Embedded system design with multiple languages

Rolf Ernst
Institut für Datenverarbeitungsanlagen
Technical University of Braunschweig
Hans-Sommer-Str. 66
D-38106 Braunschweig
Germany
Tel.: ++49 531 391 3730
Fax: ++49 531 391 3750
email: ernst@ida.ing.tu-bs.de

Ahmed Amine Jerraya
TIMA
46 Avenue Félix Viallet
F-38031 Grenoble Cedx
France
Tel: ++33 476 574 759
Fax: ++33 476 473 814
email: ahmed.jerraya@imag.fr

Abstract – The use of several languages in the design of embedded systems is very convenient for application development and optimization but it can become an obstacle on the way to higher design productivity. This paper explains solutions and future trends.

I. Introduction

With increasing embedded system complexity, there is a tendency to use several languages for specification and design of a single system. A practical reason for this trend is that different teams and companies have developed certain language preferences, but more importantly, abstract languages are often suited for a certain application domain. Flow graph representations suit transformative functions, such as digital signal processing (examples COSSAP, SPW, LUSTRE), while FSM descriptions fit best to describe reactive system behavior, such as a user interface control or a telecommunication protocol. Some languages are particularly developed for an application domain, such as SDL for telecommunication.

Since more complex embedded systems such as a mobile communication terminal almost inevitably include both reactive and transformative system functions, there is a need to either combine system parts described in different languages or find a single general language. There are, in fact, languages which try to cover both state based descriptions and flow graphs, such as the languages of STATEMATE (from I-Logix), ARGOS or UML, but they result in more complex semantics which are harder to treat in analysis and implementation. Also, they basically combine separate languages, such as Statecharts and Activitycharts in STATEMATE.

Languages like VHDL which are used for implementation are certainly general enough to implement many of the semantics of the abstract languages, e.g. based on library functions. It would, however, not be efficient to replace all these languages by VHDL, Verilog or a similar language since, e.g. flow graphs are closer to the application domain and there are numerous transformations on flow graphs which can be exploited in system optimization. Similarly, there is much knowledge how to efficiently verify, merge or split concurrent FSMs.

Moreover, many abstract languages do not define the exact order in which the individual subfunctions are executed as long as the overall behavior described in the language is not violated. This partial order can be exploited in design space exploration. In contrast, languages like VHDL are based on exact timing which constrains the design space more than necessary. Such languages are, however, very useful when it comes to the final hardware and software implementation.

In this paper we will demonstrate two approaches to combine different languages, the language-based approach which couples models in different languages using a fixed communication protocol, and the compositional approach which combines the different model semantics on a unified internal representation.

The paper is structured as follows. The second chapter will give a brief overview of important models of computation with emphasis on their differences. Chapter III introduces the language-based approach which is already used in commercial tools. Chapters IV and V motivate and present the compositional approach which is investigated in research systems. The paper ends with a short conclusion in chapter VI.

II. Models of Computation

To give a better understanding of the problems which arise when coupling languages, we will start with an overview of some important languages for embedded systems. All the languages which are discussed support concurrent execution of processes. One of the main issues is the process interaction and the semantics of the overall (composed) system, i.e. the *model of computation*.

A. *Process networks*

Process networks [1] are an important model of computation (fig. 1). There is a set of processes which communicate via a *directed flow*. The communication consists of a possibly infinite sequence of tokens, the "stream". The processes perform operations on the input tokens and create output tokens. The process network is based on two languages, a *host language* that describes the processes and the *coordination language* which describes the network.

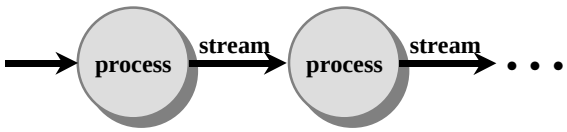

Fig 1: process network

B. *Kahn process networks*

Kahn process networks [2] are a special class of process networks. Here, the stream is a FIFO with unbounded capacity. The processes "consume" the input tokens at the start of a process execution (destructive read) and "produce" output token when the process terminates (non destructive write). A process only executes when all required input data are available (blocking read) but writes independent on the FIFO state (non-blocking write).

