MULTILANGUAGE DESIGN

Bernd Kleinjohann[1]

1. Introduction

During the last years the production costs for IT systems have steadily decreased with regard to their complexity. IT applications that had to be realized as expensive PCBs in earlier times can now be realized on a single chip, which is, of course, much cheaper. Furthermore, low cost broad band communication media are available. Typically, the market requires IT systems that realize a set of specific features for the end user in a given environment, so called embedded systems. Some examples for such embedded systems are interfaces of mobile phones, or more generally end user interfaces for communication products, set top boxes for digital TV, or control modules in cars, airplanes or buildings. For these different application domains a variety of modeling languages with domain specific features was developed in the past. C/C++ / VHDL , for instance, as used in hardware/software codesign, do not fulfil the designers requirements for modeling of embedded systems. In order to cover the entire functionality of an embedded system, subsystems have to be specified in different languages that support the modeling of application domain specific features. As an example the control system of a car may serve: The body electronic is usually specified using discrete state based languages, whereas for the specification of control subsystems for window lifts, power train, or ABS continuous data flow based languages are used. At higher levels the car itself may be seen as a system that is embedded in a network of telematic services, that will be specified in typical languages of the telecommunication domain.

In this tutorial we will present how different languages can be used together during design of embedded systems. Figure 1 describes this in principle. Starting with a specification in several language (A, B), an integration model has to be defined that allows to realize a design process that generates an implementation for the specified modules consisting of some realization parts (V, W, X, Y) and their communication. Section 2 will summarize current practice in design of embedded systems and investigate some problems arising during such multilanguage design processes. Section 3 will especially encounter semantic problems at the interfaces of system parts specified

[1] C-LAB, Fürstenalle 11, 33094 Paderborn, Germany, Email: bernd@c-lab.de

F. J. Rammig (ed.), *Distributed and Parallel Embedded Systems*

in different languages. In Sections 4 and 5 we will introduce the concepts of *language coupling* and *language integration*, that are based on two different types of integration models. For *language coupling* mechanisms for the connection of inputs and outputs of subsystems modelled in different languages have to be provided without changing the original subsystem models. For *language integration* models specified in different languages have to be transformed into a common model.

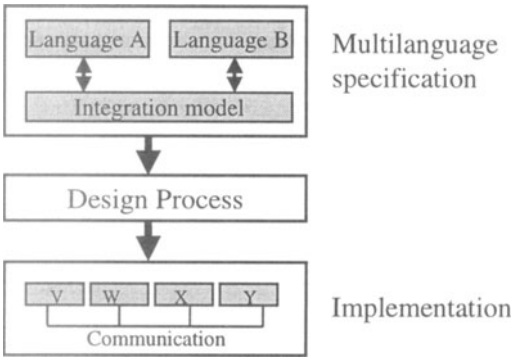

Figure 1: Multilanguage design

2. The Design Process

The design process has to consider three independent aspects: *applications, architectures*, and *design methods*. The first aspect, we will consider, are the *applications*. They usually rise various requirements and constraints, typical for the specific application domain. Embedded systems often have to fulfil a specific task in a dedicated environment like the above mentioned control system of a car or a mobile phone. Typically an embedded system has to fulfil real time constraints. Here we distinguish soft real time requirements, for instance for the transmission of audio and video data, and hard real time requirements that apply for vehicle control systems, for instance. Furthermore, many applications are extremely security critical, but nevertheless obey a strong cost pressure. Non predictable reactions of such applications cannot be tolerated due to serious damage they may cause.As an example how to cope with these conflicting requirements, the X-by-wire technology may serve, that will be introduced to mass products like cars.

The second aspect in the design of embedded systems is the *target architecture* typically consisting of a software and a hardware part. The hardware architecture, for instance in automotive applications, is usually given by a set of electronic control units (ECU) connected by a communication module (see Figure 2). In this domain the topology of the hardware architecture is often determined by the mechanical construction, having in mind that the cable tree in a vehicle is a main cost factor. In contrast, for private TV sets with video camera and stereo equipment, the topology may

change during their run time, if for instance the portable video camera is (dis)connected to (from) the equipment.

