Project 22703: SYRF

Deliverable 4/5.2

Integrating synchrony & asynchrony: distributed code generation

Abstract

Modular and distributed code generation is the main subject of this paper. Distributed code generation aims at reducing the dependency of the design with respect to the target architecture. Modularity helps reuse of existing designs, and makes it possible to address much larger systems.

Contents

1 Distributed code generation: overview of the approach

1.1 A first glance at the method

The overall method [3, 4, 6] is illustrated in the figures 1 and 2.

Figure 1 shows what the designer has to do. The designer has on her/his screen (at least) three windows (those of the top part of the figure). The first window—top left—is the (Signal, or $\mathrm{D}\textsc{c}_{+}$, or Activity Charts in StateMate) program editor. In this window, a dataflow diagram is depicted. The arrows would typically depict flows of data, but also could correspond to scheduling requests. In the bottom right window, icons are shown which allow the designer to specify her/his target architecture.

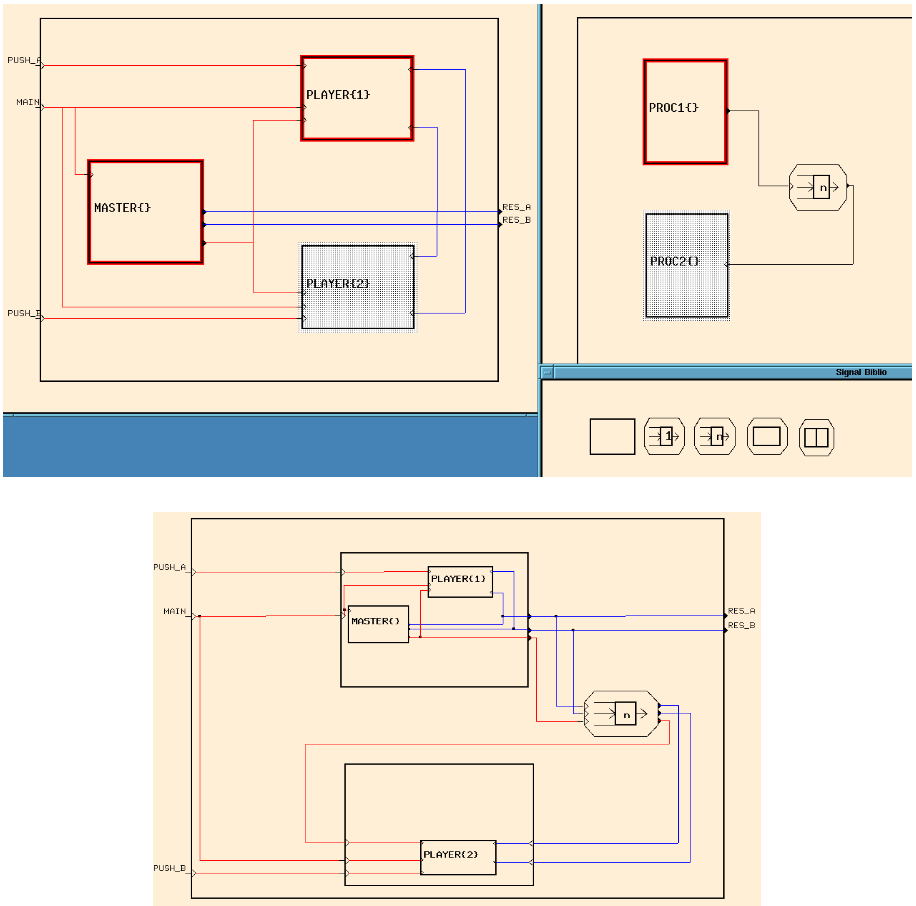

Figure 1: *Code generation: what the designer does*

This architecture has two types of constitutive elements.

The first one (on the left) is a *processor,* i.e. a machine that complies with the synchronous model of execution, in which a run is a sequence of atomic reactions. Thus processors can be pieces of sequential code (C/C_{++}-procedures, threads, etc.), or alternatively parallel machines running according to the model of perfect synchrony (e.g., synchronous hardware).

