Software Synthesis of Process-Based Concurrent Programs

Bill Lin
Electrical and Computer Engineering Department, University of California, San Diego, La Jolla, California, 92093-0407

Abstract— **We present a Petri net theoretic approach to the software synthesis problem that can synthesize ordinary C programs from process-based concurrent specifications without the need for a run-time multi-threading environment. The synthesized C programs can be readily retargeted to different processors using available optimizing C compilers. Our compiler can also generate sequential Java programs as output, which can also be readily mapped to a target processor without the need for a multi-threading environment. Initial results demonstrate significant potentials for improvements over current run-time solutions.**

I. Introduction

SOFTWARE is playing an increasingly important role in embedded systems. While high-level language compilers exist for implementing sequential programs on embedded processors [17]. e.g. starting from C [10], many embedded software applications are more naturally expressed as concurrent programs, specified in terms of communicating processes. This is because typically actual system applications are composed of multiple tasks.

Currently, the most widely deployed solution is to use an embedded operating system to manage the run-time scheduling of processes and to handle the inter-process communication. However, this solution tends to add significant overhead in program size, run-time memory requirements, and execution time.

Several alternative high-level approaches have been proposed. Static data-flow solutions [2], successfully used to design DSP-oriented systems, achieve compile-time scheduling at the expense of disallowing conditional and non-deterministic execution. Other researchers have considered hybrid approaches [7], [18] that generate application-specific run-time schedulers to handle the multi-tasking of conditional and non-deterministic computations. Reactive approaches, e.g. Esterel [1], rely on a strong synchrony hypothesis that makes two fundamental assumptions: the existence of a global clock abstraction to discretize computation over instances, and computation takes no time within each instance. This hypothesis is difficult to satisfy for distributed implementations and may not match naturally to many applications from a specification standpoint.

In contrast, our work is based on a model of *asynchrony* where the concurrent parts can evolve independently and only synchronize where specified. Recently, we introduced a new Petri net theoretic software synthesis method based on a new Petri net theoretic technique that can synthesize efficient embedded software implementations from asynchronous process-based specifications without the need for a run-time scheduler [11]. This approach has been implemented in a system under development called *Picasso*. In this paper, we further develop on our approach. We also briefly describe a new Java-based implementation, in addition to our earlier C-based implementation.

The remainder of this paper is organized as follows: Our specification model is introduced in Section II. The Petri net intermediate representation is introduced in Section III. The software synthesis method is detailed in Section IV. Initial results from an encryption example are presented in Section V.

II. Specification

Our programs are hierarchically composed of processes that communicate through synchronizing channels. The semantics is based on the CSP formalism [9], but the syntax is similar to C. Consider the following simple example composed of two processes called `ping` and `pong`.

```
1 ping (input chan(int) a, output chan(int) b) {
2   int x;
3   for (;;) {
4       x = <-a; /* receive */
5       if(x < 100) x = 10 - x;
6       else x = 10 + x;
7       b <-= x; /* send */
8 } }

9 pong (input chan(int) c, output chan(int) d) {
10  int y, z = 0;
11  for (;;) {
12      d <-= 10;/* send */
13      y = <-c; /* receive */
14      z = (z + y) % 345; /* send */
15 }}

16 system ( ) {
17   chan(int) c1, c2;
18   par {
19       ping (c2, c1);
20       pong (c1, c2);
21 } }
```

Channels are declared using the `chan` statement, as exemplified in `Line 1`. The unary receive operator, `<-`, receives data on the channel specified as its right operand. The received value may then be manipulated by other operators, e.g. it can be assigned to a variable, as exemplified in `Line 4`. The send operator, `<-=`, transmits the result of the expression provided as its right operand on the channel specified as its left operand, as exemplified in `Line 7`. Basic control-flow constructs, like `if-then-else`, `for-loops`, and `while-loops`, and basic arithmetic and relational operators, like `+`, `-`, `*`, `%`, and `>`, `>=`, `==`, `!=`, are the same as in C. There is also an `alt` construct [9], not used here, that provides a mechanism for non-deterministic execution. Finally, processes can be hierarchically composed to form larger systems, as exemplified by the process `system`. The `par` statement executes the statements in its body in parallel and joins the threads of execution at the end by waiting for all processes to terminate before proceeding.