Each node (ECU, see Figure 2) in the often predetermined topology may contain several computing kernels, memory units, and application specific periphery. The target hardware for the computing kernels (micro controller, digital signal processor, programmable gate array, or application specific hardware) has to be determined for each node. Furthermore, size and type of the memory units (RAM, ROM, FLASH) and the set of periphery components (e. g. AD-/DA transmitters, serial and parallel ports, or communication interfaces) have to be defined. These realization decisions have to be taken for each node considering its performance and cost requirements. Whenever possible, existing architecture families, that can be parameterized appropriately, should be chosen for cost reduction reasons.

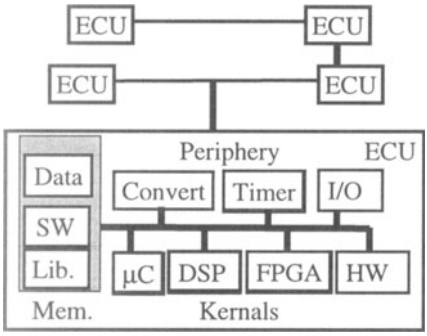

Figure 2: Hardware architecture

The dimensioning of the memory elements has to be done with regard to the expected data traffic and software size. The software to be stored consists of two parts: the *basic services* that have to be configured for the application and the actual *application software*. The *basic services* usually comprise operating system and communication functions as well as interface functions for interaction with the periphery, application specific hardware, or hardware programmable components (FPGA). For real time applications standard operating system interfaces like OSEK (OSEK, 1997) are available. However, recent trends show that even such basic services are realized for a specific application (degradable operating system). The actual *application software* is given by those functions realizing the top level specification, usually provided in different languages, with its constraints.

Figure 3 represents the principle *design method* for embedded systems. The basis for the depicted design steps is an initial specification of the system and its constraints. Embedded systems and thus also their specifications contain various types of functions or tasks. Discrete control tasks are usually specified using state based notations, whereas for the specification of analogue control tasks differential equations or in the discretized case data flow notations are frequently used. The structure of such a multiparadigm specification should follow the functional grouping of system tasks, in order to support an independent development of different system parts

The initial specification has to be split into independently executable tasks. Then, the run time and performance requirements for these tasks are estimated and the tasks

are allocated to the specific nodes in the hardware architecture considering the specified constraints. Afterwards the scheduling is done in order to estimate whether the timing constraints can be fulfilled. In the subsequent step the tasks allocated to the various nodes have to be realized. If a given controller (selected from a set of existing hardware controllers) shall be used, the software for this controller has to be generated. Otherwise, classical hardware/software codesign methods can be used. In this case, it has to be decided which parts of the task are implemented in software and which ones in hardware. Afterwards the required hardware and software have to be realized.

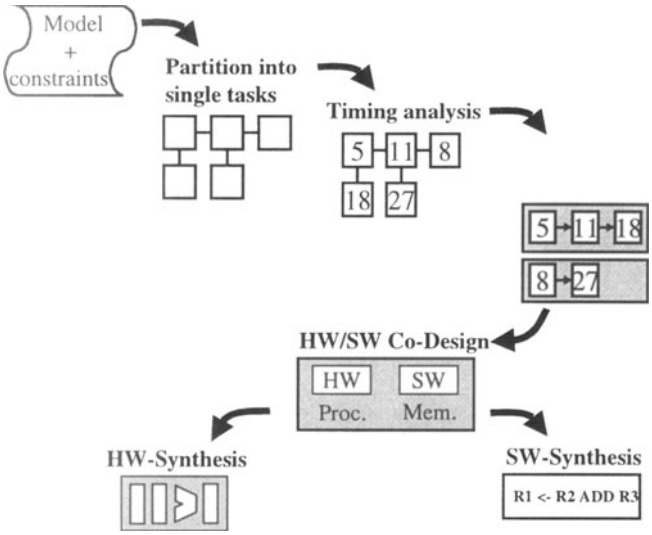

Figure 3: Simplified design flow for embedded systems

The above design process has been presented in a rather simplified manner. Usually many iteration steps are necessary to reach a satisfying result. For multilanguage design, however the specification phase plays a crucial role, since it influences the entire design process, especially if the specification consists of different languages. Only by the integration of different specification languages a well defined basis for subsequent design steps can be reached. In order to preserve existing libraries and know how in industrial design processes, also the integration of specification languages following the same paradigm is reasonable. Therefore, we will concentrate on this phase and its impact in the rest of this tutorial.

