Invited Paper

Trends in Formal Verification Techniques for C-based Hardware Designs

Masahiro Fujita†1

Three formal verification approaches targeting C language based hardware designs, which are the central verification technologies for C-based hardware design flows, are presented. First approach is to statically analyze C design descriptions to see if there is any inconsistency/inadequate usages, such as array overbounds accesses, uses of values of variables before initialization, deadlocks, and others. It is based on local analysis of the descriptions and hence applicable to large design descriptions. The key issue for this approach is how to reason about various dependencies among statements as precisely as possible with as short time as possible. Second approach is to model check C design descriptions. Since simple model checking does not work well for large descriptions, automatic abstractions or reductions of descriptions and their refinements are integrated with model checking methods such that reasonably large designs can be processed. By concentrating on particular types of properties, there can be large reductions of design sizes, and as a result, real life designs could be model checked. The last approach is to check equivalence between two C design descriptions. It is based on symbolic simulations of design descriptions. Since there can be large numbers of execution paths in large design descriptions, various techniques to reduce the numbers of execution paths to be examined are incorporated. All of the presented methods use dependence analysis on data, control, and others as their basic analysis techniques. System dependence graph for programming languages are extended to deal with C based hardware designs that have structural hierarchy as well. With those techniques, reasonably large design descriptions can be checked.

1. Introduction

Due to increased complexity and capacity of target designs, it takes more and more time to make sure their logical correctness. One way to realize shorter design periods is to start design with more abstracted representations, such as system-level designs where whole designs are represented as hardware/software combined systems. There are several advantages in starting the designs with system-level. Among them the most important issue for verification is the reduction of designer's effort to write down designs. Because design descriptions are more abstracted than the lower level ones, such as the ones in register transfer level (RTL), the quantity of design descriptions become much smaller, which makes verification efforts much simpler. Currently C based languages are commonly used to describe designs in system-level. Starting from system-level, design descriptions are gradually refined either manually or automatically all the way down to RTL. It is very important to make sure the correctness of design descriptions at each design step. Formal methods should be incorporated as much as possible along with intensive simulations.

Figure 1 shows a general design flow in high level design processes. It starts with a rather algorithmic description of the target design and tries to refine it into a high-level synthesizable model. Here it is assumed that algorithmic design descriptions written in C/C++ or their extended counterparts such as SpecC [1] and SystemC [2] language are given as specifications. In some cases these algorithmic descriptions can be found in the definitions of standards for image compression/decompression techniques, communication protocols and encryption methods. The goal of designers is to transform these algorithmic descriptions into ones that are hardware-friendly. The final transformed descriptions should be accepted

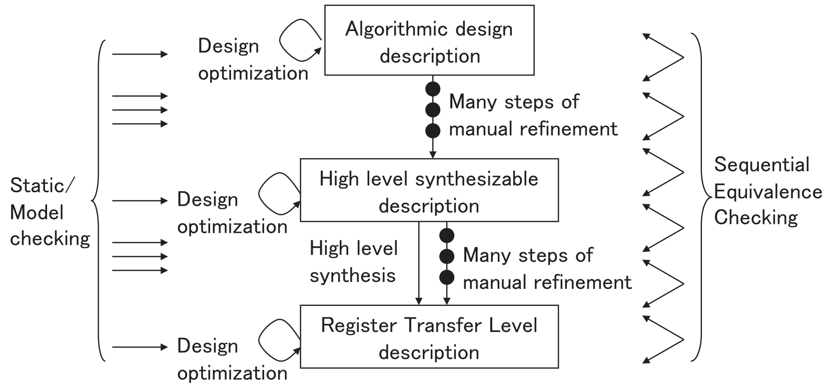

Algorithmic design description
Design optimization
Many steps of manual refinement
High level synthesizable description
Design optimization
High level synthesis
Many steps of manual refinement
Register Transfer Level description
Design optimization
Static/ Model checking
Sequential Equivalence Checking

Fig. 1 High-level design flow.

†1 VLSI Design and Education Center, The University of Tokyo

by high level synthesis tools to generate high quality RTL descriptions. These design refinements consist of many steps of transformations including changes of data types (floating point to fixed point, refinements of bit-widths in fixed point models), removal of certain types of pointer manipulations, partitioning of memories, and various types of statement transformations to make the models efficiently synthesizable. Design models higher than RTL such as algorithmic and high-level ones, are called system-level models (SLM). Once a high-level synthesizable description is obtained, it can automatically be transformed into RTL through high level synthesis tools. The RTL description then goes into the standard implementation flow using logic synthesis, placement and routing.

In order to enforce the correctness of various descriptions in the design flow shown in Fig. 1, two issues must be resolved:

- Eliminate bugs as much as possible from given levels of descriptions. This is needed at the highest level of abstraction that cannot be verified via equivalence checking, but is also needed when equivalence checking cannot verify all the behaviors of a model by comparing it against a higher level model.
- Guarantee the equivalence of the two descriptions: a validated higher level model and a lower level model obtained automatically (via high-level synthesis) or through manual refinements of the higher level model.

The aboves complement each other to assure the correctness of the descriptions as a whole.

RTL descriptions precisely define what must be computed at each clock cycle. On the other hand, high-level designs can have freedom in terms of scheduling operations. Algorithmic descriptions usually have almost no timing constraints relating to execution orders of operations. They are called “untimed” models. As designs are refined, more and more timings on the executions of various operations are inserted into design descriptions. They are called “timed” models. High-level verification methods must deal with both untimed and timed design descriptions. In the case of RTL descriptions, the notion of next time is obvious. In high-level design descriptions, however, next time may not be clearly defined. Therefore, analysis processes on timing are required to extract state transition representations that are required for formal methods. This extraction process is one of the key issues for high-level formal verification. Especially in the case of equivalence checking on high-level designs, timings of executing operations in the two designs are generally different, and those differences must be precisely specified in order to define the equivalence of the two designs. The resulting verification becomes sequential equivalence checking.

There are basically three formal verification approaches targeting C language based hardware designs, such as the ones in SpecC[1]. First approach is to statically analyze C design descriptions to see if there is any inconsistent/inadequate/non-popular usages, such as array overbounds accesses, uses of values of variables before initialization, null pointer accesses, deadlocks, and many others. It is based on local analysis of the descriptions, and hence applicable to very large design descriptions, such as the ones having more than 100,000 lines. Although the analysis is basically conservative in the sense that there can be false errors or warnings, it can practically catch many realistic bugs. Moreover it takes very short time for such various checks. The key issue for this approach is how to reason about various dependencies among statements as precisely as possible with as short time as possible. Second approach is to model check C design descriptions. Since simple model checking does not work well for large designs, automatic abstractions or reductions of designs and their refinements are integrated with model checking methods so that reasonably large designs can be processed. By concentrating on particular types of properties, there can be large reductions of design sizes, and as a result, real life designs could be targets for model checking. The last approach is to check equivalence between two C design descriptions. It is based on symbolic simulations of design descriptions. Since there can be large numbers of execution paths in large design descriptions, various techniques to reduce the numbers of execution paths to be examined are incorporated. There are many cases where two design descriptions to be compared are very similar in the sense that the actual different portions of the two descriptions are very small. In such cases, analysis only on those difference may establish the equivalence of the whole designs and hence very large design descriptions can be verified. All of the presented methods use dependence analysis on data, control, and others as their basic techniques. System dependence graph for programming languages are extended to deal with C based hardware designs that have structural hierarchy as well. With the extended dependence analysis techniques, reasonably large design descriptions can be processed and verified.