III. Intermediate Representation

A. Petri nets

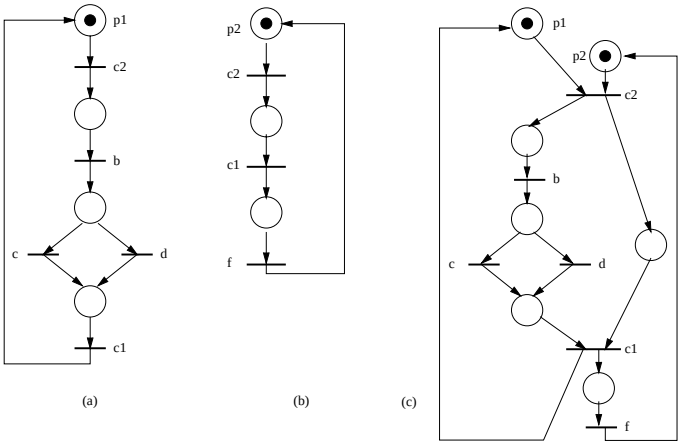
Let $G = \langle P, T, F, m_0 \rangle$ be a Petri net [14], where P is a set of places, T is a set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, and $m_0 : P \rightarrow N$ is the initial marking, where N is the set of natural numbers. The symbols $\bullet t$ and $t\bullet$ define, respectively, the set of input places and the set of output places of transition t. Similarly, $\bullet p$ and $p\bullet$ define, respectively, the set of input transitions and the set of output transitions of place p. A place p is called a *conflict place* if it has more than one output transition. Two transitions, t_i and t_j are said to be in *conflict* if and only if $\bullet t_i \cap \bullet t_j \neq \emptyset$. A state, or marking, $m : P \rightarrow N$, is an assignment of a non-negative number to each place. $m(p)$ denotes the number of tokens in the place p. A transition t can fire at marking m_1 if all its input places contain at least one token. The firing of t removes one token from each of its input places and adds a new token to each of its output places, leading to a new marking m_2. This firing is denoted by $m_1 \stackrel{t}{\rightarrow} m_2$. Given a Petri net G, the reachability set of G is the set of all markings reachable in G from the initial marking m_0 via the reflexive transitive closure of the above firing relation. The

DAC 98, San Francisco, California

corresponding graphical representation is called a reachability graph. A Petri net G is said to be *live* if $\forall t \in T$, $\exists m$ reachable from the initial marking m_0 such that t is enabled. It is said to be *safe* if in every reachable marking, there is at most one token in any place. In this case, we can simply represent each marking $m : P \rightarrow \{0, 1\}$ as a binary assignment.

B. Intermediate construction

In [4], [19], a process algebra was developed for constructing a Petri net model from a program of communication processes. Consider again the example presented in Section II. The derived Petri net models for processes `ping` and `pong` are shown in Fig. 1(a) and Fig. 1(b), respectively, along with their initial markings.



(a)		(b)		(c)	
c2:	x=<-a	c2:	d<-=10	c2:	x=10
b:	(x<0)	c1:	y=<-c	b:	(x<0)
c:	x=10-x	f:	z=(z+y)%345	c:	x=10-x
d:	x=10+x			d:	x=10+x
c1:	b<-=x			c1:	y=x;
				f:	z=(z+y)%345

Fig. 1. (a) `ping` (b) `pong` (c) `system = ping || pong`

Concurrent processes can be composed via *parallel composition*. In parallel composition, communication actions in fact form *synchronization points* and are joined together at their common transitions. This is illustrated in Fig. 1(c).