3. The Semantics of Systems

This section will investigate the problem of interfacing different system specifications in order to have a serious basis for the integration of languages. The role of interfaces and their semantics for system specification and realization will be explained.

The specification of a system part in a specific language can be regarded as the specification of an open subsystem that communicates with its environment. Even if only the behavior of the system part under design is explicitly specified, the expected behavior of the environment is at least implicitly given.

The observable behavior at an interface between a system part and its environment is indirectly given by their specifications (system part and environment). For the specification of interfaces at an open system part (no explicit environment specification is given) different notations are in use. Interfaces of continuous systems can be represented by trajectories, i. e. a curve of vectors from a continuous domain over the continuous time axis. Discrete interfaces can be characterized by streams or traces (van de Snepsheut, 1985).

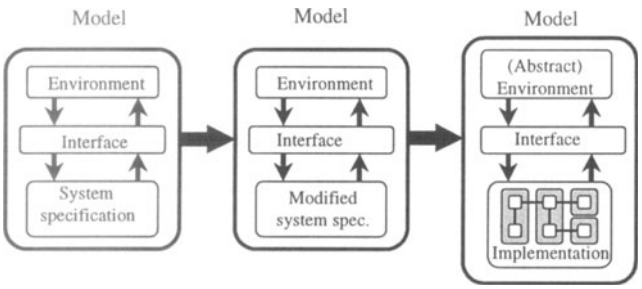

Figure 4: Interfaces between subsystems

In this context the goal of the design process can be characterized as depicted in Figure 4: The specification of a (sub)system to be realized should be transformed in such a way that it consists merely of models of easily realizable or already realized system components. This process is usually carried out in multiple steps as already mentioned in Section 2. During this process also the original environment model may be substituted by a more abstract model in order to simplify it (see below). For each step it is expected that the interface between the (sub)system and its environment remains the same or is in a well defined relation to the original interface definition (see Figure 5).

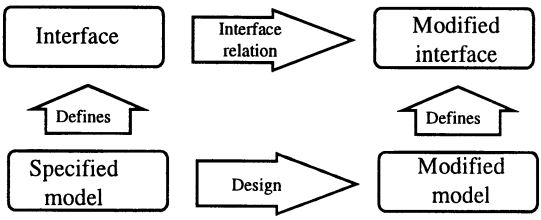

Figure 5: Specification and interface relation

Usually it is expected that the diagram presented in Figure 5 commutes. However, for continuous interfaces described by trajectories tolerable deviations from the origi-

nal one may be defined. Consequently, all realizations whose trajectories are within the allowable deviation range are correct realizations.

For systems with discrete interfaces the realization is expected to show exactly the originally specified interface behavior. However, this expectation leads to surprising results, since several specification languages like, for instance, Statecharts contain inherently nondeterministic behavior. By nondeterministic behavior we mean that a possible simulation run is not exactly predetermined by the specification.

In practice this inherent nondeterminism is resolved by the simulation tools in a specific design environment, or it is at least restricted by so called 'don't care' values. Thus, depending on the applied simulation tool different realizations may be postulated for the same specification. If a system realization should behave exactly as predicted by a simulation run, the simulation semantics must at least partially be considered. In cases where such a simulator and the real time operating system of the target hardware follow different strategies, the envisioned realization is partly impossible.

An alternative to the postulation for the interface relation stated above (diagram in Figure 5 commutes) is the following one.

If two possible simulation runs distinguish at a certain position for the first time, one of the following conditions must hold:

1. *If the according value is an* input *(of the system to be realized), then the system to be realized must be able to continue its work with both values.* In this case the environment model shows nondeterministic behavior. By the use of nondeterminism the environment model may be simplified. In these cases the environment model is an abstraction of the real environment and may show behavior that cannot occur in reality. Vice versa, it may happen that the environment model does not completely specify the behavior of the real environment. For these omitted environment behavior the realized system will show no reaction or incorrect reactions.
2. *If the simulation runs distinguish first in an* output *value, only one alternative reaction must be implemented by the system to be realized.* In this case the system to be realized shows nondeterministic behavior, that can be used to optimize the system realization, since only one alternative reaction must be realized. The knowledge about this nondeterministic behavior is also needed for defining an integration semantics. However, classical simulators do not suffice for detecting such cases, but a tool would be needed that can navigate through the design space characterized by that nondeterminism.

