Towards a Model-based Toolchain for the High-Confidence Design of Embedded Systems

Péter Völgyesi, Gábor Karsai, Sandeep Neema, Harmon Nine,
Joseph Porter, Ryan Thibodeaux, and János Sztipanovits
Vanderbilt University
Institute for Software Integrated Systems
Nashville, TN 37235, USA
peter.volgyesi@vanderbilt.edu

Abstract

While design automation for hardware systems is quite advanced, practical embedded systems software badly needs such automation. The current state-of-the-art is to use a software modeling environment and 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 concepts, elements, and some early prototypes for an envisioned suite of tools for the development of embedded software that integrates verification steps into the overall process.

1. Introduction

Constructing embedded software for applications that require high confidence in the software's functionality, reliability, and safety is hard. We have to not only design, implement, and verify the software, but also need to provide arguments for certifying its behavior. The problems of embedded software development are well-known [13], yet industrial practice falls short of these expectations.

Existing development practices often fall into two categories. In one category we find the use of a model-based development environment (which typically includes a graphical model building tool supporting block-diagram and statechart-like notations, a simulation engine, and a code generator), which allows programming and the generation of code for the embedded system. The two most frequently used such environments are Simulink/Stateflow of Mathworks, and Matrix-X of National Instruments. In these toolchains the prevailing paradigm for development is to use dataflow diagrams and statecharts to specify code functions and perform simulations of the expected environment, then to generate code for the 'controller' part of the models, and finally to deploy that code on some real-time OS platform. The second category of software develpment uses a UML-oriented modeling tool, possibly a code generator, as well as an interactive development environment (IDE). Examples here include numerous products from UML tool and IDE vendors. The prevailing paradigm here is to use the UML tool to create models of the software, optionally using the UML code generator to translate the models into an object-oriented language, then debug and deploy the code using the IDE.

Both of these approaches have shortcomings. For the first case, advanced software engineering concepts are rarely used, platforms and their properties are rarely captured in models, and non-functional properties (e.g. fault-tolerance) are not explicit and very hard to reason about, just to mention a few. For the second, UML does not seem to be suitable for modeling the environment of the system (i.e. the physical dynamics of the 'plant'); there is often a disconnect between the models and the code as deployed, and the same problem occurs with non-functional properties. In both cases, support for verification and certification of the end-product is often missing.

Accordingly we need tools and techniques that could address the wide variety of problems arising in embedded software development. Because of the great variability of embedded software (ranging, for instance, from MP3 players to avionics systems to distributed air-traffic control) and the continuous evolution of the available tools, we believe a single tool will never be sufficient – i.e. we need to think in toolchains. These toolchains should contain elements that address the various expectations of embedded system developers and organizations that are interested in the quality of the results.

This paper introduces the conceptual design of such a toolchain and describes some early results in its construc-

tion. The next section describes the envisioned development flow and the expected toolchain elements, and the subsequent sections describe the individual elements already constructed or being developed. The paper concludes with a comparison to other toolchains and a discussion of future work.

2. The vision

In this work, we envision a sophisticated, end-to-end toolchain that supports not only construction, but also the verification of the engineering artifacts (including software) for high-confidence applications. The development flow provided by the toolchain shall follow a variation of the classical V-model (with software and hardware development on the two branches), with some refinements added at the various stages. Figure 1 illustrates this development flow.

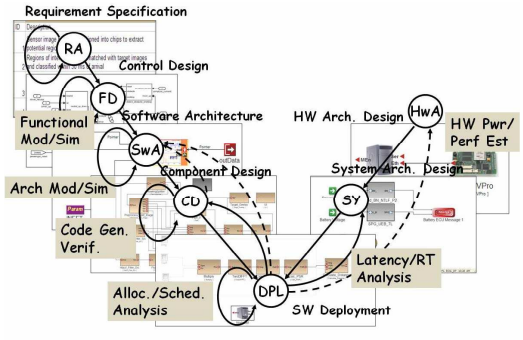

Figure 1. Conceptual model of the toolchain: Development flow

In the development process, requirements analysis (RA) is followed by functional design (FD) for the controllers, typically enacted using a controller modeling tool like Simulink/Stateflow or Matrix-X. Controller models typically involve (synchronous) dataflow (block) diagrams and some variant of Statechart [11] models. This step is followed by laying out the overall software architecture (SwA) as well as the hardware architecture (HwA) (on the other side). We assume that models that capture both the SwA and HwA aspects of the system are available, or can be constructed. Controllers (produced in the form of models) are componentized in the component design (CD) phase - this usually involves 'wrapping' the code generated from the block diagrams such that they become integratable components on some run-time component framework. On the hardware side, the architecture design is closely related to system architecture (SY) design (which is about the integration of hardware and software aspects), and software components must be deployed on hardware elements (DPL). We envision that multiple, yet interlinked models will be used in all stages of the design process, and engineers work through the design flow via models to construct the system.

