論 文 の 内 容 の 要 旨

論文題目:　Formal Verification of Synchronization in System-Level Design
(システムレベル設計における形式的同期検証に関する研究)

氏 名　サクンコンチャック　タンヤパット

1. はじめに

現在、多くの電子機器は、ハードウェアとソフトウェアを組み合わせたシステムとして設計されている。これらのシステムの設計は、半導体微細化技術の進歩と要求される仕様の高度化により、急速に複雑さが増している。一方で、集積度向上の伸びに比べて設計生産性の伸びは低く、その差が広がっており、計算機による設計支援が重要になっている。集積度向上の結果、一つのシリコンチップ上にシステム全体を集積するシステムオンチップ（System-on-a-Chip）が実現されている。このような大規模システムでは、システム全体の実行プロセスは、いくつかの並列に動作する実行プロセスの集まりとして成り立っている。そのため、１つのプロセスに注目するだけでは、システム全体の動作を把握することが困難である。このような並列プロセスを設計者の意図通り実行させるためには、プロセス間の実行順序のスケジューリング(同期)が不可欠である。

本論文では、システムレベル設計で用いられる C ベース設計言語の１つである、SpecC 言語におけるモデル検査法に基づく同期検証を提案する。検証するモデルの規模を小さくするために、提案手法では述語抽象化法を利用している。提案手法では、SpecC プログラム中でデッドロックが起こりうるかどうか、つまり、あるプロセスの実行が停止したままになりうるかどうかを形式的に検証することができる。述語抽象化を施した抽象モデルにおいてデッドロックが起こらないことが証明された場合には、元の SpecC プログラムでもデッドロックが起こることはない。しかし、モデル検査の結果として反例（デッドロックを起こす実行シーケンス）が得られた場合には、その反例を解析し、元の SpecC プログラム中で反例に対応する実行シーケンスが存在しない場合には、抽象モデルを改良し、さらに検証を続ける必要がある。また、反例に対応する実行シーケンスが元のプログラム中に存在する場合には、デッドロックが起こりうるという結果を得る。なお、提案する手法は完全に自動化することが可能である。

2. モデル検査法における抽象化技術

モデル検査法は、分かりやすく、完全に自動化が可能な手法であるため、広く用いられている検証手法である。モデル検査法では、有限状態から成る並行動作システムにおいて、あるプロパティ（仕様）が成り立つかどうかを証明することができる。しかし、この手法では、状態数が膨大になってしまうと検証が現実的な時間では終了しないという問題点があり、これを解決するために、検証するモデルの状態数を減らす多くの手法が提案されている。その中でも、抽象化は非常に有用な手法である。抽象化とは、具体モデルからより状態数の少ない抽象モデルを導出する手法である。そのため、抽象モデルでは高速にモデル検査法によって検証を行うことができる。しかし、抽象モデルでは、具体モデルでは存在しない実行シーケンスが生じる場合がある。そのため、抽象モデルで証明されたプロパティは具体モデルでも成り立つが、抽象モデルで証明できなかったプロパティであっても具体モデルでは成り立つ可能性がある。

モデル検査器があるプロパティの検証に失敗した場合には、反例が作られる。しかし、抽象化が適用されていると、この反例が false negative である可能性がある。そのため、この反例を解析し、その結果 false negative である場合には、この反例が存在しなくなるように抽象モデルを改良する必要がある。

3. SpecC 言語における同期

SpecC 言語は、仕様設計から RTL(register transfer level)に至るシステム全体を記述するためのシステム設計言語である。そのため、ハードウェアとソフトウェアをシームレスに記述することができるという性質を持つ。SpecC 言語では、par 文を用いて並列プロセスを記述することができる。例えば、図 1 のように par { a.main(); b.main() }と記述すれば、behavior a と behavior b が並列に動作することになる。それぞれの behavior 内では、各文は C 言語のように順次実行される。その結果、behavior a におけるタイミング制約は、

$$Tas \leqq T1s < T1e \leqq T2s < T2e \leqq Tae$$

となる。なお、T1, T2 は図中の文 st1, st2 に対応するタイミングであり、添え字 s, e は実行の開始と終了に対応する。この段階では、各文の実行順序は st1→st2→st3, st3→st1→st2, st1→st3→st2 のいずれであるか決定することはできない。この場合、意図しない結果や同じ変数 x への同時アクセスが起こり得る。この問題に対応するため、SpecC 言語ではイベント受け渡しによる同期の構文（notify/wait）が提供されている。wait が実行されると、対応するイベントを引数とする notify が実行されるまで、wait 側の実行が停止する。図 2 では、図 1 のプログラムに wait/notify を挿入することによって同期を実現している。その結果、st3 が st2 の後に実行されることが保証される。

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

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

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

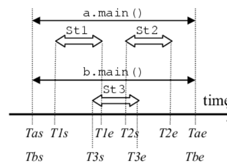

図 1: behavior a と behavior b の実行タイミングダイアグラム

```
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*/}}
```

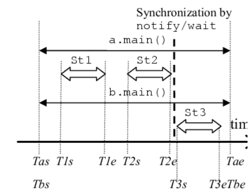

図 2: 図 1 のプログラムに対する同期（notify/wait）の導入

4. 同期検証

ここでの同期検証の目的は、与えられた並列実行を含む SpecC プログラムが正しく同期されているかどうかを検証することである。SpecC プログラムの抽象化手法として、ブールプログラムの概念を応用している。検証の流れを図 3 に示す。この検証全体は、完全に自動化することが可能である。以下では、検証の各ステップを簡単に述べる。