Having in mind the principle problems that may arise at the interfaces of system specifications, even if only one language is used, we will now deal with the composition of multiple languages and introduce the concepts of language coupling and language integration.

4. Language Coupling

One way of composing several specification languages is *language coupling*. For language coupling the interfaces of subsystems specified in different languages have to be connected. This can be reached by explicitly describing the connections between the subsystems with a corresponding tool or implicitly by naming conventions for

identifiers (using the same identifiers in different subsystem specifications). The subsystem models are not changed at all and follow the paradigms of those languages used for their specification. Hence, assertions regarding the entire system are restricted to the subsystem interfaces. The design process works independently on the model parts as depicted in Figure 6.

The following example shows some problems that may arise during language coupling. In this example Statecharts and discretized differential equations will be coupled.

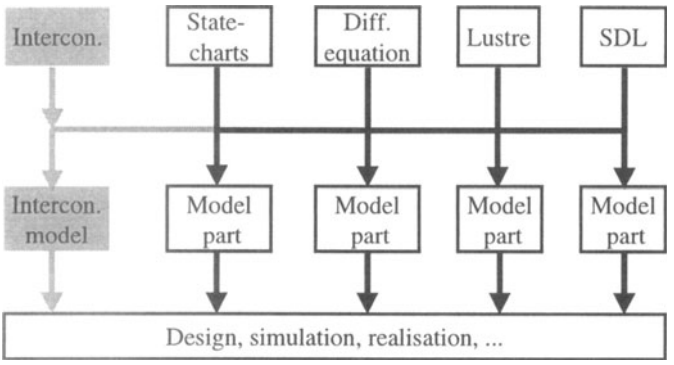

Figure 6: Language coupling

Statecharts (Harel, 1977) are a state based formalism. Figure 7 shows an example modelling a remote TV controller. Hierarchical and orthogonal states are supported. The state *TV* has the parallel substates *Standby* and *On*. *On* is divided into two orthogonal states separated by a dashed line, which again have two substates each. A broadcast mechanism is used for inter state communication. State transitions can be annotated with events (*On, Off, TXT, Sound, Mute*) and actions, and signal values. Arrows starting at a dot denote default states. For example, *Standby* is the substate of TV which is entered by default, if a state transition to TV occurs. Value changes (events) and variable values can be checked in conditions. Signal values can be changed in actions. Longer lasting activities may be controlled by start, stop or activate. The timing model is causal, the definition of timers is possible (events). The synchronous simulation of Statecharts is based on the so called *synchrony hypothesis*:

- All actions are executed in zero time and
- the system reacts quicker than its environment.

Input/output for Statechart models is realized by the change/reading of external signals. The interface behavior of Statecharts can be described by interpreting changes of input or output signals (including the new value) as symbols in a trace. Such a trace corresponds to a simulation run of a Statechart. However, when interpreting these simulation runs, one should be aware, that the simulation semantics may differ between different Statechart simulators.

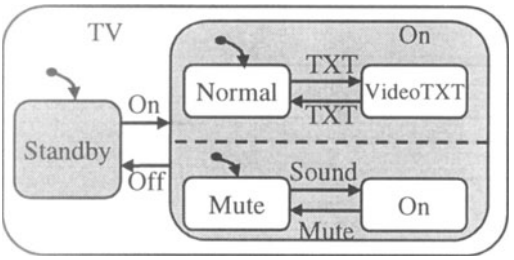

Figure 7: Statechart example

For instance, the Statemate Magnum tool from i-Logix (i-Logix, 1997) distinguishes between *steps* and *supersteps* during simulation for resolving conflicts arising from the specification (see right side of Figure 9). Given a system state consisting of the set of all active states, variable values, signals, internal (timeout) events and history information a *step* is calcualted in the following way:

- Reactions within a step will be executed in the next step.
- Events are alive for one step only.
- Calculations are based on the state at the beginning of the step.

Variables change their values at the end of a step. A *superstep* is completed when no further steps can be executed without external input. The simulation semantics is not compositional.

Other Statechart simulators realize a different semantics not based on the notion of steps and supersteps and hence do not follow the synchronous paradigm any more. Coupling of Statechart modules following different semantics usually raises the same problems like coupling of entirely different specification languages.