Other icons refer to (generally *a*synchronous) communication media. Using these two windows, the designer builds, in the third window (top right), her/his execution architecture: the source dataflow diagram is partitioned as shown in the figure, and corresponding subdiagrams are mapped onto "processors" by click-and-point. Also, models of communication links are specified by the designer, by clicking-and-pointing to the appropriate icon.

The result of this mapping is built automatically as illustrated in the bottom part of the figure.

Figure 2 shows what the tool generates for each processor. From the specifications provided by the designer as in figure 1, the tool will generate, for embedding into each processor, the following: 1/ a suitable communication protocol which guarantees that the

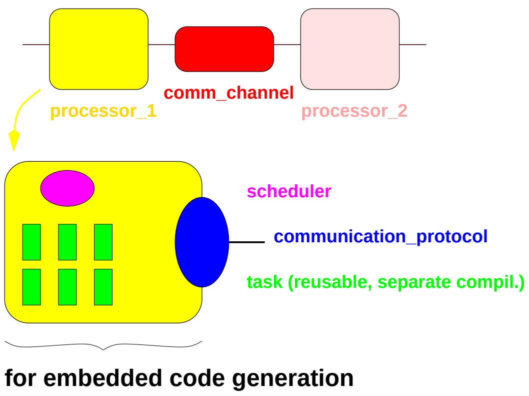

Figure 2: *Code generation: what the tool generates*

semantics of synchronous communication will be preserved even though an asynchronous communication medium is used; 2/ a structuring of the code into pieces of sequential code and a scheduler, aiming at guaranteing separate compilation and reuse.

Finally, the whole model (processors and channel models) can be used for architecture simulation and profiling.

1.2 Summary of the data needed to model the architecture : the deployment map

We do not address the partitioning/scheduling problem against quantitative optimizations (as for traditional understanding in hardware/software partitioning). Instead, we assume that an map is given, assigning nodes and flows to processors and communication links, we call it the *deployment map*. The user interface provided to describe this map differs for specific environments (SILDEX, STATEMATE)—in the figure 1, the Inria environment for SIGNAL is used.

We summarize here which data have to be provided by the users of architecture-dependent implementation of applications our approach :

1. the deployment map.

2. a definition of the set of system-level primitives used for communications (readings and writings to the media). This amounts to describing the calls to the library of OS services to which the code will be linked.

2 Modular approach to the distributed code generation: main steps

2.1 Assumptions

- We assume the deployment map is available.

- We assume the DC$_+$ program has a single activation clock, from which other clocks follow deterministically. This guarantees *endochrony.*

- Every flow which is not an input is functionally defined (no circular definition, the program is deterministic).

2.2 The deployment map

This is illustrated in the figures 3 and 4. Figure 3 depicts the functional architecture.

