Synchronous Block Diagrams

Stavros Tripakis*

June 27, 2008

1 Introduction

In the field of embedded, real-time systems, like in many other fields, specialized (sometimes called "domain-specific") languages are used. Simulink from The MathWorks[1] and SCADE from Esterel Technologies[2] are two successful commercial products in this field, and they are especially widespread in the automotive and avionics domains. These environments include a graphical model editor, a simulator and a code generator. Automatic generation of code that implements the semantics of a model is useful in different contexts: the code can be used for simulation; but it can also be embedded in a real-time control system such as X-by-wire. In fact, usages of the latter type are increasingly adopted by the industry.

In this context, it is natural to view these tools as programming tools, and the languages (often graphical) that these tools use as programming languages. It is tempting to think of these languages as one of the next steps in the evolution of programming languages from assembly to structural and object-oriented code.

The fundamental model behind notations such as Simulink and SCADE is that of *synchronous block diagrams*. These are hierarchical data-flow diagrams with a synchronous semantics, similar to that of languages such as Lustre [8] or Esterel [9]. Block diagrams are an intuitive model that has its roots to system and control theory. As such many students graduating, for instance, from electrical engineering departments, are familiar with this model, as they have seen applications of it, for example, from the signal processing or control domains. Synchronous block diagrams (SBDs) can be viewed as the discrete-time subclass of this model, which allows to capture difference equations. Tools like Simulink also allow continuous-time blocks (e.g., integrators) which allow to describe differential equations (see Figure 1 for an example[3]). This part, however, is used mainly for modeling and simulation, and not for implementation. Thus it cannot directly be considered a language for programming. We will consider SBDs in the rest of this document.

2 Description

We briefly describe the SBD model, its syntax and semantics. For simplicity, we consider only the "purely synchronous" version of the model here, where all blocks are executed at the same rate. This simple model can be extended with *triggers*, which allow to capture *multi-rate* designs in an easy and modular way. See [2, 3] for details.

An example of an SBD is shown in Figure 2 (left). There are $5 + 2 + 1 = 8$ blocks in total shown in this diagram. Block Top is the top-level block, with sub-blocks P and Q. Q has two sub-blocks C and D, and so on. Each block has a set of input ports and a set of output ports. The links connecting these ports are called *signals*.

*Cadence Research Laboratories, 2150 Shattuck Avenue, Berkeley, CA 94704, tripakis@cadence.com. The author is currently on leave from the Verimag Laboratory, where part of the work referenced in this document was done. This work is joint with a number of colleagues, see references [1, 2, 3, 4, 5, 6, 7].

[1] www.mathworks.com/products/simulink/

[2] www.esterel-technologies.com/products/scade-suite/

[3] picture copyright The Mathworks

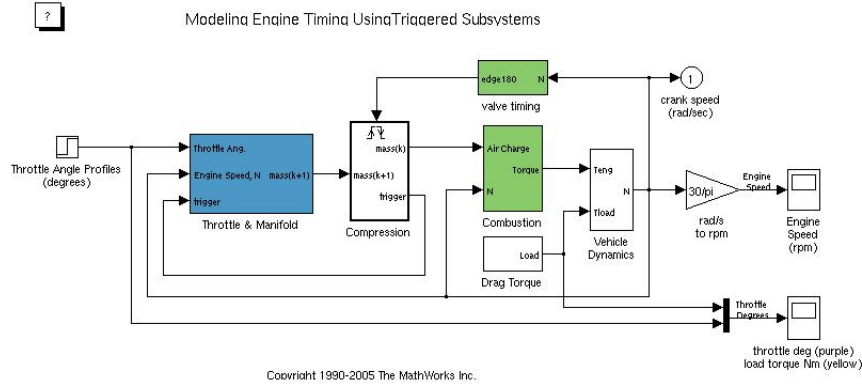

Figure 1: An engine timing model in Simulink

Notice that this model is *static* in the sense that dynamic creation of blocks and links is not allowed. Dynamically changing the connections is also not allowed. Also not that hierarchy is added to the model to make descriptions modular and tractable. Composite, or macro blocks (e.g., Top, P, Q in the figure) are hierarchical blocks, that encapsulate a number of sub-blocks connected in an internal diagram. The latter is "hidden" at the level of the macro block.