We also envision that such a model-based design flow integrates a number of verification activities. For instance, in FD today we often use simulation for verifying the functionality of the controllers. In the future this could be extended with the use of advanced model checking tools. When code is generated from the models, the generated code could be subjected to code-level verification regimes to check, for example, that the generated code will work with a bounded stack. When components are allocated to hardware and software resources (e.g. 'tasks'), the resulting system model could be used for schedulability analysis (assuming the problem is decidable and data for worst-case execution time is available).

In summary, we envision that model-based embedded software development toolchains will be extended with verification and validation tools such that models and code could be continuously subjected to these checks, and the results could be used in the certification of the embedded system produced. Below we discuss some design rationales for the actual implementation of the toolchain.

2.1. Platform

In order to reduce design complexity, we believe a toolchain needs to follow the principles of platform-based design [19]. A 'platform' in this case means a 'component framework' that provides some common services used by all applications. Existing commercial real-time operating systems typically provide elementary services (concurrent tasks, task synchronization and communication, partitioning, etc.). We believe embedded systems require better software platforms with higher-level services, e.g. remote method invocation, time-triggered execution, support for composing systems from components, fault-tolerant real-time communication, replica determinism, and others. The main expected feature of a component framework is to be able to compose systems from components in such a way that the properties of the system could be determined from the properties of the components and the way they are composed. However, this composability property is tightly interwoven with the provided services of the framework, and they cannot be separated.

Component frameworks for embedded systems are an active area of research, and there are numerous research prototypes as well as industrial implementations. In high-confidence systems we need platforms that provide services

and guarantees for needed properties (e.g. fault containment, temporal fire-walls, etc.) These services should be provided by the platform because they are critical (like partitioning), and system developers should not re-implement them from scratch.

Note that the platform, as a component framework, also defines a 'Model of Computation' [17]. An MoC governs how the concurrent objects of an application interact, how synchronization and communication takes place, and how these activities unfold in time. The MoC also determines how composition works, and, implicitly, how the system-level properties could be derived from the component level properties and the composition.

The above discussion underlines the need for the careful selection of a platform. Initially in our work we wanted to focus on the design of digital controllers that operate periodically, according to precise timing specifications. These controller components could communicate with other controller components as well as software components that encapsulate peripheral interfaces. For high-confidence designs, distributed fault tolerance (while maintaining real-time properties) is essential, and, according to our knowledge, currently there is one approach approach that provides all of these: the Time-Triggered Architecture (TTA). Hence, in the initial version of the toolchain our chosen platform was the TTA.

2.2. Modeling Language

Model-driven development is based on the use of models expressed in modeling languages, while model-integrated computing refines this paradigm by emphasizing the use of *domain-specific* modeling languages. Industrial practice, especially UML, in its default form, is capable of expressing domain-specificity through the use of UML profiles; however, we did not find profiles adequate for our applications. Hence, we have chosen to develop a domain-specific language for modeling and deploying high-confidence control systems.

A modeling language to support the development flow described above should have several desired properties, as listed here: (1) it should be able to "understand" (and import) functional models from existing design tools, (2) it should be able to capture the relevant aspects of the hardware system and architecture, (3) it should support the modeling of components and the componentization of functional models, (4) it should be able to model the deployment of the software architecture onto the hardware architecture. The need for being able to import existing models from functional modeling tools is not a deeply justified requirement, it is merely pragmatic.

The design of the language was influenced by two factors: (1) the MoC implemented by the platform, and (2) the need for connectivity towards legacy modeling and embedded systems tools. We have chosen Simulink/Stateflow as the "legacy" tool with which we wanted to maintain the integration. As our chosen MoC relies on periodically scheduled, time-triggered components, it was natural to use this concept as the basis for our modeling language and interpret the imported Simulink blocks as the implementation of these components. Communications between Simulink blocks (the signals), are mapped onto TTA messages communicated between the tasks. The resulting language provides a componentized view of (discrete-time) Simulink models that are scheduled periodically (with a fixed rate) and communicate with each other using time-triggered messages.

Note that this is only the initial implementation of the modeling language, and we plan to extend it towards other MoC-s. For instance, we need provisions for event-triggered communications and event-triggered components. Such event-triggered component structures give rise to interesting communication patterns that seem to be very useful in practical systems (e.g. publish-subscribe, rendezvous, and broadcast). We are planning to integrate support for these in our modeling language in the near future.

2.3. Code generators and synthesis tools

Model-based development is often called higher-level programming, i.e. programming using models instead of an algorithmic language. This means that we need to generate executable code from the models. This also means that the generator could be more than just a simple model-to-code translator; it could be a code 'synthesizer' that synthesizes low-level code from higher-level specifications. Synthesis is different from translation as it can involve some notion of search during the code generation process. For instance, if scaled fixed-point arithmetic is to be used, the synthesizer could automatically select the appropriate scaling factors for every operational step, (in, say, a dataflow model) based on the value ranges for the input and output variables and the number of bits available for representing fixed point numbers on the hardware.

