to perform actions after the whole compilation process of multiple modules is finished. This would avoid having the `lastModule` property for the plugin proposed in this thesis and probably would be useful for plugins for other platforms as well.

E Machine

In the presented tool chain the E machine implementation differs from former Giotto case studies. Due to the static nature of the TTP protocol and the development tools it made sense to make a static approach concerning the E machine as well. The E code produced by the compiler was not used to feed an interpretative E machine. Instead the compiler plugin processes each module and generates scripts for the TTP tools and glue code that together realize the timing and functionality that is given by the TDL code. This solution might again serve as a prototype for applications or platforms with a similar static nature or where safety and reliability or easier verifiability is preferred over flexibility.

Bus Schedule

The thesis identified typical problems of distributed scheduling, especially in combination with the FLET assumption of TDL. It was demonstrated that maintaining the FLET property in distributed environments is a complicated task because a lot of constraints have to be obeyed. Although no optimal generic solution was found these constraints were identified and a way was found to meet those constraints for simple applications. There is still room for optimizations here in order to use the bus and CPU time on each node more efficiently.

TTP Tools

Developing the TDL plugin tool chain also showed the capabilities of TTTech's TTP tools. It was possible to realize TDL requirements like the FLET property with the help of the powerful programming interface of those tools. On the other hand also some weaknesses like the rather inefficient realization of the E machine-like task with TTPbuild were identified that can be seen as a suggestion to further improve the tools or to go for a proprietary implementation of the required functionality.

Tool Chain Comparison

The comparison between the TTP tool chain and the newly developed TDL-TTP tool chain for the TTP platform demonstrates the differences and advantages of TDL over traditional design approaches. The clearest advantage of TDL is its platform independence. Existing modules and functionality code can be reused with no or minor changes on different platforms for which a corresponding compiler plugin exists. In contrast, applications developed for the TTP hardware do not have this flexibility and a radical redesign is needed when changing the underlying platform.

5.2 Restrictions

The restrictions of the current plugin implementation and of the current version of the TTP tools are listed in this section.

Not all language constructs of TDL are supported by the plugin for the TTP platform. This is partly due to specific reasons as for example for public sensors as explained in 3.5, and partly because some constructs were not regarded as critical for proofing that the integration of TDL with the TTP tools is possible. An example for the latter would be state ports.

TDL supports functions as initialization values for output ports and actuators which are called at runtime. As the plugin needs to know the actual init values for all ports that need to be transferred via the TTP bus, in order to set the corresponding values in TTPplan and TTPbuild, only constants are supported as initialization values.

Another significant restriction is that only one mode per TDL module is supported. This is due to the limitation of the TTP tools to one application mode and the lack of any possibility for a workaround to overcome this limitation.

A notable restriction that is due to a specific limitation in the plugin interface of the TDL compiler is the need to have unique port names throughout the whole application. The limitation prevented the plugin from using fully qualified names with module and task name for the message names in the TTP tools and so identical port names of public tasks in different modules are not allowed.

Apart from these clear limitations, not every TDL program that obeys them is necessarily executable on the TTP platform. This might be either because the TDL plugin is not smart enough to generate suitable cluster and node scripts or because the TTP tools don’t find a feasible schedule. The latter can be due to the rather inefficient implementation of the E machine-like tasks, especially when there are a lot of different FLET periods as discussed in 3.5.

It must be said that the developed plugin is meant as a proof of concept and not as a fully-fledged tool that is ready for production usage. It has not been tested thoroughly enough and there is no guarantee that is works with all kinds of TDL modules that obey the restrictions mentioned above. TDL itself is still under development and its plugin interface might change as well, which would require adopting the TTP platform plugin accordingly.

5.3 "TDL vs. TTP Tools"

This section tries to identify the differences of developing a distributed real-time application from design to implementation with the TTP tools and with the TTP TDL plugin and points out the advantages and disadvantages of both choices.

A basic difference between the TTTech TTP tool chain and TDL is the two-level design approach used by TTTech. The two levels are realized by using TTPplan for cluster design and TTP build for node design. The idea is that a system integrator, who employs a number of subsystem manufacturers, first designs the cluster schedule and assigns bandwidth for the specification of messages used by the various

subsystems. Also fault-tolerance properties have to be specified. This design approach requires at lot of information regarding the system at this point of development, but once the first level has been finished, the subsystem developers are independent from each other and it is possible to test and verify subsystems without the need to have the whole system available. This is possible as the final cluster schedule is already known at this point. In TDL there exists no comparable mechanism and so when using TDL as front-end for the TTP tools as described in this thesis, the two levels are transparent and hidden from the user. So a fundamental difference is that in the TTP tools the cluster schedule can be seen as part of the system specification, whereas in TDL it is derived from the communication requirements of the modules that form the system.

Additional differences that TTTech points out regard information hiding and responsibility concerns. When using TDL, the subsystem developers would need to provide TDL modules to the system integrator who eventually compiles them together. Only then can be determined if they will actually run on a specific hardware platform. TTTech argues that this would no be acceptable for their customers, as this way on one hand a higher risk of integration is introduced and on the other hand it is not possible to hide information of the actual implementation from another. An additional argument is that in case of errors in the system it is harder to identify the module developer responsible for the non-working system.