There have been a number of efforts in the above three approaches, and some of them have been already used in industrial designs. Static checking methods are widely used as initial verification efforts in high level descriptions. Since they can deal with millions of lines of codes in practical time, static methods can compensate simulations for quick identification of buggy descriptions. Although Model checking for RTL designs have been used in industry for several years, its application to high level descriptions are mostly in introductory phases to industry. As for equivalence checking for high-level descriptions, some industrial tools have been developed, and they are now used in some real designs. The state-of-the-art model checking and equivalence checking tools can deal with design descriptions for block level of designs. As more research efforts, such as the ones shown in this paper, are included in the industrial tools, large design descriptions can be processed.

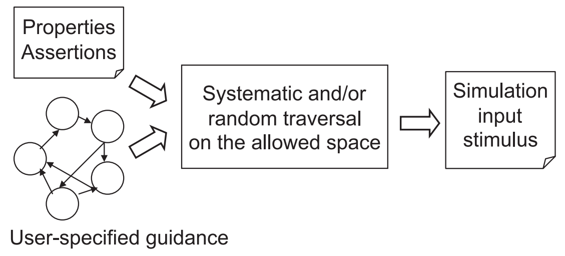
In this paper, mostly our works on them are presented as example efforts in the above three approaches. This paper is organized as follows. In the next section basic technology and algorithm for high-level verification with their performance are reviewed. In the following section after introducing SpecC language as a representative of various C based design languages, basic data structure for various data/control/other dependence analysis, called system dependence graph, is introduced. This data structure is the base for the analysis shown in the following sections. Then in the following three sections the above mentioned three formal verification approaches are presented. The last section gives concluding remarks.

2. Technology for High-level Verification

In this section, techniques for high-level design descriptions are briefly reviewed. Sequential equivalence checking methods for high-level design descriptions for C programs are discussed in later sections with details.

2.1 Simulation Based Verification

From the viewpoint of ensuring the correctness of the descriptions as much as possible, both simulation and formal verification are employed in each verification step. Intensive simulations may be performed on the descriptions with randomly and manually generated input stimuli The key issue here is the quality of such input stimuli, and there are techniques for generating good ones through the use of formal verification techniques. There are always corner case problems in simulations. That is, some rarely exercised execution paths may not be activated by the given stimuli. Thus bugs in those execution paths may not be eliminated. Therefore, instead of using pure random stimuli, designers may specify sorts of templates for event sequences relating to the behaviors of the target designs including corner cases from which input stimuli can be automatically generated as shown in **Fig. 2**. This is a user-guided generation of input stimuli and easily fits into any design flows such as the one in Fig. 1. Systematically traversing search spaces specified by such templates is a common analysis technique used in formal verification techniques. Since this is a more systematic way to generate input stimuli than normal simulation, it is sometimes called intelligent simulation. Those templates can be created either by specifying I/O behaviors for them or analyzing internal behaviors of the designs. For example, in the case of communication protocols, such as on-chip bus protocols, I/O behaviors are completely specified as part of standard bus protocols from which their templates can be generated. Also, assertions and properties on the designs can be used to generate templates by analyzing their allowed execution sequences. An assertion specifies how the target design should behave in multiple time frames, and so it can be used to let the input stimuli follow those sequential behaviors.


Properties
Assertions
Systematic and/or random traversal on the allowed space
Simulation input stimulus
User-specified guidance
Fig. 2 Input stimuli generation based on user-guidance.

Intelligent simulation pattern generations can be applied to larger descriptions in general, although coverages of the generated simulation patterns may decrease as description sizes increase.

2.2 Static Analysis Methods

Static analysis does not execute and follow behaviors of designs globally. It checks on behaviors of the designs only locally. The most basic static methods are the ones used in various lint-type tools. First of all, control/data flow graphs are generated from given design descriptions, and dependencies on control/data are examined which results in various dependence graphs. By analyzing these generated graphs, which can automatically be generated from C/C++ and SpecC descriptions, various items can be checked very quickly by locally analyzing them. The key issue here is what portions of the descriptions are analyzed by such checks. Whereas model checking methods basically examine the design descriptions exhaustively starting from initial or reset states, static program checking analyzes design descriptions only in small and local areas. Instead of starting from initial states, it first picks up target statements and then examines only small portions of design descriptions which are very close to those target statements as shown in the left part of **Fig. 3**. Since analysis is local, it can deal with very large design descriptions, although accuracy of the analysis is less. This means that static checking methods can produce false errors or warnings, that is, even if errors or warning are generated, they may not be real bugs. Static checking methods are based on conservative analysis for quick processing.

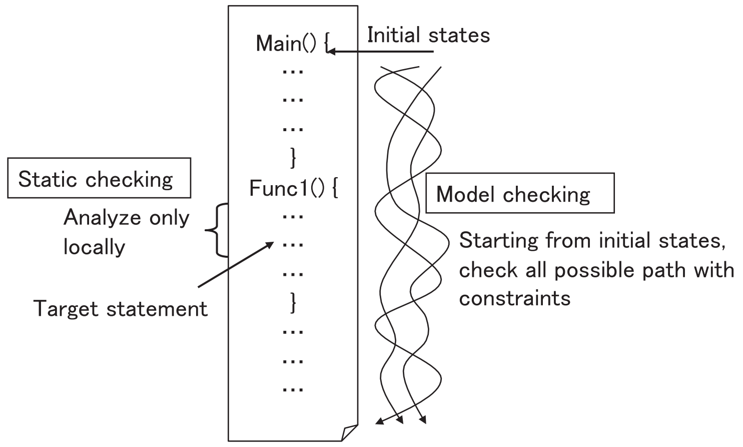


Fig. 3 Static checking and model checking approaches.

Many items can be quickly checked with such static methods. Non-initialized use of variables, null pointer references, relative execution orders among concurrent statements, and other issues which are easily modeled as local traversals on control/data/dependence graphs can efficiently be checked. Static checking methods have been applied to various software verification tasks and have proved to be very effective even for very large descriptions having millions lines of code. Since the analysis is conservative, there can be many false error messages generated by static checking methods, and how to eliminate most of them is one of the key future research topics in static checking methods.

Static checking methods can be applied to very large descriptions, e.g., descriptions having more than millions of lines of codes with practical processing time, although they may generate false waring/errors in some cases.

2.3 Model Checking Methods

After static checking methods are applied to the design descriptions, model checking methods can be used to detect more complicated bugs that can only be found through systematic traversals starting from initial states or user specified states. Given a property, model checking methods determine whether all possible execution paths from initial states satisfy the property, as shown in the right side of Fig. 3. Several formal verification methods for C programs, such as CBMC[3], SLAM[7], BLAST[8], MAGIC[9], and others, have been proposed. These are using SAT such as Chaff[4] ad SMT solvers such as Z3[6] as basic reasoning engines for state space traversal. Moreover, in order to process larger descriptions, given descriptions are first reduced for model checking. This automatic reduction, such as CEGAR paradigm[10], is generally target property dependent. That is, given a property to be verified, portions of design descriptions that do not influence the correctness of the property are automatically eliminated. Depending on characteristics of target properties, the amount of reduction varies.

Model checking methods can deal with reasonably large descriptions with automatic reductions of descriptions with respect to given properties to be verified. Thousands of lines to tens of thousands lines of codes have been verified within practical time by the methods mentioned above.

