07ATC-240

Model-Based Development of Distributed Embedded Real-Time Systems with the DECOS Tool-Chain

Wolfgang Herzner, Rupert Schlick
Austrian Research Centers GmbH - ARC

Martin Schlager, Bernhard Leiner
TTTech Computertechnik AG

Bernhard Huber
Vienna University of Technology

András Balogh, György Csertan
Budapest University of Technology and Economics

Alain LeGuennec, Thierry LeSergent
Esterel Technologies

Neeraj Suri, Shariful Islam
Darmstadt University of Technology

ABSTRACT

The increasing complexity of distributed embedded systems, as found today in airplanes or cars, becomes more and more a critical cost-factor for their development. Model-based approaches have recently demonstrated their potential for both improving and accelerating (software) development processes. Therefore, in the project DECOS[1], which aims at improving system architectures and development of distributed safety-critical embedded systems, an integrated, model-driven tool-chain is established, accompanying the system development process from design to deployment. This paper gives an overview of this tool-chain and outlines important design decisions and features.

INTRODUCTION

Today, the development of embedded systems still follows a customized design approach, resulting in rather isolated applications, reinvention of system design concepts, and little reuse of code across diverse application domains and even product families. So, for example in modern cars a number of sub-systems co-exist like the power-train control, the body electronics, or driver-assistance systems, each equipped with its own electronic hardware, communication cabling etc. While this federated approach eases system composition and supports fault encapsulation (a failure in one subsystem will not affect others), it implies at least increased hardware costs, weight, and power consumption. It also severely hampers the sharing of resources like sensors among the different sub-systems.

Based on these observations, the European project DECOS aims at developing basic enabling technology for moving from *federated* to *integrated* distributed architectures to reduce development, validation and maintenance costs, and increase the dependability of embedded applications in various application domains. An integrated architecture is characterized by the integration of multiple application subsystems within a single distributed computer system. When integrating subsystems into one platform, however, it has to be guaranteed that each of them can be executed in a protected environment that resembles the environment of the federated architecture, i.e. as it would not share resources with other subsystems. In particular, subsystems must not disturb each other, and faults in one subsystem must not propagate to others.

Efficient development of applications relies on a proper system architecture and on appropriate tool support. Within the DECOS project, a tool-chain has been

[1] DECOS (Dependable Embedded COmponents and Systems) is an integrated project partially funded by the EU within priority “Information Society Technologies (IST)” in the sixth EU framework programme (contract no. FP6-511 764)

developed for design, development, configuration and integration of applications [1]. In this paper we discuss the model-based development of distributed embedded real-time system based on the final version of this tool-chain.

The following section shortly outlines the DECOS architecture and gives an overview of the tool chain. Thereafter, the constituents for system modeling, configuration establishment, and behavior modeling are described. Finally, in the last section a conclusion is drawn and a short outlook is given.

DECOS OVERVIEW

CONCEPTS AND ARCHITECTURE

The basic principles for achieving dependability in a DECOS system [2] are strong fault encapsulation, fault tolerance by means of replication and redundancy, and separation of safety-critical from non-safety-critical functionality. These principles, in particular redundancy, lead to functional distribution. Additionally, supporting hard real-time applications requires guaranteed response times. Therefore, the functional structuring of a DECOS system comprises a number of (nearly) independent Distributed Application Subsystems (DASs), each realizing a part of the overall system service. DASs can be further subdivided into jobs, which represent the smallest executable software fragment in the DECOS model. Jobs communicate with each other by the exchange of messages via virtual networks.

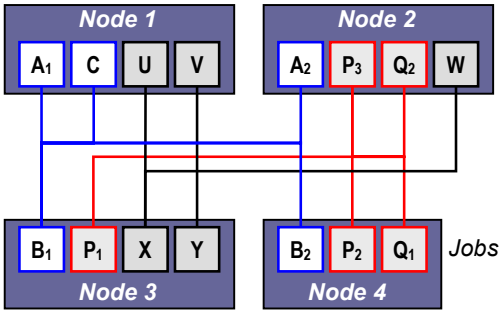

Fig. 1: DECOS cluster with four nodes and three DASs: one with jobs A, B, C, another one with P and Q, and a third one with U, V, W, X, and Y. A, B, and Q have two replicas each, while P has three.

Furthermore, a DECOS cluster is physically structured into a set of distributed node computers interconnected by a time-triggered network. Each node computer comprises several encapsulated partitions, which serve as the protected execution environment for jobs. During the development of a DECOS system, a mapping of jobs to partitions has to be established as indicated in Fig. 1. Due to the guaranteed non-interference of individual partitions, a DECOS node is able to host multiple jobs belonging to different DASs, even exhibiting different levels of criticality. Conceptually, all jobs could be allocated to a single processing node, as long as resource limits and hardware fault-tolerance do not require distribution.

Taking the achievements of research in the area of dependable system architectures into consideration, DECOS does not intend to design the complete system architecture from scratch. Instead, it presumes the existence of a *core architecture*, providing the *core services:*

- Deterministic and timely transport of messages.
- Fault tolerant clock synchronization.
- Strong fault isolation.
- Consistent diagnosis of failing nodes.

Any architecture that provides these core services can be used. It has been demonstrated that the Time-Triggered Architecture (TTA) [3] is appropriate for the implementation of applications in the highest criticality class in the aerospace domain according to RTCA DO-178B [4] and consequently meets the DECOS requirements as core architecture.

On top of the core services, DECOS provides a set of *architectural (or high-level) services:*

- Virtual Networks (VN) and Gateways.
- Encapsulated Execution Environment (EEE).
- Diagnostics.

