Journal on Satisfiability, Boolean Modeling and Computation 5 (2008) 57-82

Dependence Graph Based Verification and Synthesis of Hardware/Software Co-Designs with SAT Related Formulation

Masahiro Fujita fujita@ee.t.u-tokyo.ac.jp
Kenshu Seto seto@cad.t.u-tokyo.ac.jp
Thanyapat Sakunkonchak thong@cad.t.u-tokyo.ac.jp
VLSI Design and Education Center,
The University of Tokyo,
Japan

Abstract

Program slicing is a software-analysis technique that generates System Dependence Graphs (SDGs) by which dependencies among program statements can be identified through their traversal. We have developed a program slicing tool for SpecC, a C-based system level design language for hardware/software co-designs, on top of a program slicer for C/C++. This program slicing tool can generate SDGs from any combined descriptions in C, C++, and SpecC, and can be used to analyze design descriptions for hardware/software co-designs uniformly and smoothly in all the combined descriptions. In this paper, after reviewing our program slicer that generates SDGs, we present verification and synthesis techniques for hardware/software co-designs through various analyses on SDGs and generations of SAT/ILP problems from them. For analysis and verification of the combined descriptions, we examine SDGs statically by traversing them with SAT/ILP solvers as verification engines. With this method, many static checks can be efficiently realized even if the target design descriptions are fairly large. We first present techniques for checking synchronization among concurrent processes described in SpecC through symbolic analysis on SDGs. The techniques could verify the synchronization of MPEG4 descriptions with about 48,000 lines within 10 seconds. The techniques can be applied to automatic conversions between sequential and parallel computations. One such application to automatic program serialization is presented. By using the technique for automatic program serialization, we could serialize Vocoder descriptions with about 10,000 lines in around 1 minute. As for synthesis from the combined descriptions, we present an optimal code generation method based on SAT formulation. It can generate codes for reconfigurable processors with minimum code lengths. The sizes of problems as SAT formulation range from 10,000 to 100,000 variables and 100,000 to 500,000 clauses for the largest configurations. With appropriate uses of the state-of-the-art SAT solvers and related tools, we show that fairly practical sizes of verification and synthesis problems can be solved by analyzing SDGs generated from the combined descriptions.

Keywords: *SAT-solver, lookahead, equivalence reasoning, local learning*

Submitted February 2007; revised May 2008; published June 2008

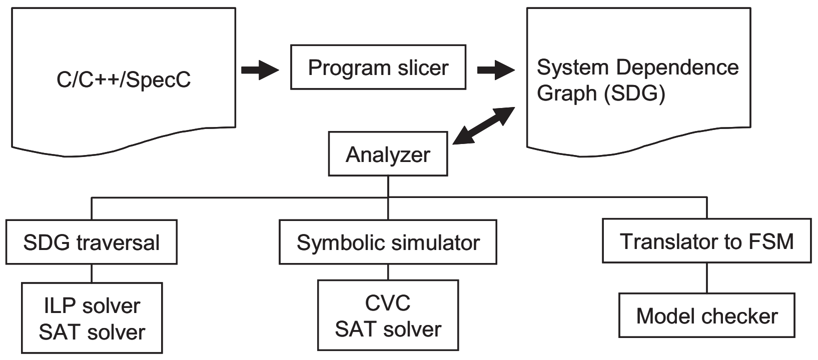


Figure 1. Overall synthesis and verification flow

1. Introduction

In system-level designs of electronics systems, both hardware and software must be taken into account. Since the standard ANSI-C language cannot effectively describe hardware parts, the SpecC language [1] has been proposed as an extension over the ANSI-C language for system-level design descriptions. SpecC, which is syntactically a super-set of ANSI-C, provides functions to describe system-level designs effectively and efficiently. For smooth and uniform hardware/software co-designs, it is very important to support descriptions in any of C, C++ and SpecC languages, and we are developing a uniform representation for those language descriptions. It is based on System Dependence Graphs (SDGs) that have been used in program slicing technology in the program analysis field [2]. Our verification and synthesis techniques work on extended SDGs [3] and other related data structures.

Figure 1 shows an overview of our verification and synthesis frameworks for combined design descriptions in C/C++/SpecC. The input to our flow is any design description in C, C++ and SpecC. We use program slicer [3] to generate System Dependence Graph (SDG). Our flow includes three types of analyses of SDGs for synthesis and verification of the design description: SDG traversal, symbolic simulation and translator of SDGs to finite state machines. Each analysis uses verification engines such as ILP solver and SAT solver.

In this paper, we describe the methods for the verification and synthesis for system-level designs that were published by the authors [3][13][15][11]. The first method is based on traversals of SDGs and mainly checks static aspects of the design descriptions. In this case, the static checking problems are converted into SAT formulae or integer linear programming problems. The second method is to check various partially dynamic behaviors of the design descriptions by symbolically simulating SDGs. An example of such partially dynamic behavior analysis is to check synchronization of concurrent processes. In SpecC, “wait event” and “notify event” statements are used for such synchronization, and we present a method to check if they are consistently synchronizing with each other or not. The method could verify the synchronization of MPEG4 descriptions with about 48,000 lines within 10 seconds. A method for automatic program serialization using the second method is also presented, and we could serialize Vocoder descriptions with about 10,000 lines in around 1 minute. The last method is to automatically generate optimum program codes for reconfigurable architectures from SDGs. The last method first generates finite state machine models from SDGs and then uses them to synthesize optimum program codes by using a SAT-based formulation. The last method can handle DFGs with up to around 20 operations. The sizes of problems as SAT formulation range from 10,000 to 100,000 variables and 100,000 to 500,000 clauses for the largest configurations.

This paper is organized as follows. In the next section we briefly review our program slicer for C/C++/SpecC combined descriptions. Then we present the three methods shown in Figure 1 in Sections 3 and 4. The last section gives concluding remarks including future directions.

2. Background

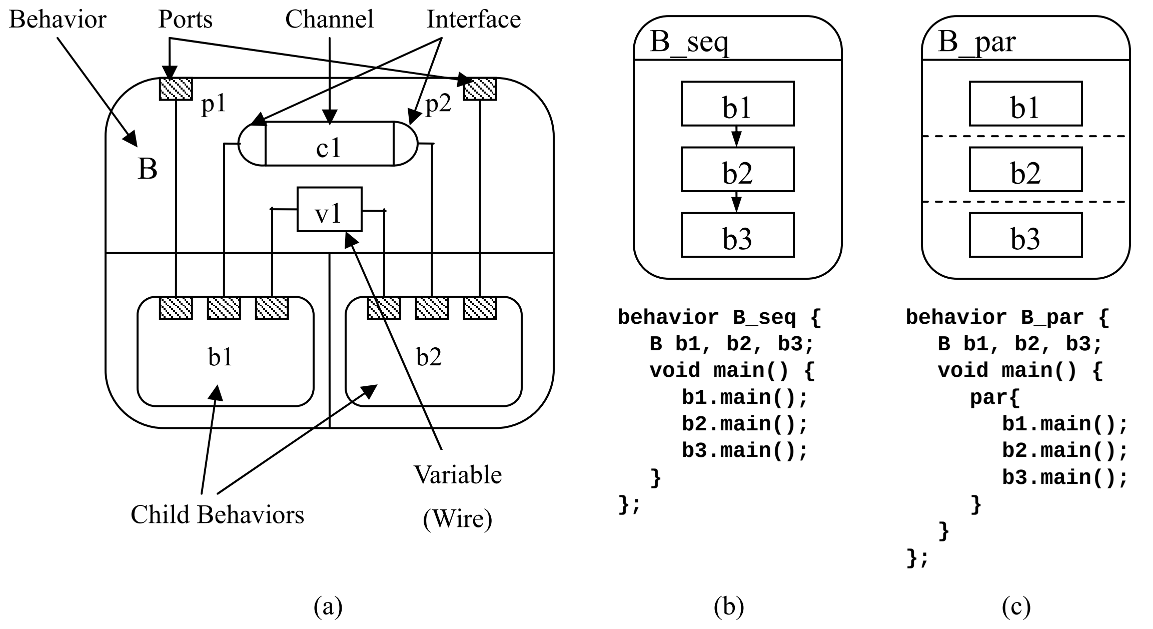
2.1 The SpecC Language

The SpecC language [1] has been proposed as a standard system-level design language for adoption both in industry and academia. It has been promoted for standardization by the SpecC Technology Open Consortium (STOC, `http://www.SpecC.org`). The SpecC language was specifically developed to address the issues involved with system design, including both hardware and software. Built on top of C, the de-facto standard for software development, SpecC supports additional concepts needed in hardware design and allows IP-centric modeling. Unlike other system-level languages, the SpecC language precisely covers the unique requirements for embedded systems design in an orthogonal manner. In SpecC, the *par* construct allows parallel behaviors to be expressed. For example, $par\{a.main(); b.main();\}$ in Figure 6 indicates that threads a and b are running concurrently (in parallel). Within each thread, statements run in the sequential manner just as in the C programming language.