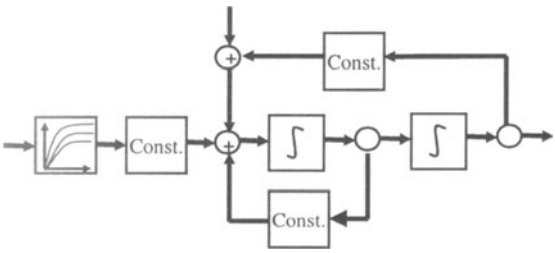

Figure 8 Block diagram example

The second notation used in our example are *differential equations*. Many tools are available that represent differential equations by block diagrams (see Figure 8). Each block may contain mathematical operators, differential or integral operators, and timing delays. The specification of characteristic curves is usually supported, too. Design tools and methods that perform discretization of continuous differential equa-

tions are beyond the scope of this tutorial. For further processing the results obtained by these tools can be represented as dataflow graphs, whose outputs have to be provided at defined points of time. Real time applications often require a periodical evaluation depending on a fixed sampling rate.

For the coupling of event based specifications and solutions of discretized, timed differential equations the first step is to determine which events should be exchanged between these models at which times. An event may for instance be triggered by an interval exceeding of a differential equation. The event driven model part may for instance determine input values, activate continuous input functions, or even change entire parameter sets of the model part specified by differential equations. Furthermore, it may switch between semi continuous models.

A second step is to determine the timing points for data exchange (see Figure 9). For Statecharts data exchange is only reasonable at the beginning of a superstep, since the system reacts in zero time (synchrony hypothesis). For system parts specified by differential equations, for instance, a concept based on timers may be developed to guarantee the correct execution of periodic events.

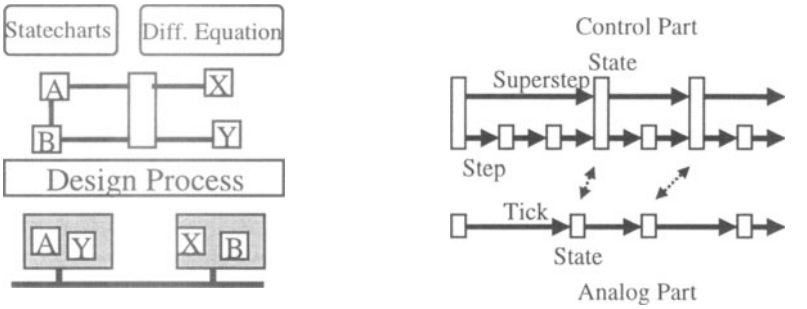

Figure 9: Coupling of Statecharts and differential equations

Using the above explained concept of language coupling, usually, has the following consequences for subsequent steps in the design process (left part of Figure 9). *Optimization* steps can only optimize the distribution of entire model parts in the overall controller topology or perform local optimizations w.r.t. a single model part (A, B, X, Y in Figure 9). A shift of functionality between model parts specified in different notations cannot be done automatically but has to be done manually by altering the specification. Furthermore, the simulation of coupled model parts causes performance loss due to the communication overhead of (at least) two simulator kernels involved for the corresponding model parts. Commercial tools supporting the coupling of Statecharts and block diagrams are already available (ISI, 1998, Tanurhan, 1995). The scientific community often uses the concepts of language coupling to integrate tools with reasonable effort. An example is the approach described in (Rust, 1998) which is based on Software Circuits (Hansson, 1997). Here existing, pre optimized modules are used for system realization. The control part is realized by software running on a real time operating system. The interconnection of existing hardware modules is done by predefined interface modules and protocols.

The goal of a design method should be reuse support at even higher levels of abstraction with largely automatic generation of realizations from a specification.

5. Language Integration

Language integration assumes that all involved specification paradigms are mapped onto a common internal model. Henzinger describes, for example, the integration of finite state machines with differential equations into a common model called hybrid automata (Henzinger, 1996). Hybrid automata combine the specification of event based state transitions (e. g. due to exceeding of intervals) with differential equations. For this purpose states in the state transition graphs can be annotated with systems of differential equation. This work describes also refinement rules for sub graphs fulfilling certain constraints. Hybrid automata do not provide constructs for the specification of complex information processing tasks like e. g. SDL. Therefore, they cannot be used as a common model for embedded systems.