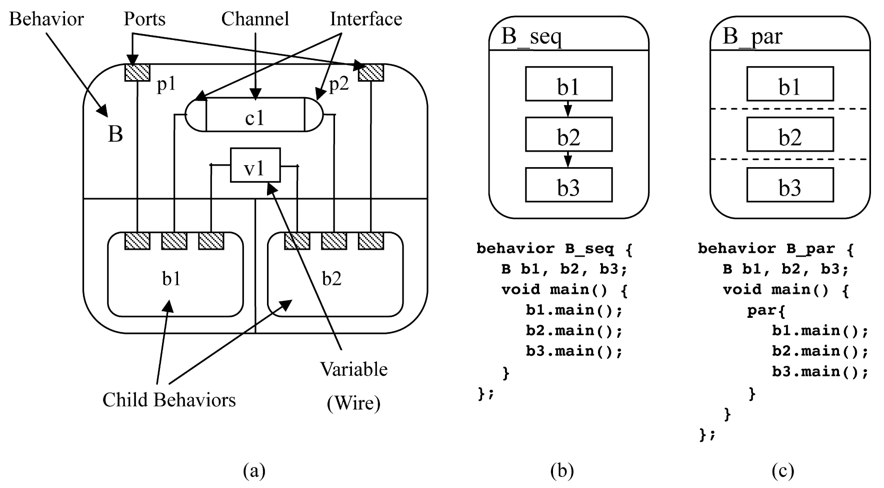
3. Dependence Analysis Techniques

3.1 C Based Design Language

There have been lots of efforts to use C or C++ languages to describe hardware parts of the target designs as well as their software parts. Although there are differences in their details, the ways to describe hardware parts in C and C++ languages share common features. In this paper we target SpecC language [1] when presenting the verification methods as a representative of C based hardware design descriptions. SpecC is an extension over C language in the following ways:

- Parallel (concurrent) descriptions with $par, notify, wait$ statements
- Introduction of modules and ports for structural hierarchy
- Various statements to describe hardware oriented controls

The features of SpecC language are summarized in **Fig. 4**. 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. A behavior corresponds to a module in RTL hardware description languages. In order to communicate, a behavior can be connected to other behaviors or channels through its ports or interfaces. The structural hierarchy of such a behavior is shown in Fig. 4 (a). The sequential and



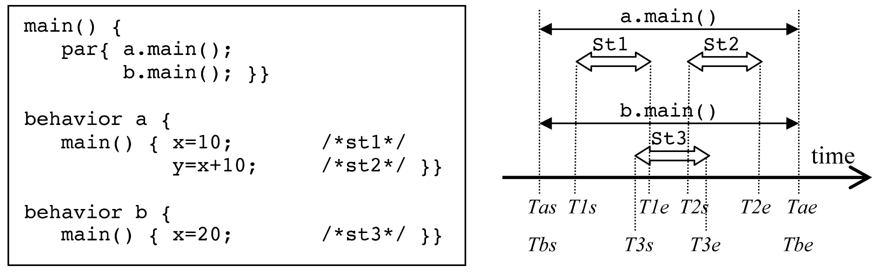
```
behavior B_seq {
   B b1, b2, b3;
   void main() {
      b1.main();
      b2.main();
      b3.main();
   }
};
```

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

Fig. 4 SpecC language.

parallel constructs of SpecC are shown in Fig. 4 (b) and (c), respectively.

Concurrency and synchronization among concurrent behaviors are represented in SpecC by the *par* and *notify/wait* constructs, as seen in the **Figs. 5** and **6**. As can be seen from Fig. 5, there is no constraints on execution orders among statements in parallel behaviors. In a single behavior running in isolation, correctness of the result is usually independent of the timing of its execution, and deter-

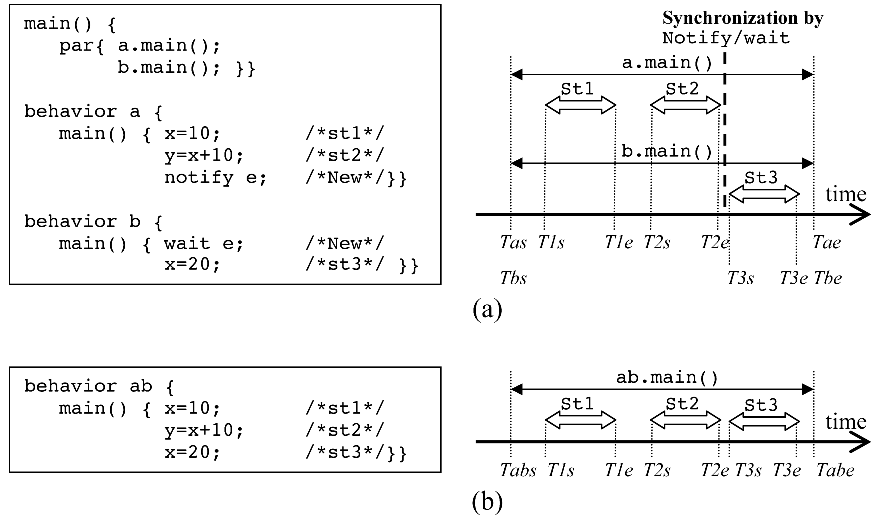


```
main() {
   par{ a.main();
        b.main(); }}

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

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

Fig. 5 Parallel execution in SpecC.



```
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*/ }}
```

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

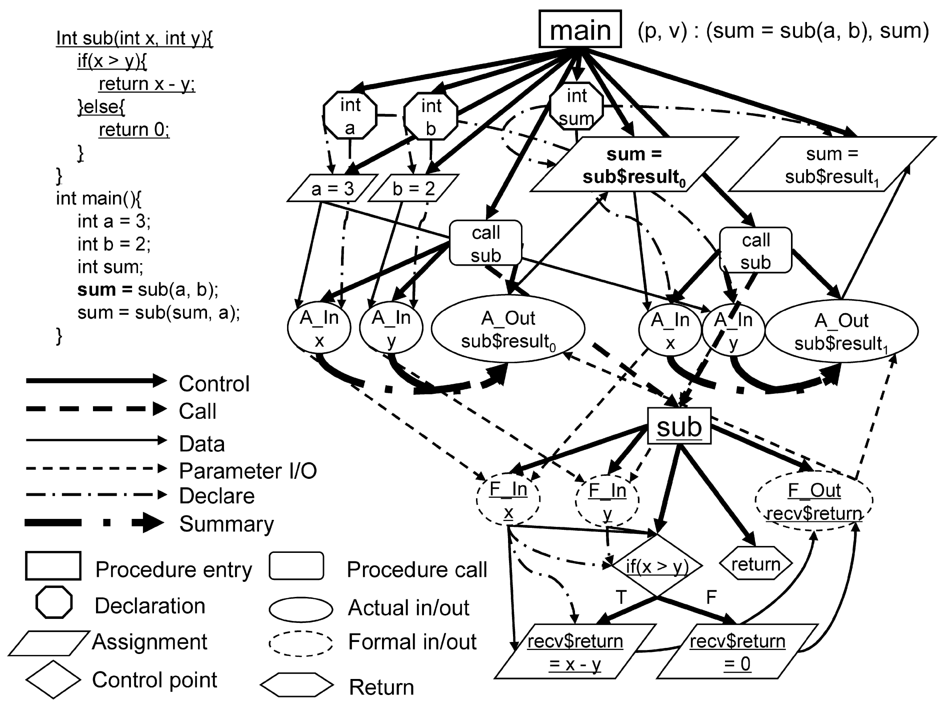
Fig. 6 Synchronization between parallel processes in SpecC.

mined solely by the logical correctness of its functions. However, when several behaviors are running in parallel, execution timing may have a great affect on the correctness of the execution results, that is, 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, and *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. With these synchronization statements, the descriptions in Fig. 6 (a) behaves in the same way as the one in Fig. 6 (b).

3.2 Program Slicing and Dependent Analysis for SpecC Descriptions

Program slicing is a technique to extract portions of the original programs which are relevant to the variables at some statements specified by users. Slicing is computed given two parameters, program point *p* and the set of variables *v* which appear in *p*. Program slicing first computes various dependence graphs where slicing for the given parameters are computed. Horwitz, et al. in Ref. 11) defines System Dependence Graphs (SDGs), which contains multiple Procedure Dependence Graphs (PDGs) and expresses dependencies between procedures. An example system dependence graph generated from a simple C description is shown in **Fig. 7**. As for the details of various edges and nodes defined in system dependence graphs, please refer to Ref. 11).

In 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, an SDG for SpecC with new nodes and edges are added. To deal with concurrent executions realized by using *par* statements, a node for *par* as a control point node, similar to that of *if*, *while* and *for*, is defined. 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 **Fig. 8**, since *b1.main()* and *b2.main()* are executed concurrently, there must be control edges from *par* node to each of them. Extra data dependence edges for representing the shared ports and parameters are also required. In the figure, *b1* and *b2* are running in parallel and there is a shared variable *p1*, hence, the data edge for