Let a process in a Kahn process network consume a stream of input tokens X = x1 x2 ... xn and create a stream of output tokens F(X). A Kahn process is *continuous* if F(X) = F(x1) F (x2) F(xn). This means that a continuous process can be executed iteratively (stepwise), i.e. it can start before the whole input token stream is present. A typical example is a digital filter process where new output data can be generated for each data sample arriving at the input instead of having to wait for all samples to arrive before the output stream can be generated. An example requiring a non-continuous process is sorting, where all data must be present before the largest (or smallest) element is known. Kahn networks which only contain continuous processes have a very nice property: The functions are independent of the order of process execution (proof see [1]). This property opens a wide design space with numerous scheduling opportunities, in particular because the streams are FIFO buffers which reduces the effect of data dependencies and supports pipelining.

Data flow (DF) process networks are Kahn process networks where the processes are controlled by *firing rules.* Firing rules define the conditions for process execution, i.e. the number and type of input tokens required and the corresponding number of output tokens which are generated when the process is executed for a specific firing rule. Data flow process networks with data independent firing rules only (synchronous DF process networks, SDF) allow to determine an efficient static order of process execution. Such DF process networks are frequent in standard signal processing applications. The same holds if the network is cyclo-static, i.e. if it cycles through a data independent sequence of token consumptions and productions [3]. The more general boolean DF network [1] which uses a control input to select a firing rule requires knowledge of the token values or must be implemented with a dynamic process order or leads to less efficient schedules. DF process networks demonstrate that we do not always need full knowledge of the process behavior for design space exploration and process scheduling but that it may be sufficient to abstract few properties such as the token consumption and production. In addition, the execution time on the respective target architecture is required which must be estimated or analyzed. Fig. 2 shows the graphical representation of DF processes.

Commercial examples of DF process network design tools are COSSAP (Synopsys), SPW (Cadence), or the DSP Station (Mentor).

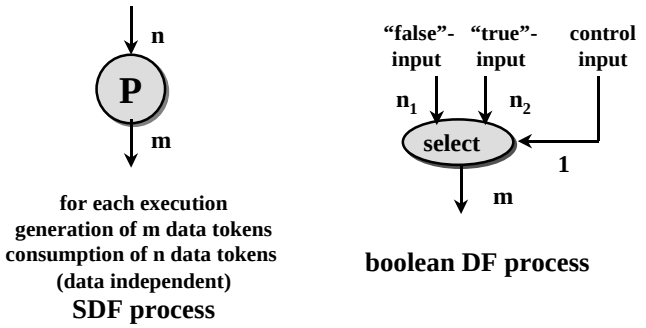

Fig 2: DF process

C. *Synchronous reactive systems*

Synchronous reactive (SR) systems [4] are in many respects at the other end of the spectrum of process network semantics. Fig. 3 shows such an SR system. Here, processes communicate via events, which are defined with a duration of $t_{ev} \rightarrow 0$. In contrast to DF processes which are only executed when all required tokens are available, an SR process is executed upon arrival of any input event. The SR processes react instantaneously (i.e. in zero time) creating output events. This is unlike DF processes which can be delayed and can take a finite execution time as a consequence of buffering and execution order independent network function. The SR output events are immediately visible to all other processes in the network, i.e. there is no explicit event flow. Fig. 3 shows this implicit communication in dotted lines. Timing is introduced using time events. All other delays are zero.

Exact timing events and instantaneous signaling and execution define a total order of events and process executions of an SR system rather than a partial order as in DF systems. This is convenient for verification and simulation since the order of events for a given input pattern becomes unique. Due to the instantaneous process execution assumption, hi-

erarchical expansion of nodes becomes straightforward since the all leaf nodes are, again, executed in zero time preserving the zero-time execution of the parent node. So, an SR system clearly specifies the system reaction. On the other hand, total order and exact timing limit the design space. The impact of this limitation very much depends on the target architecture and system requirements. If computation is fast compared to the specified system timing, such as in many control applications, then the implementation of the specified timing and event order is easy to implement, while in the case of computation intensive tasks and tight timing requirements, such as in digital signal processing, the limi-