Generating code from dataflow and Statechart models is a well-defined and solved problem, and there are many actual implementations. Code synthesis from higher-level models, however, is an active area of research with many open questions regarding code efficiency and correctness.

In our toolchain we created a number of code generators (no synthesizers yet). In the construction of the two main code generators (one for Simulink-style models and another one for Stateflow-style models), we have used a higher-level approach based on graph transformations. This approach is based on the assumption that (1) models are typed and attributed graphs with some specific structure (governed by

the metamodel of the modeling language) and (2) the executable code can be produced in the form of an abstract syntax graph (which is then printed directly into source code). This graph transformation-based approach allows a higher-level representation of the translation process and lends itself to algorithmic analysis of the transformations.

2.4. Verification and analysis

As discussed above, verification and analysis must be an inseparable part of the development process for high-confidence system developers. There are a number of well-known verification techniques and tools; however we must emphasize two points of concern here: One, the verification tools often operate on the models (i.e. an abstraction of the system), not the code of the system. Two, verification results are meaningful only if the model transformations (that map design models into analysis models or into code) are correct, i.e. properties are carried over by the transformation and the transformation does not add any new artifacts that extend the behavior of the generated code compared to that of the model.

As we envision it, there are several opportunities and challenges for verification. First, verification can be done directly on the models; in this case the design language and the analysis language are one and the same. Unfortunately, this is not a common case, except in simulation-based approaches. Second, verification can be done on analysis models (i.e. models re-expressed in the language of some verification tool), but in this case there is a need to show that the transformation of design models into analysis models is correct, i.e. the analysis results are true for the original models. Third, verification can be done on the level of the (generated) code, but in this case we typically lose the higher-level abstractions of the original models that could make reasoning efficient and its results close to application problem. In the first two cases we also need to address the question of how the verification results carry forward to the (executable) code such that we can claim that the code (the final product that runs on an embedded system) satisfies properties of interest.

Construction of such verification-based tools and the verifying of model translators is an active area of research. The correctness of a model transformation is a very complex problem, but some early results indicate a promising direction[10]: instead of proving the correctness of a transformation in general, one can show that a particular instance of a transformation preserves the properties of interest, and thus could conclude that the properties hold for the input of the transformation as well.

Another promising direction that we plan to pursue later with our toolchain is a connection between the models and the generated code. We are working on extending the code generator such that it carries forward model-level information to the generated code (in the form of annotations) that provide help for the code level verifier to check model-level properties on the code. This connection could potentially be used to improve performance, as the verifier could reason using higher-level abstractions than those immediately available from the code.

3. Elements of the toolchain

Below, we describe the implemented elements of the toolchain. Figure 2 illustrates the toolchain as it exists today, and it shows how the services needed (e.g. scheduling) are implemented by the specific element.

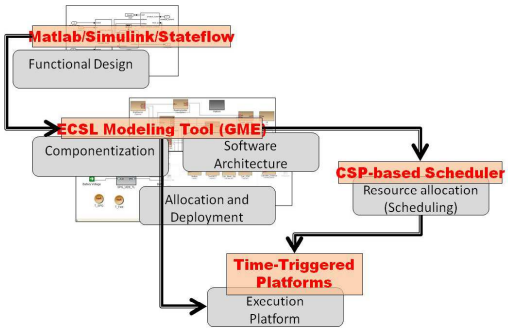

Figure 2. Existing elements of the toolchain

3.1. Platforms

The carefully chosen set of target platforms have great impact on the design of the proposed toolchain. As we further refine the design environment—particularly the feedback mechanisms from the deployment stages—this will remain so in the future. Based on our initial experience with various platforms, we recognize two fundamentally different computational models in the task scheduling and communication domains. We believe that many of the practical distributed embedded environments can be described by a combination and blending of these models.

Event-triggered (ET) real-time systems are driven solely by the occurrence of various events—like the reception of a message, the assertion of an external interrupt line, or a timeout. In this computational model, the system responds to these events as soon as possible, resulting in a very flexible but highly unpredictable environment. A notable purely event-triggered embedded distributed platform is TinyOS [18].

In stark contrast to event-triggered 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.

One of the most important conclusions we reached in the early stages of the project was to separate the communication models and task scheduling models on the target platforms. This enabled us to mix and match these approaches and provided the flexibility to characterize real-world embedded execution platforms. The interactions between these models and domains are an interesting and important research area [8]. However, in the prototype version of the proposed toolchain, we focused on the time-triggered scheduling and communication aspects of the target platforms.