```
Int sub(int x, int y){
    if(x > y){
        return x - y;
    }else{
        return 0;
    }
}
int main(){
    int a = 3;
    int b = 2;
    int sum;
    sum = sub(a, b);
    sum = sub(sum, a);
}
```

Fig. 7 A simple C description and its system dependence graph.

M_A_Out p1 to *M_A_In p1* are constructed and vice versa. In **Figs.** 8 and **9**, 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.

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 *wait* statement is constructed. In Fig. 9, 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*. Typically an event variable is

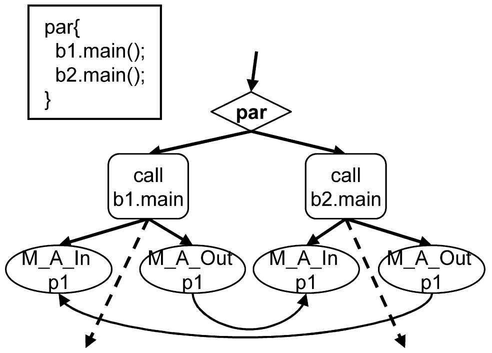


Fig. 8 Extended nodes and edges for par statement.

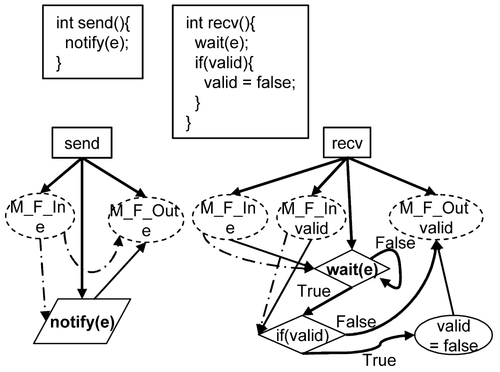


Fig. 9 Nodes and edges for notify and wait statements.

communicated through *channel* connected to *behavior*'s port. Therefore we can traverse data dependence edges from *notify* node to *wait* node across *behaviors* and *channels* in the SDG.

With the above extensions over system dependence graphs for C languages, various dependencies in SpecC descriptions can be represented as Extended System Dependence Graphs. In the following analysis methods, SpecC descriptions are first converted into Extended System Dependence Graphs.

4. Static Program Checking

By locally analyzing dependence graphs generated from C based design descriptions, various properties can be checked or verified very quickly. The key issue here is how much portions of the descriptions are analyzed for such verification. While model checking methods basically examine the design descriptions exhaustively starting from initial or reset states, static program checking analyze design descriptions very locally. Instead of starting from initial states, it first picks up target statements and then examine only small portions of design descriptions which are very close to those target statements. Since analysis is local, it can deal with very large design descriptions, although accuracy of the analysis may be decreased. This means that static program checking methods may generally produce false errors or warnings, that is, even if errors or warning are generated, they may not be true. This is due to the fact that static program checking methods use conservative analysis. In this section, we present several items that can be relatively effectively checked along with their associated verification algorithms [12].

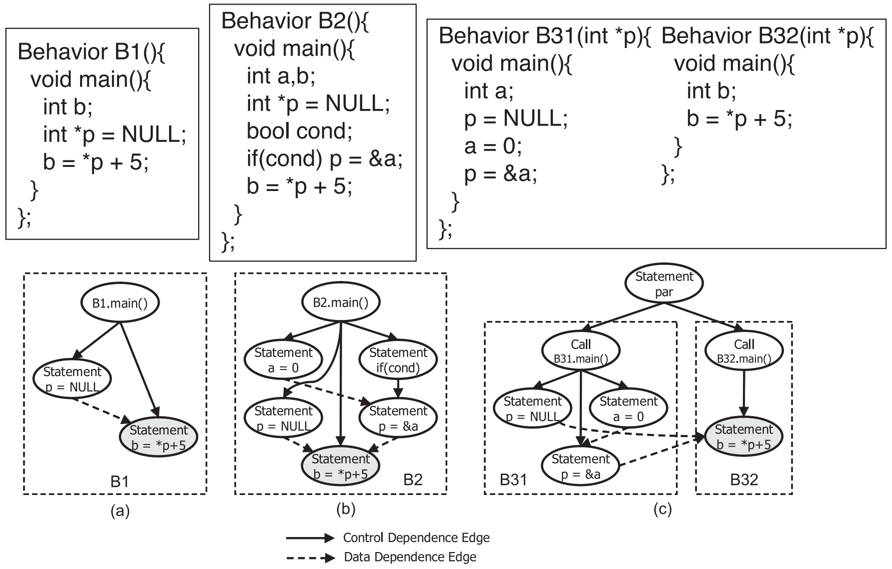
4.1 Detection of Null Pointer Dereferences

Null pointer dereferences are more likely to happen wherever pointer variables point to nothing. Normally, pointer variables are used after initializations which assign addresses of variables. Null pointer dereferences are classified into three types as given in **Fig. 10**. In each case, a pointer variable "p" is used at "b = *p + 5". However, in (a), there are no nodes initializing "p" except for "p = NULL". In (b), although "p = &a" initializes "p", it may not be executed since it is under a conditional branch "if(cond)". In (c), the execution order of "p = &a" and "b = *p + 5" is not decidable when behaviors "B31" and "B32" are running concurrently. Therefore, "p" can be dereferenced as null pointer in each case.

Null pointer dereferences can be detected by the following procedure.

(1) For each pointer variable used in nodes, nodes which initializes the pointer are collected by traversing data dependence edges backwardly.
(2) Whether there are no nodes initializing to null is checked.

For example, a node "b = *p + 5" in Fig. 10 (b) is checked as follows.



```
Behavior B1(){
  void main(){
    int b;
    int *p = NULL;
    b = *p + 5;
  }
};
```

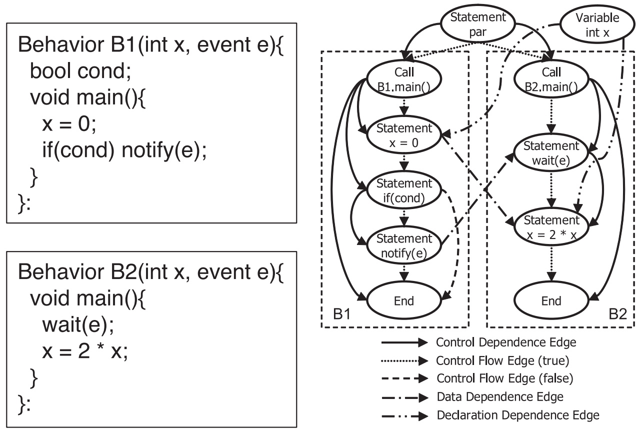
```
Behavior B2(){
  void main(){
    int a,b;
    int *p = NULL;
    bool cond;
    if(cond) p = &a;
    b = *p + 5;
  }
};
```

```
Behavior B31(int *p){ Behavior B32(int *p){
  void main(){            void main(){
    int a;                  int b;
    p = NULL;               b = *p + 5;
    a = 0;                }
    p = &a;             };
  }
};
```

Fig. 10 Example of null pointer analysis.

(1) A pointer variable "p" is used in the node.
(2) Whether "p" can be null at the node is checked.
  (a) Dependence edges about "p" are traversed backwardly from "b = *p + 5", and "p = NULL" and "p = &a" are found as nodes initializing "p"
  (b) Since there is a node where "p" is defined to null, "p" is found to have possibility to be null.

In the null pointer dereferencing detection, false warning problems may happen. This is due to the fact that the algorithms mentioned above examine the description only locally and also analysis on conditions may not be accurate. That is, some erroneous execution paths can be actually false paths if the execution starts from initial states. In order to solve these false warnings, reachability analysis from initial states must be performed. Model checking methods analyze that way and explained later.