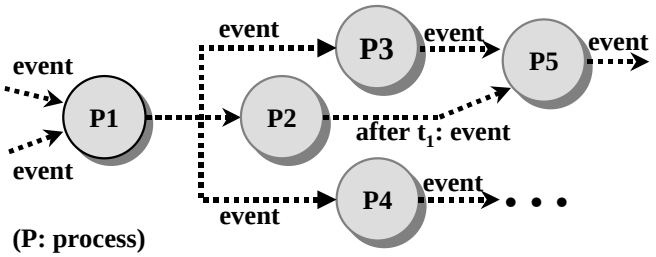

Fig 3: SR systems

tations will be severe. Therefore, SR and DF process networks are suited for complementary system classes.

Examples of languages based on an SR or similar models are Esterel [5], Argos [4] or Statecharts [6].

D. Other models of computation

SDL is a language which is very popular in telecommunication system design. It uses a process network where the process executions are locally controlled. Communication uses buffers which can be controlled by the processes, such that SDL does not correspond to a Kahn graph. The model semantics are rather complex.

MATLAB/SIMULINK (from MathWorks) combines several semantics. Processes communicate via shared registers (destructive write, non-destructive read). They can be executed periodically or upon arrival of an event. Communication via registers hides the actual communication patterns in the host language and constrains the optimization at the network level. It is best suited for software implementation where periodic execution and communication via shared variables is a well known and very general model used in real-time operating systems.

The POLIS system [7] which is specifically targeted to hardware-software co-design, is based on concurrent FSMs with execution time $t > 0$ (Co-design finite state machines, CFSM). Process communication uses registers.

E. Systems without systematic models of computation

Even in up-to-date design processes, there are system functions which are not described with systematic models of computation. Such functions might be legacy code which must be included, non-critical maintenance functions, 3rd party software or large software packages which shall not be touched. These functions will have to be implemented with unknown or data dependent communication and running times. For reliable system design, we should find some way to include such system function in the formal system design process.

III. Language-based approach

The current approach to combine multiple languages in simulation and synthesis is shown in fig. 4. The subsystems 1 to n which may be described in different languages at different levels of abstraction are individually designed and optimized. The subsystems are coupled as communicating processes via a common protocol “backbone.” The backbone protocol can be used for simulation as well as for implementation using communication synthesis. This two-level approach allows to use existing tool environments for each of the subsystems.

The language-based approach is the state of the art that is reflected in tools such as CoWare [8] which uses a client-server approach to communication or AREXSYS [9] which uses a remote procedure call protocol. These EDA systems

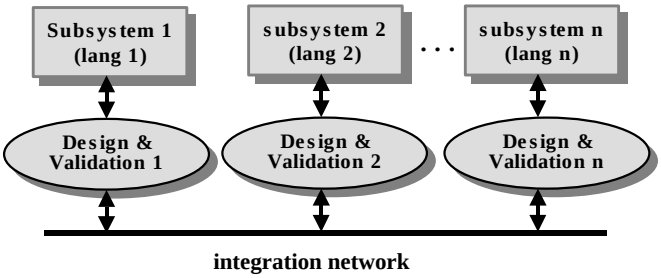

Fig 4: Language based approach

provide libraries with communication primitives implementing the backbone protocol. Object oriented languages, such as C++ or Java are well suited for this task since they allow to hide interface details.

The individual processes can then be mapped to hardware or software components while the communication protocol is mapped to target system communication primitives, such as bus transactions or operating system functions.

The language-based approach is a systematic and very flexible approach to system integration of individually designed subsystems. It is already effective in early design phases since it can combine models at different levels of abstraction.

IV. Design space exploration, global analysis and optimization

To understand the differences between the language-based and the compositional approach, we should have a closer look at the problems of design space exploration and global analysis and optimization. The design process takes an abstract system specification as an input. This specification

defines the design space, i.e. the set of feasible solutions. The *design space* is bound by technical and non-technical constraints which should be included in the specification. *Design space exploration* shall explore the constrained design space to determine the best feasible solution under a given objective function.

In practice, design space exploration is always limited by the design process and the available components and architectures. Examples of component parameters are processor types, memory modules and capacity or word length, while the architecture includes the coarse structure, component integration and global control and data flow as well as the software selection such as the operating system. The design process is defined by methodologies and tools.