VNs represent the communication system for DASs, embedded on the physical cluster network. Gateways provide means to exchange information between VNs, as well as with external networks, in a controlled way. The EEE is a partitioning real-time operating system that enables the execution of jobs from different DASs and of different criticality on the same hardware with guaranteed fault encapsulation. This is achieved by housing each job in its own partition with strong spatial and temporal protection [5]. Compared to other partitioning operating systems (e.g. ARINC653 LynxOS, AUTOSAR Tresos), the EEE is very small in terms of code size (< 1MB) and can run on comparably simple hardware [6]. The diagnostics service of DECOS both assists fault encapsulation and supports prognoses of hardware breakdowns.

Architectural services are implemented in a form influenced by the underlying (HW-)platform. In order to minimize dependency of application programmers on a certain implementation of these services, the *Platform Interface Layer* (PIL) provides a platform independent interface of the architectural services for application jobs.

An important DECOS feature is the support of both time- and event-triggered messages. Time-triggered (TT) messages transmit state values like the current speed periodically, while event-triggered (ET) messages transmit changes, e.g. the difference to the previous speed value whenever that change passes a certain threshold. So, while a transient transmission error of a TT message can be compensated with the next one, this is not the case

for ET messages. Therefore, the latter cannot be utilized for transmission of safety-critical information. TT messages are also denoted as *state messages*, and ET messages as *event messages*. Essentially, state messages realize the parallel computing concept of a conflict-free *distributed virtual shared memory*.

THE DECOS TOOL-CHAIN

An important activity in DECOS was the development of a dedicated tool-chain for supporting the development of certifiable DECOS applications by integrating several DASs in one cluster. As shown in Fig. 2, this tool-chain relies on a model-driven approach [7], aiming at the generation of configuration data as well as middleware and application code purely from models.

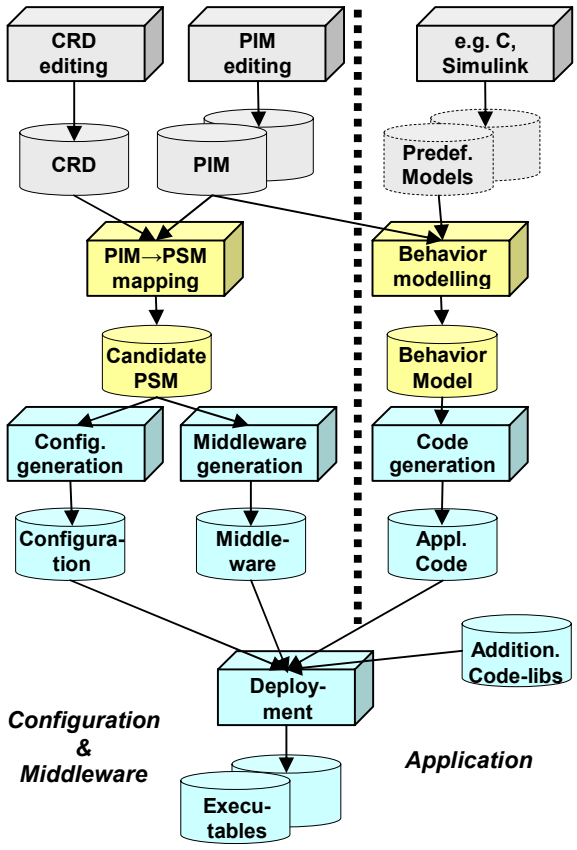

Fig. 2: DECOS tool-chain overview. Feedback loops, e.g. for failure reporting, are not shown.

First, the *Platform Independent Models* (PIMs) of the DASs are created, which serve two purposes. On the one side, together with the specification of the target cluster (Cluster Resource Description - CRD) and other information (job size etc.), they are used to derive configuration and scheduling information, as well as to generate the PIL, by transforming the PIMs into the *Platform Specific Model* (PSM). In Fig. 2, "Candidate PSM" is denoted rather than "PSM", because if scheduling fails, another allocation has to be chosen.

On the other side, PIMs are used to guide the development of jobs, by modeling their behavior, which is addressed in the section APPLICATION MODELING. Finally, the results of both activities are integrated to achieve the target executables, which can then be downloaded to the application cluster.

SYSTEM AND CONFIGURATION MODELING

This section addresses generation of relevant input for the PIM→PSM transformation process, namely the PIMs and the description of the cluster hardware and resources (CRD).

PIM AND ITS GENERATION

The purpose of the PIM [8] is to formalize the functional, dependability, and performance requirements of the DAS in an implementation platform independent manner. It is the place of the first steps of system architecture conceptualization. DECOS platform services - both at core and high-level - are handled in an abstract form that is easy to use and understand at this level of design.

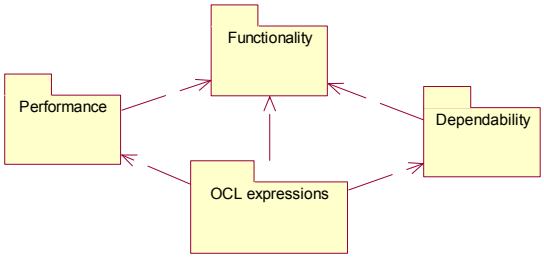

Fig. 3: PIM meta-model packages

According to the concepts of MDA [7] the PIM has its own meta-model. It is composed of three sub-models, each containing its respective object diagrams: (a) Functionality package: the basic functional elements like DAS, Job, Interface, Port, Message, Sensor/Actuator, State Variable, etc. (b) Dependability package: the dependability attributes of functional elements: reliability, availability, SIL, redundancy degree, etc. (c) Performance package: the performance attributes of functional elements: WCET, period, phase, deadline, latency, etc. Additional OCL expressions are used to specify semantic constraints: multiplicities, attribute constraints, etc.