Although these platforms differ in their implementation of the time-triggered approach—later sections will detail these differences—all follow some general rules and use similar concepts. We support a general TT component model in the modeling language we developed, called ECSL-DP. According to the TT model the timeline (both in the communication and task scheduling domains) is divided into continuously repeating periods. The task and message schedule describe the partitioning of this period. The length of the periods for task scheduling and for communication are of equal length but obviously have different partitioning. Optionally, the period can be divided into subperiods of equal lengths (hardware implementations of TTA function in this way). Both the communication subsystem and task scheduler wait for the beginning of the period—in a synchronized manner—and control access to the communication medium and/or release tasks according to a predefined timetable throughout the period. At the end of the period the whole process starts again.

3.1.1 The Time Triggered Architecture platform

One of the experimental target platforms for the toolchain is the Time Triggered Architecture [15] developed at the Vienna University of Technology and by TTTech Computertechnik AG. The primary reasons for selecting TTA in general and the TTP-Development Cluster (see Figure 3) in particular as a test platform were its fault tolerance features, performance characteristics, and its technological maturity.

Fault tolerance is supported by redundant communication channels, by replicated subsystems with constant state synchronization and voting in the value domain, and by various provisions in the communication protocol (e.g. cluster startup, distributed time synchronization, and membership management with clique avoidance). The measured 1-5 μs jitter and its relatively high bandwidth (5-25 Mbit/s) with

Figure 3. The TTP-Development Cluster with eight nodes.

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 its composability aspects. A well-designed cluster schedule (with enough spare capacity for future growth) 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 to get a deeper understanding of time triggered computational models. On the other hand, we faced various constraints and peculiarities inherent in the TTP protocol, some of which had profound effects on the system design and on the design environment. These lessons confirmed to us the 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 nodes 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*, in which granularity is limited by the precision of the distributed clock synchronization. The granularity factor (Π) 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 the phys-

ical 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 4). 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 they 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 the same ordering and lengths of the time slots for the nodes should be used. These spartan scheduling rules make mode changes and scheduling more deterministic and easier to handle, but they 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 uS (re-synchronization occurs at the end of each round). However, the number of rounds in a cycle varies substantially, as do the length and contents of the frames.

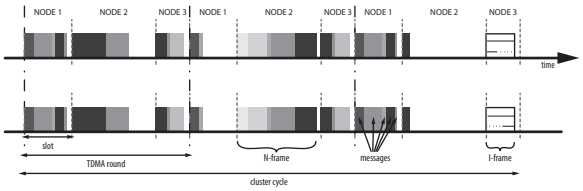

Figure 4. Message scheduling in a TTP application.

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 tasks 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. To support 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 must 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 considered in the application design. The task schedule with its important dependencies in a typical application is shown in Figure 5.

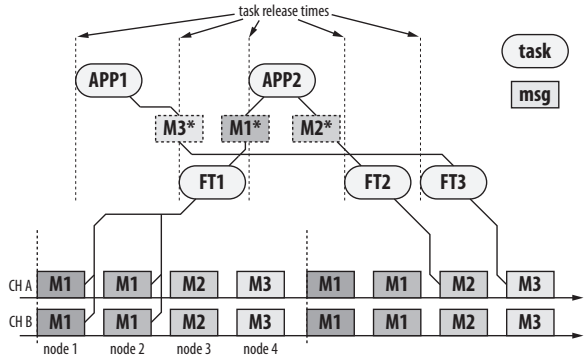

Figure 5. Message scheduling in a TTP application.

3.1.2 The FRODO platform

Although the TTA approach to fault-tolerant execution of control applications has been developed and matured over many years, its current manifestation only as a hardware specific platform restricts the flexibility and portability of possible applications developed using the toolchain. These obstacles motivated a second deployment platform for the toolchain that offered the same deterministic execution of the TT MoC with off-the-shelf components and a hardware/software independent platform API. The resulting platform uses an abstract virtual machine, called FRODO, for executing TT control tasks of high-confidence systems coupled with a message controller that provides the TT transmission of data message across networked nodes, all of which is implemented using readily available infrastructure.

Architecture

This platform's implementation architecture is based on logical separation principles similar to those of other TT platforms like Giotto [14] and TTA [15]: the periodic execution of control tasks is determined solely by the passage of time and the current operational mode of the system, i.e.

events such as the arrival of new data messages do not affect the timely task execution. Accordingly, only the most recent data values, representative of some control signals, are relevant to calculations performed by TT tasks; therefore, a globally shared memory construct is used to read and update data throughout execution. The resulting architecture is provided below in Figure 6. In the figure, bidirectional arrows represent the flow of data throughout the system whereas the unidirectional arrows indicate the sending of relevant events. The current platform implementation uses a standard Ethernet UDP network for communication, and each node is running an unmodified Linux 2.6.x kernel on an embedded 266MHz 586 processor with 256 MB of RAM.