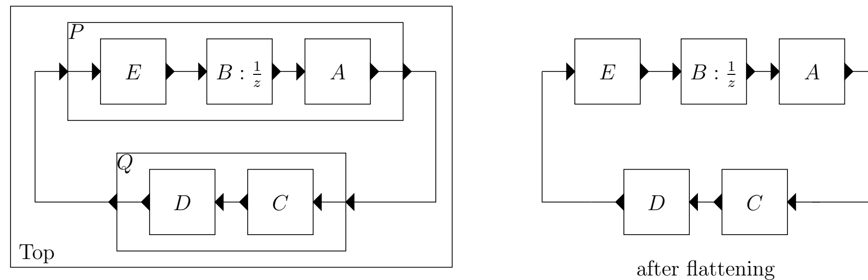

Figure 2: A hierarchical block diagram.

Atomic blocks are those that have no sub-blocks. Each atomic block A is pre-classified as either *combinational* (state-less) or *sequential* (having internal state). Some sequential blocks are *Moore-sequential*. Each output of a Moore-sequential block *only depends on the current state, but not on the current inputs*. For example a *unit-delay* block that stores the input and provides it as output in the next synchronous instant is a Moore-sequential block. The above definitions extend to macro blocks in a natural way.

An SBD is *flat* if it contains only atomic blocks. Any hierarchical SBD can be made flat by a procedure called *flattening*, which successively removes macro blocks and replaces them by their internal diagram, by appropriately re-establishing the connections. An example is shown in Figure 2 (right).

A flat diagram is called *acyclic* if every cycle in the diagram contains at least one Moore-sequential block. This guarantees that there is no cyclic dependency of the signals in the diagram in a given synchronous instant.

We assign semantics only to flat, acyclic diagrams. The semantics we use are standard synchronous semantics used also in languages like Lustre. Each signal x of the diagram is interpreted as a total function $x : N \to V_x$, where $N = \{0, 1, 2, 3, ...\}$ and V_x is a set of values: $x(k)$ denotes the value of signal x at time instant k. If x is an input this value is determined by the environment, otherwise it is determined by the (unique) block that produces x. Since the diagram is acyclic there exists a well-defined order of firing the blocks to compute the values of all signals at a given synchronous instant.

3 Code generation

Code generation from SBDs has often been regarded as a task of mere translation, not deserving the term "compilation". Even the simplest code generation methods, however, do more than just translation. For instance, consider a flat block diagram and suppose we want to generate a single, "monolithic" piece of sequential code of the form "loop { await trigger; read inputs; compute; write outputs }", to be executed on a single processor (see, for instance [1]). In this case, we still need to check whether the diagram is acyclic.[4] For acyclic diagrams, we can topologically sort the blocks to generate sequential code. We also need to make sure the worst-case execution time of the body of the loop does not exceed the minimum inter-arrival time between triggers. It can be seen that even in this simple case, code generation is more than mere translation.

Code generation becomes even more interesting in cases where the target software and hardware architecture are more advanced. We briefly mention here some recent works that have dealt with such cases, contributing in completing the picture of software synthesis from synchronous block diagrams.

One long-standing problem in the domain of synchronous models has been *modular* code generation (sometimes called "separate compilation"). In the context of block diagrams, modularity means code should be generated for a given composite block independently from context, that is, without knowing in which diagrams this block may be used in the future. Recent work [2, 3] has provided a solution to this problem. The main idea of the solution is to abstract each block into a set of interface functions plus a set of dependencies between these functions. The finer the abstraction, the more information about input-output dependencies is preserved, thus allowing the block to be reusable in more contexts. This approach allows modularity to be quantified, in terms of the size of the interface, i.e., the granularity of the abstraction. It also reveals fundamental trade-offs between modularity, reusability and size of the generated code [4].

