변수 개수 최소화를 통한 소프트웨어 모델 체킹 성능향상 연구

이낙원, 박진희, 백종문

한국과학기술원 전산학과
대전광역시 유성구 대학로 291
{skrdnjs, jh_park, jbaik}@kaist.ac.kr

요약: 모델 체킹을 통한 소프트웨어 검증은 소프트웨어의 모든 가능한 동작 상태를 확인 함으로써 소프트웨어가 결함을 가지고 있지 않다는 것을 보증할 수 있는 강력한 방법이다. 그러나 소프트웨어의 복잡성으로 인해 확인해야 하는 상태의 범위는 소프트웨어의 크기가 커질수록 기하급수적으로 증가 하게 되고 이로 인해 그리 크지 않은 크기의 소프트웨어 임에도 현재의 컴퓨터 성능으로 검증이 불가능한 상황이 자주 발생한다. 이런 모델 체킹의 부족한 확장성을 보완하고자 많은 연구자들은 검증의 유효성은 그대로 유지하면서 확인 해야 하는 상태의 범위를 줄이는 방법에 대해 고민해 왔다. 그러한 연구의 결과로 심볼릭 모델 체킹 기법이 등장 하였으나 이는 변수의 배열 순서에 성능이 좌우되는 문제를 안고 있다. 본 연구에서는 소프트웨어의 변수 개수를 최소화시키는 기법을 통하여 확인해야 하는 상태의 수를 줄이는 기법을 제안한다. 이 기법은 기존 심볼릭 모델 체킹 기법들의 전처리 과정으로 이용 될 수 있으며 변수 배열 순서와 상관없이 성능을 향상시킬 수 있다.

핵심어: 모델 체킹, 소프트웨어 검증, 정형 기법, 모델, 변수개수 최소화

1. 연구배경

안전 필수 소프트웨어의 결함은 심각한 재산 손실이나 인명 피해를 발생 시킬 수 있기 때문에 이러한 소프트웨어의 품질을 보증하는 것은 매우 중요하다. 일반적으로 품질 보증을 위해 사용되는 테스팅 기법은 소프트웨어에 내제된 결함을 찾아 낼 수는 있으나 소프트웨어가 결함을 가지고 있지 않다는 것을 보장할 수 없다. 반면에 모델 체킹을 통한 소프트웨어 검증은 소프트웨어의 모든 가능한 동작 상태를 빠짐없이 확인 함으로써 소프트웨어에 결함이 없음을 보증 할 수 있는 방법이다. 따라서 안전 필수 소프트웨어의 품질을 보증하는데 모델 체킹과 같은 방법은 매우 유용하다[10][13].

그러나 모델 체킹을 통한 검증은 매우 큰 컴퓨팅 파워를 필요로 한다는 문제를 안고 있으며 이는 소프트웨어가 가진 복잡성에서 기인한다. 초기에 모델 체킹은 전자 회로의 검증을 위해 많이 사용 되었다. 전자 회로는 제한된 논리회로의 조합에 따라 그 행위가 결정되기 때문에 회로가 가지는 상태가 비교적 적다. 반면에 소프트웨어는 많은 변수와 명령어 줄로 이루어져 있기 때문에 전자 회로와는 비교할 수 없을 만큼 다양한 동작 상태를 가질 수 있다. 여기에 통신이나 병렬 프로그램 기법까지 포함하면 매우 많은 상태의 수를 가지게 된다. 일반적으로 소프트웨어의 모델링은 변수 값의 조합을 통해 상태를, 명령어를 통해 상태의 전이를 나타내기 때문에 소프트웨어의 크기가 증가하면 즉, 변수의 개수나 명령어의 개수가 늘어나게 되면 소프트웨어의 동작 상태의 수는 기하급수 적으로 증가하게 된다.

이러한 모델 체킹의 확장성 문제를 해결하고자 많은 연구자들은 모델 체킹이 가지는 검증 능력을 그대로 유지하면서 성능을 높이고자 노력해 왔다[2][5][6][11][12]. 그리고 그러한 노력을 통해 제시된 기법중의 하나가 Binary Decision Diagram (BDD)[1]을 이용한 심볼릭 모델 체킹 기법이다. BDD 를 이용한 심볼릭 모델 체킹 기법은 하나의 자료구조로 여러 개의 상태를 저장하는 것으로 여러 개의 상태에 대한 연산을 한번에 수행할 수 있기 때문에 상태 저장에 필요한 공간과 상태 전이의 횟수를 획기적으로 감소시킬 수 있다. 그러나 BDD 자료구조를 이용한 상태 표현 및 연산은 트리 자료구조를 표현하는 변수의 순서에 따라 그 성능이 크게 좌우된다. BDD 자료구조의 최적화된 변수 순서를 찾는 것은 NP-hard 의 문제로 휴리스틱 알고리즘 등을 통해 최적의 순서와 근사한 순서를 찾으려고 노력할 뿐이다. 일부 최악의 경우에는 변수 순서에 의해서 성능 향상을 볼 수 없는 경우도 발생한다[2].

변수 순서와 관계 없이 성능 향상을 이루기 위한 방법으로 우리는 변수의 수를 줄이는 방법을 이용한다. 소프트웨어에는 프로그래머에 의해 불필요한 변수들이 선언되어 있을 것이며 프로그램의 결함과 연관되지 않은 변수들도 존재할 것이라는 가정을 토대로 이 변수들을 제거하는 기법을 통해 심볼릭 모델 체킹의 성능을 향상시킨다. 이러한 방법을 적용하여 원래의 소프트웨어를 최소 필요 변수 개수만을 이용

해 수정하고 수정된 소프트웨어를 모델 체킹을 통해 검증한다. 불필요한 변수가 모두 제거되고 나면 유지해야 하는 변수의 개수가 줄어들고 이를 통해 BDD 자료 구조의 크기도 줄어 검증의 성능을 향상 시킬 수 있다.

이 논문의 나머지 부분에서는 2 장에서 관련 연구를 소개하고 3 장에서 본 기법을 위한 기본 개념인 Control Flow Graph (CFG)와 Binary Decision Diagram (BDD)에 대하여 설명한다. 4 장에서는 제시할 기법을 위한 변수 제거의 이론적 근거를 설명하고 5 장에서 프로그램 수정 기법을 제안한다. 그리고 6 장에서 결론과 향후 연구 계획을 설명하고 마무리 한다.

2. 관련 연구

모델 체킹의 성능을 향상시키기 위한 다양한 방법이 연구되어 왔다. 심볼릭 모델 체킹에 관련된 방법으로는 [5]에서 다양한 방법을 소개하고 있다. 이 연구에서는 전체를 구성하는 개별 구조에 대하여 각각 검증을 실시한 후 전체 시스템의 성질을 추론해 내는 compositional reasoning 을 통하여 Binary Decision Diagram (BDD)기반의 심볼릭 모델 체킹 성능을 향상시키는 방법들을 소개하였다.

그 중 [6]에서는 partitioned transition relation 이란 방법을 사용하여 전이를 위한 조건 식을 표현한 BDD 의 크기를 줄이는 방법으로 성능을 향상시킨다. 이 방법에서는 하나의 큰 전이식 대신 구성하는 변수의 개수가 적은 여러 전이식의 조합으로 전체 전이를 표현한다. 이때 변수간의 의존성을 고려하여 조합의 순서를 결정하기 때문에 이 순서를 찾는 것 또한 문제로 남는다. 하지만 변수개수를 줄여 BDD 의 크기를 줄인다는 점에서 우리의 기법과 유사하며 이는 BDD 연산 내에서 일어나는 과정이기 때문에 우리의 기법은 이 기법 이전에 전처리 과정에서 사용 가능하다.