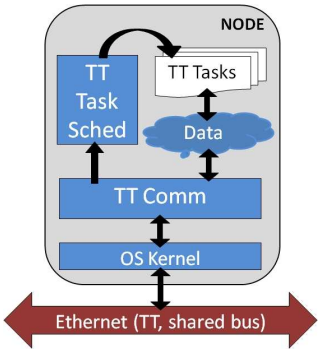

Figure 6. FRODO implementation architecture.

Time-triggered communication

Maintaining low cost and platform independence for the message controller was necessary to making this platform as portable as possible. Accordingly, the communication controller expects only a shared bus network configuration for connecting all of the nodes in the system and a library of methods that provide the basic functionality for sending and receiving messages. Unlike the TTP/C capable platform described above but similar to the TTP/A protocol [16], this platform is not yet robust enough to provide a fault-tolerant synchronization approach for maintaining a global clock; instead, a designated communications controller in the system is responsible for initially verifying all of the expected nodes are connected to the network before starting execution and synchronizing the start of the each message schedule round with all of the nodes. All other nodes on the network will not begin executing unless confirmation is received that all nodes are present at startup, and they will also fail to start a new schedule round unless the proper synchronization message is received.

Based on the previous discussion of the properties of the TT MoC, the certification/verification of distributed control systems is contingent upon the deployed system's ability to accurately follow a fixed message schedule; therefore, the use of a TT capable message controller is paramount to this platform's applicability for systems developed using the toolchain. Like the TTP approach, a static message schedule is generated offline, passed as an input to the platform, and cyclically iterated throughout the execution; however, the format of the message schedule is not identical to that used in TTP. TTP requires the use of a strict TDMA format where all nodes have a scheduled time-slot for transmitting messages in each round. Conversely, this platform does not place any minimum or maximum on the amount of data that can be sent from a node in a round except those constraints that maintain the feasibility of the message schedule. Instead, the static message schedule specifies exactly which node sends a message at an exact scheduled offset with respect to the start of the schedule round. Each schedule round is one hyperperiod long, and the hyperperiod is of fixed duration. A node will be allotted no messages/slots if it is not expected to transmit any data within a schedule round. If a message is expected to be sent with a frequency greater than one within a round, it will be listed as many times in the schedule as it is expected with each appropriate scheduled transmission time. The length of each message, in bytes, is also indicated in the message schedule.

The current steps for establishing and maintaining TT communication across a network of nodes implemented using this platform proceeds in the following steps:

1. Discovery of all nodes required by the implementation
2. Transmission of hyperperiod start signal (beginning of schedule execution)
3. Transmission of data messages in their strict scheduled order
4. End of hyperperiod is reached
5. Repeat starting from step 2 or terminate execution

Obviously, while each individual node is preparing to transmit its scheduled data messages, it must also be configured to receive the broadcast messages from the other controllers in order to maintain the most up-to-date data set. Each outgoing message includes a message indicator such that the receivers can quickly locate the appropriate memory location to update with the received value. A snapshot of the activity on the communication network over two hyperperiods of the message schedule is provided in Figure 7.

Time-triggered task execution

The application code that provides the functionality of the individual toolchain components is generated completely

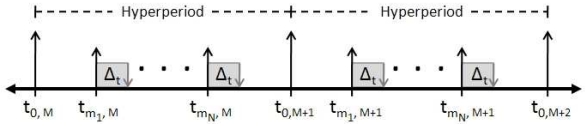

Figure 7. Activity timeline across time-triggered network.

agnostic to the choice of target platform(s). Instead, the tasks that contain the application code of the components provide only timing information regarding their execution and require access to the data relevant to the control calculations they perform. Accordingly, the platform's task scheduler need only expose an interface fitting the tasks' needs and have the capability to invoke and terminate task execution in order to provide precise controller execution. FRODO is such an abstract, platform-independent executive (virtual machine) responsible for providing the necessary TT execution of control tasks on this platform.

Most commonly, digital implementations of controllers operate in a periodic fashion – sampling control signals, computing the next control signal and the current state of the system, and performing an actuation based on the computations. Likewise, the task scheduler of FRODO is responsible for initiating the periodic execution of the control tasks based on a static task execution schedule initially passed as input to the system. The current implementation of FRODO does not allow the preemption of tasks, i.e. only one periodic control task is released for execution at any given time. In order to maintain the timely execution of the control tasks according to the schedule, FRODO is also responsible for terminating executing tasks that have yet to finish prior to reaching their worst-case execution time (WCET). The task schedule for a node is similar to the message schedule mentioned in the preceding section: each task invocation is explicitly listed in the schedule with its appropriate release time with respect to the start of the schedule and the task schedule has the same length or hyperperiod as the message schedule. Synchronizing the execution of the task schedule with the message schedule maintains the controller correctness with respect to the current control signals; therefore, each new round of the task schedule has to be initiated by the arrival of an event from the underlying communication controller indicating that a new schedule round is beginning. If no such event is received, the control tasks will not be released for execution.