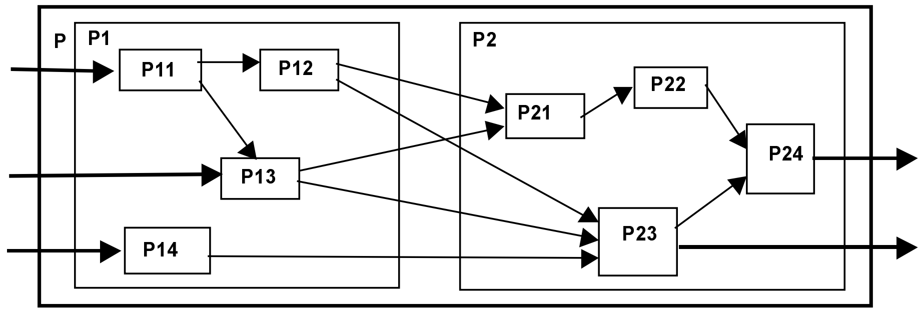

Figure 3: $P = (P_1 \,\|\, P_2)$

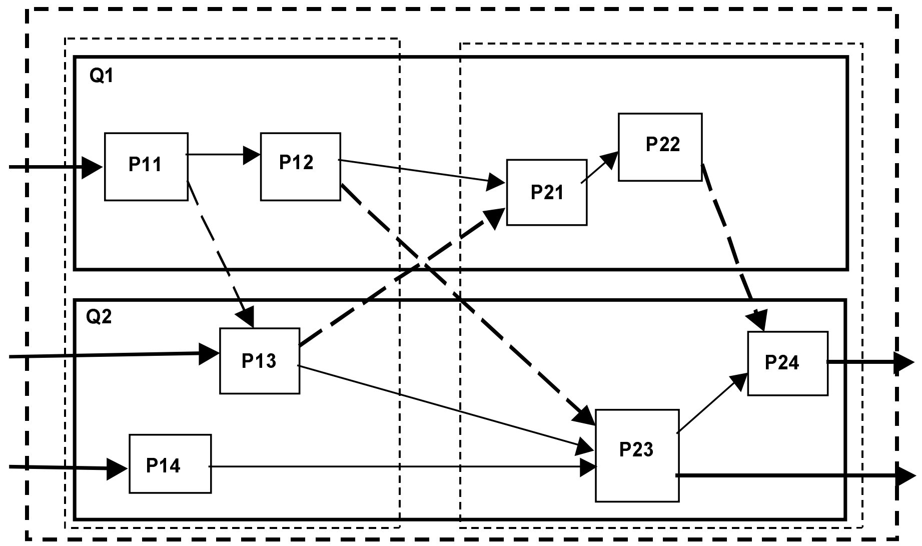

Figure 4: *Deployment map:* $P = (Q_1 \,\|\, Q_2)$, *macro-task* $Q_1 = (P_{11} \,\|\, P_{12} \,\|\, P_{21} \,\|\, P_{22})$, *macro-task* $Q_2 = (P_{13} \,\|\, P_{14} \,\|\, P_{23} \,\|\, P_{24})$

Then figure 4 shows its deployment on a two processor architecture. Resulting communication links are shown in thick.

2.3 Node abstractions

Each node, say Q, is associated with an interface containing:

- as inputs, the input flows of Q,
- as outputs, the output flows of Q

- the clock tree of the input/output flows,
- the clock-labelled dependences between external flows ; this is computed using special algorithms for building the transitive closure a of clock-labelled graph.

2.4 Building reusable tasks

Definition *We say that two nodes N_1 and N_2 are input-sensitively equivalent if and only if for each input i:*

$$\exists \text{ causality path from } i \text{ to} N_1 \quad \Leftrightarrow \quad \exists \text{ causality path from } i \text{ to} N_2$$

Note that in this definition, we do not take clocks of dependences into account. This is illustrated in figure figure 5. Input-sensitivity classes constitute our formal no-

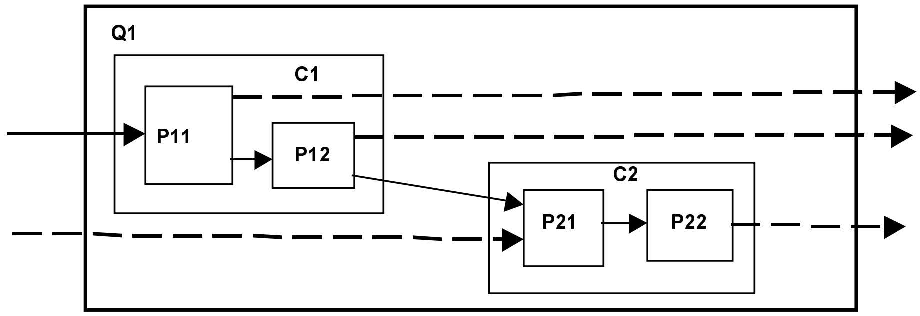

Figure 5: *Partitioning Q_1 into input-sensitivity classes C_1 and C_2.*

tion for a *task.* Each task can be executed in an atomic way using the scheme *read_inputs/compute_reaction/emit_outputs.* Any scheduling compliant with this scheme can be implemented. Tasks can be stored and reused as object code.