전이를 위한 BDD 의 크기를 줄이는 또 다른 방법은 시스템의 전체 전이를 모두 포함할 필요 없이 특정 상황에서 영향이 있는 전이만을 이용하여 BDD 를 구성 하는 것이다[7]. 이 방법은 전이를 표현한 식의 변수 개수가 줄어들지는 않으나 고려하는 전이의 개수가 줄어들어 BDD 가 간결해지는 효과가 있다. 이 기법 또한 BDD 연산 시에 사용되는 방법으로 모델을 생성하기 전에 적용되는 우리의 기법을 전처리 과정으로 사용 가능하다.

Interface processes 는 [5]에서 제시된 기법으로 우리의 기법과 유사하게 문제와 관련된 변수만을 이용하여 상태공간의 크기를 줄이는 방법을 제시한다. 이 방법은 병렬 처리를 하는 프로세스들을 기반으로 모델 체킹 시에 서로 다른 프로세스가 아주 소수의 변수에 의해 의사소통 하고 있음에도 두 프로세스를 과도하게 병렬화 시켜 처리 함으로써 상태가 많아지는 문제를 해결하고 있다. 여기서는 프로세스간에 공유하고 있는 변수만을 이용하여 공유하는 변수와 전혀 관련되지 않는 변수들로 이루어진 명령어 줄들을 제거 함으로써 한 프로세스 입장에서 다른 프로세스의 크기가 줄어드는 효과를 얻는다.

위 기법들은 대부분 모델 기술 언어를 이용한 제한된 환경에서의 모델 체킹을 지원하는 것들로 실제 소프트웨어를 그대로 이용한 모델 체킹 기법들도 존재한다. CBMC [8][9]는 bounded 모델 체킹 기법의 대표적인 도구로서 c 언어로 작성된 프로그램에 대한 모델 체킹 기능을 제공한다. bounded 모델 체킹은 프로그램에 포함된 반복문을 제한된 횟수의 분기문으로 바꾸어 수정된 프로그램을 검증하는 방식이다. CBMC 는 Satisfiability (SAT) 식 또는 Satisfiability Modulo Theories (SMT) 식의 형태로 프로그램을 해석하여 이들 식을 푸는 것으로 프로그램을 검사한다. CBMC 는 포인터와 동적 할당과 같은 연산을 모두 지원하는 강력한 도구이지만 프로그램 안에 assert 문을 이용한 조건을 넣어 검사하는 방식으로 LTL 이나 CTL 같은 표현 식을 통한 검사가 불가능하다.

3. 기본 개념

이 장에서는 소프트웨어 모델 체킹의 기본이 되는 자료구조인 Control-Flow Graph (CFG)와 Binary Decision Diagram (BDD)에 대하여 설명하고 이를 이용한 심볼릭 모델 체킹 방법에 대해서 간략하게 설명한다.

3.1 Control-flow graph

Control-Flow Graph (CFG)는 Frances E. Allen 에 의해 제안된 것으로 프로그램의 실행 경로를 방향성이 있는 그래프를 이용하여 나타낸 것이다[3]. 어떤 그래프 G 를 CFG 라고 할 때 G={B, E} 로 나타낼 수 있으며 여기서 B 는 그래프를 구성하는 노드의 집합 $\{b_1, b_2, \cdots, b_n\}$ 을 의미한다. 노드의 집합 B 안에는 하나의 시작 노드가 존재하고 여러 개의 끝 노드나 혹은 끝 노드가 존재하지 않을 수도 있다. E 는 방향성이 있는 에지 $(b_i, b_j)=\{(b_i, b_j)\,|\,1\leq i\leq n,\ 1\leq j\leq n\}$의 집합 $\{(b_i, b_j), (b_k, b_l), \cdots\ |\,1\leq i\leq n,\ 1\leq j\leq n,\ 1\leq k\leq n,\ 1\leq l\leq n\}$ 을 의미한다. CFG 에서 중요한 개념은 basic block 으로 하나의 시작 명령어 줄과 하나의 끝 명령어 줄을 가지는 일련의 명령어 줄을 의미한다. basic block 안에서는 점프 명령이 없으며 점프 명령의 목적지도 없다. CFG 에서 basic block 은 각 노드 b_i 와 1:1 로 대응된다.

그림 1의 (a)는 간단한 예제 프로그램이며 그림 1 의 (b)는 (a)의 프로그램을 CFG 의 형태로 나타낸 것으로 basic block b_1 은 시작 노드, b_2 와 b_3 는 끝 노드가 된다. 한 basic block 내에서는 명령어 줄이 순차

적으로 실행되어야 하며 분기가 일어나는 시점은 더 이상 순차적으로 실행 될 수 없는 명령어 줄이 나타났을 때이다. 위 방법을 이용하면 프로그램을 그에 정확하게 대응되는 CFG 형태로 만들 수 있으며 프로그램의 탐색을 수행하게 된다.

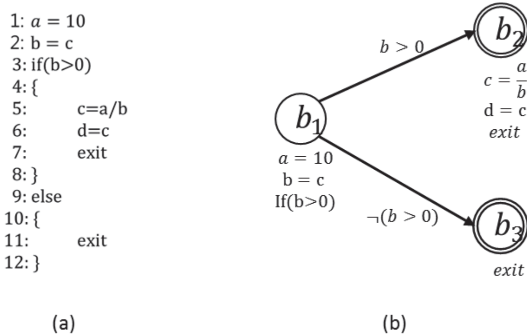

그림 1 예시 프로그램과 CFG

3.2 Binary decision diagram

Binary decision diagram (BDD) 는 Randal E. Bryant 에 의해 창안된 자료구조로서 불 대수 식을 표현하는데 사용된다. 기본적으로 방향성이 있고 사이클이 없는 그래프의 형태로 이진 트리와 유사한 구조를 가지고 있으며 각 내부 노드는는 불 대수 변수를, 트리의 말단 노드들은 0 또는 1 값을 의미한다. BDD 를 이용한 상태 표현은 반드시 불 대수 식을 이용해야 하므로 프로그램의 모든 변수를 불 대수 변수로 변환하여 모든 명령어 줄을 변환된 불 대수 변수에 대한 식으로 변경해야만 위의 방법을 이용해서 상태의 표현과 모델 체킹이 가능하다[1].

그림 2의 (a)는 불 대수 식 $f(a,b,c)=(\neg a \wedge \neg b \wedge \neg c) \vee (a \wedge b) \vee (b \wedge c)$를 BDD 의 형태로 나타낸 것이다. BDD 자료구조의 특징은 루트 노드에서부터 말단 노드까지 이르는 각 경로가 식을 이루고 있는 각 변수의 유일한 조합을 의미한다는 것이다. 모든 경로에서 나타나는 변수의 순서가 같은 BDD 를 Ordered BDD 라 한다.