Since the data model of the PIM, which is XML, is hard to edit even with XML editors, two solutions are provided for easing the generation of a PIM: (a) to use the same UML tool as for high-level system design. In this case, the DECOS-PIM-XML file is generated from the XMI output of the UML editor. Currently Rational Rose 2003 and Rational Software Modeler are supported. (b) to use a Domain Specific Editor (DSE) (see Fig. 4), which allows for creating only meta-model compliant PIMs. Such a DSE has been implemented under the Eclipse

technology[2]. It runs directly inside of VIATRA [9] which is the selected tool for PIM-PSM mapping. This way importing PIMs is not needed any more.

As indicated in Fig. 4, a PIM consists of (instances of) elements like Job, Interface, or Message, and specific associations among them, as in UML object diagrams. Furthermore, most elements and associations have specific attributes. In order to ease creation of PIMs, so-called "PIM design patterns" are provided. Inspired by (object-oriented software) design patterns, they allow adding dedicated sub-graphs to PIMs, guarded by a small number of parameters. For example, if a job shall be added to a PIM, only (a) its name, (b) its type (TT or ET), (c) whether it performs physical I/O, (d) whether it sends messages to other jobs and (e) whether it receives messages from other jobs has to be given. The sub-graph of new elements and required associations is then automatically generated and added to the PIM.

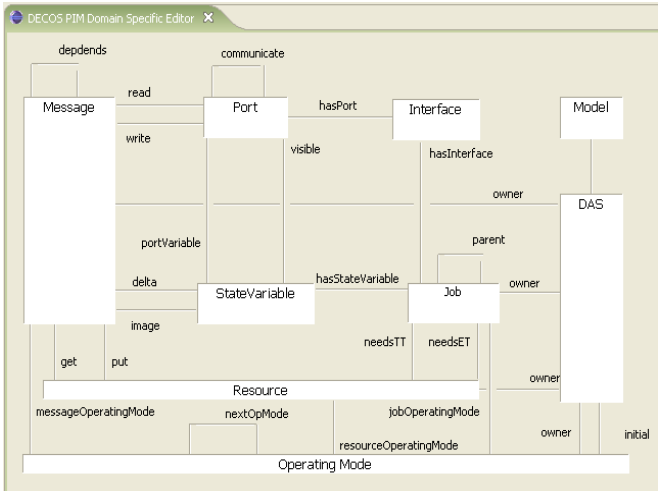

Fig. 4: DSE for the PIM under VIATRA

Before applying these patterns, some consistency checks are performed. For instance, a pattern will not be applied, if a name is given which is already used in the given PIM. Thus, besides easing the PIM capturing process, PIM patterns also help to avoid mistakes.

Finally, a service is provided (which can be activated directly from the PIM/DSE per simple mouse-click) which allows to check a PIM for semantic completeness (based on the predefined OCL constraints). The reason for this tool is that PIM/DSE does not allow for creating elements in conflict with the PIM meta-model like elements of undefined type or invalid associations. It cannot avoid, however, that elements are missing or required attributes undefined. Of course, when constructing a PIM under exclusive usage of the patterns, only a small number of attributes (e.g. execution periods of jobs or transmission periods of messages), for which no reasonable defaults exist, will have to be added manually.

[2] http://www.eclipse.org/

CRD AND ITS GENERATION

It is the purpose of the so-called *Cluster Resource Description (CRD)* to capture the relevant characteristics of the platform for the software-hardware integration in the DECOS design flow. These characteristics include amongst others computational resources (e.g., CPU and memory), communication resources, and dependability properties.

In order to ease CRD creation, a graphical, domain-specific modeling environment is developed, using the Generic Modeling Environment (GME). GME is a configurable framework for creating domain-specific modeling environments [10]. The configuration of GME is performed via the *Hardware Specification Model (HSM)*, a meta-model which formally describes the targeted modeling domain, i.e. it describes the entities, its attributes, the relationships, and constraints that can be expressed with and that are validated by the resulting modeling environment.

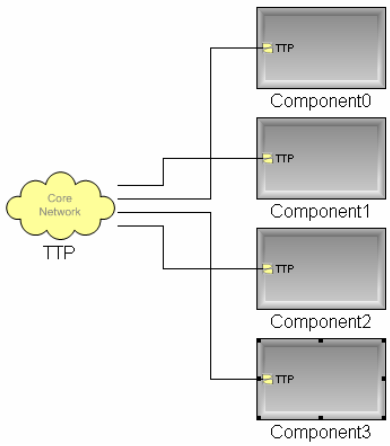

Fig. 5: Example (cut-out) of a CRD built with GME

One aim of the HSM is to facilitate reuse and hierarchical composition of the resource model. This is achieved by separating the resource modeling process into two phases: the resource capturing and the resource composition.

Resource Capturing: The specification of reusable hardware entities of a DECOS platform, so-called *resource primitives*, is addressed by the resource capturing phase. Resource primitives are the smallest physical hardware units whose characteristics are captured. Examples for resource primitives are: processors, memory elements, communication interfaces. However, the HSM provides mechanisms to extend the set of resource primitives that can be modeled in order to be flexible and extensible with respect to the types of resource primitives [11].

Resource Composition: This second phase of the modeling process is concerned with the composition of an entire CRD (including the internal setup of DECOS components and their interconnection) out of the pre-

viously modeled resource primitives. The composition is guided by the DECOS component model [2], which is fully captured by the HSM.

The interface to the subsequent tools in the development process realizing the software/hardware integration is specified using the extensible markup language (XML).

CONFIGURATION AND MIDDLEWARE

Based on PIM and CRD, the next steps involve the generation of configuration data and middleware:

- PSM-generation (mapping jobs to nodes).
- Scheduling and Fault-Tolerance Layer generation.
- PIL-generation.

PSM-GENERATION

The main purpose of the PSM (which is still a model) is to precisely specify which application jobs are to be assigned to which cluster nodes, under consideration of all constraints defined in the PIMs of the DASs and the available resources described in the CRD. Fig. 6 depicts a small part of its meta-model in UML-notation.

