Optimization and Synthesis for Complex Reactive Embedded Systems by Incremental Collapsing

Massimiliano Chiodo
Cadence Technologies
555 River Oaks Pkwy
San Jose, CA 95134, USA
maxc@cadence.com

ABSTRACT

We propose a software synthesis procedure for reactive real-time embedded systems. In our approach, control parts of the system are represented in a decomposed form enabling more complex control structures to be represented. We propose a synthesis procedure for this representation that incrementally aggregates elements of the representation while keeping the resulting code size under tight control. This method combined with heuristic strategies works very well on real-life designs and demonstrates the potential to produce results that challenge or beat hand-written implementations.

Keywords

Software Synthesis, Finite-state Machines, Embedded Systems, Real-time Systems

1. INTRODUCTION

A major bottleneck in the implementation of embedded systems is the development of software, its debugging, and its integration with the hardware components. The ability to analyze a system before a specific technology is chosen as a target implementation and thus to "get it right the first time" is of paramount importance to reduce the cost of development. This scenario fueled the quest for a design methodology that favors system-level description of the functionality and constraints, technology independent verification, and automatic optimized mapping from the system-level description and constraints to software and hardware implementation. Here, we focus our attention on automated software synthesis.

In order to understand our approach, we must informally distinguish between *specification languages* and *programming languages* and between *software synthesis* and *software compilation*. A specification language provides a high-level specification that describes the *function* that must be performed regardless of how it will be implemented.

A programming language, such as a C- or Pascal-like, imperative, non-concurrent language, has semantics that are very close to that of the implementation domain (assembly code or executable code) and in a sense it already describes the desired implementation of the software. Software compilation denotes an optimized translation process from a programming language to the implementation domain. Software synthesis denotes an optimized translation process from a specification language to a programming language.

Examples of software synthesis are the C code generation capabilities of systems based on finite state machines (FSM) such as those based on StateCharts [2], ESTERELStudio [1], or of programming environments such as SDL [10], ECL [8], and ESTEREL [7].

The main motivation for pursuing sofware synthesis is to provide designers with front-end tools and languages they can use to express a system's behavior in a way that is natural in that it is defined by the semantic domain of the application rather than dictated by the semantic domain of the implementation technology. This allows them to concentrate their efforts in specifying the desired behavior with little concern regarding the efficiency or cost-effectiveness of the implementation trusting that the automatic synthesis system will deliver a behaviorally equivalent implementation that satisfies the required cost and performance constraints. For example, a language such as ECL (which combines the reactive semantics of ESTEREL with the expressive power and data types of C) provides the user with primitive contructs for expressing synchronous concurrency and various forms of preemption along with a synthesis path that generates behaviorally equivalent C code. For someone who needs to design a highly concurrent complex real-time system, ECL has a definite advantage over C since otherwise the designer himself would have to come up with an implementation for concurrency and preemption based on C constructs, which would be time-consuming and error-prone, or RTOS support which may not be available on a typical embedded platform, or not be as well defined semantically.

The straightforward implementation of a FSM is a *flat* or *single-step implementation* consisting of a sequence of tests and assignments in which:

- a variable can be assigned or tested only once,
- a variable cannot be tested after being assigned, and
- a variable can be assigned only after all the variables it depends on have been tested.

Note that the same variable can be tested and assigned, and some tests may follow some assignments. However, any test on any given variable v must precede any assignments on that same v.

Conversely, in a *multi-step* implementation the final value of state variables and outputs is computed by degrees in that intermediate data can be stored in temporary variables that can be tested after being written, and more than once.

In the reactive real-time embedded systems, the category that we consider, the objective is to ensure sufficient execution speed within the available resources, namely data memory (normally RAM) and code memory (normally ROM). Execution speed is achieved by minimizing the number of steps performed. In general, a flat implementation will be faster and will use less data memory than an equivalent multi-step implementation but will use more code memory. [1] A multi-step implementation may be desirable if it allows to fit the program in the amount of memory available while maintaining acceptable performance.