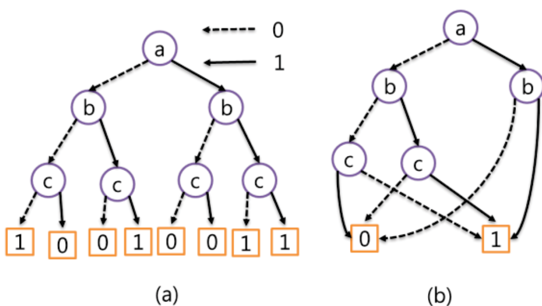

그림 2 BDD 와 ROBDD 의 예시

위와 같이 BDD 형태로 불 대수 식을 표현하려면 n 개의 변수로 이루어진 식의 표현의 위해 2^{n+1}-1 개의 노드가 필요하지만 반드시 모든 노드가 표현에 필요한 것은 아니다. 필요 없는 노드를 제거하는 두 가지 원칙은 첫째, 동형의 서브 그래프는 하나로 합치고, 둘째, 자식인 두 서브 그래프가 동형인 노드는 제거하는 것이다. 두 가지 원칙을 이용해 위 그래프를 바꾸면 그림 2의 (b)와 같은 형태로 표현할 수 있다. 이렇게 제거 원칙을 통해 축소시킨 그래프를 Reduced Ordered BDD (ROBDD)라고 하며 앞으로 BDD 라는 표현은 모두 ROBDD 와 같은 의미로 사용한다.

BDD 가 불 대수 식의 유용한 표현법으로 사용되는 이유는 ROBDD 처럼 필요 없는 노드를 제거한 형태로도 and, or, not, 포함관계, 동치관계 등의 연산을 수행 할 수 있다는 점 때문이다. 프로그램의 상태를 표현하기 위해 각각의 변수 값의 조합을 유지하는 대신 여러 상태를 하나의 BDD 에 저장하고 저장된 상태에 대해 한꺼번에 전이 연산을 수행 할 수 있게 되는 것이다. 하지만 변수를 배치하는 순서에 따라 제거되는 노드의 수가 달라지고 최악의 경우에는 노드 제거가 불가능한 경우도 있어 최적의 성능을 보장 할 수 없다는 단점을 가지고 있다.

3.3 심볼릭 모델 체킹

모델 체킹은 소프트웨어의 모든 가능한 행동을 검사하는 것으로 소프트웨어 모델은 상태와 전이로 구성된다. 상태는 (v_i, l_i)의 쌍으로 표현되며 v_i 는 프로그램의 모든 인코딩 된 변수에 특정 값을 대입한 것, l_i는 프로그램에서의 현재 위치로 CFG 상의 특정 basic block 이 된다. 상태의 전이는 (l_i, l_j, c)로 표현되며 l_i, l_j 는 현재 위치와 전이 후 위치, c 는 전이가 일어날 때의 명령어들로 변수가 변경되거나 변수의 조건이 달라지는 것을 의미한다. 심볼릭 모델 체킹에서는 개별 상태가 아닌 상태의 집합에 대해서 연산을 수행한다. 이에 따라 상태 (v_i, l_i)대신 (s_i, l_i)로 표현되며 s_i 는 변수 값 조합의 집합 $s_i=\{v_a, v_b, \cdots, v_m \mid a,b,m \leq 2n\}$ (n 은 프로그램의 불 대수 변수의 총 개수) 를 의미한다. 현재 상태가 (s_i, l_i)일 때 전이 (l_i, l_j, c)를 탐색하기 위해서는 현재 상태 s_i 를 BDD 를 이용해 하나의 자료구조로 나타내고 전이 c 또한 BDD 를 통해 나타내어 BDD 간의 연산을 통해 전이 후의 s_j를 계산한 후 상태를 (s_j, l_j)로 바꾼다. CFG 에서 분기가 발생하는 경우는 (l_i, l_j, c_j), (l_i, l_k, c_k), (l_i, l_m, c_m), ⋯ 과 같이 l_i 에서 여러 다른 위치로 전이가 있는 형태로 표현 되며 각 전이마다 (s_j, l_j), (s_k, l_k), (s_m, l_m), ⋯ 를 계산하여 각각의 상태에 대해 탐색을 이어간다[4].

명령어 c 는 프로그램의 명령어 줄을 인코딩된 변수의 조합으로 바꾸어 나타낸 것으로 불 대수 식으로 표현 되어 있어 BDD 로 표현 가능하다. 2 비트의

두 변수 a=10 과 b=01 이 있다고 가정할 때 인코딩에 의해 두 변수 a, b 는 네 개의 불 대수 변수 a_0=1, a_1=0, b_0=0, b_1=1 로 바뀐다. 명령어 줄 a=a+b 는 $(a_0' \leftrightarrow (a_0 \text{ xor } b_0)) \wedge (a_1' \leftrightarrow ((a_0 \wedge b_0) \text{ xor } (a_1 \text{ xor } b_1))) \wedge (b_0' \leftrightarrow b_0) \wedge (b_1' \leftrightarrow b_1)$ 와 같이 변경 할 수 있다. 여기서 a_0', a_1'은 a_0, a_1 의 대입 연산후의 다음 상태를 의미하며 b_0', b_1'은 b_0, b_1 의 다음 상태이다. 대입 연산으로 인해 값이 변하지 않는 변수는 이전 상태와 다음 상태가 같다는 것을 연산 식에 포함 시킴으로써 나타낸다. 대입 연산 명령어의 표현을 위해서는 현재의 변수와 대입 이후의 변수에 대한 불 대수 변수가 필요하므로 프로그램의 변수가 n 개라 할 때 대입 연산을 위한 BDD 는 2n 개의 변수를 사용해서 구성하게 된다.

검사하고자 하는 소프트웨어의 성질 또한 불 대수 식으로 정의 되어야 하며 BDD 로 표현 되어야만 연산이 가능하다. 따라서 검사 하고자 하는 성질은 프로그램을 구성하는 변수에 대한 논리 식으로 이루어 지며 프로그램의 모든 상태 중 해당 논리 식을 만족 시키는 상태가 있는지 확인 하는 것으로 실질적인 검사가 이루어진다. 이렇게 개개의 상태에 대해 성질을 만족 시키는지 검사하는 것뿐 아니라 여러 상태에 걸쳐 성질을 만족 시키는지 검사 하는 것 또한 가능한데 이런 성질을 표현하는 식에는 Linear Temporal Logic (LTL)이나 Computational Tree Logic (CTL)등이 있다[14][15].

4. 변수 개수 최소화를 위한 원리

이 장에서는 소프트웨어에서 모델 체킹시에 필요한 변수의 개수를 최소화 하는 원리에 대해서 설명한다. 변수 개수 최소화 이후에도 소프트웨어는 변수 개수 최소화 전과 동일한 입력에 대하여 동일한 출력 결과를 보여야 한다. 현재 본 기법은 포인터 연산, 배열 연산 반복문을 제외하고 함수를 포함하여 C 언어로 작성된 소프트웨어를 대상으로 하며 반복문은 다른 기법을 통한 전처리 과정을 통해 적용 가능하다. 추후 연구를 통해 포인터 등의 연산에 대해 적용이 가능한지 확인 할 것이다.

4.1 변수 유효구간

모델 체킹을 위한 소프트웨어의 변수 유효구간 정의는 다음의 가정을 기반으로 이루어 진다.

- 소프트웨어는 어떤 순간에 정의되어 있는 모든 변수 중에 그 값을 유지하지 않아도 되는 변수들이 존재한다.