IV. Software Synthesis Method

A. Classes of Petri nets

A *Marked Graph (MG)* is a net $G = \langle P, T, F, m_0 \rangle$ such that $\forall p \in P : |\bullet p| = 1 = |p\bullet|$. MGs cannot model conficts.

A *State Machine (SM)* is a net $G = \langle P, T, F, m_0 \rangle$ such that $\forall t \in T : |\bullet t| = 1 = |t\bullet|$. SMs cannot model concurrency.

A *Free-Choice Net (FC-net)* is a net $G = \langle P, T, F, m_0 \rangle$ such that $\forall t_1 t_2 \in T, t_1 \neq t_2 : \bullet t_1 \cap \bullet t_2 \neq \emptyset \Rightarrow |\bullet t_1| = 1 = |\bullet t_2|$, or $\forall p_1 p_2 \in P, p_1 \neq p_2 : p_1\bullet \cap p_2\bullet \neq \emptyset \Rightarrow |p_1\bullet| = 1 = |p_2\bullet|$. Every MG and SM is a FC-net. For FC-nets, all conflicts can be decided *locally*.

Let G' be a subset of a net G generated by a *non-empty* set $X \subseteq P \cup T$. G' is a *MG-Component* of G if $\bullet t \cup t\bullet \subseteq X$ for every $t \in X$, and G' is a strongly connected MG.

Let G' be a subset of a net G generated by a *non-empty* set $X \subseteq P \cup T$. G' is a *SM-Component* of G if $\bullet p \cup p\bullet \subseteq X$ for every $p \in X$, and G' is a strongly connected SM.

G is said to be *covered* by a set of MG-Components if every transition of the net belongs to some MG-Component. G is said to be *covered* by a set of SM-Components if every place of the net belongs to some SM-Component. Hack [8] proved that a live safe FC-net can always be *covered* by a set of MG-Components or a set of SM-Components.

B. Expansions

Definition IV.1 (Expansion) An *expansion* is an acyclic Petri net with the following properties:

- There is one or more places without input transitions.
- There is one or more places without output transitions.
- There are no transitions without at least one input place or one output place.

The places without input transitions are called *initial places*. The places without output transitions are called *cut-off places*.

Definition IV.2 (Maximal expansion) Let G be a Petri net and let m be a marking of G. The *maximal expansion* of G with respect to m, E, is an acyclic Petri net with the following properties:

- The initial places correspond to m.
- The cut-off places correspond to the set of places encountered when a cycle has been reached.
- E is transitively closed: for each $t \in E$ or $p \in E$, all preceding places and transitions reachable from m are also in E.

m is referred to as the *initial marking*.

Intuitively, the maximal expansion of G with respect to a marking m corresponds to the largest *unrolling* of G from m before a cycle has been encountered. Consider the example shown in Fig. 2(a). The corresponding maximal expansion with $m = \langle p1, p2 \rangle$ is shown in Fig. 2(b).