2. RELATED WORK

In [4, 5] a software synthesis methodology is proposed that operates on an extended FSM model called *codesign FSM* (CFSM) [9] which extends classical FSMs with arithmetic and relational operators. Structurally, A CFSM is a directed network of one combinational control node (CTR), one or more combinational datapath nodes (DPN), and one or more registers in which the combinational part is acyclic. In the software synthesis process, reduced-order BDDs (ROBDD) are used to represent a CFSM's unique CTR in the form of its characteristic function as they are an efficient way to represent a relation that reflects the cost of the software implementation in term of size (as the number of BDD nodes relates to code size) and speed (as the depth of the BDD branches relates to the number of steps perfomed by the code along the various branches). By reducing the BDD size we reduce the size of the resulting code and the main problem is to find a good BDD variable ordering that does not violate dependency relations between variables i.e. each assignment cannot precede any test on which it depends.

However, single-CTR representations do not scale well at all to real-life designs as the BDD tends to grow exceedingly big mainly because of the fixed variable ordering which must be the same on every BDD path and thus on every branch of the resulting implementation. It is desirable to extend the approach and start from an enhanced CFSM representation that allows multiple CTRs connected either in parallel (i.e. driven by the same inputs) or sequentially (i.e. one node's output drive another node's input possibly through DPNs), and then incrementally combine nodes in the representation.

[1] A typical situation is a tree program in which two or more sub-trees can be merged by adding extra temporary variables.

Nodes connected in parallel define projections of the FSM on separate outputs and their composition is equivalent to computing a product FSM. Nodes connected in sequence define intermediate steps in the computation and their composition is equivalent to flattening those steps.

In [6], Balarin *et al.* present an algorithm that only considers combining nodes connected in parallel which translates into sharing computation steps that would otherwise be duplicated. That algorithm is effective at minimizing the code which implements the data path but does not address the size explosion derived from the collapsing of the CTRs.

3. OUR APPROACH

This work proposes an approch in which a CFSM's CTRs are connected either in parallel or sequentially with two objectives: (1) maintain an optimal variable ordering *on each branch* that produces a minimal implementation, and (2) never let the BDD size grow beyond a given acceptable limit. We create a set of possibly different BDD orderings by heuristically picking the first variable v in the order and creating a BDD manager for each value of v being used in the CFSM. Each CTR in the CFSM will be implemented in just one of the managers. Within each group of CTRs that share the same BDD manager (called a *region*) the composition of CTRs is done incrementally in pairs. In order to avoid the BDD explosion, each composition will take place only if the sizes of the BDDs to be combined are below a given set limit.

As in [4] and [5], here we make the simplifying assumption that each step in the implementation (as does each BDD node) has the same cost, and that each code path (as does each BDD path) has the same probability of being traversed. Since reducing the size of the BDD reduces its average depth, we only consider the the code size as cost function.

The rest of the paper is organized as follows. Section 4 details the CFSM model. Section 5 describes the primitive operations that can be performed on it. Section 6 describes the algorithm and the heurstics. Section 7 presents and discusses the experiments and the results. Finally, in Section 8 we draw the conclusions and outline possible future developments.

4. PRELIMINARIES

Structurally A CFSM is a directed network of combinational nodes and registers in which the combinational part is acyclic. Each node in the network can have multiple inputs and outputs. Each net has a single source, which can be either a primary input or an output of some node, and possibly many destinations which can be primary outputs or inputs of some nodes. A net can be either *trigger* or *data*. With trigger nets we associate two values: *present* and *absent*, while data nets can be associated with any finite set of values. If a node in the network has at least one trigger input, then we say it is a *control transition relation* (CTR). Otherwise, we say it is a *data path node* (DPN). Both CTRs and DPNs define a function from their inputs to their outputs. However, since some outputs may be *"don't-cares"* for some inputs, it is more precise to call them relations. To *evaluate* a DPN or a CTR for a given input means to find an output assignment that satisfies the relation represented by that node.

```
v = foo(v);
present (a) {
    v = 1;
}
present (b) {
    v = v + 1;
}
emit y(v);
```

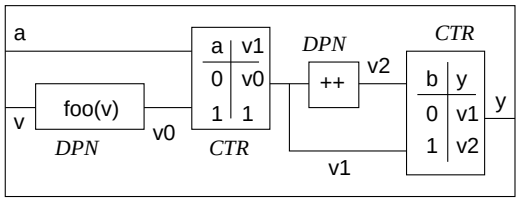

Figure 1: A simple CFSM

We require that CTRs be *reactive*: if there are no present triggers at their inputs, the trigger outputs should not be emitted (i.e. they should be absent) and data outputs should be "don't-cares".