2.1.1 Behaviors, Channels, and Interfaces

A SpecC behavior is a class consisting of a set of ports, a set of component instantiations and a set of private variables and functions. In order to communicate, a behavior can be connected to other behaviors or channels through its ports or interfaces. A channel in SpecC is a class consisting of a set of private variables and functions and is used to define a communication protocol. The structural hierarchy of such a behavior is shown in Figure 2(a). The sequential and parallel constructs of SpecC, which will be described next, are shown in Figure 2(b) and 2(c), respectively.

Figure 2(a) illustrates the basic features of the SpecC language. In the figure, behavior B hierarchically contains two child behaviors b1 and b2 that run in parallel. b1 and b2 communicate using the shared variable (or wire) v1 and the channel c1. The channel c1 provides two interfaces, and the interfaces are connected to the ports of b1 and b2. Other ports of b1 and b2 are connected to the ports of B. Figure 2(b) shows a sequential description in SpecC, and the child behaviors b1, b2 and b3 of the behavior B_seq are executed sequentially. Figure 2(c) shows a parallel description in SpecC, and the child behaviors b1, b2 and b3 of the behavior B_par are executed in parallel. The difference between the sequential and parallel SpecC descriptions is the $par\{\}$ construct.



```
behavior B_seq {
   B b1, b2, b3;
   void main() {
      b1.main();
      b2.main();
      b3.main();
   }
};
```

```
behavior B_par {
   B b1, b2, b3;
   void main() {
      par{
         b1.main();
         b2.main();
         b3.main();
      }
   }
};
```

Figure 2. (a) Basic structure of a SpecC model, (b) sequential description in SpecC, (c) parallel description in SpecC

2.1.2 Concurrency and Synchronization Semantics

Concurrency and synchronization among behaviors is handled in SpecC by the *par*{} and *notify/wait* constructs, as seen in Figures 6 and 7. In a single behavior running in isolation, correctness of the result is usually independent of the timing of its execution, and determined solely by the logical correctness of its functions. However, when several behaviors run in parallel, execution timing may have a great effect on the results' correctness: results can vary depending on how the multiple behaviors are interleaved. Therefore, synchronization between behaviors is an important issue for a system-level design language.

The *notify/wait* statements of SpecC are used for synchronization. A *wait* statement suspends its current behavior from execution and keeps waiting until one of the specified events is notified.

2.1.3 SpecC and SystemC

Where SpecC can be viewed as a super-set of C, SystemC is a C++ library extension. SpecC and SystemC share many features, e.g., behaviors (known as *Modules* in SystemC), ports, channels, interfaces, concurrency and synchronization[1].

In the next sub-section we will briefly describe our C/C++/SpecC program slicer. In addition, we are developing a new tool that can also support SystemC and RTL descriptions. Our preliminary results in this direction were published in [20].

1. The detail comparisons between SpecC and SystemC is presented in [19].

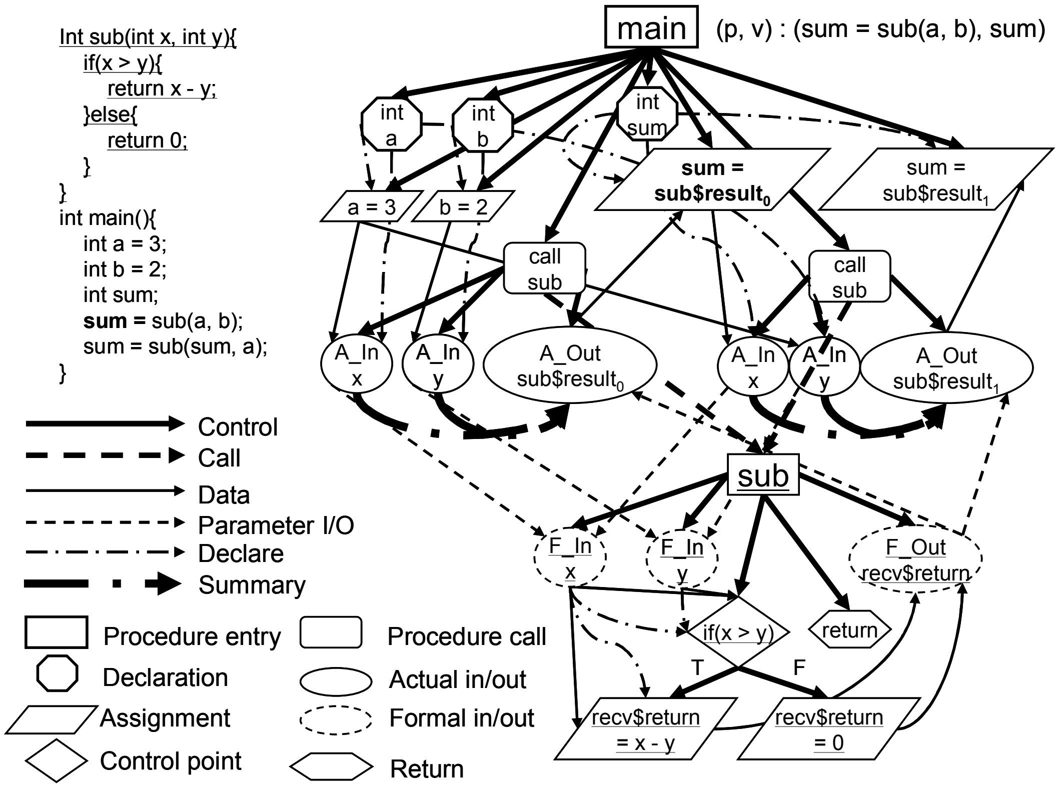


Figure 3. An example program of ANSI-C and its System Dependence Graph (SDG)

2.2 Program Slicing for C/C++/SpecC Descriptions

Program slicing is a technique to extract portions of the original programs which are relevant to the variables in some statements specified by the user of the program slicing tool.

As originally proposed by Weiser [7], slicing is computed given two parameters, program point p and the set of variables v which appear in p. Later, Ottenstein and Ottenstein [6] proposed a new method based on dependence graphs. In dependence graphs, a node represents a statement or an expression and an edge represents data or control dependence between statements. The control dependence represents conditions under which a statement is executed. The data dependence shows flow of data between statements. They constructed a dependence graph from a given program and identified the sliced codes from the variable v which is given by the user, by tracing data and control dependence edges in the graph. In this algorithm, the computation time of slicing increases linearly with the number of nodes in the dependence graph. In addition, Horwitz [2] et al. defined System Dependence Graphs (SDGs), which contain multiple Procedure Dependence Graphs (PDGs) and express dependencies between procedures. A PDG is a dependence graph that is created for each procedure. To model the dependences between procedures, SDGs use nodes related to procedure parameters, such as Actual in/out nodes and Formal in/out nodes, and edges related to the dependences between the parameters such as Parameter I/O edge. A formal in

Table 1. SpecC SDG's nodes and edges.

		Elements(*Additional element*)
Nodes	Entry	Function Entry, *Interface Entry*, *Channel Entry*, *Behavior Entry*
	Assignment	Assignment
	Control Point	Control Point (if, while, for, *par*, *wait*)
	Call Site	Function Call, *Instance Call*
	parameter in	Actual In, Formal In, *Member Actual In*, *Member Formal In*
	parameter out	Actual Out, Formal Out, *Member Actual Out*, *Member Formal Out*
	Return	Return
	Declaration	Declaration
Edges	Control	Control, Call, *Instance Call*, *Class Member*
	Data	Data, Parameter In, Parameter Out
	Declaration	Declaration

Note: Non-italic fonts represent the traditional ANSI-C nodes and edges in SDG.
Italic fonts represent the extra nodes and edges for representing SpecC in SDG.

node corresponds to a formal parameter of a procedure, and a formal out node corresponds to a formal parameter that may be modified. An actual in node corresponds to an actual parameter of a procedure at a call site, and an actual out node corresponds to an actual parameter that may be modified by the called procedure. SDGs also include summary edges to explicitly represent data dependences across procedure calls. A summary edge is connected from an actual in node N1 to an actual out node N2 when the value corresponding to N2 may depend on the value corresponding to N1. Based on the work that uses SDGs, several program slicing methods have been proposed including slicing for object oriented programs [5] and programs in JAVA with multiple threads. As an example, for the ANSI-C and C++ language, a commercial tool CodeSurfer [4] is provided by GrammaTech Inc.

Figure 3 shows an example of SDGs for the C program with the two functions: sub() and main(). The nodes represent the constructs of the C language and the edges show the dependences between the constructs.

As for slicing of programs that include concurrent executions, Clarke et al. developed a slicing tool for simulation-oriented hardware descriptions in VHDL [8]. They constructed a Control Flow Graph (CFG) from an original description in VHDL and generated an SDG of the program by analyzing dependencies on the CFG. The CFG consists of a set of nodes that represent statements in a program, and a set of arcs that represent flows of control. Our SpecC slicing [3] is also based on SDGs. The major difference between their work [8] and our work is the target language. Since the target of their work are RTL descriptions, their method cannot handle system-level design descriptions which are our target. In order to realize the SpecC program slicing, the most critical task is to represent programs as their SDGs. In the SpecC language, there are hierarchical structures such as the *behavior*, *channel* and *interface*, concurrent parallel execution syntax as *par*, and synchronization syntax as *wait* and *notify*. To address these SpecC language's features, we developed an SDG for SpecC with new nodes and edges such as the *Behavior Entry* node, as shown in Table 1.

