Compositional Software Synthesis of Communicating Processes

Xiaohan Zhu, Bill Lin
Electrical and Computer Engineering Department
University of California, San Diego
9500 Gilman Drive, Mail Code 0407
La Jolla, CA 92093-0407

Abstract – In this paper, we describe a new compositional software synthesis method for synthesizing concurrent software programs into ordinary C programs so that they can be executed on embedded processors without the need for a run-time multi-tasking operating system. The synthesized C program can be readily retargeted to different processors using available optimizing C compilers. The method works by transforming the initial input specification into a set of interacting Petri net components. It then applies a quasi-static scheduling method on each Petri net component to produce a corresponding state machine model. The resulting set of interacting state machines are then mapped into a single C program, which can finally be compiled to native machine code via conventional C compilers. In the degenerate case, each process in the initial input specification is mapped into a separate Petri net component. In this case, the size of the resulting C program is directly proportional to the size of the original concurrent specification. Thus, this technique can scale well to large applications and is immune to code explosion.

1 Introduction

Embedded software is playing in increasingly role in system ASIC applications. This trend is driven by a wide spectrum of emerging applications, ranging from wireless communications systems, to consumer electronics, to automotive.

While high-level language compilers exist for implementing sequential programs on embedded processors, e.g. starting from C, 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 a multi-tasking 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. Handcrafted solution is another commonly used approach where concurrent programs are manually rewritten in terms of a sequential program by the designer. This approach is tedious, timing consuming, and hard to debug. The resulting code is often hard to read and maintain, and is usually extremely difficult to modify to accomodate specification changes.

Recently, we proposed a quasi-static scheduling technique for synthesizing concurrent software programs into ordinary C programs so that they can be executed without the need for a run-time multi-tasking operating system [6]. The synthesized C program can be readily retargeted to different processors using available optimizing C compilers. This technique first compiles the initial input specification into a single intermediate Petri net representation. It then statically schedules the operations on the Petri net via a process called expansion. The resulting solution is a single sequential state machine that can be syntactically mapped into C code for native machine code generation on to the target processor. In practice, this technique produces efficient results. However, theoretically, it is possible for the resulting state machine to become very large, hence resulting in code explosion. To circumvent this limitation, we have developed a new compositional software synthesis technique that first transforms the initial input specification into a set of interacting Petri net components. We then apply our quasi-static scheduling method on each Petri net component to produce a corresponding state machine model. We then map the resulting set of interacting state machines into a single C program, which can finally be compiled to native machine code via conventional C compilers. In the degenerate case, we map each process in the initial input specification into a separate Petri net component. In this case, the size of the resulting C program is directly proportional to the size of the original concurrent specification. Thus, this technique scales well to large applications and is immune to code explosion.

The remainder of this paper is organized as follows. In Section 2, we briefly outline our concurrent specification model, which is based on the CSP formalism [4]. We also briefly present Petri nets as our intermediate representation and the construction procedure. In Section 3, we summarize the quasi-static scheduling method presented in [6]. We then present our new compositional synthesis method in Section 4. Section 5 describes further the handling of the degenerate cases. Finally, we describe our implementation and some experimental results in Section 6.