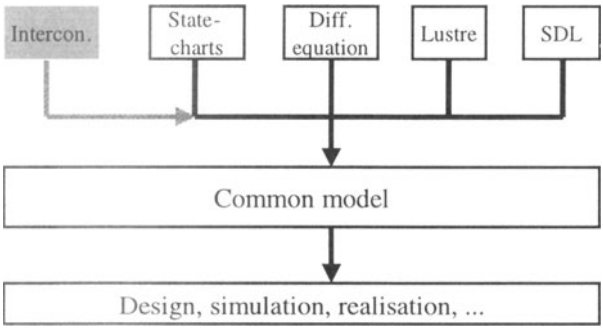

Figure 10: Language integration

A common model has to provide constructs that cover all concepts of the specification languages that should be mapped to it. Hence, a common model tends to become very complex and unwieldy resulting in the fact that established design methods cannot be easily transferred to fit with a common model of the required complexity. To overcome these difficulties and transfer at least parts of a design method some restrictions have to be considered during definition and application of a common model.

At first it should be possible to identify and separate a part of the common integration model generated from a specification (part) in a certain language from the rest of the specification. This results in the requirements for modularity and hierarchy.

Secondly, for such a part in the common model restrictions should be definable in order to allow a transfer of the design methods applicable for the original specification language to this "restricted" common model. Such a transferred design method should then be applicable for all common model parts fulfilling these restrictions. This approach makes design methods also available outside their original application domain.

Furthermore, it allows to shift functions between modules specified in different languages. Hence, the common integration model provides the basis for inter module optimizations.

Last but not least, it should be possible to recognize constructs of the original specification languages also in the common model. This feature on the one hand eases the translation into the common model, on the other hand design steps are more easily traceable.

The remainder of this section will present extended Predicate Transition Nets (Pr/T nets) as an example of such a common integration model. Pr/T nets were derived from Petri nets (Peterson, 1981) by Genrich and Lautenbach (Genrich, 1987). Pr/T nets are a reactive event driven modelling paradigm with a formally defined execution semantics.

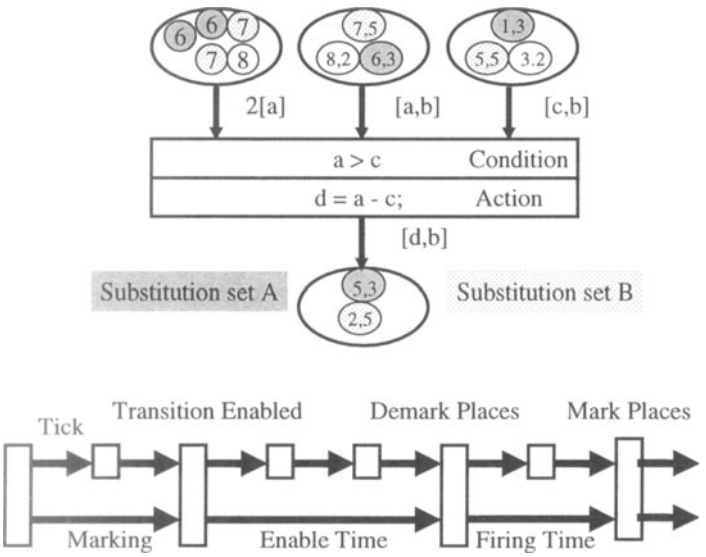

Figure 11: Pr/T net example

A Pr/T net is a bipartite directed graph consisting of *places* (represented as circles, see Figure 11) and *transitions* (represented as boxes, see Figure 11). *Places* serve as data storage and may contain distinguishable *tokens*. *Tokens* may contain structured data items (in our example (tuples of) integers). *Transitions* can be annotated with conditions (a > c) and actions (d = a-c). Edges are annotated with formal variables ([a,b], [c,b], etc.). The execution semantics is defined by the *firing* of transitions. A transition t is *enabled* or may fire in the current *marking* if appropriate tokens are available in its input places, i. e. there exists a valid substitution of the formal variables at t's input edges that fulfills also the conditions in t's annotation. The firing cycle starts by deleting the appropriate tokens from t's input places (demark places, Figure 11). Then the actions annotating t are executed and finally the calculated tokens are generated in t's output places (mark places, Figure 11). Furthermore an *en-*

able delay and a *firing delay* may be specified for a transition t. The *enable delay* specifies an interval how long the appropriate tokens may be in t's input places before t fires. The firing delay specifies a time interval for the length of the firing cycle. Delays may be infinitely long. A delay value (upper/lower bound of delay interval) may be altered by the actions annotating the transition t depending on the data values of the tokens in the actual firing cycle. These changes become valid for the next firing cycle.