A CFSM as a whole also defines an input-output function (or more precisely relation). To evaluate a CFSM for a given set of primary inputs, we need to evaluate all of its nodes in any topological order.

We do not make any assumption on how DPNs are specified or represented. A CTR defines a relation between its inputs and outputs $R = R(I_1, \cdots I_n, O_1, \cdots O_m)$. In the following, we will refer to the CTR or its characteristic function by the same name. Figure 1 depicts a simple CFSM and a fragment of ECL code that explains its behavior.

For the kind of processing that will be used in this work, it is useful to represent a CTR as

$$R = \mathcal{F}_R(x, s) \cdot \mathcal{M}_R(s, g_1, \cdots g_{last}, v) \tag{1}$$

where $\mathcal{F}_R$, called *selecting function* is a a relation that maps the input x to a selection variable s, and $\mathcal{M}_R$ is a multiplexer that selects one g_i as the value of v for a given value of s.

$$\mathcal{M}_R(s, g_1, \cdots g_{last}, v) = \sum_i [(s \equiv i) \cdot (y \equiv g_i)] \tag{2}$$

The relation R is defined by the intersection of $\mathcal{F}_R$ and $\mathcal{M}_R$. However, the representations of the two cannot be combined (namely the BDD for the composition will not be constructed) since $\mathcal{F}_R(x, s)$ represents a relation between boolean values of x and non-negative integer values of s, and $\mathcal{M}_R(s, g_1, \cdots g_{last}, v)$ is a mapping between each value k of s and a *function* $g_k = g_k(z, x)$ that defines the value of v, where z is a set of ouputs of some other CFSM node.

We will use the notation $\mathcal{F}_R \diamond \mathcal{M}_R$ to indicate the implicit intersection of the two.

$$R = \mathcal{F}_R(x, s) \diamond \mathcal{M}_R(s, g_1, \cdots g_{last}, v) \tag{3}$$

Figure 2 depicts the internal structure of a CTR. Semantically, the formulae (1) and (3) are equivalent to the simpler form below from which formula (1) can be derived by applying the Shannon decomposition.

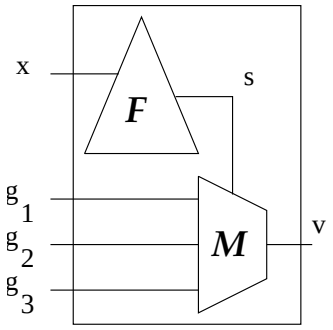

Figure 2: Structure of a CTR

$$R = \exists_s[(v \equiv g_s) \cdot (\mathcal{F}_R(x, s))] \tag{4}$$

The $\mathcal{F} \diamond \mathcal{M}$ form is not used for binary outputs since the multiplexer $\mathcal{M}$ can be easily rewritten as a trivial two-case multiplexer of the form $s \cdot v + \bar{s} \cdot \bar{v}$ and the selecting function is simply $\mathcal{F}_R(x, v) = R(x, v)$.

In a multi-output node there can be a shared selecting function which drives a $\mathcal{M}$ only for some of the outputs. That is, there can be a CTR R defined as

$$R(x, v_1, v_2) = \mathcal{F}_R(x, s_1, v_2) \diamond \mathcal{M}_R(s_1, g_1, \cdots, g_{last}, v_1) \tag{5}$$

where $\mathcal{F}_R(x, s_1, v_2) = \mathcal{F}_A(x, s_1) \cdot \mathcal{F}_B(x, v_2) = \mathcal{F}_A(x, s_1) \cdot B(x, v_2)$

5. PRIMITIVE OPERATIONS

The CFSM minimization algorithm that will be outlined later performs parallel and serial composition of CTRs. The parallel composition of two CTRs A and B is denoted by $A \circ B$. The sequential composition is denoted by $A \triangleright B$.

By definition, the parallel composition of two CTRs is the intersection of their characteristic function. When A and B do not have a multiplexer their parellel composition is thus simply the intersection of the two selecting functions, that is

$$A \circ B = \mathcal{F}_A \cdot \mathcal{F}_B \tag{6}$$

In the parallel composition of two $\mathcal{F} \diamond \mathcal{M}$ nodes only the selecting functions are to be composed.

$$A \circ B = (\mathcal{F}_A \cdot \mathcal{F}_B) \diamond [\mathcal{M}_A \parallel \mathcal{M}_B]$$

where the $\parallel$ operator denotes the fact that the multiplexers are independent of each other.