```
Behavior B1(int x, event e){
  bool cond;
  void main(){
    x = 0;
    if(cond) notify(e);
  }
};
```

```
Behavior B2(int x, event e){
  void main(){
    wait(e);
    x = 2 * x;
  }
};
```

Fig. 11 An example for deadlock detection.

4.2 Detection of Deadlocks

Deadlocks occur when all concurrent processes are suspended and their executions cannot proceed any more. In SpecC designs, deadlocks happen when a suspended process by an wait node is not resumed by at least one of the *notify* nodes connected with the *wait* node through data dependence edges. Therefore, deadlocks are detected by checking whether at least one of the corresponding *notify* nodes is executed for each *wait* node.

Figure 11 gives an example source code and its SDG for the deadlock detection. In the example, there is an *wait* node "wait(e)" in a behavior "B2". The deadlock checking for "wait(e)" starts from collecting corresponding *notify* nodes by backwardly traversing data dependence edges of "e", and then the corresponding *notify* node "notify(e)" in behavior "B1" is found. Next, whether "notify(e)" nodes exist in each paths of "B1" is checked. In the example a "notify(e)" is in the "then" or "true" path of "if(cond)", and no "notify(e)" exists in the "else" or "false" path of "if(cond)". Therefore, we can say that deadlock may occur in this case.

There are many other items to be checked by static checking methods. For example, array overbounds accesses should be checked as much as possible since they may cause critical and security errors. For such analysis on overbounds of indexing variables must be computed as accurately as possible without spending much time[13].

Static checking is becoming a common tool for initial verification of the C based design descriptions. As can be seen from the above discussions, static checking is generally very quick since it analyzes only locally the design descriptions. On the other hand, static checking generates errors or warnings if there are local execution paths that lead to the errors or warning as can be seen above. Those errors or warnings are not true if the execution paths are false, that is, they are never activated. If the conjunction of conditions for an execution path is not satisfiable, that path becomes false. Since C descriptions have word variables, such as integer variables, both SAT solvers (e.g., Chaff[4]) and SMT solvers (e.g., CVC[5] and Z3[6]) are used. In order to completely check whether an execution path is false or not, the description must be model checked and all possible execution paths from initial states must be examined, which is generally infeasible. There are two difficulties. One is that execution paths are very long and have lots of conditions. If the conjunct of them becomes too large, neither SAT nor SMT solvers can deal with them. For efficient static checking, the length of execution paths must be kept small and that is the reason why the analysis must be very local. The other is that there are too many execution paths to be checked. If each execution path is checked one by one, the analysis simply never terminates. Recently false execution path analysis methods which can deal with sets of execution paths instead of analyzing them individually are proposed[14]. This could allow static checking methods to work more globally (or less locally).

5. Model Checking Methods

5.1 Model Checking Algorithm

Given a property, model checking methods check whether all possible execution paths from initial states satisfy the property. Several formal verification methods for C programs have been proposed. CBMC[15] verifies that a given ANSI-C program satisfies given properties by converting them into bit vector equations and solving satisfiability using SAT solver, such as Chaff[4]. All statements in C descriptions are automatically translated into Boolean formulae, and they are analyezd up to given specific cycles. Although CBMC and similar approaches can generally verify C descriptions, they cannot deal with large ones due to blow-up of SAT processing time. In C descriptions, there are word variables, such as integer variables, which have multiple bit-widths. If all such word variables are expanded into Boolean as CBMC does, resulting Boolean formulae to be analyzed simply become too large. It is much better to keep word variables as words instead of expanding them into Boolean. Some SAT solvers, called Satisfiability Modulo Theory (SMT) solvers, such as CVC[5] and Z3[6] can deal with multi-bit variables as they are. They are sorts of theorem provers and consist of several decision procedures. Although reasoning about C description can be much more efficient with SMT solvers, realistic sizes of C design descriptions are still simply too large to be processed.

Therefore, there have been proposed a number of automatic abstraction techniques, such as SLAM[7], BLAST[8], MAGIC[9], and others, by which large C design descriptions can sufficiently be reduced, and SAT/SMT solvers quickly reason about the reduced design descriptions. This automatic reduction is generally target property dependent. That is, given a property to be verified, portions of design descriptions which do not influence the correctness of the verification are automatically eliminated. Depending on characteristics of target properties, amount of reduction varies. For example, if the target is to check whether array accesses never exceed array bounds, most of the design descriptions are irrelevant and can be omitted for model checking. In the following, one such approach targeting synchronization mechanisms for parallel processes are presented[16],[17].

5.2 Abstraction-refinement Based Model Checking: Synchronization Verification

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. *Deadlock* is an error that is caused by synchronization failure. Static checking methods which have been introduced in the previous section can also be applied to checking deadlocks, race conditions for shared variables, and other synchronization verification items. Although static checking methods can deal with large descriptions, they may generate false errors or warnings due to their local analysis.

On the other hand, simple model checking cannot deal with large descriptions due to explosions of execution paths. Therefore, it is essential to apply abstraction methods that can reduce the sizes of descriptions to be model checked. The method presented here is based on application of the state-of-the-art model checking with counterexample-guided abstraction refinement (CEGAR) [10] and constraints solving with integer linear programming (ILP) techniques. Generally speaking, statements that control synchronization of concurrent processes are only small portions of the whole descriptions. By concentrating on synchronization verification, large portions of the given descriptions can be eliminated for model checking.

In SpecC, the computation and communication parts are clearly separated. Computation is encapsulated in *behaviors*, while communication is encapsulated in *channels*. Now let us consider *concurrency* in SpecC. Expressing behaviors within a *par* statement results in parallel execution of those behaviors. For example, *par{a.main(); b.main();}* in Fig. 5 implies that behavior a and b are concurrently running (in parallel). Within behaviors, statements are running in the sequential manner just like in the C programming language. Without any $notify/wait$ statements as shown in Fig. 5, the final values of x and y varies depending on schedulings. The two parallel behaviors a and b are shown in Fig. 6 (a) where the synchronization statements, *notify/wait*, are inserted into the one in Fig. 5. The statement *wait e* in behavior b suspends the statement $st3$ until the specified event e is notified. That is, it is guaranteed that statement $st3$ is safely executed right after statement $st2$. This eliminates the ambiguous results. Synchronization verification is to check if the corresponding "wait event" and "notify event" statements synchronize with each other as designers intend.

Behaviors and statements in SpecC can be defined on *time interval* which defines starting time and ending time of each behavior or statement described as *T(behavior/statement)s* and *T(behavior/statement)e*. The `behavior a` and `behavior b` in Fig. 6 can be represented with the following timing constraints.

- $Tas <= T1s < T1e <= T2s < T2e <= Tae$
  (sequentiality in a)
- $Tbs <= T3s < T3e <= Tbe$
  (sequentiality in b)

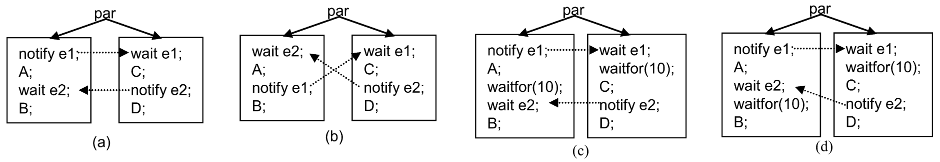


Fig. 12 Synchronization verification with ILP solvers.

- $Tas = Tbs$, $Tae = Tbe$
  (concurrency between a and b)
- $T2e < T3s$
  (synchronization of *notify* and *wait* statement)

Note that the length of each statement, $st1$, $st2$, $st3$, and the gap between them are non-deterministic. These equalities/inequalities are precisely representing the non-determinism of the execution of statements in SpecC.

Figure 12 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 mixed integer linear programming problems with real number representations for the timing constraints on behaviors and statements as shown above and integer representations for logical constraints derived from conditional statements 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 of the design descriptions. Counterexample-Guided Abstraction Refinement (CEGAR) [10] 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 generated 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. Please note that due to the abstraction

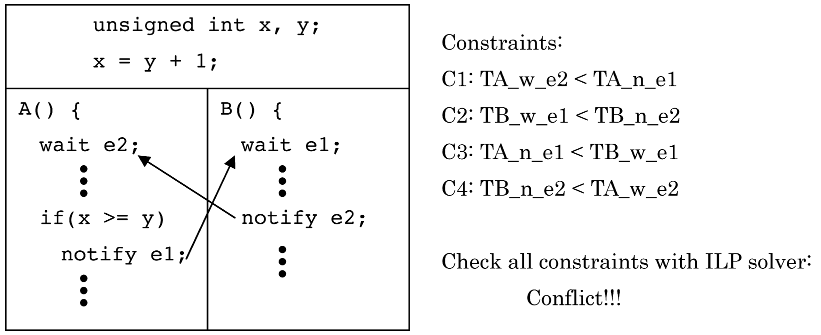


Fig. 13 A and B are running in parallel. Constraints $C1$ and $C2$ represent sequentiality in A and B, respectively. Constraints $C3$ and $C4$ represent synchronization of *notify/wait* of $e1$ and $e2$, respectively.