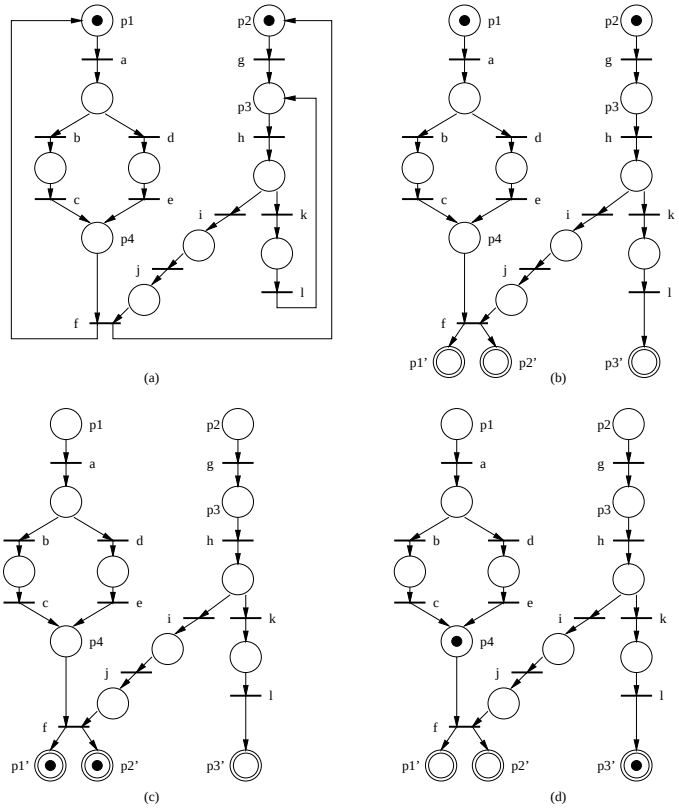


Fig. 2. (a) Petri net example. (b) Its maximal expansion. (c) A cut-off marking. (d) Another cut-off marking.

Definition IV.3 (Cut-off markings) Let G be a Petri net, and let E be a maximal expansion of G with respect to the initial marking m. A marking m_c is said to be a *cut-off marking* if it is reachable from m and no transitions are enabled to fire. The set of cut-off markings is denoted by $CM(E)$.

For the example shown in Fig. 2, there are two possible cut-off markings $m_{c_1} = \langle p1', p2' \rangle$ and $m_{c_2} = \langle p3', p4 \rangle$, shown respectively in Fig. 2(c) and Fig. 2(d).

Our synthesis procedure works by generating code from a maximal expansion segment E obtained by using the initial marking m_0 as the initial marking for the expansion. Then from each cut-off marking

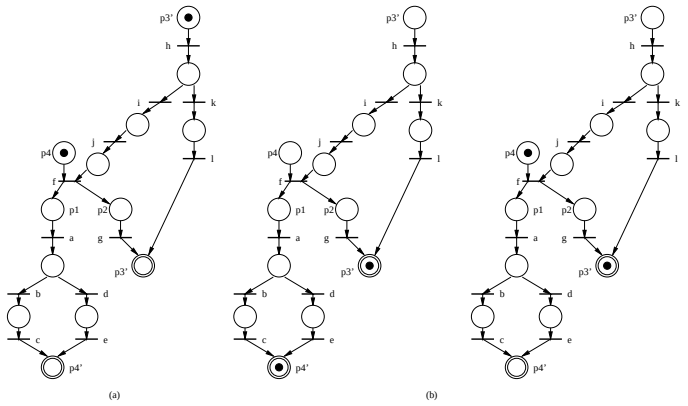


Fig. 3. (a) Maximal expansion. (b) Cut-off marking.

$m_{c_i} \in CM(E)$, a new maximal expansion segment E_i is generated using m_{c_i} as the initial marking. This iteration terminates when all cut-off markings have already been visited. The pseudo-code for the overall algorithm is shown below.

```
soft-synt (G, m0) {
   EM = {m0};
   push (m0);
   while ((m = pop()) ≠ ∅) {
      E = maximal-expansion (G, m);
      static-scheduling (E, m);
      foreach mc ∈ CM(E) {
         if mc ∉ EM {
            EM = EM ∪ mc;
            push (mc);
}  }  }  }
```

The `static-scheduling` step is applied to each expansion segment to produce the actual code.

In the example shown in Fig. 2, only two expansion segments are needed. From the initial marking $m = \langle p1, p2 \rangle$, the only cut-off markings reachable are $m_c = \langle p1, p2 \rangle$ and $m_c = \langle p3, p4 \rangle$. However, from $m = \langle p3, p4 \rangle$, the only cut-off marking reachable is $m_c = \langle p3, p4 \rangle$ itself, as shown in Fig. 3.

However, in the example shown in in Fig. 1, only *one* expansion segment is needed since the only cut-off marking reachable from the initial marking is the initial marking itself (i.e. $m = \langle p1, p2 \rangle$)[1].

C. Properties

The expansion procedure described in Section IV-B is guaranteed to converge since the number of possible markings in a Petri net is finite. Hence, the number of expansions or iterations is also finite. Typically, very few expansions are required.