Pr/T nets that contain only a bounded number of tokens can be mapped to finite state machines. The above presented timing model fulfills the synchrony hypothesis if enable delays are restricted to the interval [0, 0] and firing delays are restricted to the interval [1,1]. In this case the specification of nondeterminism (forward conflicts) in the net structure is not allowed. In principle these concepts allow to map Statecharts to Pr/T nets. If additionally hierarchical constructs as described in (Cherkasova, 1981, Kleinjohann, 1996) are used, states may be mapped to hierarchical places and state transitions to net transitions. Then the resulting net model resembles also in its graph structure the original Statechart. During design the hierarchy concept allows, for instance, the use of different integration mechanisms simply by instantiating the corresponding net part.

The translation of block diagrams for differential equations into Pr/T nets is also feasible (Kleinjohann, 1996) preserving the original graph structure of the block diagram. Figure 12 presents the nets for a signal sum and a rectangle integrator.

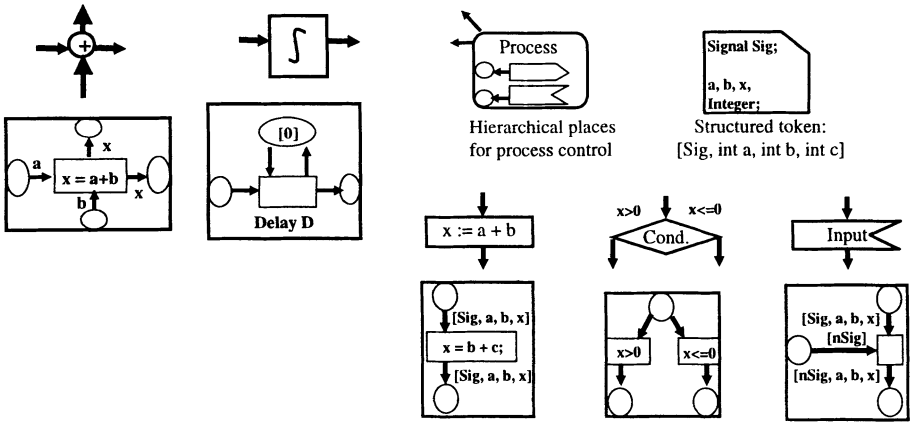

Figure 12: Mapping of block diagrams and SDL to Pr/T nets

The right part of Figure 12 shows the mapping of some SDL constructs to Pr/T nets. SDL is an asynchronous language widely used in telecommunication applications. The control diagrams of SDL provide among others, rectangles for statements, rhombs for conditions, and arrows for communications. Data types are represented in separate diagrams. For a mapping of SDL into Pr/T nets, at first, the data types and defined variables have to be mapped to token types and appropriate annotations. The evaluation of conditions and the execution of statements can be directly mapped to transition annotations. Process diagrams can be transferred similarly to block diagrams for differential equations. The asynchronous communication mechanisms of SDL can be easily mapped to the asynchronous execution of Pr/T nets.

We have developed the SEA Environment for specification and animation of extended Pr/T nets (Kleinjohann, 1996). The handling of foreign language constructs is supported by an abstract graphics that can be linked to the underlying Pr/T net. This graphics can be animated in the desired way when the according net is simulated. By providing appropriate libraries for Statecharts, SDL, block diagrams etc. users can directly use the constructs they are familiar with, without knowledge of the underlying Pr/T net.

Currently some other integration approaches are developed. On the basis of UML (Rational, 1997, Booch, 1996) efforts are undertaken to integrate Statecharts, SDL, Message Sequence Charts, and synchronous data flow notations. Many specification languages relying on the synchronous paradigm like Lustre, Signal, or Esterel (Halbwachs, 1993) can be integrated on the basis of a common synchronous automaton model (Maffeïs, 1996). VHDL and C in the hardware/software codesign domain are often integrated by means of control data flow graphs. However, a commonly suitable solution is not yet known. Here additional research is necessary.

