# Towards a Model-based Toolchain for High-confidence Design of Embedded Systems

Gábor Karsai, János Sztipanovits, Joseph Porter, Ryan Thibodeaux, Péter Völgyesi
Vanderbilt University
Institute for Software Integrated Systems
Nashville, TN, USA
gabor.karsai@vanderbilt.edu

## Abstract

*While design automation for hardware systems is quite advanced, the software world, and especially the embedded systems software badly needs such automation. The current state-of-the-art is to use some modeling environment for software modeling and an integrated development environment for code development and debugging, but these rarely include the sort of automatic synthesis and verification capabilities available in the VLSI domain. This paper introduces the concepts, the elements, and some early prototypes for an envisioned chain of tools for the development of embedded software that integrates verification steps into the overall process.*

**Figure 1. The TTP-Development Cluster with eight nodes.**

## 1. Platforms

### 1.1. The Time Triggered Architecture

One of the experimental target platforms for the toolchain is the Time Triggered Architecture [3] developed at the Vienna University of Technology and by TTTech Computertechnik AG. In stark contrast to event-triggered (ET) systems, in time-triggered (TT) systems the control signals are derived from the progression of a system-wide global time according to a static schedule defined in the design phase. These control signals might trigger the sending and receiving of messages, the activation of application tasks or mode changes. The primary reasons for selecting TTA in general and the TTP-Development Cluster (see Figure 1) in particular as a test platform were its fault tolerance features, performance characteristics and its well established and thoroughly researched history.

Fault tolerance is supported by redundant communication channels, by replicated subsystems with constant state synchronization with voting in the value domain and by various provisions in the communication protocol (eg: cluster startup, distributed time synchronization, membership management with click avoidance). The measured 1-5 $\mu$s jitter and its relatively high bandwidth (5-25 Mbit/s) with 60-80% efficiency are definitely superior to similar performance metrics measured in general purpose (ET) computer systems. However, one of the most crucial advantages of TTP for the system designer lies in the its composability aspects. A well-designed cluster schedule (with enough spare capacity for future grow) guarantees that existing services are not affected by the removal or the addition of other services both in the value and in the time domains. Furthermore, due to the dedicated and autonomous protocol controller, these changes are transparent to the application level software. Also, for similar reasons, application level errors are not propagated to other nodes and do not disrupt the communication channels.

These characteristics made the TTA platform and the Time Triggered Protocol prime targets for the proposed toolchain for deeper understanding of time triggered computational models. On the other hand, we faced with various constraints and peculiarities inherent in the TTP protocol some of which had profound effects on the system design and thus on the design environment. These lessons confirmed us in our basic concept of a need for an integrated environment, where (late) deployment decisions are able to influence the (early) design phases via feedback in the toolchain. This section gives a brief overview on the protocol to understand some of the special characteristics of TTP better and to show their effect on the system design.

In TTP all protocol operations—except during the *startup* phase—are initiated at *a priori* known points in time. All node that are integrated in the cluster agree on a global time. This requires approximately synchronized clocks with a fault-tolerant synchronization protocol. The global clock is defined in a *sparse time base*, which granularity is limited by the precision of the distributed clock synchronization. The granularity factor ($\Pi$) is an important parameter in the system design and depends on the accuracy of the local oscillators, the frequency of the synchronization points (TTP does not use skew compensation techniques), on the speed of the communication controller and of the physical medium. The communication controller has complete knowledge of the message schedule—defined in a special descriptor table located in the memory of the controller—and provides a shared memory interface for the host processor to access the messages to be sent and received.

The message schedule defines a *cluster cycle*—or multiple cluster cycles if cluster mode changes are allowed—which is executed periodically (see Figure 2). The cluster cycle is further divided into *TDMA rounds*. Within a TDMA round each node is assigned to exactly one slot—a time interval where it can send messages packed into a single frame on the communication bus. The length of these slots may differ for different nodes, but are constant throughout the TDMA rounds in the cluster cycle (thus the length of the TDMA rounds are of the same length in the cycle). Mode changes may alter the length of the cluster cycle (the number of rounds in a cycle) and the message contents of the frames, but should use the same ordering and lengths of the time slots for the nodes. These spartan scheduling rules make mode changes and scheduling more deterministic and easier to handle but have crucial effects on the system design. In a typical TTP application the length of the TDMA rounds are around 1-2 ms and the precision of the synchronization is below 5 $u$S (re-synchronization occurs at the end of each round). They however vary substantially in the number of rounds in a cycle and—obviously—in the length and contents of the frames.

The application design on the TTP platform may follow two different approaches. In the simpler case, the application level tasks are not synchronized to the communication bus and access the messages via the shared memory interface as they wish. This approach has obvious limitations (no guarantees on end-to-end jitter and latency) and can hardly be considered real-time. More interesting is the second alternative, where application tasks are activated in sync with the message schedule. For supporting this method the communication controller provides interrupts to the host processor at well defined time instants while executing the protocol. The real-time operating system running on the host processor is able to align the task schedule with the bus schedule. Evidently, the limitations on the message schedule affect the task schedule also in these systems. The application cycle has to follow (match in its length) the cluster cycle. Also, tasks cannot be released before the arrival of their input message(s) and shall finish before the transmission of their output(s). Furthermore, fault management for subsystem replication (tolerance against faults in the value domain) is implemented as higher level services running on the host processor, thus additional tasks and their dependencies have to be reckoned with in the application design. The task schedule with its important dependencies in a typical application is shown in Figure 3.

TO BE DRAWN IF NEEDED.