By definition, the serial composition of two CTRs connected by variables $v_1, \cdots v_n$ is the projection onto the variable space orthogonal to $v_1, \cdots v_n$ of the intersection of their characteristic function. Therefore, the sequential composition of two nodes $A(x, v)$ and $B(x, v, y)$, in the case A has no multiplexer, only involves the intersection of the selecting functions and elimination of v, that is

$$A \triangleright B = \exists_v(\mathcal{F}_A \cdot \mathcal{F}_B) \diamond \mathcal{M}_B \tag{7}$$

or

$$A \triangleright B = \exists_v(\mathcal{F}_A \cdot \mathcal{F}_B) \tag{8}$$

if also B does not have a multiplexer.

The sequential composition of two nodes A and B both in $\mathcal{F} \diamond \mathcal{M}$ form is a little more complicated. In this situation, one input case of the multiplexer of the driven CTR is the output of the driving multiplexer. (Note that, due to type consistency, an input of $\mathcal{M}_B$ cannot be driven by an output of $\mathcal{F}_A$'s). For example, let the factors be

$$\mathcal{F}_A(x,s) \diamond \mathcal{M}_A(s, g_1, \cdots g_{last_A}, v)$$

and

$$\mathcal{F}_B(x,z) \diamond \mathcal{M}_B(z, h_1, \cdots h_{last_B}, y)$$

where h_j=v. Their sequential composition is defined as

$$A \triangleright B = \mathcal{F}_{A \triangleright B}(x,u) \diamond \\ \mathcal{M}_{A \triangleright B}(u, h_1, \cdots h_{j-1}, g_1, \cdots g_{last_A}, h_{j+1}, \cdots h_{last_B}, y) \quad (9)$$

where $\mathcal{M}_{A \triangleright B}$ is a multiplexer obtained by replacing the input h_j with the list of possible values assigned to v by A, $\mathcal{F}_{A \triangleright B}(x,u) = \exists_s \exists_z (\mathcal{F}_A(x,s) \cdot \mathcal{F}_B(x,z) \cdot G_{A \triangleright B}(s,z,u))$, and $G_{A \triangleright B}(s,z,u)$ is a mapping that returns the value of the selection variable u of $\mathcal{M}_{A \triangleright B}$ for a given assignment of s and z. The straightforward definition of G is one that enumerates the cases of $\mathcal{M}_{A \triangleright B}$. That is

$$G = \sum_{u=1}^{last_A + last_B} \left(\begin{array}{l} (u < j) \cdot (u \equiv z) \\ + \\ (u \geq j) \cdot (u < j + last_A) \cdot (u \equiv z + s - 1) \\ + \\ (u \geq j + last_A) \cdot (u \equiv z + last_A - 1) \end{array} \right)$$

The implementation of $A \triangleright B$ can be optimized by eliminating the redundant h_k cases from $\mathcal{M}_{A \triangleright B}$, and substituting in G each value of u that selects a redundant case with the value of the unique case not removed. This optimization can reduce significantly the size of the multiplexer and the size of the BDD that represents the selecting functions.

A DPN driven by the output of CTR can be distributed over the inputs of the CTR. For example consider a net $w = f(v)$ where the driving net v is the output of a CTR $R = \mathcal{F}_R(x,s) \diamond \mathcal{M}_R(s, g_1 \cdots g_{last}, v)$. R defines a function $v = (x ? g_1 : g_2)$. The combination of R and f is

$$w = f(x ? g_1 : g_2)$$

which can be rewritten as

$$w = (x ? f(g_1) : f(g_2))$$

We will call this transformation of the CFSM structure *expression replacement*. It may be applied to the case in which one or more DPNs are driven by a CTR A and drive another CTR B so as to make A and B suitable for sequential composition.

6. THE ALGORITHM

The algorithm is sketched as follows:

1. The CFSM structure is partitioned into separate regions chosen in such a way that some regions are insensitive to some inputs or current state variables. Each region $r_{v_i \in K}$ is obtained by restricting the initial CFSM to some set $K = \{\cdots k_j \cdots\}$ of values of variable v_i. To be eligible, v_i must be *testable* i.e. be an input or a current state. For the partitioning to be acceptable, the regions must not overlap i.e. no CTR can be in more than one region. For feasibility, the number of values of v_i that yield a non-empty region must be small. For each region a separate BDD manager will be created. In each BDD manager the variables v_i will be ordered at the top and in the same order. All other variables may be reodered differently in each manager. (The default is to have an empty set of such v_i's, thus only one region and one BDD manager.)