The PSM generation process encompasses a number of steps like PIM marking, feasibility checks, and the allocation process. The significant part of the mapping process is to allocate jobs with different criticality to a shared HW platform (HW nodes) subject to constraints and requirements of fault-tolerance and real-time.

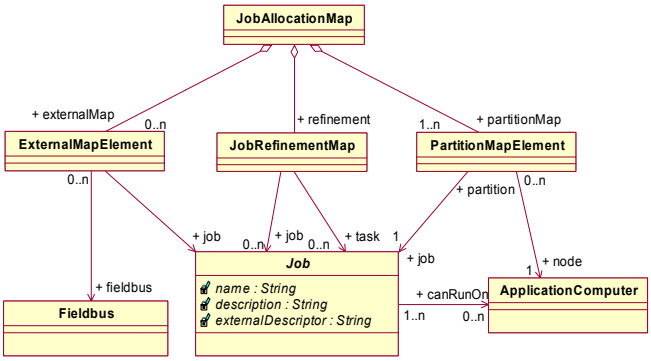

Fig. 6: Part of PSM meta-model

PIM marking is necessary for incorporating additional information to the PIMs that reflect designer decisions (hardware sensor/actuator allocation, job pre-allocation) and legacy information (job interface type, message protocol definition, etc.) The result is a *marked PIM* containing elements and associations reflecting the additional information.

Feasibility checks are executed both after the marking and after the allocation step to validate the (partial) models in order to achieve early detection of design problems. If a design constraint is violated, the designer can step back in the PIM/PSM mapping process and modify markings and/or extra-functional requirements to get a feasible system design.

The main (automatic) step is to assign jobs to nodes under the considerations of the functional and non-functional (i.e. performance and dependability) constraints given in the PIMs. Examples for such constraints are:

- Resource requirements (e.g. memory, CPU, sensors, actuators, bandwidth).
- Dependability constraints (e.g. replicas must be assigned to different nodes).

A dual-track approach is taken in DECOS to generate the PSM. First, a transformation based mapping process has been developed which deals constraints one-by-one. It finds a feasible solution for resource allocation while satisfying different constraints. A heuristics based systematic resource allocation approach has been explicated for this and presented in [12]. Considering dependability and real-time as prime drivers, we presented a schedulable allocation algorithm for the consolidated mapping of SC and non-SC applications onto a distributed platform.

Although the allocation problem is NP-hard [13], exploiting symmetry (job replicas, identical nodes, etc.) can improve the performance of such approaches [14], but it is difficult to assess solutions with respect to certain criteria like reliability maximization or cost minimization.

Therefore, in a second phase, a Multi-Variable Optimization (MVO) approach is implemented where multiple objectives are optimized together with satisfaction of constraints. Here, a so-called MVO function is used, which associates a scalar-valued function v(q) to each point q in an evaluation space, representing the system designer's preferences, provided that choosing a feasible alternative from a set of contenders such that v is maximized or minimized.

See [15] for more information about the generation of the PSM.

SCHEDULING AND FAULT-TOLERANCE

As mentioned previously, the DECOS Integrated Architecture consists of several nodes that communicate via a time-triggered physical network (cf. Fig. 1). DECOS does not make further assumptions about the specific time-triggered protocol that has to be used. Within the project, TTP as well as FlexRay and TT-Ethernet have been successfully used for the core network.

Virtual networks (VN) are built upon this time-triggered physical network, implying that all information transfer takes place via messages of the underlying time-triggered physical network. Thus, each DECOS node must be able to send/receive messages (cf. Fig. 7). Furthermore, DECOS nodes that share the same communication medium are required to coordinate the trans-

mission of messages, i.e., at each point in time, exactly one node is allowed to send a message.

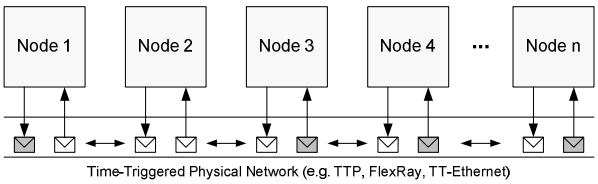

Fig. 7: DECOS nodes communicating via a time-triggered physical network

In addition to the scheduling of messages, partition- and task-scheduling is required. The scheduling of partitions generates a proper arrangement of DAS jobs that are assigned to EEE partitions (see CONCEPTS and ARCHITECTURE). Task scheduling is concerned with arranging tasks within each single partition (DAS jobs can consist of several tasks).

For that purpose, the TTTech tool suite[3] has been adapted to cope with partitions, and has been integrated into the DECOS tool chain.

The TTTech tool suite provides a graphical front-end for the specification of partitions as well as a programming interface that is used for importing the PSM. From that, it computes valid schedules and generates configuration data structures for the communication controller and for the hardware implemented fault-tolerance layer, as well as optimized fault-tolerance layer tasks, and the respective configuration files for the operating system configuration including the protection parameters of the partitions.

The hardware supported *fault-tolerance layer* (FTL) of the DECOS project comprises all functionality required for fault-detection and fault encapsulation. It consists of software (the generated fault-tolerance tasks) and hardware (implemented on a FPGA) part. High-level operations such as comparing and voting on message replicas (if a sending job is replicated, recipients receive the same message from all replicas) are handled in software whereas the hardware part takes over low-level operations like message (un)packing, comparing content received on redundant channels, setting frame status and byte ordering. Experimental evaluation of the DECOS FTL recently showed that “built-in error detection and recovery mechanisms including different RDA functions are able to detect, mask or recover from errors both internal in a redundant node or on a replicated communication network” [16].

PIL GENERATION

As already mentioned, the *Platform Interface Layer* (PIL) provides a technology invariant interface to the DECOS architectural services for application jobs. Following "native" services are offered by PIL:

- generic message transfer (TT and ET),
- global time service, and
- membership service (to get information about health states of nodes and jobs).

In addition, the usage of domain-specific application *middleware* like for CAN-support is possible.