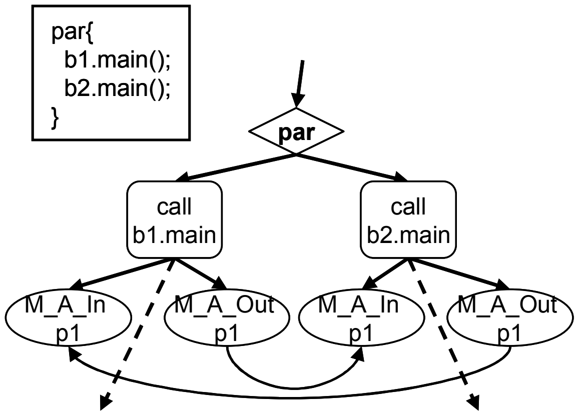


Figure 4. A dependence graph of *par*.

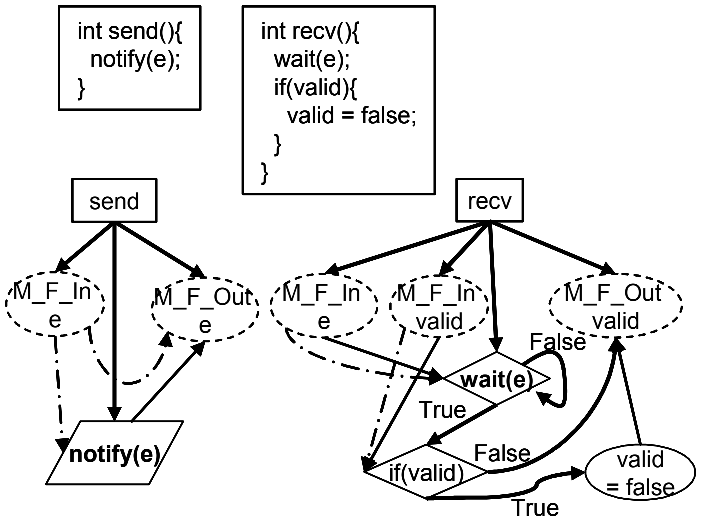


Figure 5. Dependence graphs of *wait* and *notify*.

To deal with concurrent executions realized by using *par* statements, we define a node for *par* as a control point node, similar to that of *if*, *while* and *for*. From *par* node, control dependence edges labeled **true** are drawn to every node corresponding to the statements that are executed concurrently under the semantic *par*. For example, in Figure 4, since *b1.main()* and *b2.main()* are executed concurrently, there must be control edges from *par* node to each of them. We also need the extra data dependence edges for representing the shared ports and parameters. In the figure, *b1* and *b2* are running in parallel and there is a shared variable *p1*, hence, the two data edges from *M_A_Out p1* to *M_A_In p1* are constructed. *call b1.main* and *call b2.main* nodes have outgoing call edges since the nodes are both calling function b1.main() and b2.main(). In Figures 4 and 5, the prefix M_ is used to represent member variables. For example, *M_A_Out* and *M_F_In* mean *Member Actual Out* and *Member Formal In*, respectively. Here, *Actual* parameters mean the parameters that are supplied by the caller, and *Formal* parameters mean the parameters that are declared in the function header.

In table 1, elements in non-italic letters define nodes and edges for ANSI-C's SDG in CodeSurfer [4], and we define new nodes and edges in italic letters for SpecC. For example, entry nodes that represent procedures are created for behaviors, channels and interfaces. In SDGs, Control Point nodes determine if a set of statements are executed or not. The SpecC construct representing the parallelism (*par*) is defined as Control Point nodes. This is because statements in *par*{} are executed only when the *par* is passed.

The *Wait* statement is also defined as a control point node in a dependence graph, and control dependence edges labeled **true** are constructed to every node executed after it. This is because whether *wait* is passed or not affects the executions of those statements, just like *if* or *for* statements. In addition, a data dependence edge of an event variable used in the *wait* statement is constructed, in order to build dependences between a *Notify* statement and its corresponding *wait* statements. Here, an event variable is a variable used as an argument of *Wait* statements or *Notify* statements.

Figure 5 illustrates dependence graphs of *Wait* and *Notify*. In Figure 5, there is a *control edge* from the *Wait* node to the control point node for *if(valid)*. The *Notify* statement is defined as an assignment node to an event variable. In the figure, a data dependence edge is constructed from *notify(e)* to the formal out node corresponding to the event variable *e*. In designs where channels are used for communication between parallel behaviors, event variables are communicated through *channels* connected to *behavior*'s ports, therefore we can traverse data dependence edges from *Notify* node to *Wait* node across *behaviors* and *channels* in the SDG.

3. Verification with Symbolic Simulation-Based Reasoning

This section describes our method to dynamically analyze concurrent behaviors or so called *synchronization verification* [13]. In system-level design languages such as SpecC, extra constructs are added to C in order to describe the characteristics of hardware. These extra constructs support description of parallel behaviors, pipelined behaviors, finite state machines, and operations on arbitrary-length bit-vectors. System-level models are organized as a collection of cooperating processes running in parallel. In order to keep all processes executing as the designer intended, proper scheduling of statement execution in all processes (known as *synchronization*) is necessary.

Our verification method is based on applying the state-of-the-art model checking with counterexample-guided abstraction refinement (CEGAR) [14] and constraints solving with integer linear programming (ILP) techniques. While classical automata can model the transitions of a design, these transitions convey no information about the delay between two actions. It is therefore not possible to directly model a design with timing constraints. Alur and Dill [16] proposed *timed automata* as a way to incorporate quantitative information on the passage of time in automata. Model checkers for timed automata have severe constraints on their capacity, so our approach is to capture timing constraints with equalities/inequalities that can be solved by integer linear programming (ILP) tools. In general,

```
main() {
   par{ a.main();
        b.main(); }}

behavior a {
   main() { x=10;        /*st1*/
            y=x+10;      /*st2*/ }}

behavior b {
   main() { x=20;        /*st3*/ }}
```

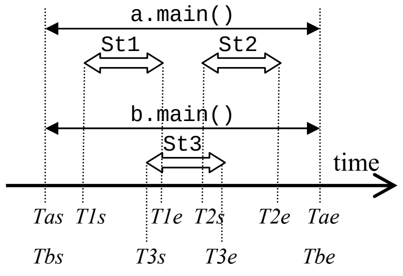
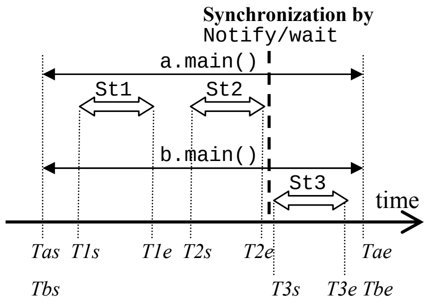


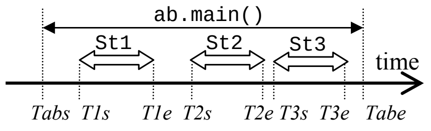
Figure 6. Timing diagram of the behaviors `a` and `b` under the `par`{}

```
main() {
   par{ a.main();
        b.main(); }}

behavior a {
   main() { x=10;        /*st1*/
            y=x+10;      /*st2*/
            notify e;    /*New*/}}

behavior b {
   main() { wait e;      /*New*/
            x=20;        /*st3*/ }}
```



(a)

```
behavior ab {
   main() { x=10;        /*st1*/
            y=x+10;      /*st2*/
            x=20;        /*st3*/}}
```



(b)

Figure 7. (a) Insertion of synchronization statement `notify/wait` of Fig. 6, (b) sequential description which is equivalent to the description in (a)

large processes are organized as collections of processes running in parallel. Without knowing how each process is interacting with other processes, verification of the entire systems may be extremely difficult and can be significantly time-consuming. Synchronization verification method as described in Section 3.1 focuses on checking whether all the processes are synchronized as intended. Instead of considering the entire design, we target only on the synchronization portions. Therefore, the size of verification problem can be reduced. After guaranteed that all synchronization is behaved as intended, a sequentialization which is a process to sequentialize concurrent processes into one single process will be explained in Section 3.2.