2. In each region, the CTRs are combined incrementally in pairs. The candidates for the next parallel composition are found by a breath-first output-to-input traversal algorithm that computes the groups of CTRs that are *not* topologically ordered and that share at least one input. The candidates for sequential composition are found by a depth-first output-to-input recursive traversal algorithm that returns the first pair of CTRs that are topologically sorted via direct nets only (i.e. there does not exist a path of nets that reaches the driven CTR from the driving CTR that traverses a node outside the pair. Composing CTRs indirectly connected would result in a cycle). At any stage of the algorithm, we can apply expression replacement to any DPN that sits between two CTRs thus allowing those two CTRs to be composed.

3. Each CTR of the resulting CFSM is synthesized as C code with the same basic technique used in [5], but the blocks of code for all the CTRs are inlined in one function and are ordered consistently with the topological order of the CTRs in the CFSM.

The user can control some aspects of the process such as:

- Choose the variables v_i by which the partitioning is done (The default is none). Not surprisingly, for a design specified in FSM form, choosing the variable representing the graphical state as the partitioning variable produces good results as the FSM is designed to react to different inputs in different states.
- Set the order in which the various types of operations should be attempted and whether the selected operation should be repeated until convergence. [2]
- Set the maximum acceptable size (in number of BDD nodes) for two nodes to be composed.

Since the optimal ordering is CTR specific, in principle one could use a different BDD manager for each CTR and create a new manager every time a new combined CTR is created. That turns out to be extremely expensive in terms of memory use and CPU time and is not viable for anything but the simplest designs. Using a unique BDD manager for each region requires an ordering that maintains the combined size of all the BDDs that are built during incremental composition process within an acceptable size. We use a generic

[2]In the case of full collapsing the order of the operations is irrelevant to the final result but can influence the speed of the collapsing process.

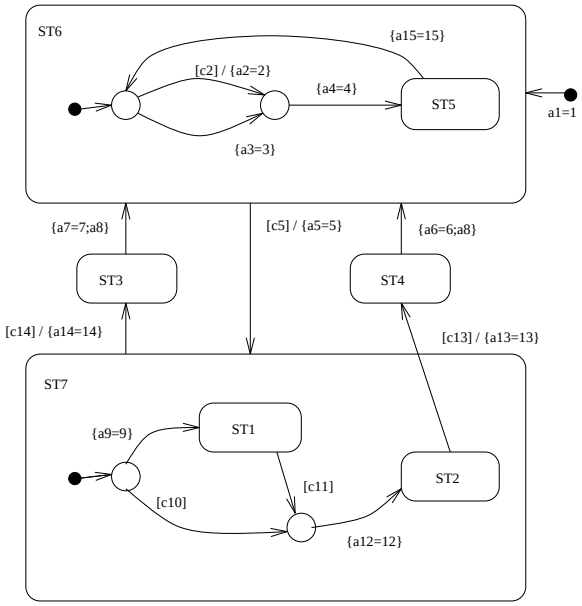

Figure 3: The Test1 example

heuristic criterion that is acceptable, if not optimal, for any generic design: the nets in CFSM are ordered topologically from input to output and the BDD variables are created in that order. This criterion guarantees that in each CTR in the original CFSM, as well as in each CTR that is created during the collaping process, no output will be ordered before its support. Dynamically reordering the BDDs while collapsing the CFSM is theoretically possible as long as it is constrained so as not to violate the property above - but it is currently not implemented. At the end of the collapsing process the BDD for each remaining CTR will undergo a constrained reordering right before the code generation step.

7. EXPERIMENTAL RESULTS

The examples are a mix of simple test cases and excerpts form real industrial applications. For each test, an ESTEREL program was created and compiled into a CFSM via an esperimental version of the ESTEREL compiler that generates a CFSM output. For each test, a a correponding hand-written implementation in C was written for comparison. The test machine was an IBM ThinkPad T21 equipped with a Pentium III running at 800 MHz. The various versions were all compiled with gcc from egcs-2.91.57 using the "-O3" optimization level. For each test, the table shows the number of CTRs in the initial CFSM representation, the number of CTR after the optimization, the object code size of the hand-written version we compare to, and the object code size of the code generated by the synthesizer. All the tests are synthesized as one region except for cruisecommL which was split in two regions based on the values of a state variable. In all tests, each region was fully collapsed.