The language-based approach is strong in supporting optimization and design space exploration in each language domain and puts a systematic and reliable integration on top. In effect, it uses process networks in their general form as a model of computation. The integration backbone defines the coordination language while there are several host languages corresponding to the subsystem design languages. This model is very flexible but it contains little information at the coordination language level which could be used for analysis or optimization. This is no problem for simulation or interface synthesis, but it limits global analysis. Hence, manual global design space exploration is still possible, but tool support for global analysis and optimization is limited.

There are many reasons to ask for global analysis and optimization. Input-output timing constraints often reach across more than one subsystem. Large memories can possibly be shared by more than one subsystem which requires global memory and buffer optimization. The same holds for system buses and even processor resources which may be controlled by an operating systems which needs a strategy for optimal scheduling. If no such knowledge is available the best strategy is to resort to the well known Rate-Monotonic Schedule, a Round-Robin Schedule, or a dynamic schedule (software) or to keep the subsystems on different components (hardware). The compositional approach described in the following tries to overcome this limitation by collecting and exploiting global information at higher levels of abstraction.

V. Compositional approach

The compositional approach combines the semantics of the subsystem languages to obtain a common representation, the composition format, which can be used for global analysis and optimization. As fig. 5 shows, the language and application specific optimizations at the subsystem level are complemented by analysis and optimization on the abstract common representation.

We can distinguish approaches

1) which map several input descriptions to a common model of computation,

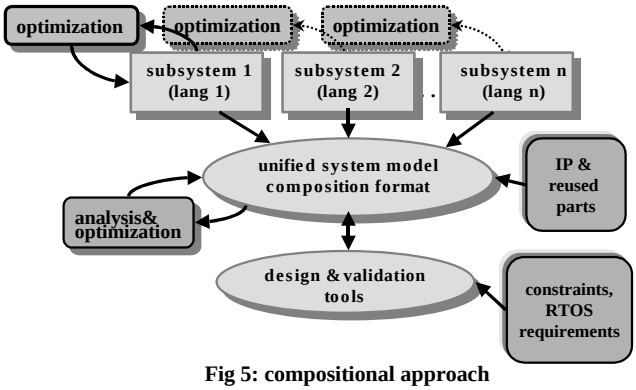

Fig 5: compositional approach

2) which coherently couple several models in a model hierarchy,

3) which capture the design space for a subset of analysis and optimization tasks.

A. *Process coordination calculus PCC*

The *process coordination calculus, PCC* [10] approach which belongs to type 1) combines 2 process types in a single model, as shown in fig. 6. There are *data driven processes* which *may* be activated if the required tokens are available (DF semantics) and event driven processes, which are executed at any input event (see SR). The data token are communicated via data streams which are depicted as solid

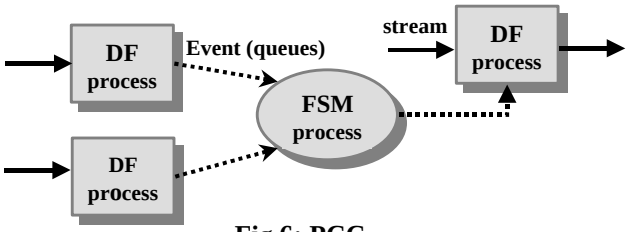

Fig 6: PCC

edges in fig. 6, while event communication is shown with dotted edges. Events can be queued.

Obviously, the function of the rightmost SDF process in fig. 6 depends on the order in which the leftmost SDF processes are executed. To avoid such situations, the authors introduce additional scheduling constraints to control the network behavior. Such constraints, however, limit the design space.

B. **charts*

**charts* [11] are hierarchical process networks. Each process network uses a single model of computation. A single process or state in that network can be refined by a process network with a different model of computation as shown in fig. 7. There is a fixed order in the hierarchy, where either the parent node is a state of a finite state machine (FSM) or the child network is a finite state machine. In other words, every second level is an FSM model. The other levels can be SDF networks, SR systems and some other models of computation which have not been explained here, such as DE (discrete event) systems.