For certain classes of Petri nets, the convergence property is even stronger. In the case of a strongly connected live safe MG, the number of expansions is exactly one. This is because in the case of a strongly connected live safe MG, the initial marking m_0 forms a minimal feedback arc set. The number of tokens along any directed cycle in the MG in the initial marking is exactly one. Thus, according to Definition IV.2, the maximal expansion of a MG G with respect to its initial marking m_0 is exactly defined as the acyclic Petri net E where both the initial places and the cut-off places correspond exactly to the places marked by m_0. Thus, the set of cut-off markings for E contains only the initial marking m_0.

In the case of a strongly connected live safe FC-net G that can covered by a set of strongly connected live safe MG components $G_1 \ldots G_n$ such that the initial marking m_0 of G restricted to G_i is also a live safe initial marking for the MG component G_i, the number of expansions is also exactly one. The argument follows a similar line as the argument for the MG case. That is, the initial marking m_0 corresponds to both the initial places and cut-off places if we maximally expand G with respect to m_0. Thus, convergence is guaranteed after one expansion since the set of cut-off markings contains only m_0.

[1] Here, we do not distinguish between p_i and p_i' because they simply denote different instances of the same place.

D. Static scheduling

Give an expansion segment E, represented as an acyclic Petri net fragment, our software synthesis method performs a *static scheduling* of the operations[2] in that segment. During scheduling, a *step* assigned to every operation in E. More formally, static scheduling is defined as follows:

Definition IV.4 (Static scheduling) Let E be an expansion segment. t_i is said to *precede* t_j in E, denoted as $t_i \prec t_j$, if there is a directed path from t_i to t_j. Let $\pi : T \rightarrow N$, be a *schedule function* that assigns a non-negative integer $\pi(t) \in N$ to every $t \in E$. A schedule is said to be *valid* iff it satisfies the following condition: $\forall t_i, t_j \in E$, if $t_i \prec t_j$, then $\pi(t_i) < \pi(t_j)$.

To illustrate this process, consider the expansion segment shown in Fig. 4(a) (corresponding to the example depicted in Fig. 4). Two valid schedules are shown in Fig. 4(b) and Fig. 4(c). It is not the intention of this paper to discuss in details the different possible scheduling heuristics. The interested reader can refer to [3], [5] for a survey of example techniques.

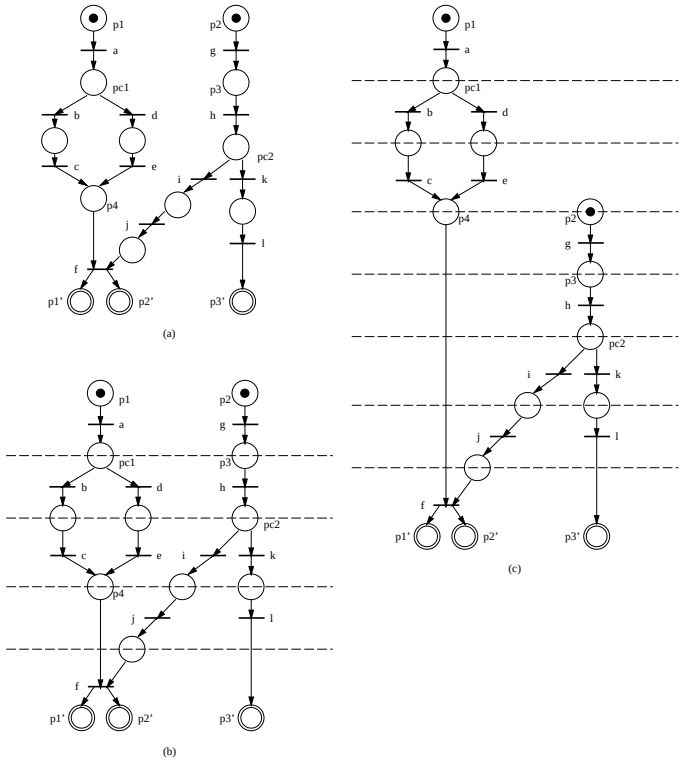


Fig. 4. (a) An expansion segment. (b) A valid static schedule. (c) Another valid static schedule.