소프트웨어에서 특정 시점에 불필요한 변수는 프로그램을 작성하는 프로그래머의 한계에서부터 기인한다. 일반적으로 프로그래머는 가독성이나 이해도를 높이기 위해 여러 변수를 이해하기 쉬운 이름으로 생성하여 사용하곤 한다. 이런 프로그래밍 방식은 사람 입장에서는 이해도를 높이고 오류를 줄일 수 있는 방법이지만 컴퓨터 입장에서는 불필요한 메모리 공간을 낭비하는 행위이다. 그래서 컴파일러는 작성된 프로그램에 대해 최적화를 하면서 변수 저장에 필요한 공간을 최소화 한다. 이런 최적화 기법은 특정 시점에 선언되어 있는 많은 변수들 중에 일부 변수는 아직 사용되는 시점이 아니거나 혹은 이미 다 사용이 되어 다음 정의되는 시점까지 의미가 없다는 점을 토대로 이루어진다.

```
1 int main(){
2   int a,b,c, match=0;
3   int result=-1;
4   if(a==b)
5     match=match+1;
6   if(a==c)
7     match=match+2;
8   if(b==c)
9     match=match+3;
10  if(match==0){
11    if(a+b<=c)
12      result=2;
13    else if(b+c<=a)
14      result=2;
15    else if(a+c<=b)
16      result=2;
17    else
18      result=3;
19  }else{
20    if(match==1){
21      if(a+b<=c)
22        result=2;
23      else
24        result=1;
25  }else{
26    if(match==2){
27      if(a+c<=b)
28        result=2;
29      else
30        result=1;
31    }else{
32      if(match==3){
33        if(b+c<=a)
34          result=2;
35        else
36          result=1;
37      }else
38        result=0;
39  }}}
40  printf("result=%d₩n",result);
41 }
```

그림 3 예제 프로그램

위에서 설명한 불필요한 변수의 조건을 프로그램의 입장에서 정의 하면 변수의 유효구간은 값을 대입한 위치부터 다음 값 대입 이전에 마지막으로 사용된 위치까지 이고 비 유효구간은 나머지 모든 구간이다. 여기서 사용이란 대입 연산에서 등호의 오른쪽에 위치하거나 분기문의 조건으로 사용되는 경우를 의미한다. 프로그램에서 변수에 값이 대입되기 전에는 변수가 아직 정의되지 않은 것과 마찬가지 이므로 유효하지 않은 것이 당연하다. 다음 대입이 일어나기 전 마지막 사용부터 대입 까지는 마지막 사용 시점부터 대입까지 사이에는 다른 사용이 없고 대입 연산이 일어나면서 이전에 저장된 변수의 값이 사라지게 되므로 마지막 사용 시점 이후 값은 역시 유효하지 않다.

그림 3은 세 정수 변수를 입력으로 받아 세 변수 값이 삼각형의 각 변을 이룰 수 있는지 그리고 어떤 형태의 삼각형이 되는지 확인하는 프로그램으로 이 프로그램을 통해 변수 값의 유효구간에 대해 확인해 보도록 한다[18]. 위 프로그램의 대입 시점과 사용 시점을 명확히 분석하기 위해 그림 4의 CFG 그래프

로 나타내었다. 그림에서 각 사각형은 CFG 그래프의 노드를 나타내며 노드 내부에서 명령어는 순차적으로 실행 된다. 명령어 오른쪽에 나열 된 변수는 해당 명령어 줄에서 유효한 변수를 나타낸 것이다. 여기서 유효한 변수는 앞서 정의한 대로 식별하여 표기 하였다. 그래프를 살펴보면 각 명령어 줄마다 유효한 변수에 차이가 있는 것을 확인 할 수 있다. 앞서 확인한 바와 같이 해당 명령어 줄에서 유효하지 않은 변수는 그 값을 유지하지 않아도 프로그램의 동작에 전혀 문제가 없다. 예를 들어 7 번 노드 에서 8 번 노드로 이동한 후에는 더 이상 match 변수의 값은 저장하고 있을 필요가 없으며 8 번 노드에서 10 번 노드로 이동한 후에는 a, b, c 세 변수의 값은 저장하고 있을 필요가 없다. result 변수는 1 번 노드에서 값이 대입되어 유효한 것처럼 보이지만 그 이후로 사용이 없이 노드 10 번, 14 번, 16 번, 20 번 등 모든 경로에서 새롭게 정의되고 있기 때문에 유효하지 않다. 이렇게 각각의 명령어 줄에서 유효하지 않은 변수를 정의한 후 이를 이용하여 변수 개수를 최소화 할 수 있다.

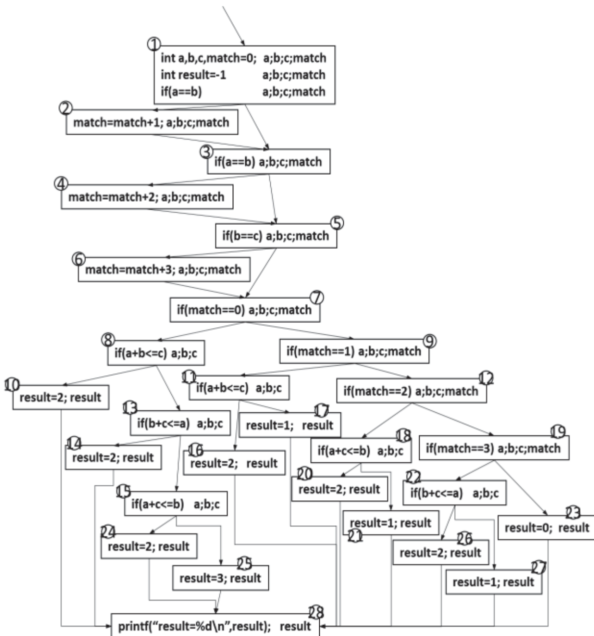

그림 4 예제 프로그램의 CFG

4.2 최소 필요 변수개수

4.1 에서 정의한 변수 유효구간을 토대로 해서 프로그램 전체의 최소 필요 변수개수를 구할 수 있다. 프로그램에 필요한 최소 변수개수는 가장 많은 유효 변수 개수를 가진 명령어 줄의 유효변수 개수로 결정된다. 즉, 가장 많은 유효변수 개수를 가진 명령어 줄의 유효변수 개수만큼 변수를 가지고 있으면 그 변수들 만으로도 전체 프로그램을 동작시키기 위한 변수 값 저장에는 문제가 없다는 것이다. 그림 4에서 명령어 줄 중 가장 많은 유효변수 개수를 가진 것은 4 개의 유효변수를 가진 명령어 줄들이다. 따라서 예제 프로그램은 실제로는 다섯 개의 변수를 포함하고 있지만 4 개의 변수만 사용하도록 바꾸어 표현해도 5 개의 변수를 사용 할 때와 동일한 결과를 얻을 수 있다. 이를 다르게 표현하면 변수의 유효구간이 전혀 겹치지 않는 변수들은 하나의 변수에 차례대로 저장하여 사용 할 수 있다는 것이다.