Since C is still the most common programming language for embedded systems, including generated code like that from SCADE (see next section), a C-binding for the PIL API is provided by default. One means to improve safety in a C-environment is to make intensive use of types and names, and to avoid the usage of error prone types like void* or char* for parameters and return values. Hence, for each job/message combination an own set of functions is provided, forcing to generate the PIL individually for each job.

For instance, if a job X receives a state message S of type t_S and may send event messages E of type t_E, then essentially the following C-API will be generated for it:

```
PIL_RetCode PIL_get_S(
   t_S *out_S,
   UINT16 *out_validity,
   const PIL_WaitMode in_wait);

PIL_RetCode PIL_queue_E(
   const t_E * in_E,
   PIL_WaitMode in_wait);
```

"out_validity" returns the number of message replicas used to yield out_S; if 0, out_S is invalid, or outdated, respectively. "in_wait" can be used for controlling whether the call shall return immediately if access to the internal buffer for S is blocked.

Some more functions are generated according to the same principle, e.g. for providing control on data access.

In order to achieve platform and programming language independency, PIL generation conceptually *binds* the PIL to the target environment. This is achieved, for instance, by using coding templates for the respective platform/ language pair. This leads to a *bound PIL* tailored to interface platform and applications.

APPLICATION MODELING

In addition to modeling system structure, configuration, and scheduling as described so far, specification of behavior is another issue. Though using C-code for that purpose is of course possible, in safety-critical environments safer approaches are highly recommended. Therefore, SCADE [17] has been chosen to be the primary DECOS tool for behavior modeling and development, which is described in this section.

[3] http://www.tttech.com/products/software.htm

SCADE

Based on a formally defined data-flow notation [18], SCADE offers *strong typing*, *explicit initialization*, *explicit time management* (delays, clocks, etc), and *simple expression of concurrency* (data dependencies). By means of a graphical data flow graph editor, it supports model-based development. This not only allows for simulation at model level, accompanied by dedicated testing [19] or formal proof techniques by the SCADE Design Verifier [20] to prove safety properties, it also enables qualified code generation, using the KCG code generator. KCG has been certified against DO178B level A [4] and IEC 61508 at any SIL level.

The basic SCADE modeling elements are predefined *operators* and user-definable *nodes*. Both have input and output parameters, through which they are connected with other nodes and operators (see Fig. 8). Of course nodes can be nested. Fairly obviously, nodes will be used to represent DAS jobs.

To assure that "job nodes" adhere to their interface definition in PIM, SCADE's UML gateway is used to import PIMs into SCADE. And to enable usage of behavior models developed in Simulink[4], another SCADE gateway can be used to import Simulink models into such nodes (see clause "Simulink Import" below). In the following, these gateways are addressed.

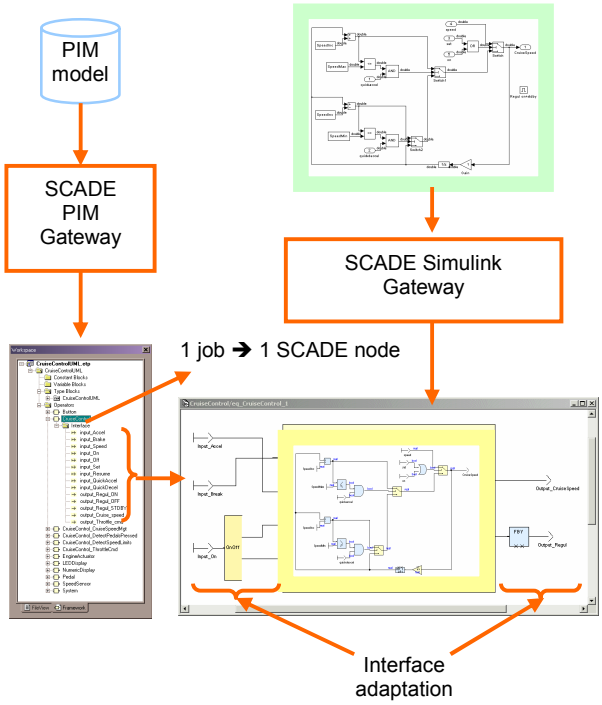

Fig. 8: Importing PIM and Simulink models into SCADE

PIM-IMPORT

The SCADE UML gateway is a flexible tool for import and reuse of software architectures specified in other modeling languages as SCADE node skeletons. It allows for easy addition of extension modules to support any modeling language that has similar architectural concepts as UML: Architectures consist of functional blocks, with one or several levels of hierarchical decomposition; and blocks are connected at specific interaction points, with corresponding communication protocols defined.

The SCADE UML gateway is being extended to support PIM-based modeling by adding a new module specifically tailored for DECOS PIM, which realizes the following mapping:

- Each DAS represents a namespace for its contained jobs.
- Jobs of a DAS are the architectural blocks of the DAS.
- Ports (grouped in interfaces) of a job are the interaction points of the corresponding block.
- Event-triggered and time-triggered messages form the protocols at the interaction points.

SIMULINK-IMPORT

If the behavior of the DAS jobs is originally modeled in Simulink, it has to be imported into SCADE, to "fill the contents" of the nodes created by the PIM to SCADE gateway. The DAS jobs behavior can be defined in several Simulink models, and/or in several parts of a Simulink model. The current version of the Simulink translator only allows one Simulink import (one part of one Simulink model), so it is upgraded to support this requirement. When a modification of a Simulink job model is made, thanks to this "modular translation" feature, only the respective part needs to be re-imported.

Since re-translation may introduce incoherencies in the SCADE model, semantic inconsistencies (on names, types, or used clock units) are automatically checked by the SCADE checker.