The results indicate that our synthesizer has the potential to produce code that is comparable in size with, and sometimes better then a hand-written implementation.

In controlFuelA the synthesized version's big size is explained by the fact that the CFSM was generated from an ESTEREL program that made liberal use of *await* statements. The ESTEREL compiler creates a one-hot encoding of the resulting state machine (one boolean state variable per await, where states in which more than one such variable is set are unreachable) with consequent explosion of the next-state relation and hence of the synthesized code. The situation improves slightly by perfoming state re-encoding (via UC Berkeley's *sis* [3]) which reduces the number of reachable states, but since such re-encoding is done in the boolean space the number of tests on the state variables is still high and that affects the code size. The same considerations apply to u1pulse and sch1pulse. The initial ESTEREL program for controlFuelF, behaviorally equivalent to controlFuelA, uses a single await statement within a loop which translated to a single state variable used to encode the system's implicit state.

Example	Initial CTRs	Final CTRs	Size hand	Size auto
abro	50	1	188	240
test1	94	1	642	640
test2	53	1	352	272
u1pulse	225	1	752	1456
sch1pulse	187	1	788	1584
controlFuelF	117	1	700	992
controlFuelA	99	1	584	4176
cruiscommL	592	2	2590	1424

Figure 4: Tests used in the experiments

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a software synthesis approach based on BDDs and CFSMs that works well on real-life designs and has the potential to beat hand-written implementations. By allowing the construction of BDDs in separate managers we create implementations that can visit variables in different orders on different branches and thus are more compact and efficient than those based on a single ROBDD. By incrementally combining nodes in the CFSM we prevent size explosion.

There are many possible developments of the the approach described in this paper which will need to be explored such as:

- Resolving the ROBDD limitations by using, in the incremental composition process, either (1) a non-BDD internal representation for CTRs while still using BDDs for the final synthesis performed on the resulting set of CTRs, or (2) a BDD model more flexible then ROBDD to achieve better variable ordering on a per-branch basis.
- Testing our methodology extensively and comparing the results to other automatic synthesis tools, both commercial and academic.
- Linking our synthesizer to alternative front ends not based on the ESTEREL compiler e.g. tools based on XMI.
- Integrating the incremental composition approach with run-time scheduling of DPNs.

Acknowledgments

I would like to thank Ellen Sentovich, Luciano Lavagno, Felice Balarin, Roberto Passerone, Alberto Sangiovanni-Vincentelli, and Jerry Burch for their help and our many useful discussions.

9. REFERENCES

[1] Information on EsterelStudio available at *http://www.simulog.fr*.

[2] D.Harel et al. Statecharts: A visual formalism for complex systems. *Scientific Computer Programming*, pages 231–274, July 1987.

[3] E.Sentovich et al. *SIS: A System for Sequential Circuit Synthesis*. Technical Report UCB/ERL M92/41 University of California, Berkeley, California, 1992.

[4] F.Balarin et al. *Hardware-software co-design of embedded systems: the POLIS approach*. Kluwer Academic Publishers, Boston, 1997.

[5] F.Balarin et al. Synthesis of software programs for embedded control applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):834–849, June 1999.

[6] F.Balarin and M.Chiodo. Software synthesis for complex reactive embedded system. In *Proceedings of ICCD '99*, pages 634–639, 1999.

[7] G.Berry and L.Cosserat. The Esterel synchronous programming language and its mathematical. In *S. Brookes et al., Seminar on Concurrency*, pages 369–448. Carnegie-Mellon University, 1984.

[8] L.Lavagno and E.Sentovich. Ecl: A specification environment for system level design. In *Proceedings of 36th DAC*, pages 511–516, June 1999.

[9] M.Chiodo et al. A formal specification model for hardware/software codesign. In *Proceedings of the International Workshop on Hardware/Software Codesign*, 1993.

[10] R.Saracco et al. *Telecommunication Systems Engineering Using SDL*. North Holland/Elsevier, 1989.