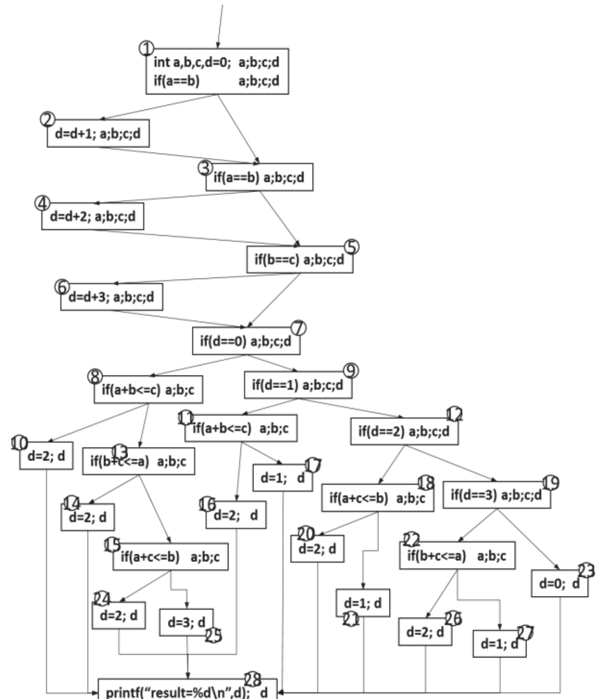

그림 5 예제 프로그램의 변수 개수 최소화 CFG

예제 프로그램에서도 전체 변수가 다섯 개이지만 최소 필요 변수는 4 개이므로 특정 변수들은 하나의 변수에 순차적으로 사용이 가능할 것이다. 그래프의 시작 노드 1 번에서부터 종료 노드 28 번에 이르는 모든 경로를 살펴보면 match 변수와 result 변수는 어떤 명령어 줄에서도 동시에 유효하지 않은 것을 확인할 수 있다. 그러므로 어떤 변수를 이용하여 match 변수가 유효한 구간에서는 match 변수의 값을 저장하고, match 변수가 더 이상 유효하지 않고 result 변수가 유효할 때는 같은 변수에 result 변수의 값을 저장할 수 있다. 이와 같은 방법을 토대로 예제 프로그램을 재구성하면 그림 5의 CFG 로 나타낼 수 있다.

이전 프로그램에서 result 변수와 match 변수는 새로운 프로그램에서 d 변수를 사용하여 하나의 변수에 표현 되었고 1 번 노드의 result=-1 명령어 줄은 유효하지 않은 값의 대입이기 때문에 삭제 되었다. 수정된 프로그램은 CFG 상의 구조 변화가 전혀 없고 서로 동시에 유효하지 않은 변수에 대해서만 같은 이름을 사용하였기 때문에 이전 프로그램과 동일하

게 동작 할 수 밖에 없다.

4.2 결함 상태 관련 변수

모델 체킹은 검사를 위한 대상이 되는 모델과 대상 모델에서 검사하고자 하는 성질을 입력으로 받아 대상 모델이 검사하고자 하는 성질을 만족하는지 확인하여 만족의 유무를 알려주는 기법이다. 여기서 만족이란 모델의 특정 상태에서 성질에 대한 식이 참이 되느냐 하는 것으로 모델의 상태에서 성질을 표현한 변수들의 값을 모두 알고 있어야만 해당 상태에 대한 만족 여부를 판단할 수 있다. 모델 체킹에서는 결함 상태를 식으로 표현하여 결함 성질을 만족하는 상태가 있으면 소프트웨어에 결함이 있는 것으로 판단하므로 결함 상태 관련 변수란 검사하고자 하는 성질을 표현한 식을 구성하고 있는 변수들로 정의될 수 있다. 실제 프로그램에서는 assert 문에 포함된 조건을 구성하는 변수들이나 검사하고자 하는 성질의 LTL, CTL 식을 이루는 변수들을 의미한다.

결함 상태 관련 변수가 중요한 이유는 결함의 측면에서 보았을 때 결함과 관련된 변수의 변경을 일으키는 명령어 줄 만이 프로그램이 결함 상태로 빠지는 전이를 일어나게 할 수 있기 때문이다. 소프트웨어를 모델로 변화 시켰을 때 결함과 관련된 변수가 제외되어 있다면 모델은 결함 상태를 표현할 수 없고 검사 또한 불가능하다. 앞서 설명한 변수개수 최소화의 원리를 그대로 적용할 수 없는 이유가 여기에 있다. 반대로 결함과 관련되지 않은 변수에 값을 대입하는 명령은 결함 상태로 가는 과정일 뿐이므로 이러한 상태에 대해서는 굳이 명시적인 확인이 필요하지 않다.

결국 결함 상태 관련 변수의 변화로만 프로그램의 상태 공간을 구성하더라도 검사하고자 하는 성질과 관련된 상태들은 모두 존재하므로 프로그램이 성질을 만족하는지 검사가 가능하다는 것이다. 이런 방법은 전체 프로그램의 변수 중 일부만을 사용하여 검사를 진행하게 되므로 상태를 표현하기 위한 공간의 크기를 작아지게 하는 효과가 있다.

하지만 결함 상태 관련 변수만을 사용한 상태 공간의 구성은 BDD 를 이용한 연산을 불가능하게 한다. 현재 상태를 결함 상태 관련 변수만을 사용해 표현하게 되면 상태의 전이인 명령어를 BDD 로 표현했을 때 현재 상태에 저장되어 있지 않은 변수들이 포함되어 있기 때문에 다음 상태를 계산할 수가 없다. BDD 를 사용하지 않으면 모든 상태를 별도로 저장해야 하기 때문에 결국 심볼릭 모델 체킹의 장점을 이용할 수 없다. 따라서 결함 상태와 관련된 변수들은 모두 모델에 포함시키면서도 BDD 를 구성하는 변수의 개수를 줄일 수 있어야만 심볼릭 모델 체킹의 장점을 그대로 이용하면서 모델 체킹의 성능을 높일 수 있다.

5 프로그램 수정 기법

변수개수 최소화 기법은 프로그램의 동작에는 영향을 미치지 않으면서 전체 프로그램에 사용되는 변수의 개수를 줄이는 방법이다. 그러나 이 과정에서 변수의 이름과 저장 공간이 달라지기 때문에 상태의 관점에서는 이전 프로그램과 다른 상태 공간과 전이를 보이게 된다. 이번 장에서는 결함 관련 상태를 고려한 변수개수 최소화 원리를 통해 프로그램 수정 기법을 제안하고 이 기법으로 수정된 프로그램이 수정되기 전 프로그램과 같은 입력에 대해 동일한 출력을 보이며 모델 체킹 결과에서도 수정 전과 후에 동일한 결과를 보인다는 것을 확인한다.

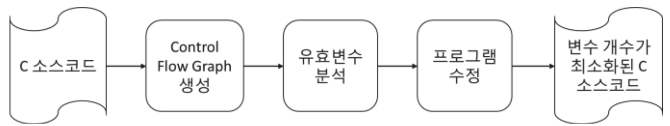

그림 6 프로그램 수정 기법 진행 단계

5.1 유효변수 설정

앞에서 설명한 바와 같이 결함 상태 관련 변수의 수정은 프로그램의 모델 체킹 결과에 영향을 끼치므로 수정 후에도 결함 상태 관련 변수들은 모델에 모두 존재해야 하며 나머지 변수들 중 변수개수 최소화 기법을 통해 유효 구간이 겹치지 않는 변수들에 대하여 기법을 적용한다. 먼저 CFG 를 통해 프로그램의 각 명령어 줄에서 유효한 변수들을 찾아야 한다. 모든 CFG 노드의 명령어 줄은 기본적으로 결함 상태 관련 변수들을 반드시 유효변수에 포함 하도록 한다. 각 명령어 줄에 대한 연산을 쉽게 처리하기 위해 여기서 사용하는 CFG 는 각 노드마다 단 하나의 명령어 줄 만을 포함하는 것으로 가정한다.