3.1 Synchronization Verification

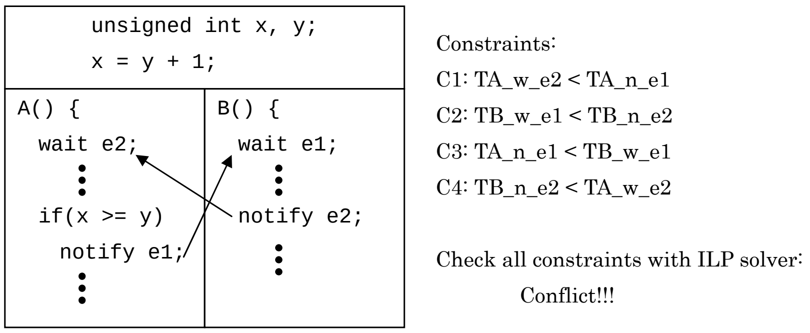
Let us describe how synchronization and constraints formulation can be done for verification. In SpecC, the computation and communication parts are clearly separated. Computation is encapsulated in *behaviors*, while communication is encapsulated in *channels*. Also, structural hierarchy is supported where the structure of the design can be described as a hierarchical network of behaviors and channels. Now let us consider *concurrency* in SpecC. Built on top of C, the de-facto standard for software development, SpecC supports additional concepts needed in hardware design and allows IP-centric modeling. Unlike other system-level languages, the SpecC language precisely covers the unique requirements for embedded systems design in an orthogonal manner. In SpecC, the *par* construct allows parallel behaviors to be expressed. For example, $par\{a.main(); b.main();\}$ in Figure 6 indicates that behaviors a and b are running concurrently (in parallel). Within each behavior, statements run in the sequential manner just as in the C programming language. The timing constraints that must be satisfied for the behavior a are $Tas \leq T1s < T1e \leq T2s < T2e \leq Tae$, where Ta, $T1$ and $T2$ stand for the timing of a, $st1$ and $st2$ respectively, and the postfix notations s and e stand for starting and ending time. In other words, $st1$ and $st2$ execute after a starts and before a ends, and no overlap is allowed in the execution of $st1$ and $st2$. All behaviors under $par\{\}$ are having the same starting and ending time. Hence, the timing constraints for starting and ending time of a and b are $Tas = Tbs$ and $Tae = Tbe$.

Note that it is not determined when $st3$ is scheduled relative to $st1$ and $st2$: any of "$st1 \rightarrow st2 \rightarrow st3$", "$st3 \rightarrow st1 \rightarrow st2$", and "$st1 \rightarrow st3 \rightarrow st2$" are allowed. In this case, an ambiguous result or an access violation error can occur since both $st1$ and $st3$ assign a value to the same variable x. The event manipulation statements in SpecC, $notify/wait$, can be used to synchronize behaviors a and b to achieve any desired scheduling. Figure 7(a) shows a modified version of Figure 6 with insertion of *notify/wait* statements. Let us focus on the $/*New*/$ labels in Figure 7 where the event manipulation statements are used. We can see that *wait* e prevents execution of $st3$ until the event e is notified by *notify* e. Due to sequentiality in behavior a, *notify* e is scheduled right after the completion of $st2$. The *notify/wait* pair therefore introduces the additional constraint $T2e < T3s$. That is, it is guaranteed that statement $st3$ is safely executed right after statement $st2$. This enforces the scheduling "$st1 \rightarrow st2 \rightarrow st3$". The timing constraint that describes this synchronization is $T2e < T3s$.

All timing constraints from Figure 6 can be represented as follows.

- $Tas <= T1s < T1e <= T2s < T2e <= Tae$
  (sequentiality in a)
- $Tbs <= T3s < T3e <= Tbe$
  (sequentiality in b)
- $Tas = Tbs$, $Tae = Tbe$
  (concurrency between a and b)
- $T2e < T3s$
  (synchronization of *notify* and *wait* statement forces $st3$ to be after $st2$)

Figure 9 shows typical synchronization examples which may be found in iteration bodies of loop-type statements in design descriptions. These synchronization problems can be analyzed as integer linear programming problems with the timing constraints on behaviors and statements as shown above and logical constraints derived from conditional statements



```
        unsigned int x, y;
        x = y + 1;

A() {                  B() {
  wait e2;               wait e1;
    ⋮                      ⋮
  if(x >= y)             notify e2;
    notify e1;             ⋮
    ⋮
```

Figure 8. A and B are running in parallel. Constraints $C1$ and $C2$ represent sequentiality in A and B, respectively. Constraints $C3$ and $C4$ represent synchronization of *notify/wait* of $e1$ and $e2$, respectively. After analyzing all constraints with ILP solver, there is a conflict due to the use of *wait e1* and *wait e2* prior to *notify e1* and *notify e2*.

in the design descriptions, such as if-statements. The conditional statements are modeled with constraints on integer variables. Since we are concentrating only on synchronization verification, we can abstract away portions of the design descriptions which are not related to synchronization statements. As a result, we can deal with fairly large design descriptions with the state-of-the-art ILP solvers. First, we can extract very small portions of the original design descriptions which directly influence the synchronization statements, such as "wait" and "notify". We call this an abstraction process[2] of the design descriptions. Then by using symbolic simulations we can generate ILP formulae to be checked. If the results of ILP solvers are negative, we proceed to the process of refinement of abstraction. For this refinement, we need to investigate dependencies among statements where SDGs provide very efficient mechanisms. This abstraction-refinement process may be repeated until we get real verification results. A complete discussion of abstraction refinement for model checking is beyond the scope of this paper. The interested reader is directed to [14, 13] for a detailed treatment of this subject.

Figure 8 shows a simple example that illustrates the use of CEGAR and ILP solver for synchronization verification. Both A and B are called under *par* and x and y are shared global variables. Note that, in Figure 8, we consider only the synchronization statements and omit others. Two pairs of synchronization points of events $e1$ and $e2$ where both *wait e1* and *wait e2* were placed prior to both *notify e1* and *notify e2*. There is also the $if(x >= y)$ guarded *notify e1*. As mentioned earlier, synchronization verification can be conducted in two steps. First, we use CEGAR to validate that every pair of *notify/wait* statements are eventually synchronized. Particularly, we are interested in validating the guarded condition of statement *notify e1*. The result from CEGAR tells that the condition $if(x >= y)$ is

2. Counterexample-Guided Abstraction Refinement (CEGAR) [14] is a method to automate the abstraction refinement process. More specifically, starting with a coarse level of abstraction, the given property is verified. A counterexample is given when the property does not hold. If this counterexample turns out to be spurious, the previous abstract programs are then refined to a finer level of abstraction. The verification process is continued until there is no error found or there is no solution for the given property.

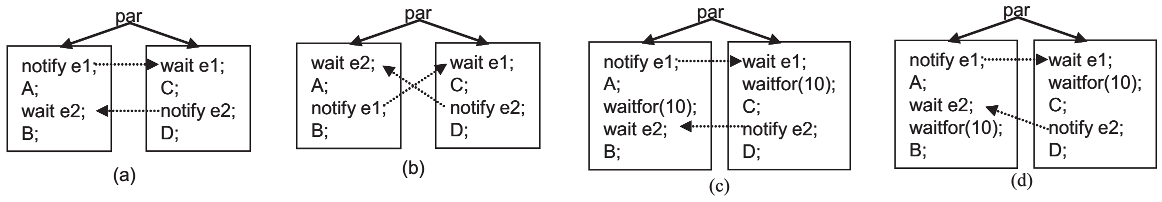


Figure 9. Synchronization verification with ILP solvers

always true ($x = y + 1$ makes condition $x >= y$ always true). The second step is to validate all equalities/inequalities formulae. In this case there is a conflict in the formulae and a deadlock occurs due to the *wait e1* and *wait e2* are executing prior to *notify e1* and *notify e2*, respectively. The detailed algorithm and explanation of synchronization verification can be found in [13].

Several experiments were conducted on a Pentium4 2.8-GHz machine with 2 GB of RAM running Linux. The results of synchronization verification are shown in Table 2. A counterexample was generated whenever a property did not hold. This counterexample showed a path leading to each inserted deadlock in the descriptions. The column LOC denotes the lines of codes of the original descriptions and the descriptions after abstraction. The column "# of Behaviors and Iterations" denotes the number of concurrent behaviors and the number of times the CEGAR refinement loop was executed. The last column denotes the number of deadlocks detected. Since all of these benchmarks are the real industrial designs, We found no deadlock in all benchmarks.

According to the results in Table 2, the verification of MPEG4 descriptions considered only a portion of the descriptions (about 800 lines) instead of the entire description (about 48,000 lines). We would like to point out that focusing on the synchronization verification can significantly reduce the size of the model that needs to be considered. We also believe that once the synchronization correctness is guaranteed, we can also use this framework to verify other properties.

As a final remark in this section, we use a Cooperating Validity Checker (CVC) [18] to compute abstraction when generating the abstract model. This is working well for the word-level data types. However, when considering the types of data with finite precision, some properties, e.g., overflow detection, cannot be verified using our method. This issue can be handled by using Satisfiability Modulo Theories (SMT) [17].

3.2 Sequentialization of Concurrent Behaviors

We can use the synchronization verification method mentioned in Section 3.1 to guarantee that all concurrent processes are running as the users intend. This section describes its usage for sequentializing concurrent behaviors which can be extended to check the equivalence of the two sequentialized descriptions as shown in [15].

To sequentialize the design descriptions, the following conditions must be satisfied.

Table 2. Experimental results

Benchmark	LOC		# of		Runtime	Deadlock
	Original	After abs.	Behaviors	Iterations		
FIFO	260	240	5	3	18.2	0
Point-to-point protocol	844	724	13	2	50.1	0
Elevator control system	2000	819	6	2	21.1	0
MPEG4	48126	781	5	1	9.7	0

- **No deadlock.** Every *wait* statement is always executed with at least one corresponding *notify* statement being eventually executed. This can be checked by the method explained in Section 3.1.
- **No race condition.** There is no possibility for any shared variable to be accessed at the same time, i.e., have no read/write, write/read or write/write accesses.