NATIVE BEHAVIOR MODELING

Instead of importing Simulink models, job behavior can, of course, also be modeled in SCADE directly. Besides a rich set of basic operators (arithmetic, boolean, set, temporal etc.), SCADE offers a number of libraries with predefined nodes, which can be easily extended. DECOS takes advantage of this feature by providing an own library of DECOS specific nodes; for instance for fault-tolerant treatment of sensor input.

[4] http://www.mathworks.com/

INTERFACE ADAPTATION

The input/output interface of a job at PIM-level is made of a set of state messages and/or event messages carrying information about various state variables together with validity flags provided by the platform. (PIL's validity number is mapped to boolean, with 0 mapped to false and any positive value mapped to true). A single message can carry information about an arbitrary number of state variables and is therefore represented as a structured value with many fields, the first field being the validity flag for the whole message.

Various application-level strategies exist for handling fault tolerance. The simplest one is to do the actual computation with the latest valid values of the state variables as extracted from input messages and then propagate the validity flags to dependent output messages.

Usually, the first step during behavior computation is to check the validity flags of messages and extract the values of the various state variables that they carry. Then the actual computation is performed, often through an auxiliary node coming from an import from Simulink. The PIM gateway automates parts of this strategy, as follows:

In the SCADE model, the state variables (and possibly also the event flags) as extracted from the I/O messages must be connected to the right inputs and outputs of the auxiliary node doing the computation. This task is tedious and error prone, which is the reason why the PIM gateway does the state variable extraction automatically, by creating corresponding local variables in the SCADE job. If the names happen to be consistent in the PIM versus in the algorithm node, a simple “connect by name” command in the SCADE editor will be enough to complete interface adaptation in just one click.

Besides state variable extraction, the other aspect of interface adaptation is type conversion: Indeed, the state-variables are typed by PIM-level types, which are opaque from a SCADE point of view (they are “imported types” in SCADE terminology), while the computation node itself is normally using various SCADE primitive types. Values of state variables must therefore be converted, mapping each opaque PIM-level type onto the most appropriate SCADE primitive type. The mapping is expressed as SCADE-dedicated PIM annotations. For each PIM types, the annotations provide its name as a SCADE imported type, the name of the most appropriate SCADE primitive type on which it should be mapped, and the name of the two conversion operators (from PIM type to SCADE type and vice-versa). Most of the time, the conversion operators can be implemented by a simple cast at C-level. The PIM gateway uses this information to automatically insert type conversions between the values as extracted from input/output messages and the values as manipulated by the algorithm node.

Extraction of state variable information and type conversions are the two facets of interface adaptation. Most of this step is automated already, relieving the user of some tedious manual work, while limiting the risk of introducing errors.

TESTING

A particular strength of SCADE is its provision of powerful tools for simulation and testing. In addition to an intrinsic simulation feature with a broad variety of control possibilities (single step, time range, etc.), it offers a set of dedicated tools like SCADE MTC for evaluating test coverage or model checking by means of DESIGN VERIFIER™ by Prover Technology.

In DECOS, two further tools are under development for testing SCADE models. One allows for checking the correct use of physical units and dimensions in SCADE's data-flow models [21], the other serves for simulating complete DASs based on their PIMs. This can be used to quickly evaluate whether, e.g., temporal data like periods and phases of job executions or message transfers are sufficiently complete, or to examine whether the integration of modeled job functionality into a DAS does not provoke unintended emergent system behavior. It also allows simulating effects of faults like failing message transfers at application level.

CODE GENERATION

As already mentioned, KCG is used to generate the code from each individual SCADE node job. To get the code of the complete distributed application, the job code must be linked to the middleware code via the PI API.

Since SCADE nodes do not access DECOS services via the PI API directly, but instead work on ‘context objects’ which contain their input and output parameters, they have to be embedded into so-called *wrappers*, written in C, which at each execution of the job

- fetch received messages and put them into the input fields of context objects,
- activate the node code,
- forward output fields of context objects to PI API as send messages.

Since node code may not always produce requested outputs, a specific flag mechanism is implemented to inform the wrapper which messages to send.

At the moment, the wrappers do not handle type conversions, which are to be done in the SCADE model itself using imported type conversion operators, as explained previously. Pushing type conversions into the wrapper code would simplify the SCADE model and remove the need for imported types and conversion operators in the SCADE model. This improvement is currently under study.

DEPLOYMENT

As the final step in the tool chain, all parts (application code, either generated from SCADE or written manually, generated middleware and configuration data) are put together into executables for the target platform. For the primary DECOS platform (EEE on TriCore TC1796) this is a single file per node which can be loaded into the flash memory of the node.

Typical DECOS systems are built from parts contributed by/bought from different vendors and put together by the system integrator, e.g. a car manufacturer. While components developed by the integrator itself are usually available in source code, third party vendors in general only provide the subsystems PIM, C header files and precompiled object files or libraries, together with documentation for the integrator. To keep these parts organized, a standard project directory structure is defined (see Fig. 9 for three DASs and a cluster with four nodes).

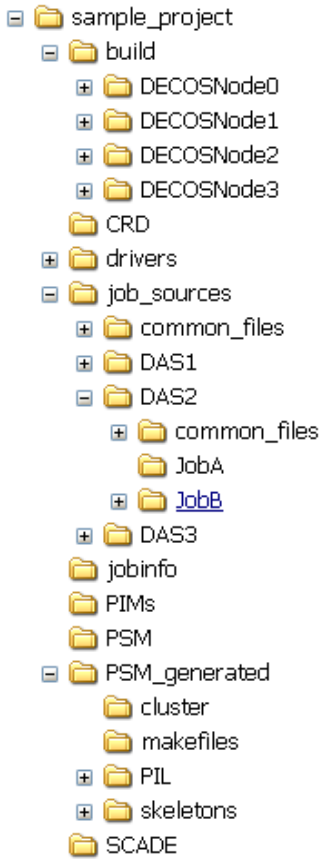

Fig. 9: DECOS deployment directory hierarchy

The EEE build environment expects all source code in one single build directory per node. Therefore, using a global (static) deployment control makefile, augmented via a sub-makefile generated from PSM, with respect to the specific allocation, only the necessary input files are put into the build directories. Subsequently, the compiling and linking steps for EEE applications are done for each node and produce binary files to upload into the nodes.