Only as a scheduled task begins and (properly) ends execution, it will require the necessary access to the globally shared memory to read/update the control signal data used throughout the system. Accordingly, FRODO provides two methods for performing such operations; however, internally FRODO must use proper access-control locks to ensure that it is not updating values concurrently with the message controller.

3.2. Modeling languages

As mentioned earlier, graphical modeling languages and transformations bind together functions of the toolchains. These languages are defined using metamodels in the Generic Modeling Environment (GME) [4], which is also used directly to edit models thus represented. Graph transformations are specified using GReAT [5]. These generic tools aim to provide the necessary framework to enable an end-to-end development solution: from function design using Simulink/Stateflow, through verification of critical operational properties, down to certified code that can be used confidently on the specified platform. A simple illustrative example demonstrates the current state of the tools, the intended design flow, and the integration of disparate functions using the model-integrated computing (MIC) approach.

Consider a simple system consisting of a digital compass sensor, a servomotor, and a few small embedded processors connected by a single bus. The goal is the creation of a simple, digital open-loop controller that drives the servomotor to track the orientation of the compass. The model of computation for network communications is TTA. The compass only provides data through half of the full circle, so the translation must account for the valid range of sensor values. We will hold the last valid value when the measurement is out of range. The design begins in Simulink with the realization of two functions:

1. For input orientation data $o_k \in [0, 179.9)$, application of a small smoothing filter should clean up high-frequency noise:

$$o'_k = a_0 o_k + a_1 o_{k-1} + a_2 o_{k-2}$$

2. The smoothed orientation must be converted to a pulse-width modulation (PWM) value to control the servomotor position. We also must hold the previous PWM value indefinitely if the measured orientation is out of range.

$$p_k = \begin{cases} p_{k-1} & o'_k < 0 \\ p_{k-1} & o'_k \geq 180 \\ 1440 + \frac{6227 \cdot (1800 - o'_k)}{1800} & 0 \leq o'_k < 180 \end{cases}$$

Figure 8 shows a simple Simulink model realizing these two functions. They are placed in a test harness with producer/consumer functions to simulate them over a range of inputs. Once simulation results are satisfactory, we are ready for software design. The ECSL-DP (Embedded Control System Language for Distributed Processing) contains

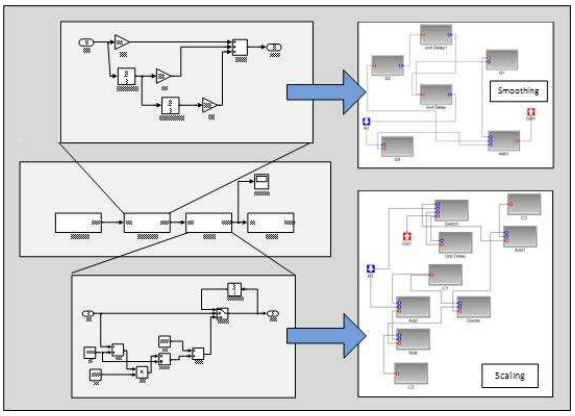

Figure 8. MDL2MGA – Simulink to ECSL-DP model translation.

modeling paradigms representing Simulink/Stateflow models. The MDL2MGA tool invokes MATLAB and creates an ECSL-DP model that captures the structure and function of our two subsystems. The Simulink and Stateflow paradigms in ECSL-DP attempt to faithfully model all of the structures and parameters of Simulink and Stateflow. Data types and bus widths are inferred from the configuration, as they are in Simulink.

Once the Simulink dataflow models are imported into our modeling language, they are assigned to individual software components by the software designer. These components refer to the dataflow models, and they serve to specify subsets of the Simulink/Stateflow functions for software implementation. Later (when mapping components to hardware) we will add operational parameters such as execution period and worst-case execution time for the functions. One component block is created for each subsystem, and connections between the ECSL-DP components represent channels for exchanging signals/data between software components.

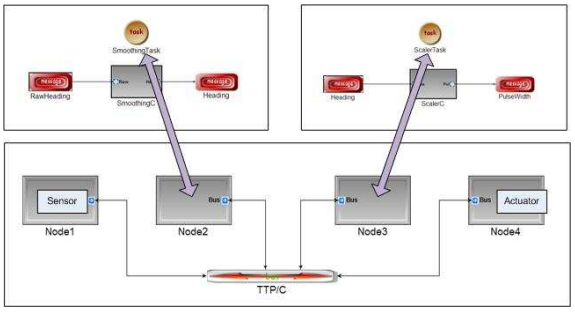

Figure 9. Mapping software tasks to hardware nodes in ECSL-DP.

Finally, the ECSL-DP modeling language includes deployment concepts. We capture a description of the TTA platform, and assign components to run on it. Note that a central feature of the toolchain is platform modeling – TTA and its supporting hardware are appropriate for our current applications, but the modeling language is designed for change. The hardware and/or the distributed model of computation could be replaced if different guarantees or levels of performance are needed. Task configuration for TTA includes execution periods and worst-case execution times. Hardware configuration data includes processor and bus parameters, most notably bus data rates. Figure 9 shows the hardware topology and the mapping of tasks to hardware nodes. Data is sent between components via bus messages, which contain implementation-specific information as well as the data values specified in the data flow models.