The above two conditions are the necessary conditions required to generate the sequential design that is equivalent to the original one. If a given design has a deadlock, its execution can halt somewhere in the design, while the execution of the sequential design should never halt. In this case, behaviors of the two designs are not equivalent for any sequentialization. For 'no race condition', if there exists any global variable which is local to parallel behaviors, accesses to this variable (known as read/write, write/read or write/write accesses) at the same time can cause the design to perform different functionalities.

Race Condition Check Before introducing the race condition check, we define *basic block* and *synchronization point.*

- **Basic block (BB):** A series of statements that do not include conditional branches nor synchronization points.
- **Synchronization point (SP):** A pair of *notify* and *wait* statements of the same *event* is considered as a synchronization point.

Now, assume that there are two basic blocks, *BB1* and *BB2*, where $T(BB1_{starttime}) < T(BB1_{endtime})$ and $T(BB2_{starttime}) < T(BB2_{endtime})$ are the timing constraints for each block. Together with these constraints, we can check the race condition between *BB1* and *BB2* by checking the following pair of properties.

$$\begin{array}{l} \textbf{Prop1}: \ T(BB1_{starttime}) < T(BB2_{endtime}) \\ \textbf{Prop2}: \ T(BB1_{endtime}) > T(BB2_{starttime}) \end{array}$$

We will not consider the condition when both **Prop1** and **Prop2** are unsatisfied at the same time, since it is obvious that this condition cannot occur. If **Prop1** is satisfied and **Prop2** is not, $BB1$ is proved to be executed prior to $BB2$ ($BB1 \rightarrow BB2$). In contrast, if **Prop1** is not satisfied and **Prop2** is, we can say that $BB1$ is proved to be executed after $BB2$ ($BB2 \rightarrow BB1$). If both of the properties are satisfied, *BB1* and *BB2* can be interleaved and it is possible for a race condition to occur. All variables in *BB1* and *BB2* must be checked

Algorithm 1 Sequentialization(SC)

```
declare
1: $SC$: SpecC descriptions
2: $error$: a Boolean variable
3: $Sync$: a set of synchronization point
4: $SET_{par}$: a set contains heuristic depth of $par$
begin
5: /* Heuristically search for $par$ */
6: $SET_{par}$ := HeuristicSearch($SC$)
7: foreach $par$ in $SET_{par}$ (start from the deepest one) do
8:    /* Check if there is any deadlock */
9:    ($error$, $Sync$) := SynchronizationVerification($SC$)
10:   if $error$ then
11:      exit ("There is a deadlock")
12:   end if
13:   foreach synchronization point in $Sync$ do
14:      /* Check if there is any race condition */
15:      $error$ := RaceConditionDetect($SC$)
16:      if $error$ then
17:         exit ("There is a race condition")
18:      end if
19:   end for
20: end for
21: return (SequentialGen($SC$))
end
```

for data dependency. If, for any variable, there is data dependence (read/write, write/read or write/write) between *BB1* and *BB2*, then there is a race condition. Otherwise, *BB1* and *BB2* are race condition free, and we can sequentialize in either ($BB1 \rightarrow BB2$) or ($BB2 \rightarrow BB1$) order.

Algorithm 1 can be explained briefly as follows. Variables are declared in line 1-4. In line 6, we heuristically search the entire descriptions to find all behaviors that were called under $par\{\}$ and store in variable SET_{par}. The outer loop (line 7-20) is the core sequentialization process. We check for synchronization error as shown in line 9. If there is any error, then terminating the process with error report. Otherwise, all the synchronization points are stored in variable *Sync*. The inner loop (line 13-19) checks whether each synchronization point can cause race condition. Once we exit from outer loop without any error, the process returns with the sequentialized descriptions as shown in line 21.

According to Algorithm 1, the detailed sequentialization process is described as follows.

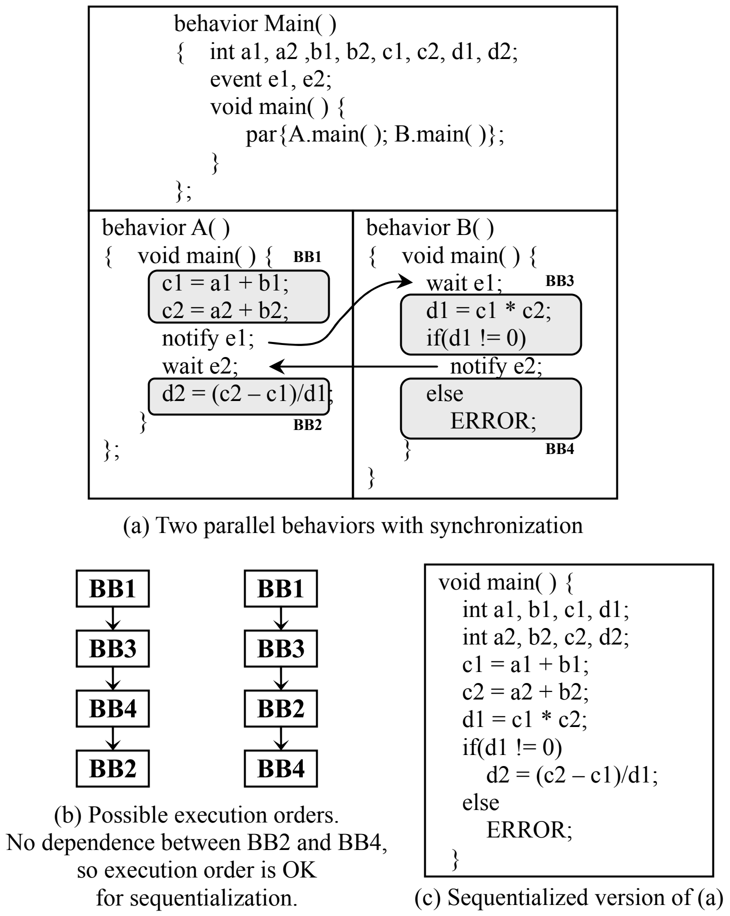
- Since SpecC supports design hierarchy and concurrency, we store all *par* statements in a set SET_{par} and, for each *par*, we mark its depth as we explore the design to lower hierarchy. The depth of a *par* statement is the depth from the top *par* statement of the hierarchy. If a *par* statement is not nested, the depth is 0.

- For each *par* statement, the deepest depth of SET_{par} is taken from the set and sequentialized by the following procedure. Let $Bhvr1$ and $Bhvr2$, for example, denote the two behaviors under the *par* statement to be sequentialized.
  - With the method for race condition check as described above, find all pairs of basic blocks $BB1$ in $Bhvr1$ and $BB2$ in $Bhvr2$ and put them into a set BB_{pair}.
  - For each pair in BB_{pair}, check **Prop1** and **Prop2**. If **Prop1** is true and **Prop2** is not, *BB1* and *BB2* can be sequentialized directly as *BB1* before *BB2*, and vice versa. However, if both properties are satisfied and there is no data dependence, *BB1* and *BB2* can be sequentialized in any order. Either *BB1* before *BB2* or *BB2* before *BB1*.
  - Otherwise, race condition is found and verification terminates with an error report.

Examples Figure 10 and 11 show examples of sequentialization of concurrent behaviors. Let us begin with the example in Figure 10. Behaviors A and B are called under *par*. There are two pairs of events $e1$ and $e2$ used for synchronization. Since $d2$ is computed by dividing $(c2 - c1)$ by $d1$, the value of $d1$ should never be equal to zero. This is a fatal error and we cannot allow it to occur. Therefore, the $ERROR$ statement was asserted to catch this error whenever it occurs. Hence, the condition $(d1 \mathrel{!}= 0)$ must always be true, and therefore both synchronization points always exist. According to these synchronization points, we can divide portions of code into four basic blocks $(BB1, \ldots, BB4)$. Synchronization point of *notify e1* and *wait e1* makes $BB1$ executed before $BB3$. In contrast, with the synchronization point of *notify e2* and *wait e2*, both $BB2$ and $BB4$ are executed after this point. It is not certain whether one will be executed before another, i.e., $BB2$ and $BB4$ can be interleaved with each other. Therefore, two possible execution orders of these basic blocks are shown in Figure 10(b), $BB1 \rightarrow BB3 \rightarrow BB2 \rightarrow BB4$ or $BB1 \rightarrow BB3 \rightarrow BB4 \rightarrow BB2$. There is no race condition, since data dependency (no read/write or write/write condition) between $BB2$ and $BB4$ does not exist. This is because there is no shared variable in $BB2$ and $BB4$. Both of the execution orders are functionally equivalent, hence generation of either of them is the sequentialization of Figure 10(a). Sequentialization of $BB1 \rightarrow BB3 \rightarrow BB2 \rightarrow BB4$ is shown in Figure 10(c).

On the other hand, let us consider the similar example shown in Figure 11. Both behaviors A and B of Figure 11(a) are slightly different from those of Figure 10(a). By using the same approach, both $BB1$ and $BB3$ are executed before synchronization point of event $e1$ and both $BB2$ and $BB4$ are executed after synchronization point of event $e2$. The possible execution orders are ($BB1$ *interleaved with* $BB3$) $\rightarrow$ ($BB2$ *interleaved with* $BB4$) as shown in Figure 11(b). Both $BB2$ and $BB4$ are exactly the same as in Figure 10. However, interleaving of statements between $BB1$ and $BB3$ has data dependencies that caused different behaviors and the read/write access violation of variables $c1$ and $c2$. Therefore, in this case, we terminate the sequentialization process and report that this design contains a race condition error.