Given a schedule π, a control-flow-graph fragment is constructed. In contrast to the traditional scheduling problem, where typically only *data-flow blocks* are considered, the control-flow-graph mapping step is much less straightforward. This is because we can have complex concurrent conditionals where the *firing* of a *transition* is dependent on the concurrent conflow flow and must obey Petri net firing rules. Essentially, the control-flow-graph generation step is based on a traveral of E, but we modify the Petri net firing rules so that we proceed in accordance to the levels defined by π. For example, the schedule shown in Fig. 4(b) will result in the control-flow-graph fragment depicted in

[2] Previously called pre-ordering [11].

Fig. 5(a). Similarly, Fig. 5(b) shows the resulting control-flow-graph for the schedule shown in Fig. 4(c).

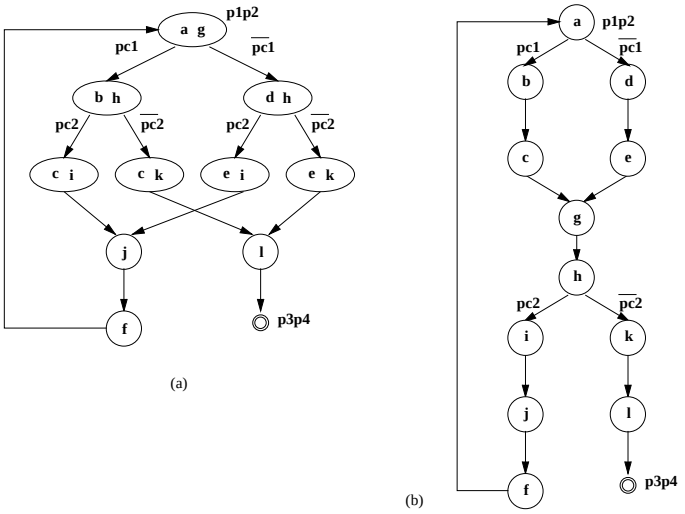


Fig. 5. (a) Control-flow-graph fragment. (b) Another control-flow-graph fragment.

E. Code generation

Once the overall control-flow-graph has been generated, it can be syntactically translated into plain C or sequential Java code for implementation. This last code generation step can leverage upon well-studied standard code optimization techniques [17].

V. Implementation and Results

A. Implementation

The synthesis method presented in this paper has been implemented in a system called *Picasso*. The compiler is implemented as a preprocessor that generates either plain C [10] or Java [6]. Both solutions are highly portable.

In the case of the C output, any available optimizing C compiler can be used to produce the target machine code. For comparisons, we have also implemented a multi-threading approach that uses the Solaris thread library to implement the processes. In principle, any real-time multi-threading packages may be used.

In the case of the Java output, the Java produced by our synthesis method is *sequential* in that it does not make use of any multi-threading feature in Java. Thus, a much lighter weight Java Virtual Machine without multi-threading support may be used. Also, any Just-In-Time compiler or Java-To-C translator (e.g. [12], [15]) can be used to produce native executables, again without the need for multi-threading support. For comparisons, we have also implemented a multi-threading approach in Java by mapping processes to Java threads.

B. Results

We have applied our C implementation to an example derived from the RC5 encryption algorithm. RC5 is widely used by RSA Data Security in a range of Internet security products [16]. A novel feature of RC5 is the heavy use of data-dependent rotations. The top-level view of the example is shown in Fig. 6. It consists of two encryption/decryption chains that are merged together by a monitor process.