6. Conclusion

In this tutorial we investigated the semantic problems in multilanguage design with regard to the composition of system parts specified in several languages and with regard to their environment behavior. Secondly, we introduced the concepts of language coupling and language integration and explained them by two examples. The third part of the tutorial introduced Pr/T nets as an integration model. Furthermore, we showed exemplary how Statecharts, block diagrams for continuous models, and SDL as an asynchronous language can be integrated by mapping some constructs of these specification languages onto Pr/T nets. If such subnets fulfill some restrictions the design methods provided for the original models can be adopted for Pr/T nets as the integration model.

Solutions for language coupling are already available for specific subsets of languages. But no general solution exists. Common integration models also exist for specific subsets of application domains. However, considerable research is necessary in this field. This research should be directed towards the extension of application domains and more general common models, which cover a broader set of modeling paradigms. By applying appropriate restrictions in the use of such a broad common model the design methods used in the original application domains can be transferred to the common model.

Actually we perform some work on language coupling in the COMITY project (EU 2305) based on previous nationally funded work in the METRO project. We perform further research on language integration and an according design method in a nationally funded project (SFB 376 on massive parallelism).

References

Booch, G., Rumbaugh, J., Jacobson, I. (1996). The Unified Modeling Language for Object Oriented Development, Version 1.0.

Cherkasova L. A., Kotov V. E. (1981). "Structured nets". J. Gruska and M. Chytil, editors, *Mathematical Foundations of Computer Science.* Volume 118 of Lecture Notes in Computer Science. Springer Verlag.

Genrich, H. J. (1987). *Predicate/Transition Nets.* Advances in Petri Nets Part I. Volume 254 of Lecture Notes in Computer Science. Springer Verlag.

Halbwachs, N. (1993). *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers.

Harel, D. (1977). "StateCharts: A visual formalism for complex systems". *Science of Computer Programming* 8(3), pp. 231-274.

Henzinger, T.A., (1996). " The theory of hybrid automata". *Proceedings of the 11th Annual Symposium on Logic in Computer Science.* IEEE Society Press.

I-Logix (1997). *STATEMATE, User Reference Manual, Magnum 1.0.* I-Logix Inc.

ISI (1998). "Design Automation Solutions". Integrated Systems Inc.. http://www.isi.com/Products/DAS

ITU-T (1992)."ITU-T Specification and Description Language SDL", Recommendation Z.100 (SDL-91), ITU General Secretariat, Geneva. http://www.itu.ch/ITU-T.

Kleinjohann, B., Kleinjohann, E., Tacken, J. (1996). "The SEA Language for System Engineering and Animation". *Applications and Theory of Petri Nets.* Volume 1091 of Lecture Notes in Computer Science. Springer Verlag.

Kleinjohann, B., Tacken, J., Tahedl, C. (1997). "Towards a Complete Design Method for Embedded Systems Using Predicate/Transition-Nets". *Proc. of the XIII IFIP WG 10.5 Conference on Computer Hardware Description Languages and Their Applications*, Toledo, Spain, Chapman & Hall.

Maffeïs, O., Morley M., Poigné, A. (1996), " The Synchronous Approach to Designing Reactive Systems". *Arbeitspapiere der GMD*, No. 973. GMD, St. Augustin, Germany.

OSEK (1997), *OSEK/VDX operating system.* ftp://www-iiit.etec.uni-karlsruhe.de/pub/osek/os20ra.pdf

Peterson, J.L. (1981). *Petri Net Theory and the Modelling of Systems.* Prentice Hall.

Rational (1997). *UML Summary, Version 1.1.* http:/www.rational.com/uml/resources/documentation/ summary

Rust, C., Stroop, J., Tacken, J. (1998). "The Design of Embedded Real-Time Systems using the SEA Environment". *Proc. of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98),* Adelaide, Australia.

Tanurhan, Y:, Schmerler, S., Müller-Glaser, K.D. (1995). "A Backplane Approach for Co-simulation in High-Level System Specification Environments". *Proceedings of EURO-DAC.*

van de Snepsheut, J.L.A. (1985). *Trace Theory and VLSI Design.* Springer Verlag.