In the following the development process of the TTP tools and TDL are compared using three categories: Determinism, compositionality and software standardization. A more detailed analysis including a comparison to other commercial tools for the development of distributed real-time systems can be found in [13].

Determinism

If a software component is called twice with the same input values at the same time instances, it both times has to produce the same output values at the same time instances. An example would be if a control component receives the same input, for example sensor values, it always has to react exactly in the same way. The consequences of determinism are minimal jitter and good testability, as each behavior can be reproduced.

In TDL value and time determinism and close to zero jittering are guaranteed by the FLET assumption. The sensors are read and the output values for actuators are available at specific points in time (based on FLET), that are not influenced by the OS scheduler or the moments in time when the tasks are actually executed. The behavior of a TDL program is solely determined by its physical environment and not by CPU performance, bus load or scheduling strategy.

When using the TTTech TTP tools, value and time determinism is also ensured to some degree. The transmission of messages and the invocation of tasks are done with minimal jitter as both are static at runtime and the clock synchronization of the TTP protocol is accurate. However when sensor readings or actuator updates are implemented inside tasks, the exact invocation time depends on how long the tasks actually run in each period. This problem also occurs if the calculation performed by a task is modified and its execution time changes. In order to ensure determinism, it would require executing sensor readings and actuator updates in a separate task, similar to the E machine-like task used by the TDL plugin for the TTP platform described in this thesis.

Compositionality

Compositionality is given if the behavior of a software component is independent of the overall system load and configuration. This means that for example a new component can be added to a system without influencing the behavior of the original components. The consequences of compositionality are the extensibility of systems and the reuse of components.

With TDL each component can be developed independently and compositionality is guaranteed because of the precise timing model based on FLET. Since the parallel composition of timed programs does not change the timing behavior of the individual program, compositionality is guaranteed in TDL.

The TTP tool chain by TTTech also aims at compositionality, but it has certain restrictions on it. It is required that components that require communication on the TTP bus or fault tolerance mechanisms are already known when designing the first level of the two-level design approach, as their communication and fault tolerance needs to be specified at this point of development. This limits the extensibility of the system but it does ensure that the components are independent of overall system load regarding the communication system.

Software standardization

Software standardization is given when the behavior of a software component is specified independently of its implementation. For example, the hardware, the operating system, or the bus architecture can be changed without changing the behavior of the application components. The consequences of software standardization are upgradeability of hardware, portability of software and the possibility to move software components between nodes of the system.

TDL separates the functionality and the timing behavior and platform specifications. It is easy to change the platform without changing the model of the software and its functionality by changing only the platform annotations. This separation will maintain the behavior of the whole system although the underlying hardware platform may be changed or become distributed.

It is clear that the TTP tools cannot provide software standardization to the degree that TDL provides, simply because the fact that they are tailored to the specific requirements of the TTP protocol on cluster-level and the TTTech operation system TTPos and the FT-Com layer on node-level. Only a limited range of hardware targets is supported and the user has no possibility to extend that range.

Bibliography

[1] Hermann Kopetz. Real-Time Systems: Design Principles for Distributed Embedded Applications. Kluwer, 1997.

[2] Thomas A. Henzinger and Christoph M. Kirsch. The Embedded Machine: Predictable, Portable Real-Time Code. Proceedings of the International Conference on Programming Language Design and Implementation (PLDI), ACM Press, 2002, pp. 315-326.

[3] Pankaj Jalote. Fault Tolerance in Distributed Systems. Prentice Hall, 1998.

[4] Thomas A. Henzinger, Benjamin Horowitz and Christoph M Kirsch. Giotto: A Time-triggered Language for Embedded Programming. Proceedings of the First International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2211, Springer-Verlag, 2001, pp. 166-184.

[5] TTTech. TTPos: The Time-Triggered Fault-Tolerant Operating System. Manual Edition 2.9.10 for TTPos Product Version 4.4, 2003.

[6] TTTech. TTPplan: The Cluster Design Tool for the Time-Triggered Protocol TTP. Manual Edition 4.4.0 for Product Version 4.4, 2003.

[7] TTTech. Time-Triggered Protocol TTP/C High-Level Specification Document. Edition 1.0.0, July 2002.

[8] TTTech. TTPbuild: The Node Design Tool for the Time-Triggered Protocol TTP. Manual Edition 3.3.1 for Product Version 4.4, 2003.

[9] Josef Templ. TDL Specification and Report. Technical Report, University of Salzburg, Computer Science Department, 2004.

[10] TTTech. TTPload: The Download Tool for the Time-Triggered Protocol TTP. Manual Edition 5.3.9 for Product Version 5.4, 2003.

[11] TTTech. TTPview: The Monitoring Tool for the Time-Triggered Protocol TTP. Manual Edition 5.0.8 for Product Version 5.10, 2003.

[12] TTTech. TTP-Powernode: A TTP Development Board for the Time-Triggered Architecture based on the TTP-Chip C2. Manual Edition 2.0.08 for Product Version 1.0, 2003.

[13] C. Farcas, M. Holzmann, H.Pletzer, G. Stieglbauer. The TDL Advantage. University of Salzburg, 2004.