We chose this example because it contains both data-dependent loops as well as non-deterministic choices. Table I compares the results of our method with a multi-threading library approach. The table compares the execution times of both approaches on different size input streams. The first row corresponds to a 40K bytes input file, the second row corresponds to a 400K byte input file, and the third row corresponds to a 4M byte input file. The CPU-times are reported in seconds

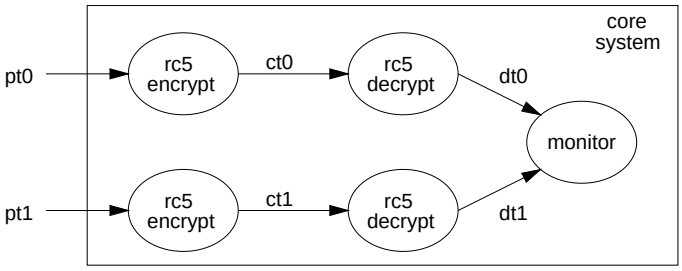


Fig. 6. RC5 encryption chain example.

size	threads	synthesis
40K	2.6	0.04
400K	26.0	0.33
4M	256.6	3.30
rate	15.4KB/s	1.21MB/s

TABLE I
Comparative results on a Sun/Solaris Ultra-1.

on a Sun Ultra-1 workstation running Solaris. The row labeled "rate" summarizes the execution of the two solutions in terms of bytes per second. Comparing CPU-times, the Solaris thread based implementation is significantly slower than our software code synthesis approach, due to the significant overhead introduced by Solaris threads.

References

[1] G. Berry et al. "The synchronous approach to reactive and real-time systems", *IEEE Proceedings*, 1991.

[2] J. T. Buck et al. "Ptolemy: A framework for simulating and prototyping heterogeneous systems", *International Journal on Computer Simulation*, January 1994.

[3] R. Camposano and W. Wolf (editors), *Trends in High-Level Synthesis*, Kluwer Academic Publishers, 1993.

[4] G. de Jong, B. Lin. "A communicating Petri net model for the design of concurrent asynchronous modules", *ACM/IEEE Design Automation Conference*, 1994.

[5] H. De Man, F. Catthoor, G. Goossens, J. Vanhoof, J. Van Meerbergen, S. Note, J.A. Huisken, "Architecture-driven synthesis techniques for VLSI implementation of DSP algorithms", *Proceedings of IEEE*, vol.72, no.2, pp.319-335, February 1990.

[6] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*, Addison-Wesley, 1996.

[7] R. K. Gupta. "Hardware-software cosynthesis of microcontrollers", *Proc. Codes/CASHE*, 1996.

[8] M. Hack. *Analysis of production schemata by Petri nets*. M.S. Thesis, MIT, February 1972.

[9] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[10] B. W. Kernighan, D. M. Ritchie. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[11] B. Lin. "Efficient compilation of process-based concurrent programs without run-time scheduling", *Proc. of DATE'98*, February 1998.

[12] B. Morgan. *Visual J++ Unleashed*. Sams.Net, 1996.

[13] J. Morse, S. Hargrave. "The increasing importance of software", *Electronic Design*, vol. 44, no. 1, Jan. 1996.

[14] J.L. Peterson. *Petri net Theory and Modeling of Systems*, Prentice Hall, 1981.

[15] T. A. Proebsing, G. Townsend, P. Bridges, J. H. Hartman, T. Newshan, S. A. Watterson. "Toba: Java for applications, a way ahead of time (WAT) compiler", *ftp://ftp.cs.arizona.edu/sumatra/report/toba.pdf*.

[16] R. L. Rivest. "The RC5 Encryption Algorithm", *Proceedings of the 1994 Leuven Workshop on Algorithms*, Springer-Verslag, 1994.

[17] R. M. Stallman, *Using and porting GNU CC*, Free Software Foundation, June 1993.

[18] F. Thoen et al. "Real-time multi-tasking in software synthesis for information processing systems", *Proc. of ISSS'95*, 1995.

[19] S. Vercauteren et al. "Derivation of formal representations from process-based specification and implementation models", *Proc. of ISSS'97*, September 1997.