**Figure 2. Message scheduling in a TTP application.**

TO BE DRAWN IF NEEDED.

**Figure 3. Message scheduling in a TTP application.**

## 2. Generators

### 2.1. TTP Code Generator

The Time Triggered Platform presented several challenges to the code generators. It has intricate constraints on the task and message schedule and the message schedule is not contained by the application code but a special message descriptor file (MEDL), which is downloaded to the controller at startup. Due to the closed source model of the TTP software toolset we faced an important decision in supporting this platform. We either had to replicate (some of) the efforts in the TTP tools or somehow integrate them to our toolchain. For the first prototype the second approach seemed to be more reasonable. Each of the TTP tools provide a Python-based API for automating the design process. These APIs proved to be rich enough to integrate the tools to the prototype toolchain. The *TTP Code Generator* is responsible for the following steps:

**glue code generation:** The deployment model of the ECSL-DP language defines the tasks and the contained software components. The software components refer to Simulink blocks in the dataflow model. The TTP code generator is responsible for generating the ”body” of the time-triggered tasks, read the input variables, collect (through the deployment and software component models) and invoke the functions generated by the Simulink/Stateflow code generators and update the output variables in the communication controller. The generated glue code is also responsible for the cluster startup and re-integration tasks.

**message scheduling:** In parallel with the code generation steps, the TTP code generator builds a python script as an input for the *TTPplan* tool. The generated script defines the global properties of the cluster, the messages, their types and required frequencies, the nodes and a default cluster mode. The source of this information is the deployment model.

**task scheduling:** The code generator also builds pythons scripts for the *TTPbuild* tool—one for each node. The script defines the application level tasks (based on the deployment model), their worst case execution times and builds the connection to the input and output messages.

**compiler execution:** Finally, the TTP software tools are executed with the generated script files. TTPplan generates the message schedule (if the scheduling requirements can be satisfied), TTPbuild customizes the MEDL for each node, generates a configuration file for *TTPos*—a real-time embedded operating system running the cluster nodes—and the source code for the fault tolerant communication layer. Provided that the previous steps succeeded, the TTP code generator invokes the compiler on the generated source files (Simulink/Stateflow code, TTP glue code, fault tolerant layer and operating system configuration).

## 3. Future work

Although the presented toolchain in its current early prototype stage addresses many of the challenges inherent in a unified design and implementation environment, it lacks many of the desired features and workflow procedures to be useful in real applications.

The most critical functionality, which is missing or is present only implicitly is the feedback from the implementation phases into the high level design. The chosen target platforms enabled us to understand the nature of these influential properties (eg.: jitter, quantization, constraints on the schedule, fault tolerant properties, computational models). It remains an essential part of our ongoing quest to widen this set by experimenting with different deployment platforms. On the other hand feedback mechanisms will require formal definitions of such properties. Thus, one of the main focus areas in future development is the formalization of platform properties and their utilization in the design phases as early as possible.

A related area in our plans is the design and implementation of an *adaption layer* on top of the different target architectures. This layer shall hide many of the differences of the platform APIs but should be flexible and thin enough not to try to hide the important characteristics of the underlying platform. The emphasis is on API adaption not emulation and is essential to avoid or to lessen the need for ”glue code” generators. We have already made the first steps in the design of this layer and our Linux-based virtual machine can be considered as an early prototype of this attempt. Ideally, the toolchain should be able to handle any target platform if the proper adaption layer and a formal set of platform properties are available.

Finally, the presented tools mostly support a one-way ”single shot” development workflow. Changes to the input MATLAB/Simulink model require the ECSL-DP model to be regenerated with the MDL2MGA tool. Even though this affects only the dataflow aspects of the model, this step invalidates existing references from the software component model and deployment models. Clearly, the MATLAB/Simulink model importer should be prepared to handle existing output models and should execute the changes only. With properly identifying the Simulink elements and their corresponding ECSL-DP pairs and by restricting the set of supported changes to the dataflow part of the model we will be able to overcome this limitation without bloating the importer tool.

Along a typical workflow many files are generated and/or modified by the designer. Most of these files are final output or report files, but some of them serve as input for later stages in the workflow process. Currently, the toolchain does not address versioning and dependency issues between the different tools and intermediate files. Tool integration in its current form consists of common APIs, modeling languages and file formats. The underlying development tools (GME, UDM, GReAT) provide an excellent framework for developing an integrated toolset, but this level of integration is not evident on the user level. We shall provide a high level framework—ideally in the form of GUI application—which executes the elements of the toolchain, tracks changes and dependencies and supports the user in understanding the different stages of the design process. This approach is very similar to current HDL design environments[2][1], which are built primarily on command line tools with a high level GUI driver application.

A related but somewhat more complicated issue with various files and models is that these files—ideally—contain related and/or redundant information. MATLAB/Simulink models have almost the same information as ECSL-DP dataflow models. Component and deployment models refer to the dataflow elements and to each other. The generated code and/or SFC models have clear origins in the ECSL-DP projects. Because of the different languages and different file formats one can found it extremely hard to navigate along these intricate connections. The high level integration framework (GUI) would ideally support the user in this task by maintaining a cache of meta information and present these links to the designer on-demand. Understanding the exact effects (or just the magnitude of the size of the affected part of the system) and knowing where a particular information or code artifact is coming from is surely invaluable in the envisioned design environment.

## References

[1] Altera Quartus HDL Design Environment. http://www.altera.com/.

[2] Xilinx ISE HDL Design Environment. http://www.xilinx.com/.

[3] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, Oct. 2001.