그림 7은 CFG 를 이용하여 각 명령어 줄의 유효변수를 설정하는 알고리즘으로 기본적으로 Depth First Search (DFS)알고리즘을 이용한 그래프 탐색을 기반으로 구현할 수 있다. 이 알고리즘은 CFG G 의 시작 노드를 입력으로 받아 모든 연산이 끝난 후에 CFG 의 각 명령어 줄에 유효변수 목록이 추가된 CFG G′이 완성되는 알고리즘이다. 현재 노드의 유효변수 설정은 그 노드 이후에 나오는 노드들을 탐색한 후에 결정 할 수 있기 때문에 DFS 알고리즘을 사용한다. SetDefUse(n) 함수는 CFG 의 노드를 입력으로 받아 그 노드의 명령어 줄의 대입되는 변수와 사용되는 변수를 식별한다. 대입과 사용에 대한 식별 후에는 DFS 알고리즘의 탐색 순서에 따라 다음 노드에 대해서 재귀적으로 함수를 호출하여 다음 노드들에 대한 대입과 사용 식별을 수행한다.

실제로 명령어 줄에 대한 유효변수 설정이 일어나는 것은 다음 노드들에 대한 탐색을 모두 마치고

다시 해당 노드에 돌아온 다음으로 SetValid(n,e.dst) 함수를 호출하여 유효변수를 설정한다. SetValid 함수에서는 결함 관련 변수들은 모두 유효 변수로 설정하고 나머지 변수들에 대하여 유효변수를 식별한다. 현재 노드 n 과 다음 노드 e.dst 를 비교하여 다음 노드에서 유효하면서 현재 노드에서 대입되지 않는 변수들을 모두 유효변수로 설정하고 다음 노드에서 유효하지 않더라도 현재 노드에서 사용된 변수이면 유효변수로 설정한다. 다음 노드에서 유효하면서 현재 노드에서 대입이 일어나는 변수는 유효하며 다음 노드에서 유효하지 않고 현재 노드에서 대입이 일어나는 변수는 유효하지 않다. 대입이 일어나는 변수 중 해당 명령어 줄에서 유효하지 않은 경우는 의미 없는 대입연산으로 다음 프로그램 수정 단계에서 삭제할 수 있다. 다음 노드에서 대입이 일어난 변수의 경우에는 현재 노드에서 사용이 없다면 유효하지 않다. 현재 노드에서 갈 수 있는 다음 노드들은 여러 개 있을 수 있기 때문에 SetValid 함수의 호출은 여러 번 일어날 수 있다. 이때 모든 다음 노드들 중 하나의 노드라도 유효한 변수가 있다면 현재 노드에서도 그 변수는 유효해야 하므로 한번 유효한 것으로 확인된 변수에 대해서는 더 이상 확인할 필요가 없다. 이미 방문한적이 있는 노드에 대해서는 이전 방문에서 필요한 필요한 연산이 모두 수행 된 다음이므로 바로 반환한다.

```
function FindValidRange(n)
Input : start node of CFG G n
Output : valid range CFG G'

1  if n was visited
2    skip
3  else
4    n.visited<-true
5    SetDefUse(n)
6    if n has no edge
7      SetEndNodeValid(n)
8    else
9      e<-for each edge in n
10       FindValidRange(e.dst)
11       SetValid(n,e.dst)
```

그림 7 유효변수 설정 알고리즘

SetEndNodeValid 함수는 방문한 노드가 프로그램의 끝을 나타내는 노드일 때 호출되는 것으로 SetValid 함수에서 다음 노드에 유효 변수가 하나도 없는 것과 같은 동작을 취하게 된다. 따라서 현재 노드의 명령어 줄에서 사용된 변수를 모두 유효변수에 추가하고 반환한다. 이렇게 유효변수가 추가된 CFG G'은 모든 노드에서 결함관련 변수를 유효변수로 가지면서 나머지 변수에 대하여 유효변수를 식별하고 저장한 형태가 된다.

5.2 유효변수를 이용한 프로그램 수정

5.1 절에서 설명한 알고리즘을 통해 완성된 G'을 통해 원래 프로그램의 결함 상태를 모두 포함하면서 본래의 변수 개수보다 적은 수의 변수만으로 원래 프로그램과 같은 동작을 보이도록 프로그램을 수정한다.

```
function VariableReduction(P,n,M,F,E)
Input : original program P, start node of valid range CFG G' n,
variable mapping M, temporary variable mapping set F,
temporary branch mapping stack E
Output : modified program P'

1  n.visited<-true
2  if n has definition
3    a<-getDefVar(n)
4    if a is valid in n
5      M.setMapping(a)
6      modifyVariables(P,n,M)
7    else
8      removeDef(P,n)
9  else
10   modifyVariables(P,n,M)
11 if n have more than two in edge
12   F. insert(n.index,M)
13 if n has more than two out edge
14   E.push(M)
15   e<-for all out edges in n
16     if e.dst.visited is true
17       c<-F.get(e.dst)
18       addVariableChangeLine(c,M)
19     else
20       M<-E.pop()
21       E.push(M)
22       VariableReduction(P,e.dst,M,F,E)
23   E.pop()
24 else
25   e<-getEdge(n)
26   if e.dst.visited is true
27     c<-F.get(e.dst)
28     addVariableChangeLine(c,M)
29   else
30     VariableReduction(P,e.dst,M,F,E)
```

그림 8 유효변수를 이용한 프로그램 수정 알고리즘

그림 8은 각 명령어 줄에 대한 유효변수가 표시된 CFG G'과 본래 프로그램 P 를 이용하여 결함상태 변수를 보존하면서 변수개수 최소화 기법을 적용한 프로그램 수정 알고리즘을 나타낸 것이다. 전체적으로는 DFS 알고리즘을 이용한 그래프 탐색을 기반으로 프로그램 수정을 진행한다. 입력으로 들어가는 M 은 본래 프로그램의 변수와 수정 후 바뀐 변수와의 사상관계를 표시하는 것으로 비어있는 상태로 시작한다. F 는 특정 노드의 M 을 저장하기 위한 것으로 두 개 이상의 들어오는 에지를 가진 노드의 변수수

정을 돕기 위한 공간이다. E 는 두 개 이상의 나가는 에지를 가진 노드의 M 을 저장하기 위한 것으로 DFS 를 이용한 탐색에서는 스택을 이용해 저장하여 사용할 수 있다.

전체적인 과정은 CFG G′의 노드를 방문하여 해당 노드에 변수에 대한 대입 연산이 있을 때 그 변수를 변수 사상 M 의 유효하지 않은 변수 자리에 추가하고 M 에 사상된 대로 프로그램의 변수를 수정한 후 다음 노드에 M 을 전달하는 과정을 반복하여 프로그램의 변수이름을 변경하게 된다. 이렇게 수정된 프로그램은 CFG G′에 있는 모든 노드중 가장 많은 유효변수 개수를 가진 노드의 변수개수만큼의 변수로 모두 표현이 되게 된다. 프로그램에서 사상 M 에 변수가 추가되는 경우는 명령어 줄에 대입 연산이 있을 때 뿐이고 어떤 대입 연산이 있는 노드에서 유효변수 개수가 최다가 아니라면 사상 M 에는 반드시 유효하지 않은 자리가 있게 되므로 최대 유효변수 개수 이상의 사상이 일어나지 않고, 대입 연산이 있는 노드가 최다 유효변수 개수를 가진 노드일 때는 대입되는 변수가 반드시 사상 M 에 포함되어 있는 상태이거나 사상에 유효하지 않은 자리가 하나 있는 상태이기 때문에 역시 최다 유효변수 개수 이상의 사상은 일어나지 않는다.