We tested our tool on several designs written in SpecC. Different levels of implementation, e.g., specification level, architecture level before scheduling and architecture level after scheduling, of the Inverse Discrete Cosine Transform (IDCT) and the Vocoder, provided by



```
behavior Main( )
{   int a1, a2 ,b1, b2, c1, c2, d1, d2;
    event e1, e2;
    void main( ) {
        par{A.main( ); B.main( )};
    }
};
```

```
behavior A( )
{   void main( ) {   BB1
       c1 = a1 + b1;
       c2 = a2 + b2;
       notify e1;
       wait e2;
       d2 = (c2 – c1)/d1;
    }                BB2
};
```

```
behavior B( )
{   void main( ) {
       wait e1;      BB3
       d1 = c1 * c2;
       if(d1 != 0)
          notify e2;
       else
          ERROR;
    }                BB4
}
```

(a) Two parallel behaviors with synchronization

(b) Possible execution orders.
No dependence between BB2 and BB4,
so execution order is OK
for sequentialization.

```
void main( ) {
   int a1, b1, c1, d1;
   int a2, b2, c2, d2;
   c1 = a1 + b1;
   c2 = a2 + b2;
   d1 = c1 * c2;
   if(d1 != 0)
      d2 = (c2 – c1)/d1;
   else
      ERROR;
}
```

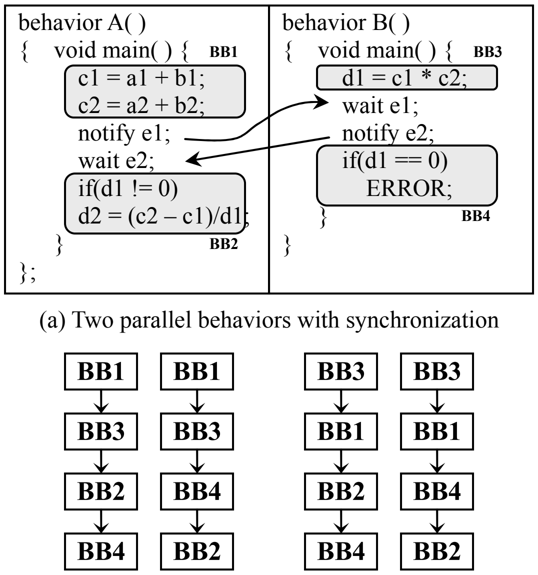
(c) Sequentialized version of (a)

Figure 10. Sequentialization example 1

Table 3. Sequentialization results of IDCT and Vocoder benchmarks

Benchmark	LOC	Bhvrs	Chnls	D-Lock	R-Cond	SymSim	ILP		
						Time(s)	Constrs	Vars	Time(s)
IDCT1_1	300	4	1	0	0	0.71	181	169	0
IDCT1_2	314	6	1	0	0	0.75	241	169	0
IDCT2_1	300	4	1	0	0	0.69	181	169	0
IDCT2_2	256	18	1	0	0	0.75	688	103	0
VocSpec	9165	102	4	1	0	38.97	1158606	2555	20.29
VocArch	10178	144	14	1	0	48.54	1179122	2749	19.99
VocSched	10139	144	14	1	0	41.97	1179182	2767	20.66

the University of California at Irvine (UCI), were tested. We performed the experiments on a Pentium4 2.8-GHz PC running Linux with 2 GB of RAM. Table 3 shows the results of sequentializing all the designs according to Algorithm 1. For each benchmark, we give the



```
behavior A( )                          behavior B( )
{   void main( ) {  BB1                {   void main( ) {  BB3
        c1 = a1 + b1;                          d1 = c1 * c2;
        c2 = a2 + b2;                          wait e1;
        notify e1;                             notify e2;
        wait e2;                               if(d1 == 0)
        if(d1 != 0)                                ERROR;
        d2 = (c2 – c1)/d1;                 }               BB4
    }               BB2                }
};
```

(a) Two parallel behaviors with synchronization

(b) Possible execution orders. There is no data dependence between BB2 and BB4, but there is between BB1 and BB3 where race condition occurs due to dependencies of shared variables c1 and c2.

Figure 11. Sequentialization example 2

code size in LOC (lines of code), total number of behaviors and channels utilized, reported errors for deadlock and race condition, and the total runtime by symbolic simulation in seconds. The last three columns present the results of the number of constraints/variables and the total runtime solved by the ILP solver.

As the results from sequentialization, we do not find any race condition in all benchmarks. However, we found one deadlock bug in all Vocoder designs. It was caused by a use of core communication channel that has a corner condition that caused a deadlock (both *sender* and *receiver* in the communication channel are waiting for the acknowledge signal from each other). We fixed this error and continued to sequentialize the design. Note that although the number of behaviors in the design is large, not all of them are concurrently running. And because in synchronization verification we heuristically explore the whole design to find only the behaviors running in parallel and focusing on behaviors that contain synchronization semantics. Therefore, the design size and the verification time can be reduced accordingly.

4. Optimum Code Generation with Translation of SDGs to Finite State Machines

In the previous section we described an analysis and verification method for SDGs, whereas in this section, we describe a method to synthesize optimal software from SDGs with SAT-based formulation.

Customized embedded processors can be attractive solutions in terms of performance and power, compared to general purpose embedded processors. In hardware/software designs, adding application-specific custom instructions is a popular approach to the customization. In the typical flow, target applications are profiled and bottleneck portions in the applications are identified. Then custom instructions that can efficiently execute these bottlenecks are generated and used. It is often the case that these bottlenecks are loop kernels.

Dynamically reconfigurable architectures can serve as fabrics to implement these custom instructions. They are more flexible than ASICs and more efficient than FPGAs for some applications. Dynamically reconfigurable architectures are mainly comprised of arrays of functional units (FUs), such as multipliers, ALUs, memory blocks, and registers. Connections between them are programmable. Because these connections can be changed even at run-time and in very short time (one cycle or so), it is easy to adapt the architectures to different bottleneck portions at run-time. To exploit dynamically reconfigurable architectures, availability of efficient compilers is vital. Without them, it is hard to utilize such architectures. Dynamically reconfigurable architectures can be viewed as special types of VLIWs, DSPs, or FPGAs, so some of the compiler techniques developed for these architectures may be reused. However, little work has been done developing compilers for them because this area of study is very new and the problem is challenging.

The basic flow of the compilers for dynamically reconfigurable architecture is proposed as follows. First, an input C program (SDG in our case) to be mapped to the architecture is parsed and translated into an intermediate representation (IR). (The automation of this part is our future work, but will not be difficult with the help of an existing compiler front-end.) Then, traditional compiler optimizations such as dead code elimination and loop optimizations, such as loop unrolling are performed to the IR using the existing compiler framework [9]. Architecture-aware optimizations are necessary to exploit the architectures. In this work, we have implemented a compiler back-end in which the optimized IR is mapped to a target architecture and the optimal code (configuration) is generated.

We focus on the compiler back-end for dynamically reconfigurable architectures. In the back-end, we have to perform *temporal mapping* (scheduling operations to time steps), and *spatial mapping* (mapping operations to functional units, assigning variables to registers or memories, mapping data transfers to connections). Here, we present a method that generates optimal code for dynamically reconfigurable architectures. It first extracts possible matching rules from a target architecture description. Then, code generation is solved by using the formulation based on finite state machines (FSMs) [11]. Our method can generate optimal code when the size of the problem is moderate. Because there are many moderate-size kernel loops in real embedded applications, we believe that our work is worthwhile.