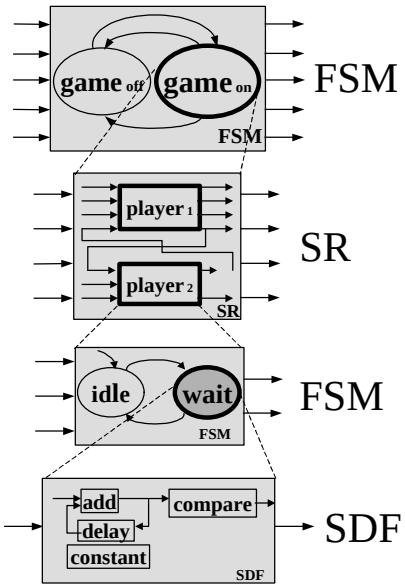

Fig 7: *charts

There are few rules for process refinement which are necessary to guarantee the required semantics of the refined process. As an example, an FSM state transition takes a finite time. If an FSM state is refined by an SDF process network, then there must be a termination time of the SDF network. An SDF process network can contain complex loops and therefore does not have a trivial termination condition. Instead, the minimum cycle which is unique and can be determined from the network is used as a termination condition. Since every FSM state can be defined by a different SDF, FSM timing becomes state dependent. To be as precise as possible, the timing is propagated to higher levels of the hierarchy eventually ending up with a potentially large set of global state dependent equations which can be used for scheduling.

*charts exploit the features of the different languages. It keeps the language domains separate but propagates optimization constraints such that global analysis becomes possible. *charts are used as internal representation for the new version of PTOLEMY.

Both PCC and *charts can be used as a general representation. They are executable (can be simulated) and can, given the limitations which were mentioned above, be used for design space exploration. *charts have the additional capability that global states and constraints can be propagated through the model hierarchy which PCC does not support. On the other hand, both adhere to fixed models of computation and do not offer a solution to include legacy code, 3rd party software or partially documented system parts.

C. System property intervals SPI

System property intervals (SPI) [12, 13] is a single abstract, non executable process network model which is only targeted to design space exploration and system synthesis. Process communication uses FIFO buffers or registers.

The main feature of SPI is that system properties are annotated as intervals. These properties include communication, timing and constraints. This way, SPI can cover systems with conditional or incompletely known behavior, such as in legacy or 3rd party IP components. The SPI modeling approach, as shown in fig. 8, maps the process networks of the input languages to the SPI process network. SPI uses intervals whenever it cannot directly reflect the input language semantics. Similar to the firing rules of DF graphs, some of the SPI properties are derived from the process descriptions of the input languages. These process descriptions may be given in a completely different host language. Virtual processes are used to describe input language semantics which cannot directly be modeled, such as periodic process execution (see Matlab/Simulink). SPI allows to model process modes [14] which are communicated between processes to capture system state dependent timing and communication. Similar to *charts, the knowledge of global system states can be used to optimize scheduling but unlike *charts, SPI does not require minimum cycle termination for mode changes which supports faster system reaction. Furthermore, SPI supports process merging and refinement by adaptation of the intervals.

SPI covers a wide range of model semantics. Translation rules have been derived for periodic processes, DF networks, SDL, or Statecharts.

On the other hand, SPI is not executable due to the uncertainty effect of property intervals. Its only intended purpose is system implementation including process scheduling