1. 与えられた SpecC プログラムを抽象化し、boolean SpecC を導く。この抽象化では、
   ・イベントに関する文（wait/notify）はそのまま残す
   ・条件分岐の条件を表す式は互いに独立な新たな変数によって置き換える（例えば、if(x>0)は if(C0)で、if(y＜3)は if(C1)で置き換えられる）
   ・その他の全ての文は、何も実行を行わない skip 文と置き換えられる
2. boolean SpecC から Tas≦T1s＜... のような各文の実行のタイミング制約を表す数学モデルを導く
3. モデル検査法に基づいてタイミング制約を表したモデルを検証する。ここでは、Difference Decision Diagram（DDD）を用いて検証を行う。DDD は、decision diagram の一種であり、等号・不等号で表される条件式を効率的に表現できるという特長を持つ。モデル検査のプロパティとしては、「デッドロックが起こらない」ことを与える。モデル検証の結果は、
   ・デッドロックは起こりえない（この場合、「検証成功」となり終了する）
   ・デッドロックが起こりえる（この場合、反例もともに出力される）
   のどちらかとなる。
4. Cooperating Validity Checker (CVC) のような式の成否を判定するツールによって、得られた反例が元のプログラム中でありえるかどうかを解析する。反例が元のプログラムではありえないシーケンスであることが証明されれば、次の改良の段階へと進む。そうでない場合には、「検証失敗」として、元の SpecC プログラムでの反例を出力して、検証を終了する。
5. 抽象モデルでの反例が、元のプログラム中で対応するシーケンスが存在しない場合、過度の抽象化を行っていることが原因である。そのため、この「偽の」反例を作る原因となっている条件変数を特定して改良を行い、新たな boolean SpecC プログラムを導き、検証を繰り返す。

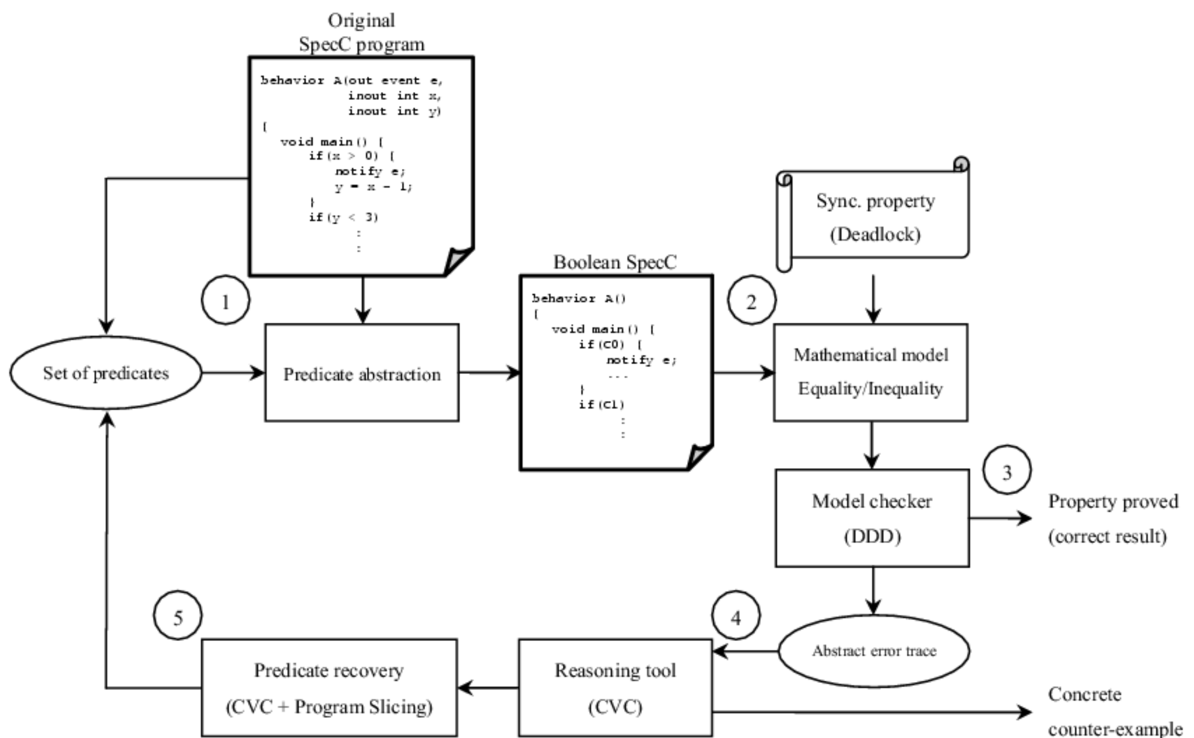

図 3: 提案する同期検証手法

6. 結論

本論文では、SpecC 言語で記述されたシステムの同期検証の手法を提案した。提案手法は、SpecCプログラムの抽象化と反例に基づいた抽象モデルの改良に基づいたものである。ブールプログラムの概念を取り入れることによって、同期に関係する wait/notify 文と条件分岐に関する条件文以外の全ての部分を抽象化し、高速なモデル検査を実現した。モデル検査では、デッドロックに関するプロパティを与えて検証を行うが、検証に失敗した場合には、デッドロックが起こるような抽象モデル上での反例が返される。そこで、その反例の解析を行い、もし元のプログラムでは存在しない反例である場合には、この「偽の」反例が生じる原因となっている条件式を特定し、抽象モデルを改良する手法を示した。

システム設計において、同期検証は最も重要な検証事項の一つである。提案手法は、同期に関する部分のみを残す抽象化手法を適用することにより、同期検証に特化した、大規模設計記述に対応可能な検証手法となっている。