2 Models

2.1 Specification

Our programs are hierarchically composed of processes that communicate through synchronizing channels. The semantics is based on the CSP formalism [4], but the syntax is similar to C. Below, a simple example composed of two processes called ping and pong is illustrated.

```c
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, $\langle \cdot \rangle$, 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, $\langle \cdot \rangle \rightarrow$, transmits the result of the provided expression 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 $\ast$, $\div$, $=$, and $\neq$, $\geq$, $\leq$, are the same as in C. There is also an alt construct [4], 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. This construct provides a mechanism for invoking concurrency.

## 2.2 Petri Nets and Intermediate Concurrency

Let $G = (P, T, F, m_0)$ be a Petri net [7], 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 $\circ t$ define, respectively, the set of input places and the set of output places of transition $t$. Similarly, $\circ p$ and $\bullet p$ 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 $m_1 \rightarrow t m_2$.

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 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. A Petri net $G$ 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.

In [2, 10], a process algebra was developed for constructing a Petri net model from a program of communication processes. Consider again the example shown above. 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.

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).

## 3 Quasi-Static Scheduling

In this section, we summarize the quasi-static scheduling algorithm presented in [6]. The algorithm is based on a systematic algorithm for generating acyclic Petri net segments from an initial cyclic Petri net representation. For each acyclic Petri net segment, a quasi-static scheduling procedure is applied to schedule the operations (transitions) in that segment. Based on the schedule, a portion of the state machine (representing the control-flow) is generated. After the overall procedure is completed, the resulting state machine is mapped to an ordinary C program for final machine code generation.

The procedure for systematically generating acyclic Petri net segment is based on a concept called maximal expansion. A maximal expansion is defined with respect to some initial marking $m$. Starting from this marking, we identify a set of places encountered when a cycle has been reached. These places are referred to as cut-off places. Intuitively, the maximal expansion of a Petri net $G$ 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 $m = \langle p_1, p_2 \rangle$ is shown in Fig. 2(b).

Let $G$ be a Petri net, and let $E$ be a maximal expansion of $G$ with respect to the initial marking $m$. A marking $m_i$ 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 $m_{c_1} = \langle p_1', p_2' \rangle$ and $m_{c_2} = \langle p_3', p_4 \rangle$, shown respectively in Fig. 2(c) and Fig. 2(d).

The overall 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 $m_{c_i} \in CM(E)$, 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.

In the example shown in Fig. 2, only two expansion segments are needed. From the initial marking $m = \langle p_1, p_2 \rangle$, the only cut-off markings reachable are $m_{c_1} = \langle p_1, p_2 \rangle$ and $m_{c_2} = \langle p_3, p_4 \rangle$. However, from $m = \langle p_3, p_4 \rangle$, the only cut-off marking reachable is $m_{c_3} = \langle p_3, p_4 \rangle$ itself, as shown in Fig. 3. However, in the example shown 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.
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). A valid schedule is shown. It is not the intention of this paper to discuss in details the different possible scheduling heuristics. The interested reader can refer to [1, 3] for a survey of example techniques.

Given a schedule $\pi$, a state machine fragment $S M_\pi$ is constructed. The control-flow-graph generation step is based on a traversal of $E$ using Petri net firing rules, but we modify the firing rules so that we proceed in accordance to the levels defined by $\pi$. For example, the schedule shown in Fig. 4(a) will result in the state machine fragment depicted in Fig. 4(b).

In constructing the state machine fragment for the schedule, we distinguish between two types of states: anchor states and non-anchor states. They are defined as follows.

**Definition 1** Let $E$ be an expansion with respect to an initial marking $m_{c_i}$, $C M(E)$ be the set of cut-off markings, $\pi : T \rightarrow N$ be the schedule function for $E$, $S M_\pi$ be the corresponding state machine fragment induced by $\pi$, and $sm_1, sm_2, \ldots, sm_k$ be the corresponding set of states in the state machine fragment $S M_\pi$. A state $sm_i$ is said to be an anchor state iff $sm_i = m_{c_i}$ or $sm_1 \in C M(E)$. It is said to be a non-anchor state otherwise.

---

**Footnote 1** Here, we do not distinguish between $p_i$ and $p'_i$ because they simply denote different instances of the same place.
In Fig. 4(b), anchor states are shown pictorially in double ovals, namely states \( p_1 p_2 \) and \( p_3 p_4 \). Once the overall state machine has been generated, it can be syntactically translated into plain C program for implementation. There are several ways to perform the syntactical mapping. One way, using a switch-case structure, is illustrated in Fig. 5. Using this construction, each case statement corresponds to an anchor state. Finally, we can leverage upon well-studied standard code optimization techniques [9] to perform the native code generation step.

4 Compositional Software Synthesis

The direct quasi-static scheduling procedure outlined in Section 3 can be summarized in Fig. 6(a). In this procedure, we compose all the processes in the system directly into a single intermediate Petri net representation. We then use static scheduling to transform the Petri net into a single sequential state machine model \( SM \) for implementation. The quasi-static scheduling procedure 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. The intermediate Petri net representation is usually comparable in size to the initial process-based specification. However, under some circumstances, the resulting state machine (representing the control flow) may be very large, hence causing the resulting C code to become very large in size as well. This problem can occur because the resulting state machine must represent explicitly the different combinations of states that the processes can be in with respect to the static schedule.

To circumvent this problem, we developed a new compositional software synthesis technique that first transforms the initial input specification with \( n \) processes, \( P_1, P_2, \ldots, P_n \), into a set of \( k \) interacting Petri net components, \( N_1, N_2, \ldots, N_k \), such that \( k \leq n \). In the degenerate case, \( k = n \), each process \( P_i \) in the initial specification is mapped to a separate Petri net component \( N_i \). We then apply our quasi-static scheduling method on each Petri net component to produce a corresponding state machine model. We then map the set of interacting state machines into a single C program for implementation. This is achieved by generating a time-loop in software that systematically steps through the interacting state machines without the need for a context-switching run-time operating system. This new design flow is depicted in Fig. 6(b). Currently, the choice of composition is decided by the user.