TOOL-CHAIN INTEGRATION

ASSEMBLING THE TOOL-CHAIN

As shown in the previous sections, a rather wide variety of tools are involved in the DECOS tool-chain. In order to ease the handling of all these tools, VIATRA [9] is used as "backbone" for PIM capturing and PIM→PSM transformation. It not only allows for developing model transformation conveniently, it is also possible to generate code with it, e.g. PIL, as well as to develop domain-specific editors, which e.g. ease PIM creation. So, basically four tools constitute the DECOS tool-chain – GME, VIATRA, SCADE, and the TTP/TTX-tools, as indicated in Fig. 10. The interchange formats among these tools is also shown in Fig. 10. Boxes denote interactive activities, rectangles denote automated steps.

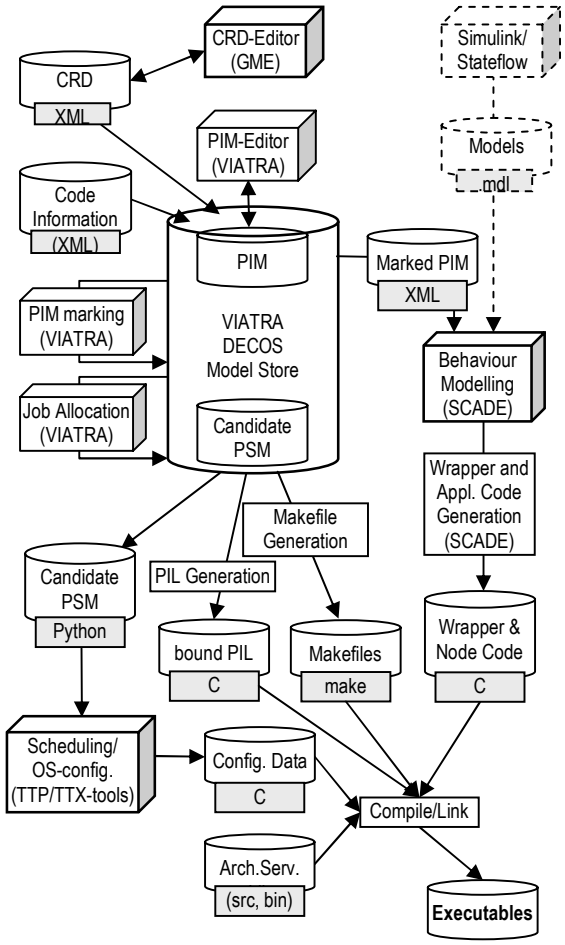

Fig. 10: DECOS tool-chain: involved tools and interchange formats

USING THE TOOL-CHAIN

As stated above, the tool chain is used by both the system integrator and the subsystem developer/vendor. While the subsystem vendor has to perform the same steps as the system integrator to test the built subsystem, the main development steps done by the vendor using the tool chain are:

1. With PIM DSE in VIATRA: Define PIM.
2. With SCADE:
   a) Import PIM.
   b) Fill job skeletons with behavior (either import from Simulink or model in SCADE).
   c) Validate models.
   d) Generate code.
3. Manually: add I/O code.
4. (With build environment: build object code/libraries).

The system integrator will mainly use GME, the PIM-PSM-mapping editor, the TTP/TTX tools, and the command line build system (probably facilitating gnumake, and for the DECOS primary platform, the TriCore 1796 by Infineon, Altiums tasking compiler and linker. The system integrator will basically perform the following activities:

1. With GME: build/adapt the CRD for the target cluster
2. With Eclipse + Viatra + PIM-PSM-mapping editor:
   a) Build a new eclipse project (with DECOS project wizard) and import pre-existing files (CRD, PIMs).
   b) Create a new PSM.
   c) Map PIM datatypes to platform datatypes.
   d) Define interface types (e.g. virtual CAN API for legacy CAN applications).
   e) Define job type (e.g. for jobs running on other nodes connected via CAN).
   f) Attach non-DECOS jobs to physical fieldbus interfaces.
   g) Connect I/O-jobs to I/O resources (sensors/actuators).
   h) Connect gateways for inter-DAS communications.
   i) Manually restrict the possible allocations, if required.
   j) Run automatic allocation.
   k) Run TTP/TTX-tools scheduler input file generation.
   l) Run PIL code generation and makefile generation.
   m) Put job sources and library object files into their respective places in the project.
3. From build command line: running make automatically takes care of the following steps:
   a) Import schedule input data into DECOS version of TTP/TTX-tools and create schedule as well as configuration data for EEE.
   b) Copy middleware (PIL, FTL) and application source code as well as prebuilt libraries and drivers to the node build directories according to allocation.
   c) Compile and link everything together into the binary file for each node, together with control files for uploading them with the debugger.

CONCLUSION

The paper presents a tool-chain for the design, modeling, development, testing and deployment of integrated embedded applications of mixed criticality. The existence of such a tool-chain is an important prerequisite for the migration from federated to integrated distributed embedded systems as targeted within the DECOS project.

In this paper we presented a strict model-driven approach by using models for all design and development phases and steps, from which all required source code – application, middleware, and system architecture configuration – is generated. It is presumably the first time that for (dependable) embedded, distributed systems such a purely model-based approach has been realized.