2.5 Task scheduling and abstractions

Tasks must be properly scheduled within a macro-task. To this end, task abstraction is used, as depicted in figure 6, in which causality arcs have been substituted to the original dataflow sub-graphs C_1 and C_2. From the point of view of the scheduler, a task is seen

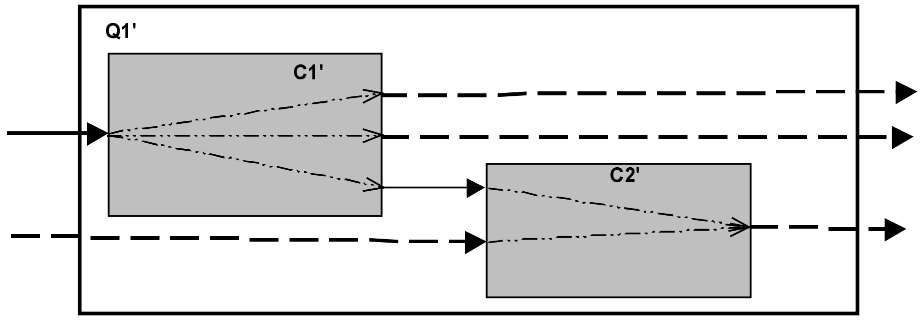

Figure 6: *Task abstractions*

as a procedure call. The scheduler must schedule the tasks in a way compliant with the scheduling constraints resulting from the abstraction shown in figure 6.

2.6 Code generation for each macro-task

Communications

Inter macro-task communication uses the communication layer of the architecture. Communication generation is performed by the way of calls to the right OS primitives, as given by the architecture description or by the user. Communication features can be modelled as (possibly nondeterministic) synchronous process abstractions (some *grey box*), cf. figure 7. To represent the new graph obtained by adding communications, we have to add

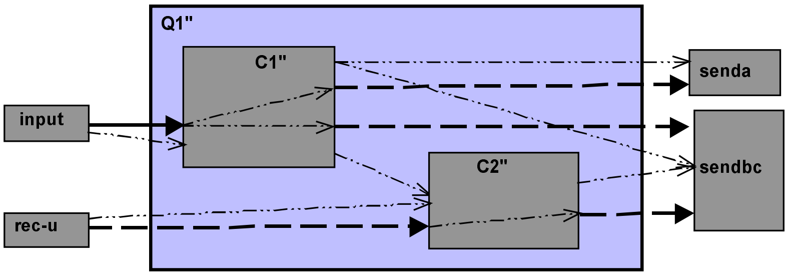

Figure 7: *macro-task abstraction with communications*

causality links with the communication nodes.

Task scheduling

In the case where the target architecture is an OS, the following design choices can be made. For a task such as Q_1'' (cf. figure 7) it is possible to generate parallel processes scheduled and synchronized (synchronization gates) by OS primitives as shown for example in figure 8. In this case, nothing is known about the interleaving of elementary actions in T_1 and T_2. Variations of this techniques can be performed, using scheduling constraints.

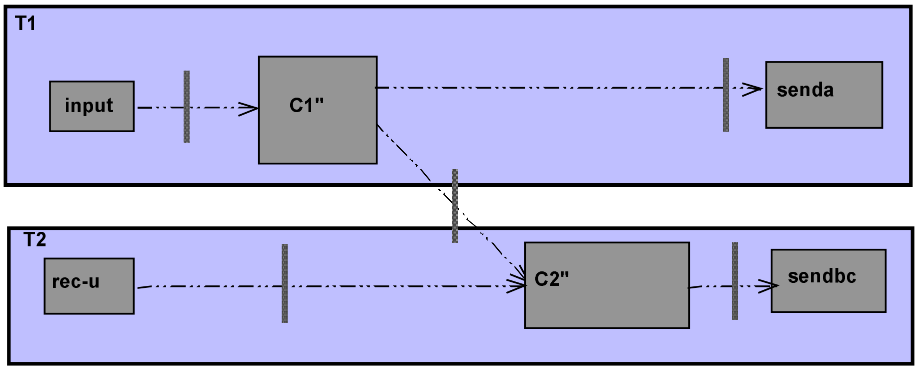

Figure 8: *OS macro-task scheduling*

Synchronization and communication

The synchronization between tasks has to be defined by the user. Different schemes can be implemented.