Figure 12 shows the overall back-end flow of the proposed approach. The inputs to the flow is loop kernel C code identified by profiling and the architecture description for a target architecture. The loop kernel C code is parsed and optimized by CoSy compiler framework [9], and optimized intermediate representation (IR) is obtained. The optimized IR is represented as abstract syntax trees (ASTs). We construct a data-flow graph (DFG) from the ASTs. The matching rules used in the code generation process are extracted

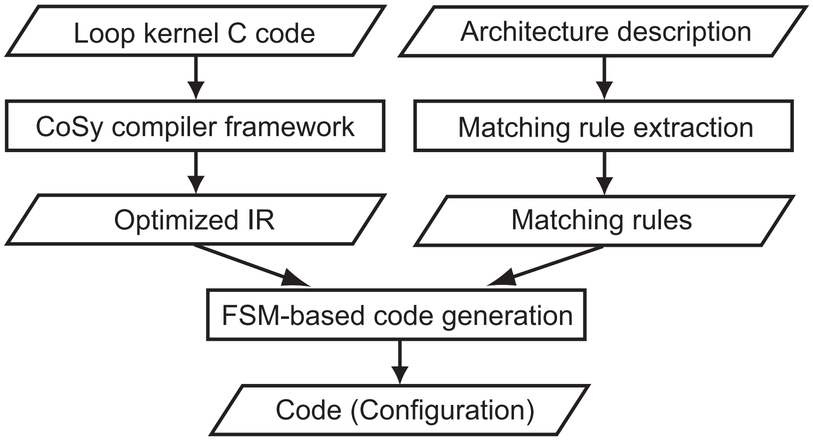
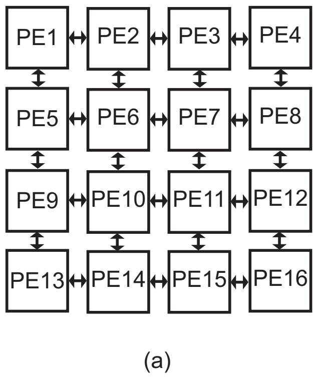
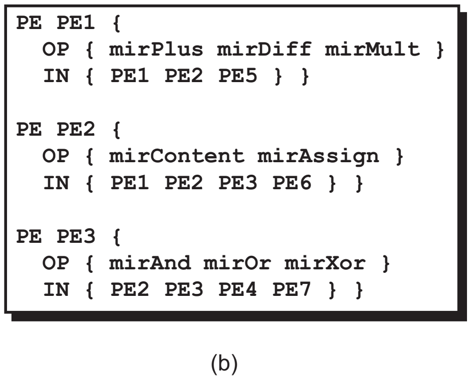


Figure 12. Overall flow of optimum code generation with translation of SDGs to FSMs





```
PE PE1 {
  OP { mirPlus mirDiff mirMult }
  IN { PE1 PE2 PE5 } }

PE PE2 {
  OP { mirContent mirAssign }
  IN { PE1 PE2 PE3 PE6 } }

PE PE3 {
  OP { mirAnd mirOr mirXor }
  IN { PE2 PE3 PE4 PE7 } }
```

Figure 13. (a) Net-list and (b) Architecture description

from the architecture description. By using the DFG and matching rules, FSM-based code generation [11] is performed. Finally, software pipelined code (configuration) is generated.

The target dynamically reconfigurable architecture is comprised of arrays of functional units (FUs), such as multipliers, ALUs, memory blocks, registers and register files, etc. Connections between them can be either bus-based or multiplexer-based and are programmable. We assume that each operation finishes execution in one cycle. In our method, a target architecture is basically represented by a net-list of components (FUs, multiplexor's, buses, etc.). From the architecture description, matching rules used in code generation are extracted.

In Figure 13(a), we show a simple target architecture. In the figure, processor elements (PEs) are connected to their nearest neighbors. Each PE has its own register file, and the computation results of a PE are written to its register file. Each PE can read operands from the register files owned by the nearest neighbors, as well as its own register file. Each PE

```
# resource and storage description
Resource {
  PE1(1),PE2(1),...}
RegisterFile {
  reg1(4),reg2(4),...}

# matching rules
Rule1 {
  pattern { mirMult(reg2,reg5)->reg1 }
  resource { PE1 } }
Rule2 {
  pattern { mirPlus(reg2,reg5)->reg5 }
  resource { PE1 } }
Rule3 {
  pattern { mirContent(reg1)->reg2 }
  resource { PE2 } }
Rule4 {
  pattern { mirAssign(reg3,reg6) }
  resource { PE2 } }
...
```

Figure 14. Code generator description

can execute some of the micro-operations such as addition, multiplication, memory read, memory write, etc. We assume that each micro-operations can be executed in one cycle.

Figure 13(b) shows an excerpt of the architecture description for the architecture in Figure 13(a). The simple micro-architecture information of each PE is described in the architecture description. In the description, `OP` shows the types of micro-operations the PE can perform, and `IN` shows input connections of the PE. For example, the first 3 lines of Figure 13(b) describe the processing element `PE1`. It can perform addition (`mirPlus`), subtraction (`mirDiff`), multiplication (`mirMult`), and it can take input operands from the register files in `PE1`, `PE2` and `PE5`. Although omitted in Figure 13(b), the information of register files in each PE is also described in the architecture description. For example, `PE1` has a register file `reg1` with 4 registers. From the architecture description, matching rules are extracted automatically. For example, `Rule1` is extracted, since `PE1` can perform multiplication (`mirMult`), and is connected to `PE2` (`reg2`) and `PE5` (`reg5`).

A code generator description (CGD) is a file that contains necessary information for code generation (such as code selection, register allocation and scheduling). An example CGD (just an excerpt) for the target architecture shown in Figure 13 is shown in Figure 14. The first part of CGD are resource and storage description. That part describes which type of and how many resources are available in the target architecture. In the example, we have resources `PE1`, `PE2`, etc. for processing. The parentheses represents the number of PEs of specific type. For example, `PE1(1)` means one `PE1`. As for storage, we have register files `reg1`, `reg2`, etc. The number written in parentheses describes the number of registers in the register files. For example, `reg1(4)` means the register file `reg1` of `PE1` has 4 registers.

The second part of a CGD is describing matching rules. They are extracted from the architecture description shown in Figure 13. We can extract such matching rules by enu-

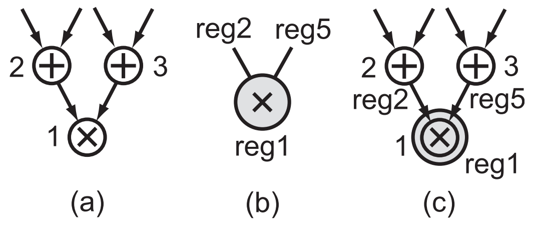


Figure 15. (a) DFG (b) pattern graph (c) match

merating all possible register transfers in the target architecture. In each matching rule, the `pattern` section shows a pattern graph that can be executed by the target architecture. A pattern graph is a graphical representation of a micro-operation. The `resource` section describes resources used when executing the pattern graph. For example, in `Rule1`, multiplication is possible whose source operands come from register files `reg2`, `reg5`, and the destination is `reg1`. To perform the multiplication, the resource `PE1` is used.

Now, we define a concept called *match*. It is simple and is used intensively in our formulation. When a sub-graph g of a DFG is isomorphic to a pattern graph i in a matching rule, we call the pair (g, i) a match. Matches can be obtained by performing graph matching between a DFG and pattern graphs. For each match (g, i), the sub-graph g can be executed by a micro-operation represented by the pattern graph i. We define the execution of a match (g, i) as the execution of the micro-operation p to compute the sub-graph g. To execute a match, proper input operands must be available.

Figures 15(a),(b),(c) show an example of a DFG, a pattern graph, and a match, respectively. A sub-graph consisting of the node 1 in the DFG can be executed by the pattern graph shown in Figure 15(b). The resulting match is depicted in Figure 15(c). To execute the match, the results of node 2 and node 3 must be available in register files $reg2$ and $reg5$ respectively.

In [11], a formulation of code generation for DSPs based on a finite state machine (FSM) was proposed. In the following, we present the formulation. The formulation has been extended to be able to deal with "software pipelining" in [12]. Due to the space limit, we skip details of the "software pipelining" formulation in this paper.

In the FSM, an input $\mathbf{w} = (w_0, ..., w_q)$ roughly corresponds to an instruction to be executed in a cycle and a state $\mathbf{x} = (x_0, ..., x_p)$ roughly represents a machine state that represents which (temporary) results are stored in which storage locations in a cycle. The shortest input sequence that brings an initial state to a final state corresponds to the assembly code with the minimum number of steps. In the following, we explain the state variables, input variables, and state transition functions of the FSM, followed by the constraints that must be satisfied by the variables.

The state variables of the FSM are defined as

$$\{a_{n,l} \mid n \in N, l \in L\} \cup \{v\} \tag{1}$$

where N is the set of all nodes in a DFG, and L is the set of all storage locations in a target processor. $a_{n,l}$ is referred to as *result availability variable* and it is one when the result of a node n is available in a storage location l, and it is zero otherwise. v is referred to as *constraint violation variable* and it is one when constraints such as a resource constraint has been violated.

The input variables of the FSM are defined as

$$\{e_m \mid m \in M\} \cup \{f_{n,l} \mid n \in N, l \in L\} \tag{2}$$

where M is the set of all matches. We refer to e_m as *match execution variable* and it is one when a match m is executed, otherwise it is zero. We refer to $f_{n,l}$ as *result free variable* and it is one when the result of a node n in a storage location l is being thrown away, otherwise zero. The result free variable is necessary to accommodate the situation when register files are full of operands, and some of them must be discarded to satisfy the register capacity constraints.

The transition function of a result availability variable $a_{n,l}$ is defined as follows.

$$a_{n,l}{}' = \begin{cases} 1 & (\bigvee_{m \in M_{n,l}} e_m = 1) \\ 0 & (f_{n,l} = 1) \\ a_{n,l} & (otherwise) \end{cases} \tag{3}$$

$a_{n,l}{}'$ is the next state variable of $a_{n,l}$, $M_{n,l}$ is the set of matches that compute the result of a node n in a storage location l and $\bigvee$ represents disjunction (OR). When the first and the second transitions are enabled at the same time, the first transition is executed. The first line of (3) means that the result availability variable $a_{n,l}$ will become one in the next cycle when any match m that computes the result of the node n in the storage location l is executed. The second line of (3) means that the result availability variable $a_{n,l}$ will become zero in the next cycle, when the result free variable $f_{n,l}$ for the same n and l is one. The third line of (3) means that the result availability variable $a_{n,l}$ will keep the previous value when the conditions of the first line and second line do not hold.