REFERENCES

1. Herzner, W., Schlager, M., LeSergent, T., Huber, B., Islam, S., Suri, N., Balogh, A. “From Model-Based Design to Deployment of Integrated, Embedded, Real-Time Systems: The DECOS Tool-Chain” Proc. (“Tagungsband”) of Microelectronics Conference ME’06, 11.-12.10.2006, Vienna/A)
2. Kopetz, H., Obermaisser, R., Peti, P., Suri, N. "From a Federated to an Integrated Architecture for Dependable Real-Time Embedded Systems." Technical report 22/2004, TU Vienna, July 2004.
3. Kopetz, H. and Bauer, G. (2003). "The Time-Triggered Architecture." *IEEE Special Issue on Modeling and Design of Embedded Software*
4. RTCA (1992) DO-178B: "Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics." Inc. (RTCA), Washington, DC.
5. Schlager, M., Herzner, W., Wolf, A., Gründonner, O., Rosenblattl, M., Erkinger, E. "Encapsulating Application Subsystems Using the DECOS Core OS." Proc. of SAFECOMP'06 (26.-29.9.2006, Gdansk/P), 386-397, Springer LNCS, vol. 4166, Springer, 2006
6. Leiner, B., Schlager, M., Obermaisser, R., Huber, B. „A Comparison of Partitioning Operating Systems for Integrated Systems.” Accepted for publication at SAFECOMP’07 (18.-21.9.2007, Nuremberg/G)
7. OMG. "Model driven architecture, A technical perspective." Technical report, OMG Document No. ab/2001-02-04, Object Management Group.
8. Pataricza, A. "Report about decision on meta model and tools for PIM specification." DECOS deliverable D 1.1.1, Dec 2004.

9. Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., and Varro, D. "VIATRA: Visual automated transformations for formal verification and validation of UML models." Proc. of the 17th IEEE Int. Conf. on Automated Software Engineering (ASE 2002), 267–270, IEEE (2002)
10. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garret, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P. "The Generic Modeling Environment." Proc. of WISP, Budapest Hungary, May 2001
11. Huber, B., Obermaisser, R., and Peti, P. "MDA-Based Development in the DECOS Integrated Architecture – Modeling the Hardware Platform." Proc. of the 9th IEEE Int. Symp. on Object and Component-Oriented Real-Time Distribued Computing (ISORC), 2006.
12. Islam, S., Lindström, R., and Suri, N. “Dependability Driven Integration of Mixed Criticality SW Components." Proc. of the 9th IEEE International Symposium on Object and Component-oriented Real-time distributed Computing (ISORC), 485-495, 2006.
13. Fernandez-Baca, D. "Allocating Modules to Processors in a Distributed System." *IEEE Trans. on Softw. Eng.*, 15(11), 1427–1436, 1989
14. Weißenbacher, G., Herzner, W., Althammer, E. "Allocation of Dependable Software Modules under Consideration of Replicas." ERCIM Workshop on Dependable Software Intensive Embedded Systems, Porto, Portugal. Sep.2005
15. Islam, S., Csertan, G., Herzner, W., LeSergent, T., Pataricza, A., and Suri, N. "A SW-HW Integration Process for the Generation of Platform Specific Models." Proc. of ME´06, 194-203, ÖVE Schriftenreihe Nr.43, Oct. 2006
16. Vinter, J., Eriksson, H., Ademaj, A., Leiner, B., Schlager, M. „Experimental Evaluation of the DECOS Fault-Tolerant Communication Layer“ Accepted for publication at SAFECOMP’07 (18.-21.9.2007, Nuremberg/G)
17. SCADE Suite Technical and User Manuals, Version 5.0.1, June 2005, Esterel Technologies
18. Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. "The Synchronous Dataflow Programming Language Lustre." Proc. of the IEEE, 79(9), 1305–1320, September 1991.
19. Dion, B., Gartner, J. "Efficient Development of Embedded Automotive Software with IEC 61508 Objectives using SCADE Drive." Proc. of VDI Conf. of Electronic Systems for Vehicles, Baden-Baden, Oct. 2005
20. Bouali, A., Dion, B., and Konishi, K. "Using Formal Verification in Real-Time Embedded Software Development." Proc. of Japan SAE, Yokohama, 2005
21. Schlick, R., Herzner, W., Le Sergent, T. "Checking SCADE Models for Correct Usage of Physical Units." Proc. of SAFECOMP'06, Sep. 2006, Gdansk/P, 358-371; Springer LNCS, vol. 4166, Springer, 2006

CONTACT

Wolfgang Herzner: wolfgang.herzner@arcs.ac.at
Rupert Schlick: rupert.schlick@arcs.ac.at

Martin Schlager: martin.schlager@tttech.com
Bernhard Leiner: bernhard.leiner@tttech.com

Bernhard Huber: huberb@vmars.tuwien.ac.at

Andras Balogh: abalogh@mit.bme.hu
György Csertan: csertan@mit.bme.hu

Alain Le Guennec: alain.leguennec@esterel-technologies.com
Thierry Le Sergent: Thierry.LeSergent@esterel-technologies.com

Neeraj Suri: Suri@Informatik.tu-darmstadt.de
Shariful Islam: ripon@deeds.informatik.tu-darmstadt.de

ABBREVIATIONS

API	Application Programming Interface
CAN	Controller Area Network
CRD	Cluster Resource Description
DAS	Distributed Application Subsystem
DSE	Domain-Specific Editor
EEE	Encapsulated Execution Environment
ET	Event-Triggered
FTL	Fault-Tolerance Layer
GME	Generic Modeling Environment
HSM	Hardware Specification Model
KCG	Qualified Code Generator
MDA	Model-Driven Architecture
MTC	Model Test Coverage
NP	Non-deterministic Polynomial-time
OCL	Object Constraint Language
PIL	Platform Interface Layer
PIM	Platform Independent Model
PSM	Platform Specific Model
SCADE	Safety-Critical Application Development Environment
TT	Time-Triggered
TTP	TT Protocol
UML	Unified Modeling Language
VIATRA	VIsual Automated TRAnsformations
VN	Virtual Network
WCET	Worst-Case Execution Time