Strong synchrony. The first implementation that can be done is to set a global gate to synchronize tasks. At each step, each activated task signals its completion to its predecessors in the graph when it has been completed and has received completion acknowledge from each of its successors. Thus at most one logical step is computed during the same instant.

Weak synchrony. A second scheme is to set local gates to synchronize tasks. At each step, each activated task signals to its predecessors in the graph that it is ready to proceed a new step when its previous step has been completed and a "ready to proceed" has been received from each of its successors concerning this previous step. This implementation allows pipeline.

Bounded asynchrony. The most permissive scheme for embedded systems is to allow communication implementation as bounded FIFO. This is guaranteed correct [1] thanks to the satisfaction of isochrony of communications which is provided by the synthesized communication protocols.

2.7 Code instrumentation

Finally, we briefly mention our approach to code instrumentation, to evaluate e.g. performance [5].

Given an implementation Q of a program and a model of time consumption for each of the atomic actions in Q, we automatically generate a program $T(Q)$ homomorphic to Q; $T(Q)$ is the parallel composition of the images $T(Q_i)$ of the components Q_i (including communications) of Q. $T(Q_i)$ are given by the user as $\mathrm{D}\textsc{c}_+$ components whose interfaces are composed of integer flows $T(x)$ instead of the original flows x. $T(x)$ represents the sequence of the availability dates for the occurrences of the original flow x.

$T(Q)$ is thus a model of real time consumption of the application (functional specification and architectural support). Some real time properties to be satisfied can be described as predicates in $\mathrm{D}\textsc{c}_+$.

3 Conclusion

This paper presents a framework for distributed code generation of synchronous programs that allows relaxing synchrony, thanks to the properties of endochrony and isochrony. On the other hand, the definition of precise abstractions of the programs permits reuse of separately compiled programs. So the method is also a method for separate compilation. It has been implemented in the framework of the SACRES project, through a number of software modules applicable to $\mathrm{D}\textsc{c}_+$ programs.

The method can be applied to many possible targets, using different real-time kernels for instance (let us mention Posix, VxWorks, OSEK for automotive, ARINC for avionics...).

[1] leaving aside timeliness issues, of course!

Also, thanks to the translation of StateMate to Dc_+ developed in the SACRES project [1, 2], it is applicable not only to dataflow programs such as Signal ones, but also to StateMate and Lustre designs.

References

[1] J.-R. Beauvais, T. Gautier, P. Le Guernic, R. Houdebine, E. Rutten, "A translation of StateCharts into Signal", in *Proceedings of the International Conference on Application of Concurrency to System Design (CSD'98)*, IEEE Publ., Aizu-Wakamatsu, Japan, March 23–26, 1998, 52–62.

[2] J.-R. Beauvais, R. Houdebine, P. Le Guernic, E. Rutten, T. Gautier, *A Translation of Statecharts and Activitycharts into Signal Equations,* Inria Research Report RR-3397, April 1998, `http://www.inria.fr/RRRT/RR-3397.html`.

[3] A. Benveniste (& contributors), "Safety Critical Embedded Systems Design: the SACRES approach", in *Proceedings of Formal Techniques in Real-Time and Fault Tolerant systems, FTRTFT'98,* Lecture Notes in Computer Science, Springer Verlag, September 1998.

[4] A. Benveniste, T. Gautier, P. Le Guernic, E. Rutten, "Distributed code generation of dataflow synchronous programs: the SACRES approach", in *Proceedings of The Eleventh International Symposium on Languages for Intensional Programming, ISLIP'98,* Sun Microsystems, Menlo Park, California (USA), May 1998.

[5] A.A. Kountouris, P. Le Guernic, "Profiling of SIGNAL Programs and its application in the timing evaluation of design implementations", in *Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems,* HP Labs, Bristol, UK, IEE, February 1996.

[6] P. Le Guernic, S. Machard, E. Rutten, "Répartition de programmes Signal", in *Actes des Rencontres Francophones du Parallélisme des Architectures et des Systèmes, RenPar'10,* Strasbourg, 9–12 juin 1998 *(in French).*

[7] SACRES consortium, *The Declarative Code DC+, Version 1.4,* Esprit project EP 20897: Sacres, Nov. 1997.