To facilitate communication between the interacting Petri net components, \( N_1, N_2, \ldots, N_k \), we first refine the communication channels between the components using a handshak-
The pseudo code for the state machine implementation and the main time-loop is shown in Fig. 9. The explicit state machine representation acts as a built-in preemption scheme. With the help of the time-loop in the main program, the compile-code implementation eliminates the need for run-time preemption, context switching, and scheduling, which are the major performance overheads in a multi-tasking operating system.

It is important to emphasize again that the resulting program is merely an ordinary C program without any specific system calls to any underlying multi-tasking operating system or the need for one. Hence, it is highly portable and only relies on a conventional optimizing C compiler to produce the final implementation.

Since the resulting program is not intended for human processing, it need not necessarily be readable. One useful optimization is to use goto statements instead of function calls. That is, each state machine model is associated with a label. Instead of making function calls in the main time loop, goto statements are used. This can result in slightly faster implementations.

5 Degenerate Case

In the degenerate case, each process is mapped to its own Petri net component. Specifically, we consider the degenerate case where each process in the initial specification is a sequential process (i.e., no par statement used internal to a process). For example, this is the case for the ping-pong example presented in Section 2. In this class of degenerate cases, we see that the resulting Petri net component after handshake expansion is actually already a state machine [7] because each transition has only one predecessor place and one successor place. While the corresponding Petri net component is already a state machine, we still need a procedure to determine which states are anchor states and which states are non-anchor states. Although the quasi-static scheduling procedure outlined in Section 3 can perform this task, we describe in this section a simple procedure that avoids the need to perform iterative expansions.

Consider the Petri net component shown in Fig. 8(a) after handshake expansion. This corresponds to the process ping described in Section 2. Note that this is a sequential model because there is only one token flowing through the model. Starting from the initial marking shown, we identify the maximal expansion and the corresponding set of cut-off places, as shown in Fig. 10(a). Instead of an iterative expansion and scheduling procedure, we can simply use the cut-off places directly as the anchor states. In this example, this is the case for the anchor states because there is only one transition in the resulting Petri net component after handshake expansion. This corresponds to the process ping-pong shown in Fig. 11.

The reduced state machine model is shown in Fig. 10(b), where the double circles represent cut-off places. This results in the state machine model shown in Fig. 10(b), where the double circles represent anchor states. In this example, the state p1 can be eliminated because its only output transition is a dummy transition ε. The reduced state machine model is shown in Fig. 10(c). This state machine model can be syntactically mapped to C code using a switch-case structure, as depicted in Fig. 11.

It is important to note that the size of the generated state machine (e.g. Fig. 10(b)) is directly proportional to the size of the corresponding Petri net component (e.g. Fig. 10(c)), which in turn is directly proportional to the size of the initial code description of the corresponding process. This means that the size of the resulting C program is also directly proportional to the size of the original specification. Thus, this
The synthesis method presented in this paper has been implemented in a system called *Picasso*. The compiler is implemented as a pre-processor that generates plain C [5]. Available optimizing C compiler can be used to produce the target machine code.

To evaluate the effectiveness of our new approach, we applied it to an example derived from the RC5 encryption algorithm that is widely used for Internet security applications [8]. RC5 is a fast symmetric block cipher that is suitable for hardware or software implementations. It provides a high degree of security, but yet is exceptionally simple. A novel feature of RC5 is the heavy use of data-dependent rotations. Since a full discussion of the RC5 algorithm is beyond the scope of this paper, the interested reader is referred to [8].

The top-level view of the example is shown in Fig. 12. It consists of an encryption-decryption chain. A stream of plaintext is read via the channel pt. Then the RC5 encryption algorithm is applied on it to produce a stream of ciphertext at channel ct. The the RC5 decryption algorithm is applied to the ciphertext to decode it back to plaintext again, along channel dt.

We chose this example because it contains data-dependent loops. Table 2 compares the results of the new compositional method with the direct static scheduling approach [6] and a multi-tasking approach. For the multi-tasking approach, we used the Solaris thread library.

Table 2 compares the execution times of all three approaches on different size input streams. The columns are labeled Single, Composition, and Threads, respectively. For the composition-based approach, we mapped each process to its own separate Petri net component. That is, we used the degenerate case as our composition strategy.

The first row corresponds to a 0.5M byte input file, the second corresponds to a 2M byte input file, and so on, with the largest input size of 512M byte. The CPU-times are reported in seconds on a Sun Ultra-2 workstation running Solaris. The row labeled “rate” summarizes the execution of the three solutions in terms of bytes per second. Comparing CPU-times, the composition-based approach is comparable to the direct static-scheduling approach. However, the composition-based approach is able to circumvent potential code explosion. The Solaris thread based implementation is slower due to the overhead introduced by multi-tasking and context switching.

### 7 Conclusion

We described a new compositional software synthesis technique that builds upon our recent quasi-static scheduling method [6]. It can circumvent potential code explosion problems by synthesizing a set of interacting state machines instead of a single state machine, as in our earlier method. In the degenerate case, the size of the resulting C program is directly proportional to the size of the original concurrent specification. Thus, this technique scales well to large applications and is immune to code explosion problems. Currently, we are exploring heuristics to decide on the composition. We are also currently exploring extensions to handle multi-rate systems.

### References