5.2.1 변수 사상 및 프로그램 수정

현재 노드에 대입 연산이 있을 때 해당 노드에서 대입되는 변수가 유효하다면 setMapping 함수를 통해 유효한 변수를 사상에 추가한다. setMapping 함수에서는 사상된 목록 중 가장 먼저 나타나는 유효하지 않은 자리에 새로운 변수를 추가하게 된다. modifyVariables 함수에서는 사상 목록과 프로그램을 이용하여 사상 목록에 나와있는 변수 사상대로 본래 프로그램의 명령어 줄을 변경한다. 대입되는 변수가 유효하지 않을 때는 의미 없는 대입연산이므로 삭제하며 대입연산이 없는 명령어 줄일 때는 이전 노드에서 받은 사상대로 변수를 변경한다. DFS 알고리즘은 더 이상 나가는 에지가 없는 노드에 도달하기까지 깊이에 우선하여 탐색하기 때문에 시작 노드부터 도달한 끝 노드까지 일관된 변수 사상이 적용된다.

5.2.2 분기가 있는 노드 처리

if 문과 같은 분기가 있는 노드에서는 한 분기에 대한 탐색을 마친 후 다음 분기에 대해 탐색할 때 해당 노드의 사상정보를 알고 있어야만 일관된 변수 변경이 가능하다. 알고리즘에서는 공간 절약을 위해 모든 노드의 사상 M 을 저장하고 있지 않기 때문에 스택을 이용하여 분기가 있는 노드의 사상 M 을 임시로 저장한다. DFS 알고리즘에서 어떤 노드의 탐색은 그 노드의 모든 나가는 에지를 탐색하기 전에는 끝나지 않으므로 스택을 이용하면 연산속도를 높이고 사상 M 을 저장하는 공간을 최소화 할 수 있다. 스택 E 는 이런 분기를 가진 노드의 사상을 저장하기 위한 용도이며 분기에 대한 탐색을 시작할 때 스택에 저장된 사상 M 을 가져와 갱신하고 다시 스택에 저장하여 모든 분기에 대해 동일한 M 을 사용한 사상이 가능하도록 한다.

5.2.3 들어오는 에지가 두 개 이상인 노드 처리

들어오는 에지가 두 개 이상인 노드는 갈라져 진행된 분기가 하나로 만나는 노드를 의미한다. 이런 노드가 문제가 되는 이유는 서로 다른 분기를 통해 서로 다른 사상을 진행한 두 실행 경로가 한 곳에서 만나면서 일관되지 못한 사상이 충돌하게 되기 때문이다. 이를 해결하기 위해 서로 다른 사상을 일치 시키기 위하여 프로그램 동작과 무관한 대입 명령어 줄을 프로그램에 추가한다. 이를 위해 그래프 탐색 중에 두 개 이상의 들어오는 에지를 지닌 노드는 집합 F 를 이용하여 노드 번호와 그 노드의 사상을 저장한다. 두 개 이상의 들어오는 에지를 지닌 노드는 그래프 탐색을 통해 쉽게 확인 할 수 있으며 이전 유효변수 설정 단계에서 매우 작은 비용으로 계산이 가능하다.

A

New var	w1	w2	w3	W4
Original var	a	b	c	d

B

New var	w1	w2	w3	W4
Original var	a	b	d	c

그림 9 변수 사상 충돌의 예

그림 9의 A 와 B 는 두 분기를 통해 진행된 변수 사상이 각각 다를 때를 의미한다. A 분기가 먼저 진행되어 이미 끝난 상태라고 가정할 때 두 분기 A, B 가 만나는 노드에는 이미 A 사상을 통해 변수의 변경이 일어나 있는 상태이다. 이때 B 분기가 진행되어 A, B 가 만나기 바로 전 노드에 도달하면 알고리즘의 16, 26 번째 줄에서 다음 노드가 방문되었던 노드라는 것을 확인하게 된다. 이때 방문했던 노드의 사상을 F 로부터 가져와 addVariableChangeLine 함수를 호출하는데 이 함수에서는 사상을 일치시키기 위한 명령어 줄을 추가한다. 명령어 줄은 아주 간단한 구조로 A 분기에 의해 이미 사상이 결정 되었으므로 B 분기에서는 A 분기에 맞게 변수 값을 서로 뒤바꿔 주는 역할을 한다. 즉, A 분기에서는 w3 에 c 가, w4 에 d 가 사상되어 있으므로 B 분기의 명령어 줄 다음에 전이가 일어나기 전에 temp=w3; w3=w4; w4=temp; 로 값을 서로 바꿔주기만 하면 합쳐진 노

드에서 아무런 문제 없이 동작이 가능하다. 두 분기의 유효 변수설정은 합쳐지는 노드의 유효변수로부터 아래에서 위로 진행되기 때문에 두 분기의 유효변수는 사상은 다르지만 같은 변수와 개수를 가지고 있을 수 밖에 없어 이 방법은 충분히 유효하다.

5.3 반복문을 위한 전처리 과정

본 기법은 연산의 속도와 간결성을 위하여 반복문을 지원하지 않고 있다. 특히 유효변수 설정은 프로그램이 동작하는 개개의 실행 경로에 의해 결정되기 때문에 반복문이 포함 될 경우 매우 복잡한 과정을 거쳐야 한다. 이런 복잡함을 피하기 위해 반복문은 전처리 과정을 통해 지원한다.

반복문 처리를 위해 사용하는 방법은 반복문을 풀어 일반 분기문으로 모두 바꾸는 unwinding loop 기법을 이용할 수 있다. 이는 CBMC 와 같은 bounded 모델 체킹 도구에서 사용되는 것으로 모든 반복문마다 반복의 횟수를 제한하여 반복문 대신 제한된 횟수의 분기문으로 프로그램을 수정하는 것이다[8]. 프로그램 전체의 반복문을 모두 제한된 횟수의 분기문으로 표현하게 되면 프로그램을 표현한 CFG 에는 순환이 발생하지 않는다. 따라서 순환이 없는 CFG 를 이용하여 본 기법을 적용할 수 있다.

6. 변수개수 최소화의 효과 확인

4 장에서 제시한 그림 3의 예제 프로그램을 통해 변수개수 최소화의 효과를 확인한다. 실험에 사용된 모델 체킹 도구는 NuSMV 2.5.4 버전으로 NuSMV 모델링 언어를 이용하여 작성된 유한 상태 기계 모델을 BDD 를 이용하여 심볼릭 모델 체킹 한다. NuSMV 는 소스코드를 직접 분석할 수 없기 때문에 예제 프로그램을 NuSMV 언어의 형태로 변형한 후 실험을 진행하였다[16]. 소스코드를 NuSMV 언어로 변형하기 위해 기존 프로그램의 변수 외에 소스코드 라인 번호를 의미하는 loc 변수와 프로그램 실행 후의 결과값을 의미하는 tt 변수를 추가하였다. 모든 변수는 8 비트의 부호가 있는 정수로 정의하였으며 변수 값의 조합으로 상태를 표현한다.

변수 개수 최소화의 효과를 확인해야 하므로 예제 프로그램과 예제프로그램에 변수 개수 최소화 기법을 적용한 두 모델을 비교한다. 두 모델은 각각 그림 4와 그림 5에 제시된 CFG 와 동일한 흐름을 가진다. 검증하려는 속성은 LTL 식을 이용하여 G (tt != 10)으로 정의하였으며 이에 따라 결함 상태는 tt 가 10 인 상태가 된다. 여기서 tt 는 결과값을 의미하는 변수이며 LTL 식은 “모든 상태에서 tt 는 10 이 아니다”로 해석된다.