In summary, consider the different aspects of the model we began with – functional blocks connected by signals. These were wrapped, respectively, in components and their associated communication channels, which are mapped (respectively) to tasks on processors and messages on buses. The resulting layered model captures all of the necessary information to generate code, create time-triggered schedules, and configure the platform. These functions are currently available in the tool chain, and they are implemented using model interpreters and graph transformations, described below. The next steps in tool evolution will add verification concepts at different stages of the design and integrate the proper tools to enable safety-assured software designs.

3.3. Generators

General Discussion

The main role of the code generators in the proposed toolchain is to transform the dataflow and statechart models into executable code on the selected hardware/software platform as it is described by the component and deployment aspects of the ECSL-DP model. Currently, different generators exist in the toolchain for the different parts of the model and the final compilation and linking step combines the generated artifacts. The SDF and HFSM code generators are the key components of the code generation infrastructure, and they share their internal architectures and underlying technologies. Both tools are built using the GReAT engine [5], and they employ graph rewriting rules on the dataflow and statechart models to transform them into an intermediate model-based representation of the executable code. This approach has benefits. The heavyweight part of the code generation is decoupled from the concrete syntax of the platform language—in this case C or C++—consequently, it is remarkably straightforward to add support for additional procedural languages. Also, the model-based representation of the generated code (in the

form of abstract syntax graphs) provides a clear and well-defined programmatic interface to the generated code for verification tools or other third party utilities. Although the proposed toolchain elements are built on strict and formal graph transformation foundations, the code generation infrastructure shows many similarities to the overall approach taken by the GNU Compiler Collection, which partitions the compiler workflow into front-end and back-end systems with the Register Transfer Language intermediate representation between these [2].

Synchronous Dataflow code generator

The Synchronous Dataflow (SDF) code generator (*SL_CodeGen*) performs a topological sort on the input dataflow model before it executes the transformation rules which finally yield executable code. Signal paths become variables and dataflow blocks are transformed into functions. The code generator is prepared to handle many of primitive processing elements and automatically generates their implementation in the given context. The following Simulink blocks are supported currently: `Product`, `MultiPortSwitch`, `Constant`, `Signum`, `Sum`, `ToWorkspace`, `Math`, `BusSelector`, `Mux`, `RelationalOperator`, `Demux`,`UnitDelay`, `Terminator`, `Gain`, `Saturate`, `MinMax`, `Switch`, `Ground`, `From`,`Goto`, `Abs`

The internal architecture of the code generator makes it rather straightforward to add support for other blocks in the future. Scalars, vectors, matrices and structures are supported on the signal paths.

Hierarchical Finite State Machine code generator

The Hierarchical Finite State Machine (HFSM) code generator processes imported (sub)models from Stateflow language. Thus, it follows the semantics of a subset of MATLAB/Stateflow ([7]) as closely as possible while generating the executable implementation. It employs sequential evaluation semantics for state transitions. The transitions are evaluated on the state hierarchy with a depth-first search for active states. A state transition into a new state implicitly enters all of its parent states. Like in MATLAB/Stateflow, the code generator assigns and generates three functions for each state, one for executing actions upon entering the state, another for the exit actions and the third one for performing tasks while the state is active.

3.3.1 TTP code generator

The Time-Triggered Platform presented several challenges to the creation of code generators. It has intricate constraints on the task and message schedules, and the message schedules are not contained by the application code but by 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 functions of the TTP tools or somehow integrate them into 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, reading the input variables, collecting (through the deployment and software component models), invoking the functions generated by the Simulink/Stateflow code generators and updating 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 Python scripts for the *TTPbuild* tool—one for each node. The script defines the application level tasks (based on the deployment model) with 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 generates 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 tolerance layer, and operating system configuration).

Constraint-based scheduler

Commercial TTP tools provide deployment analysis and schedule generation capabilities subject to use on a particular hardware platform as described above. The toolchain includes a more general schedule generation tool that can create time-triggered schedules for distributed tasks and messages. A GReAT transformation extracts the critical details from an ECSL-DP model and models them in a simple system description language called *CSched. CSched* includes system description models, instance graph models, and constraint program models. Once the system model is created using the graph transformation, the system and constraint models are automatically created by a model interpreter. Constraint models are solved using the finite-domain constraint library Gecode [3]. Any feasible solution represents a valid schedule for all tasks and messages in the system. Constraint-based schedule generation has been shown to scale well to large problem sizes. Schild and Würtz report successful scheduling of up to 3500 processes and messages, 1 million constraints, and a search space dimension of 6 million starting times [20].