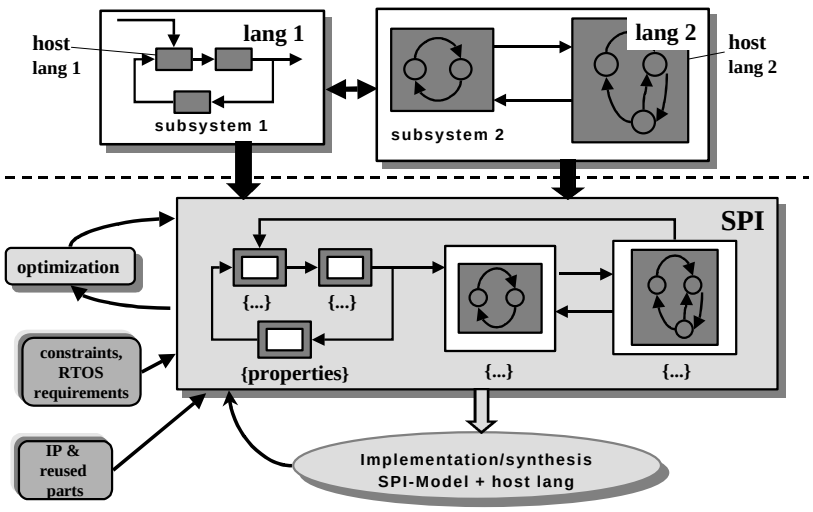

Fig 8: SPI approach

and load distribution. A second representation is required for simulation, e.g. using a language-based approach. So, the SPI modeling approach is an approach of type 3).

VI. Conclusion

There is no single design language which is suitable and convenient for all applications. The necessity to reuse parts of a design which may be described in a different language are a further incentive to consider designing with different languages. We may, therefore, expect that multi-language designs will become standard at least at the more abstract design levels. The main problem is the integration of designs in different languages. The state of the art covers multi-language simulation and subsystem optimization. Integration uses a communication backbone with a low level communication protocol. In the future, language composition is required to enable design space exploration and global optimization across input languages. We have presented several approaches to language composition which are based on different paradigms.

References

[1] E.A. Lee, T.M. Parks. *Data flow process networks*. Proceedings of the IEEE, vol. 83(5), p. 773ff,. May 95.

[2] G. Kahn. *The semantics of a simple language for parallel programming*. Proceedings of the IFIP Congress 74. Amsterdam, The Netherlands, North Holland, 74.

[3] R. Lauwereins, P. Wauters, M. Adi, J.A. Peperstraete. *Geometric parallelism and cyclo-static dataflow in GRAPE-II*. Proceedings 5th Int. Workshop on Rapid System Prototyping, Grenoble, France, June 94.

[4] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic, 93.

[5] Beneviste, G. Berry. *The synchronous approach to reactive and real-time systems.* Proc. IEEE, vol. 79, p. 1270ff, Sep. 91.

[6] D. Harel and A. Naamad. “The STATEMATE semantics of Statecharts. ACM Trans. on Software Engineering Methodology. Oct. 96, p. 293ff.

[7] Balarin, P. Giusto, A. Jurecska et al., Hardware-Software Co-Design of Embedded Systems: The POLIS Approach, Kluwer Academic Publishers, May 97.

[8] K. van Rompaey, D. Verkest, I. Bolsens, H. De Man. CoWare – a design environment for heterogeneous hardware/software systems. Proceedings of the European Design Automation Conference (EURODAC) 96, Geneva, Sep. 96.

[9] C.A. Valderrama, A. Changuel, P.V. Raghavan, M. Abid, T. Ben Ismail, A.A. Jerraya. *A unified model for cosimulation and co-synthesis of mixed hardware/software systems*. Proceedings of the European Design and Test Conference (EDAC-ETC-EuroASIC). IEEE CS Press, March 95.

[10] T. Grötker, R. Schoenen, H. Meyr,. *PCC: A Modeling Technique for Mixed Control/Data Flow Systems*, Proceedings of the European Design & Test Conference, pp. 482-486, Paris, France, March 97

[11] Girault, B. Lee, E.A. Lee. *Hierarchical Finite State Machines with Multiple Concurrency Models*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 18 (6), p. 742ff, June 99.

[12] Ziegenbein, R Ernst, K. Richter, J. Teich, L. Thiele. *Combining Multiple Models of Computation for Scheduling and Allocation.* Proc. 6. IEEE/ACM/IFIP Int. Workshop on Hardware/Software Co-Design (Codes/CASHE '98), pp. 9-13, Seattle, 98.

[13] R. Ernst, D. Ziegenbein, K. Richter, L. Thiele, J. Teich. *Hardware/Software Co-Design of Embedded Systems – The SPI Workbench*. Proceedings IEEE Workshop on VLSI‘99, pp. 9-17, Orlando, 99.

[14] Ziegenbein, K. Richter, R. Ernst, J. Teich, L. Thiele. *Representation of Process Mode Correlation for Scheduling.* Proceedings ICCAD 98, San Jose, 98.