of the design description, counter examples for the abstracted descriptions may not be realized in the original description. 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. There are several tools which are based on this strategy 7)–9). They are using similar but different abstraction techniques depending on the targets of designs and also on the types of properties.

Figure 13 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 Fig. 13, we consider only the synchronization statements and omit others. The basic idea for synchronization verification is to show inconsistency or conflict on the conjunctions of the negation of a property and the constraints extracted from the descriptions. If the conjunctions are conflicting, the negation of the property cannot be satisfied due to the constraints extracted from the descriptions. That is the property is satisfied by the descriptions. 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 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.

Table 1 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	2,000	819	6	2	21.1	0
MPEG4	48,126	781	5	1	9.7	0

Several experiments are conducted on Pentium4 2.8 GHz machine with 2 GB RAM running Linux. The results of synchronization verification are shown in **Table 1**. Runtimes are in seconds. A counterexample is generated whenever a property does not hold. This counterexample shows 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. We found no deadlocks in all benchmarks.

As can be seen in Table 1, the verification of MPEG4 descriptions considers only very small portion of the descriptions (about 800 lines) instead of the entire description (about 48,000 lines). By focusing on the synchronization verification sizes of the models that need to be considered can significantly be reduced.

5.3 Integration with Static Checking Methods

Model checking methods use automatic abstraction techniques for the reduction of design descriptions to be analyzed. This is an effort to make model checking analysis applied to as small descriptions as possible. On the other hand, static checking methods shown in the previous section try to extend the descriptions to be analyzed as large as possible without much increases of processing time.

Owning to the recent advancements of SAT and SMT solvers, model checking and static checking methods are becoming similar. Depending on how much time is allowed for analysis, either static checking or model checking methods can be used. There are efforts which integrate model checking and static checking methods. For example, F-SOFT[18] is targeting software verification and incorporate pointer analysis methods such as the one in Ref. 19) as well.

6. Equivalence Checking Methods

6.1 Equivalence Checking from Specific Aspects

Equivalence checking is a key technology to make sure that lower level design descriptions keep essentially the same behaviors as the ones in higher level descriptions and bugs are not inserted into. Although equivalence checking between the two descriptions can be performed with model checking methods by connecting the outputs of the two descriptions and checking appropriate properties on the connected outputs, this method does not scale for large descriptions. Therefore existing equivalence checking methods for C descriptions are targeting some specific types of equivalence. Several such equivalence checking methods that can be applied to C-based designs in system-level or behavior-level have been proposed. In Ref. 20), an equivalence checking method for scheduling of processes is presented. It can efficiently verify the equivalence of scheduling in which the computation algorithms are preserved. In Ref. 21), on the other hand, the equivalence of optimizations within loops is verified. In the following, an equivalence checking method which can deal with large design descriptions as long as the two descriptions are reasonably similar is presented. In a typical system level design flow, designs are refined incrementally, and design descriptions for two neighbouring design steps are naturally similar. The method shown below works well for such situations.

6.2 Equivalence Checking Method with Difference Identification

The method is based on identification of differences between the two descriptions to be compared[22] as shown in **Fig. 14**. First of all differences are recognized by some analysis. They are the first target for equivalence checking. If they are equivalent, then clearly the whole descriptions are equivalent. If they are not equivalent, however, the areas to be analyzed for equivalence must be extended

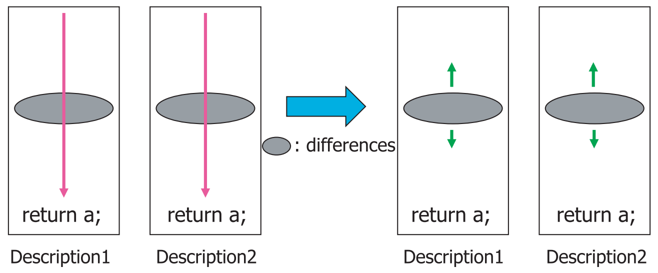


Fig. 14 Equivalence checking based on difference identification.

as shown in the figure. This is because differences may not propagate to primary outputs and/or primary inputs may not reach the differences. Therefore, the overall equivalence checking flow becomes the one shown in **Fig. 15**. Two C descriptions are given with the definition of input and output variables. In addition, the correspondence of those variables between the two descriptions is also given. The method verifies the equivalence of the output variables basically by using symbolic simulation and reports the verification result ("equivalent" or "not equivalent").

Target C descriptions should satisfy the following restrictions.

- No pointer uses (or all pointer uses are analyzed and replaced by certain variables through point-to analysis) nor dynamic memory allocation
- Loops are unrolled in a certain times in advance
- No recursive function calls

These restrictions come from the limitation of symbolic simulation. Therefore, if these statements are out of design descriptions that need to be verified by symbolic simulation, it does not matter whether given C descriptions have these statements or not. As explained later, symbolic simulations are applied only to the different portions and their neighbours of the two descriptions. A simple way to identify the difference between the two descriptions is based on textual differences, such as the ones reported by UNIX command "diff". The difference gives good hints where the equivalence of the two descriptions must be verified by symbolic simulation. If the different portions of the two descriptions are verified to be equivalent, it is clear that the whole descriptions are equivalent as well.

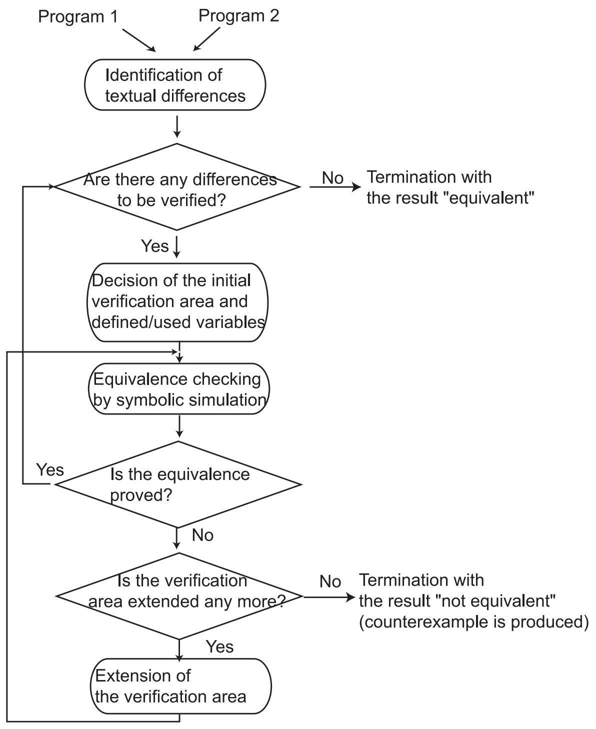

Program 1
Program 2
Identification of textual differences
Are there any differences to be verified?
No
Termination with the result "equivalent"
Yes
Decision of the initial verification area and defined/used variables
Equivalence checking by symbolic simulation
Is the equivalence proved?
Yes
No
Is the verification area extended any more?
No
Termination with the result "not equivalent" (counterexample is produced)
Yes
Extension of the verification area

Fig. 15 Our proposed verification flow.

On the other hand, even if the different portions are not equivalent, the whole descriptions can still be equivalent due to the other portions. Therefore areas to be symbolically simulated must be extended. Extensions can continue until either the two areas symbolically simulated are proved to be equivalent or extensions reach primary inputs and outputs (in this case the two descriptions are proved to be inequivalent).

For the purpose of making correspondence between statements in both descriptions, dummy statements are inserted to the descriptions in the following cases to let both descriptions have corresponding statements with each other.

- When an assignment is removed, the assignment to the same variable such as $a = a$; is inserted.
- When a conditional branch is removed, the same branch structure is inserted where all assignments are replaced by ones to the same variable.

Since these inserted statements clearly preserve the original behavior, the result of verification does not change. Even if many statements are different, the descriptions after inserted dummy statements are less than twice of the original descriptions.

Then SDGs for both descriptions are constructed. At the same time, statements are removed from SDGs when they do not affect any output variables and are not affected by any input variables just like program slicing. This reduction can be performed on SDGs. Verification areas can be represented with a set of SDG nodes, since each node corresponds to a statement in C descriptions. The initial verification area for a difference is two sets of SDG nodes corresponding to the difference (one set from each description). Note that a difference may consist of several statements. We define input variables and output variables of a local verification area as shown below.

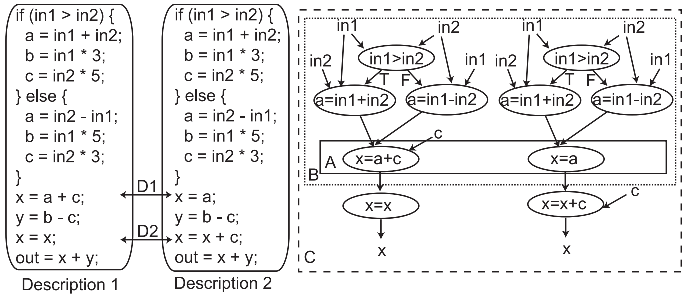
- **Local input variable** a variable corresponding to a data dependence edge coming from out of the verification area to the area
- **Local output variable** a variable corresponding to a data dependence edge coming from the verification area to out of the area

If a variable is an output variable appearing in both descriptions, its equivalence is checked by symbolic simulation. Although other local output variables (appearing only one of the two descriptions) are not checked for this difference, they will be taken into account in verification for other differences later.

A pair of corresponding local input variables is equivalent in the following cases.

- They are not affected by any differences that are proved to be inequivalent.
- They are already proved to be equivalent by the verification of other differences.

Equivalences of other pairs of local input variables are considered to be unknown. If all pairs of local output variables are proved to be equivalent, the verification area of the difference is also proved to be equivalent. On the other hand, if the equivalence of any local output variables are not proved, the verification area is extended so that preceding and/or succeeding statements are included. These are determined by tracing dependency on SDGs.



Description 1:

```
if (in1 > in2) {
  a = in1 + in2;
  b = in1 * 3;
  c = in2 * 5;
} else {
  a = in2 - in1;
  b = in1 * 5;
  c = in2 * 3;
}
x = a + c;
y = b - c;
x = x;
out = x + y;
```

Description 2:

```
if (in1 > in2) {
  a = in1 + in2;
  b = in1 * 3;
  c = in2 * 5;
} else {
  a = in2 - in1;
  b = in1 * 5;
  c = in2 * 3;
}
x = a;
y = b - c;
x = x + c;
out = x + y;
```

Fig. 16 A verification example based on difference identification.

The way to extend the verification area are shown below.

- **Backward extension** Adding a directly preceding SDG node that has a data dependence to any local input variable
- **Forward extension along data dependence** Adding a directly succeeding SDG node that has a data dependence from any local output variable
- **Forward extension along control dependence** Adding all directly succeeding SDG nodes that have a control dependence from any local output variables (This extension can be carried out if any condition nodes are proved to be inequivalent)

During extensions, multiple SDGs that represent assignments to the same variable are added to the verification area when control dependences of them are different. In such cases, the nodes that control these assignments are also added. After the extensions, the local input/output variables are generated for the new verification area, and verification by symbolic simulation is carried out.

Extensions terminate in the following ways.

- If the equivalences of added SDG nodes are already proved, no backward extension is applied from them
- If added statements are the top (or end) of programs, no backward extension (or forward extension) is applied from them

A simple verification example is shown in **Fig. 16**. We assume that the variables $in1$ and $in2$ are the primary inputs of the program, and the variable *out* is the primary output. The statement $x = x$; in Description 1 is added as a dummy statement to make a correspondence to $x = x + c$; in Description 2.

At first, the first difference $D1$ is verified. The first verification area is A in the figure, and its local input variables are a and c, and its local output variable the variable x. Since all local input variables are unknown, the equivalence of x cannot be proved. Thus, in this case, we decide to backwardly extend the area from a.

Then, the extended verification area becomes the area B, and the verification is carried out again. In this case, the local input variables are $in1$, $in2$, and c, and the local output variables are x and $(in1 > in2)$. Since the equivalence of x cannot be proved after the verification with the area B, we decide to forwardly extend the area from x and obtain the area C.

After the verification with this area C, we can prove the equivalence of x. The verification for the difference $D2$ is not carried out, since it is included the verification for $D1$. Then, as the all difference is verified, it can be said that the two descriptions are functionally equivalent.

In general, a verification area can have multiple local input/output variables. Therefore, there are a number of combinations to apply backward and forward extensions. Reasonable strategies for differences usually happening in practice are shown below.

- Applying backward extensions until the start points of the programs, then applying forward extensions until end points
- Applying forward extensions and backward extensions in turn
- First, applying backward extensions m times, then applying forward extensions n times (m, n are pre-defined number)

These strategies are similar to ones in equivalence checking of gate-level circuits, such as the ones shown in Refs. 23), 24). In some cases, designers understand which kinds of refinements are carried out. If so, a specific strategy for the refinement can be applied to improve the verification efficiency.

The method is evaluated with the following design examples written in C language.

- Common sub-expression eliminations in a differential equation solver (total 130 lines, differences are 10 parts, 30 lines)
- Refinements in IDCT(Inverse Discrete Cosine Transform) (total 420 lines, differences are 16 parts, 96 lines) from MPEG2 encoder/decoder

Table 2 Experimental results.

	result	time	verified nodes	total nodes
diffeq1	eqv	0.7 sec	60	288
diffeq2	ineqv	0.7 sec	73	288
mpeg1	eqv	1.8 sec	192	1,160
mpeg2	ineqv	0.9 sec	62	1,160
rijndael1	eqv	0.3 sec	240	4,112
rijndael2	ineqv	0.6 sec	44	4,112

- Refinements from 4-Xor into 2-Xor in the encryption function (total 1,235 lines, differences are 40 parts, 120 lines) from Rijndael encryption program

The refinements in IDCT is to reduce the computation, and it has applied combinations of common sub-expression elimination and factorization. All experiments were carried out on PC with 2.4 GHz processor and 2 GB memory.

The experimental results are shown in **Table 2**. All verification results are the same as what we have intended. As shown in the table, the numbers of SDG nodes that are symbolically simulated are much smaller than the total SDG nodes in the programs. This is seen especially in the inequivalent cases. This is because the result can be concluded to be inequivalent if a counterexample is found.

As for the comparison of verification times with the method that simply analyze the whole programs, the proposed method takes shorter times to verify when the verified programs are relatively large. For example, equivalence checking with symbolic simulation of the whole IDCT example, which has eight conditional branches, takes more than 800 sec, while the proposed method takes 1.8 sec as shown in the table.

In addition, symbolic simulation for the whole MPEG2 or Rijndael cannot be carried out in practical time. Therefore, our approach where only the portions related to the differences are symbolically simulated is effective especially when a given program is very large.

7. Conclusions

Three approaches for formal verification of C based design descriptions have been presented. They are all relying on efficient analysis of various dependence found in the descriptions. Static checking methods and model checking methods are becoming similar, and both are targeting larger descriptions by narrowing the areas to be actually analyzed. Equivalence checking methods works well as long as the two description to be compared are reasonably similar.