Another old puzzle has been how to generate non-monolithic, multi-threaded code that runs on a single processor and correctly implements the semantics of a given synchronous model. The main issue with single-processor multi-threading is that some preemptive scheduling policy is usually applied so that threads can share the CPU. Because of preemptions, there is generally no guarantee that data-flow between threads is the same as in the original synchronous model. Recent work [5] has shown that protocols can be devised to guarantee that synchronous data-flow is preserved by the preemptive implementation, at the expense of some quantifiable memory overhead.

Finally, let us mention the problem of distributed code generation, that is, code generation for a distributed execution platform. This problem has been long studied, and it involves many non-trivial decisions, such as how to distribute the function on the execution platform, whether to replicate computation or whether to communicate, how to schedule computation and communication, and so on. With the current state of the art in combinatorial optimization, it is not realistic to expect an automated tool to resolve all these choices. However, even when all choices that are combinatorial in nature are assumed to be fixed (e.g., decided by the user), it is still non-trivial to generate an implementation that preserves the semantics. In particular, the precise choice of the execution platform is crucial, and considerably influences the solution.

We have experienced this in recent works, that proposed semantics-preserving implementations of synchronous block diagrams on different distributed execution platforms: a "synchronous" one based on the Time-Triggered Architecture (TTA) [12], an "asynchronous" one based on Kahn process networks but with finite queues [13], and a "loosely" time-triggered architecture where clocks are allowed to drift but in a bounded way. Code generation for these architectures has been studied in [6, 7]. Even though in some cases the solutions can be layered, more work is needed to generalize this approach into a generic compilation suite, following traditional module-separation principles.

4 Conclusions and perspectives

To summarize, synchronous block diagrams are at the heart of widespread embedded software tools such as Simulink. Code generation for SBDs is not only important from an industrial perspective, but also an

[4]Cyclic diagrams can also be handled, using methods such as those proposed in [10, 11]. Notice that these methods require *semantical* information for the atomic blocks, e.g., whether the block computes a logical AND vs. a logical OR, etc.

interesting research problem with multiple facets, depending on the target execution platform, semantical-preservation requirements, and optimization goals. It thus rightfully deserves the status of "compilation" or "synthesis".

One direction for future work is to populate the list of code generation methods with more techniques, that target more types of execution platforms. It is also important to investigate new relations that capture what needs to be preserved when moving from the SBD model to an implementation. Sometimes strict semantical-preservation such as "no value must be lost" is not necessary (e.g., in control applications where the "freshest" value is usually what is needed). How to blend strict and more relaxed preservation requirements in the same application needs to be investigated. Extensions to the language are also necessary. SBDs are a simple model, that is good for some types of applications, but not for others. Already tools such as Simulink can incorporate other modeling formalisms, e.g., state machines (c.f., Stateflow). Code generation methods that incorporate seamlessly such heterogeneous models need to be studied.

References

[1] P. Caspi, P. Raymond, and S. Tripakis. Synchronous programming. In *Handbook of Real-Time and Embedded Systems*, pages 14–1 – 14–21. Chapman & Hall, 2007.

[2] R. Lublinerman and S. Tripakis. Modularity vs. Reusability: Code Generation from Synchronous Block Diagrams. In *Design, Automation, and Test in Europe (DATE'08)*. ACM, March 2008.

[3] R. Lublinerman and S. Tripakis. Modular Code Generation from Triggered and Timed Block Diagrams. In *14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*. IEEE CS Press, April 2008.

[4] R. Lublinerman, C. Szegedy, and S. Tripakis. Modular Code Generation from Synchronous Block Diagrams: Modularity vs. Code Size. Submitted.

[5] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis. Semantics-Preserving Multitask Implementation of Synchronous Programs. *ACM Transactions on Embedded Computing Systems*, 7(2):1–40, February 2008.

[6] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a Layered Approach for Distributed Embedded Applications. In *Proceedings of the 2003 ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, pages 153–162. ACM, June 2003.

[7] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, and M. Di Natale. Implementing Synchronous Models on Loosely Time-Triggered Architectures. IEEE Transactions on Computers (to appear).

[8] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Symp. POPL*, 1987.

[9] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[10] S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956, 1994.

[11] G. Berry. The Constructive Semantics of Pure Esterel, 1999.

[12] H. Kopetz and G. Grunsteidl. TTP – a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

[13] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.