The difficulties of representing TTA scheduling problems as finite-domain constraint models lie in capturing the variations of order that can occur among the task and message instances and in attempting to optimize the orderings to satisfy specified latencies. Schild and Würtz [20] describe a two-stage modeling technique for this problem: in the first stage, task and message dependencies are resolved and the resulting execution order is captured in constraints; in the second stage, the constraint space is searched to satisfy latency constraints within the established ordering. Our approach simplifies this technique by using additional global serialization constraints to represent the families of task/message orderings in an attempt to reduce solution effort to a single stage. This work is still in progress, but the tools do produce valid schedules for many configurations and are fully integrated into the toolchain.

4. Related work

A number of modeling tools provide similar functionality to the toolchain elements described above. We describe a few of the more notable examples here with brief descriptions and comparisons.

Mathworks' Simulink/Stateflow. [7] Mathworks tools such as Simulink and Stateflow are integrated into the toolchain, but add-ons are also available from the Mathworks that provide code generation and verification. The Real-Time Workshop add-ons allow flexible code generation for a number of embedded target platforms. Our toolchain differs in that it provides direct modeling of the distributed platform and computation of time-triggered schedules to execute the code in a deterministic distributed way. The Simulink Verification and Validation tools provide links to requirements and tests, as well as test coverage analysis. This is just one part of the whole verification problem. More recently, Mathworks has release the Design Verifier tools, but we have not had the opportunity to evaluate them. From product literature, it appears that the tools focus on model coverage and test generation functionality, while providing additional user-definable properties for more general verification.

UCB Ptolemy II. [8] Ptolemy II is a freely available multi-platform modeling and design tool emphasizing composition of heterogeneous models of computation. The library of implemented function blocks (actors) and models of computation (directors) is very sizeable, and covers a number of domains. In particular, Ptolemy implements finite state machines, synchronous data flow, and time-triggered directors, which are all relevant to this research work. Additionally, Ptolemy has a new code generation mechanism based on partial evaluation [12] that seems promising, though additional work is needed to expand the library of actors and directors for which code can be generated.

Kennedy-Carter xUML Tools. [6] xUML is a commercial UML design tool that emphasizes executable design models. xUML provides configurable code generation from models as well as links to requirements, use cases, and test descriptions. In xUML, the notion of system correctness is limited to that provided by defined tests, rather than providing formal property checking.

5. 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 lacking functionality is feedback from the implementation phases back into the high level design. The chosen target platforms enable us to understand the nature of these influential properties (e.g. jitter, quantization, constraints on the schedule, fault tolerant properties, and 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 a 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[9][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 find 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] GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[3] Generic Constraint Development Environment. http://www.gecode.org/.

[4] Generic Modeling Environment. http://repo.isis.vanderbilt.edu/.

[5] Graph Rewriting And Transformations. http://repo.isis.vanderbilt.edu/.

[6] Kennedy-Carter Excecutable UML (xUML). http://www.kc.com/xuml.php.

[7] Mathworks Simulink/Stateflow Tools. http://www.mathworks.com.

[8] UCB Ptolemy II. http://ptolemy.berkeley.edu/ptolemyII/.

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

[10] A. Narayanan and G. Karsai. Towards verifying model transformations. In R. Bruni and D. Varr, editor, *5th International Workshop on Graph Transformation and Visual Modeling Techniques, 2006, Vienna, Austria*, page 185194, Apr 2006.

[11] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.

[12] Gang Zhou, Man-Kit Leung, and Edward A. Lee. A Code Generation Framework for Actor-Oriented Models with Partial Evaluation. In Y.-H. Lee et al., editor, *Proceedings of International Conference on Embedded Software and Systems 2007, LNCS 4523*, pages 786–799, May 2007.

[13] T. Henzinger and J. Sifakis. The embedded systems design challenge. In *FM: Formal Methods*, Lecture Notes in Computer Science 4085, pages 1–15. Springer, 2006.

[14] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Lecture Notes in Computer Science*, 2211:166–??, 2001.

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

[16] H. Kopetz, M. Holzmann, and W. Elmenreich. A universal smart transducer interface: Ttp/a. *International Journal of Computer System, Science, and Engineering*, 16(2), Mar. 2001.

[17] E. A. Lee and A. L. Sangiovanni-Vincentelli. A denotational framework for comparing models of computation. Technical Report UCB/ERL M97/11, EECS Department, University of California, Berkeley, 1997.

[18] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for wireless sensor networks. *Ambient Intelligence, Sprinter-Verlag*, 2005.

[19] Sangiovanni-Vincentelli, A. Defining Platform-based Design. *EEDesign of EETimes*, February 2002.

[20] K. Schild and J. Würtz. Scheduling of time-triggered real-time systems. *Constraints*, 5(4):335–357, Oct. 2000.

[21] I. The Mathworks. *Stateflow and Stateflow Coder 7 User's Guide*. The Mathworks, Inc., 2007.