There are significant on-going research efforts on the topics targeting not only on hardware descriptions in C based languages but also their software descriptions. Owing to recent advances of SAT and SMT solvers, larger and more complicated descriptions can now be formally verified. Moreover, industrial formal verification tools following the researches shown in the paper are becoming available. Through their applications to real designs, new insights for future research direction can be obtained and further advances may be observed in the near future.

References

1) Gajski, D., Zhu, J., Doemer, R., Gerstlauer, A. and Zhao, S.: *SpecC: Specification Language and Methodology*, Kluwer Academic Publisher (Mar. 2000).
2) *IEEE 1666 Language Reference Manual*, IEEE (2005).
3) Clarke, E.M., Kroening, D. and Lerda, F.: A Tool for Checking ANSI-C Programs, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, *Lecture Notes in Computer Science 2988*, pp.168–176, Springer-Verlag (2004).
4) Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L. and Malik, S.: Chaff: Engineering an Efficient SAT Solver, *Proc. Design Automation Conference '01* (2001).
5) Barrett, C. and Tinelli, C.: CVC3, *Proc. 19th International Conference on Computer Aided Verification (CAV '07)*, *Lecture Notes in Computer Science 4590*, pp.298–302, Springer-Verlag (July 2007).
6) deMoura, L. and Bjørner, N.: Z3: An Efficient SMT Solver, *Proc. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary (2008).
7) Ball, T. and Rajamani, S.K.: The SLAM Project: Debugging System Software via Static Analysis, *Proc. POPL*, pp.1–3 (Jan. 2002).
8) Henzinger, T., Jhala, R., Majumdar, R. and Sutre, G.: Software verification with Blast, *Tenth International Workshop on Model Checking of Software (SPIN)*, *Lecture Notes in Computer Science 2648*, pp.235–239, Springer-Verlag (2003).
9) Chaki, S., Clarke, E.M., Groce, A., Jha, S. and Veith, H.: Modular Verification of Software Components in C, *IEEE Transactions on Software Engineering (TSE)*, Vol.30, No.6, pp.388–402 (June 2004).
10) Clarke, E.M., Grumberg, O., Jha, S., Lu, Y. and Veith, H.: Counterexample-Guided Abstraction Refinement for Symbolic Model Checking, *J. ACM (JACM)*, Vol.50, No.5, pp.752–794 (Sept. 2003).

11) Horwitz, S., Reps, T. and Binkley, D.: Interprocedural Slicing Using Dependence Graphs, *ACM Transactions on Programming Languages and Systems*, Vol.12, No.1, pp.26–60 (1990).
12) Sasaki, S., Nishihara, T., Ando, D. and Fujita, M.: Hardware/Software Co-design and Verification Methodology from System Level Based on System Dependence Graph, *Journal of Universal Computer Science*, Vol.13, No.13, pp.1972–2001 (2007).
13) Sankaranarayanan, S., Ivancic, F. and Gupta, A.: Program Analysis Using Symbolic Ranges, *Proc. International Static Analysis Symposium* (*SAS*) (2007).
14) Balakrishnan, G., Sankaranarayanan, S., Ivancic, F., Wei, O. and Gupta, A.: SLR: Path-Sensitive Analysis through Infeasible-Path Detection and Syntactic Language Refinement, *Proc. International Static Analysis Symposium* (*SAS*) (2008).
15) Clarke, E., Kroening, D. and Yorav, K.: Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking, *Proc. Design Automation Conference '03*, pp.368–371 (2003).
16) Sakunkonchak, T., Komatsu, S. and Fujita, M.: Synchronization Verification in System-Level Design with ILP Solvers, *IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences*, Vol.E89-A, No.12, pp.3387–3396 (Dec. 2006).
17) Sakunkonchak, T., Komatsu, S. and Fujita, M.: Using Counterexample Analysis to Minimize the Number of Predicates for Predicate Abstraction, *Proc. 5th International Symposium on Automated Technology for Verification and Analysis*, pp.553–563 (Oct. 2007).
18) Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P., Ivančić, F. and Yang, Z.: F-Soft: Software Verification Platform, *Proc. Computer Aided Verification* (2005).
19) Séméria, L. and De Micheli, G.: SpC: Synthesis of Pointers in C — Application of Pointer Analysis to the Behavioral Synthesis from C, *Proc. International Conference on Computer Aided Design* (1998).
20) Abdi, S. and Gajski, D.: Functional Validation of System Level Static Scheduling, *Proc. Design, Automation and Test in Europe '05*, pp.542–547 (Mar. 2005).
21) Shashidhar, K.C., Bruynooghe, M., Catthoor, F. and Janssens, G.: Functional Equivalence Checking for Verification of Algebraic Transformations on Array-Intensive Source Code, *Proc. Design, Automation and Test in Europe '05*, pp.1310–1315 (Mar. 2005).
22) Matsumoto, T., Saito, H. and Fujita, M.: An Equivalence Checking Method for C Descriptions Based on Symbolic Simulation with Textual Differences, *IEICE Trans. on Fundamentals*, Vol.E88-A, No.12, pp.3315–3323 (Dec. 2005).
23) Jain, J., Mukherjee, R. and Fujita, M.: Advanced Verification techniques based on learning, *32nd ACM/IEEE Design Automation Conference* (1995).
24) Matsunaga, Y.: An Efficient Equivalence Checker for Combinational Circuits, *Proc. 33rd IEEE/ACM Design Automation Conference* (1996).

(Received September 1, 2008)
(Released February 17, 2009)

(Invited by Editor-in-Chief: *Hidetoshi Onodera*)

Masahiro Fujita received the B.S. degree in electrical engineering in 1980, and the M.S. and Ph.D. degrees in information engineering from the University of Tokyo, Tokyo, Japan, in 1982 and 1985, respectively. From 1985 to 1993, he was a Research Scientist with Fujitsu Laboratories, Kawasaki, Japan. From 1994 to 1999, he was the Director of the Advanced Computer-Aided Design Research Group, Fujitsu Laboratories of America, Sunnyvale, CA. He is currently a Professor in VLSI Design and Education Center, University of Tokyo, Tokyo, Japan. He has been on program committees for major conferences dealing with digital design and is an Associate Editor of Formal Methods on Systems Design, ACM Transaction on Embedded System, and ACT Transaction on Storage. His primary research interest is in the computer-aided design of digital systems. Dr. Fujita received the Sakai Award from the Information Processing Society of Japan in 1984.