The transition function of the constraint violation variable v is defined as follows.

$$v' = \begin{cases} 1 & (v_{operand} \vee v_{resource} \vee v_{register} = 1) \\ v & (otherwise) \end{cases} \tag{4}$$

v' is the next state variable of v, and $\vee$ represents a disjunction (OR). v_{result}, $v_{resource}$, $v_{register}$ are temporary variables, representing operand constraints, resource constraints, and register constraints violations. Each variable becomes one when corresponding constraints are violated, otherwise zero.

To generate correct assembly code, certain constraints must be satisfied on the above variables along the FSM state transitions. These constraints are represented as Boolean formulas among FSM variables. The following three constraints are sufficient for generating correct assembly code.

Operand Constraint Consider a match $m = (g, i)$ whose input nodes of g are $n_1, ..., n_p$. Assume that the results of these input nodes must be available in storages $l_1, ..., l_p$ to execute the match m. We denote this constraint as $v_{operand,m}$. It is formally represented as follows:

$$v_{operand,m} = \overline{e_m \rightarrow a_{n_1,l_1} \wedge ... \wedge a_{n_p,l_p}} \tag{5}$$

The operand constraint $v_{operand}$ is then represented as follows,

$$v_{operand} = \bigvee_{m \in M} v_{operand,m} \tag{6}$$

where M is the set of all matches.

Resource Constraint Suppose that there are n matches $m_{r,1}, ..., m_{r,n}$ that use a resource type r, and the number of available resources for the resource type r is k. Then at most k out of n matches can be executed regarding this resource. This condition is translated into the following expression.

$$v_{resource,r} = T_{n,k}(e_{m_{r,1}}, ..., e_{m_{r,n}}) \tag{7}$$

$T_{n,k}(x_1, ..., x_n)$ is a *threshold function*, and it is one when more than k input variables out of $x_1, ..., x_n$ are one, otherwise zero. The resource constraint $v_{resource}$ is represented as follows,

$$v_{resource} = \bigvee_{r \in R} v_{resource,r} \tag{8}$$

where R is the set of all resource types.

Register Constraint Suppose that there are n nodes in a DFG and the capacity of a register file l is k

Then, up to k results can be stored in the register file l out of n results of all nodes in the DFG. This condition is translated into the following expression.

$$v_{register,l} = T_{n,k}(a_{1,l}, ..., a_{n,l}) \tag{9}$$

The register constraint $v_{register}$ is represented as follows,

$$v_{register} = \bigvee_{l \in L} v_{register,l} \tag{10}$$

where L is the set of all location types (register file or register).

Now, we explain the initial and final states of the FSM. In initial states, $a_{n,l} = 0$ for all nodes n except input nodes of a DFG. Initial states represent the states where inputs of the DFG are available but no operation has started yet. Final states are specified as $a_{n,l} = 1$ for all output nodes n in the DFG and for user specified locations l, and $v = 0$. Final states represent the states where the results of output nodes are computed correctly in the specified locations and no constraint has been violated.

We have implemented the presented approach and performed an experiment. To analyze the FSM, we used the pseudo-Boolean solver called PBS(v2.1)[10].

As for the target architecture, we used the same topology as shown in Figure 13(a). However, each PE not only has nearest-neighbor connections but also the second nearest-neighbor connections. For example, PE6 can read operands not only from nearest-neighbors (PE2,PE5,PE6, PE7,PE10), but also the second nearest-neighbors PE8 and PE14.

Table 4 shows the experimental results. The first column represents the name of loops. *fir* is an FIR filter code, *n_real_update* is another filter code, and *gouraud* is a shading

Table 4. Experimental results

benchmark	#ops	step	II			#vars	#clauses	#pb	time(s)
			p=1	p=2	p=3	case of p=3, II=2			
fir	7	3	3	2	2	43671	117337	80	2.7
n_real_update	9	4	5	3	2	50679	138395	80	4.5
gouraud	13	6	6	3	2	101607	281511	80	15.5

algorithm for 3D graphics. The second column (#ops) shows the number of operations in each loop body. The third column (step) is the length of the critical path for the DFG of each loop body. The fourth, fifth, and sixth columns (II) shows the initiation interval results (II) of the software pipelining when the number of replication p (i.e., the maximum number of loop iterations whose executions are overlapped) is changed as $p = 1$, $p = 2$, $p = 3$, respectively. As shown in the table, the results of II when $p = 3$ for all the benchmarks happened to be 2. The columns #vars, #clauses, #pb, time(s) represent the number of variables, the number of clauses, the number of pseudo Boolean constraints, and runtime of PBS for each benchmark's SAT instance in the case of $p = 3$ and II=2.

As shown in the table, when the number of replication p is increased, then more parallelism is gained, since the initiation interval (II) is decreasing. The CPU time spent for each loop body and for each p was not more than several minutes. Because the analysis of FSMs using Boolean satisfiability is NP-complete, the CPU time to be spent increases rapidly as the sizes of the DFG or dynamically reconfigurable architectures increase. Basically, DFGs with up to 20 nodes could be processed by the proposed approach. These solutions are the optimal initiation intervals for the given p. The sizes of problems as SAT formulation range from 10,000 to 100,000 variables and 100,000 to 500,000 clauses for the largest configurations.

5. Conclusions

We described our verification and synthesis framework based on SDGs that have been used in program slicing technology. We also showed verification and synthesis techniques based on the analysis of SDGs by translating the problems into SAT and ILP ones. The experimental results indicated that the state-of-the-art SAT and ILP solvers give practical effectiveness for reasonably large design descriptions. In order to deal with much larger design descriptions that often appear in embedded system designs, we need to incorporate various heuristics, which will be our future work. In addition, the framework should handle not only C/C++/SpecC but also SystemC and RTL. We will address these needs in our future work.

References

[1] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, "SpecC: Specification Language and Methodology," Kluwer Academic Publishers, Boston, March 2000.

[2] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs.

volume 12, pages 26–60. ACM Press, 1990.

[3] K. Tanabe, S. Sasaki and M. Fujita. Program Slicing for System Level Designs in SpecC. In *Proc. of the IASTED, International Conference on Advances in Computer Science and Technology (ACST2004)*, pages 252–258, Nov. 2004.

[4] Codesurfer. http://www.grammatech.com/products/codesurfer/.

[5] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proceedings of the 18th international conference on Software engineering*, pages 495–505. IEEE Computer Society, 1996.

[6] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184. ACM Press, 1984.

[7] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, **10**(4):352–357, 1984.

[8] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum. Program Slicing of Hardware Description Languages. In *Conference on Correct Hardware Design and Verification Methods*, pages 298–312, 1999.

[9] ACE Associated Compiler Experts. CoSy compiler development system, 2004.

[10] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Generic ILP versus specialized 0-1 ILP: an update. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 450–457, November 2002.

[11] K. Seto, M. Fujita, and K. Asada. Retargetable Code Generation Based on Finite State Machine and Boolean Satisfiability . In *International Workshop on Logic and Synthesis (IWLS)*, pages 260–265, June 2003.

[12] K. Seto and M. Fujita. Optimal Temporal Spatial Mapping for Dynamically Reconfigurable Architectures. Internal Report, 2004. http://www.cad.t.u-tokyo.ac.jp/~seto/internal_report.html.

[13] T. Sakunkonchak, S. Komatsu, and M. Fujita. Synchronization verification in system-level design with ILP solvers. In *Third ACM-IEEE International Confernece on Formal Methods and Models for Codesign (MEMOCODE2005)*, pages 121–130, July 2005.

[14] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of the International Conference on Computer-Aided Verification (CAV'00)*, ser. Volume **1855** of LNCS, E. A. Emerson and A. P. Sistla, Eds. Springer-Verlag, 2000.

[15] T. Sakunkonchak, T. Matsumoto, H. Saito, S. Komatsu, and M. Fujita. Equivalence Checking in C-based System-Level Design by Sequentializing Concurrent Behaviors. In *Proc. of the IASTED, International Conference on Advances in Computer Science and Technology (ACST2007)*

[16] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, **126**(2), April 1994.

[17] C. Tinelli. A DPLL-based Calculus for Ground Satisfiability Modulo Theories. In *Proc. of the 8th European Conference on Logics in Artificial Intelligence*, pages 308–319, 2002.

[18] A. Stump, C. W. Barrett, and D. L. Dill. CVC: A Cooperating Validity Checker. In *the 14th International Conference on Computer Aided Verification (CAV'02)*, pages 500–504, 2002.

[19] L. Cai, S. Verma, and D. D. Gajski. Comparison of SpecC and SystemC Languages for System Design. *Technical Report CECS-03-11, Center of Embedded Computer Systems, University of California, Irvine.*

[20] T. Nishihara, D. Ando, T. Matsumoto, and M. Fujita. ExSDG : Unified Dependence Graph Representation of Hardware Design from System Level down to RTL for Formal Analysis and Verification. In *International Workshop on Logic and Synthesis (IWLS)*, June 2007.