두 모델이 변수 개수가 다르면서 동일한 기능을 보인다는 것을 확인하기 위해 모델 체킹을 이용하였다. 두 모델에 입력으로 들어가는 a, b, c 변수를 동일하게 설정한 뒤 검증 속성으로 G (proc1.loc=36 & proc2.loc=36) -> (proc1.tt=proc2.tt)를 이용하여 결과값이 항상 같다는 것을 검증하였다. 여기서 proc1 과 proc2 는 각 모델을 의미하며 loc=36 은 프로그램이 종료된 상황을, proc1.tt=proc2.tt 는 두 모델의 결과값이 일치함을 의미한다. 검증 결과 검증 속성이 만족되어 두 모델이 동일한 기능을 가지는 것으로 확인되었다.

	Model 1	Model 2
Memory in use(Bytes)	5217516	5146652
Peak number of nodes	23506	20440
Number of BDD variables	113	97

A

	Model 1	Model 2
Number of reachable states	1.66094e+09	1.42606e+09
Number of total states	2^56	2^48

B

그림 10 A:모델 체킹에 사용된 BDD 의 크기, B:모델의 전체 상태의 수와 도달 가능한 상태의 수

그림 10는 두 모델에 대하여 모델 체킹을 수행한 결과로서 Model 1 은 원래의 예제 프로그램, Model 2 는 변수 개수 최소화 과정을 거친 예제 프로그램을 의미한다. A 는 모델 체킹 과정에서 사용된 BDD 의 크기를 나타내는 것이다. NuSMV 에서는 CUDD 라는 BDD 라이브러리를 사용하고 있기 때문에 이 라이브러리를 통해 결과를 확인하였다[17]. 비교한 항목은 사용된 메모리, 노드의 최대 개수, BDD 를 구성하는 변수의 개수로 세 가지 측면 모두에서 변수 개수 최소화 기법을 적용한 Model 2 가 더 나은 성능을 보이는 것을 확인할 수 있다. B 는 모델 체킹의 결과 접근 가능한 상태의 수와 모든 상태의 수를 나타낸 것이다. 모든 상태의 수는 모델의 변수 개수와 연관된 것으로 Model 1 에서는 56 개의 불 대수 변수를 포함하며 Model 2 에서는 48 개의 불 대수 변수를 포함한다는 것을 의미한다. 접근 가능한 상태의 수와 전체 모델의 상태 수 모두에서 변수 개수 최소화 기법을 적용한 Model 2 가 적은 수의 상태를 가지는 것을 확인할 수 있다.

이 결과를 통해 동일한 기능을 가지는 두 프로그램이 다른 변수의 개수를 지니고 있다면 변수 개수를 감소시킨 경우에 BDD 를 이용한 심볼릭 모델 체킹에서 더 나은 모델 체킹 성능을 보인다는 것을 확인 할 수 있다.

7. 결론 및 향후 연구계획

모델 체킹은 소프트웨어를 검증하기 위한 강력한 도구이지만 소프트웨어와 모델 체킹 자체가 가진 복잡성 때문에 사용하는데 많은 어려움이 있다. 이를 해결하기 위한 방안인 심볼릭 모델 체킹은 매우 효과적인 해결책 이지만 변수의 순서에 큰 영향을 받는 등 성능을 보장할 수 없는 문제를 가지고 있다. 우리는 이를 보완하기 위한 방법으로 검사 시점 이전에 모델을 만드는 과정에서 불필요한 변수를 식별하고 프로그램 동작에 필요한 최소 변수개수를 찾아 전체 프로그램의 변수 개수를 줄이는 방법을 제시하였다. 이 방법을 통해 전처리된 소프트웨어는 본래의 소프트웨어와 동일한 구조와 실행 경로를 가지고 있다. 또한 정의된 프로그램의 결함 상태를 모두 포함하고 있으며 더 적은 개수의 변수를 포함하고 있다. 따라서 심볼릭 모델 체킹 결과에 전혀 영향을 미치지 않으면서도 심볼릭 모델 체킹의 필수 요소인 BDD 의 크기를 줄일 수 있어 모델 체킹의 성능을 높일 수 있다.

본 방법에서 사용된 변수개수 최소화, 프로그램 수정 기법 등은 간단하고 직관적으로 구성되어 있어 쉽게 증명될 수 있으나 향후 좀더 정확하게 수학적인 증명을 통하여 이해도를 높일 필요가 있으며 실제 구현과 실험을 통하여 성능을 확인할 필요가 있다. 특히 변수개수 최소화는 실제 프로그램에서 얼마나 많은 변수개수가 줄어들 수 있는가에 따라 그 성능의 차이가 있을 수 있기 때문에 다양한 실제 소프트웨어에 대하여 기법을 적용하고 그 효과를 확인할 필요가 있다. 이를 위해 현재 구현이 진행 중에 있으며 실제 소프트웨어에 적용하기 위해 포인터 연산과 같은 복잡한 연산에 대한 처리 기법을 추가하는 연구를 진행 중에 있다.

참고문헌

[1] Bryant, Randal E. "Graph-based algorithms for boolean function manipulation." *Computers, IEEE Transactions on* 100.8 : 677-691. 1986

[2] Burch, Jerry R., et al. "Symbolic model checking for sequential circuit verification." *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 13.4: 401-424. 1994

[3] Allen, Frances E. "Control flow analysis." *ACM Sigplan Notices*. Vol. 5. No. 7. ACM, 1970.

[4] Jhala, Ranjit, and Rupak Majumdar. "Software model checking." *ACM Computing Surveys (CSUR)* 41.4: 21. 2009

[5] Berezin, Sergey, Sérgio Vale Aguiar Campos, and Edmund M. Clarke. "Compositional Reasoning in Model Checking." *Revised Lectures from the International Symposium on Compositionality: The Significant Difference*. Springer-Verlag, 1997

[6] Burch, Jerry, Edmund M. Clarke, and David Long. "Symbolic model checking with partitioned transition relations." *Computer Science Department* 435. 1991

[7] Touati, Herve J., et al. "Implicit state enumeration of finite state machines using BDD's." *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*. IEEE, 1990

[8] http://www.cs.cmu.edu/~modelcheck/cbmc/

[9] D'Silva, Vijay, Daniel Kroening, and Georg Weissenbacher. "A survey of automated techniques for formal software verification." *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 27.7: 1165-1178. 2008

[10] Young Joo Kim, Okjoo Choi, Moonzoo Kim, Jongmoon Baik, Tai-Hyo Kim, "Validating Software Reliability Early through Statistical Model Checking." *IEEE Software 30*(3): 35-41. 2013

[11] Grumberg, Orna, and David E. Long. "Model checking and modular verification." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.3: 843-871. 1994

[12] Coudert, Olivier, Christian Berthet, and Jean Christophe Madre. "Verification of synchronous sequential machines based on symbolic execution." *Automatic verification methods for finite state systems*. Springer Berlin Heidelberg, 1990

[13] Bérard, Béatrice, et al. "Systems and software verification: model-checking techniques and tools." Springer Publishing Company, Incorporated, 2010

[14] Bauer, Andreas, Martin Leucker, and Christian Schallhart. "Runtime verification for LTL and TLTL." *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20.4 : 14. 2011

[15] Clarke, Edmund M., E. Allen Emerson, and A. Prasad Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2: 244-263. 1986

[16] Cimatti, Alessandro, et al. "NuSMV 2: An OpenSource Tool for Symbolic Model Checking." *Proceedings of the 14th International Conference on Computer Aided Verification*. Springer-Verlag, 2002

[17] Somenzi, Fabio. "CUDD: CU decision diagram package release 2.3. 0." *University of Colorado at Boulder,* 1998

[18] Jorgensen, Paul. *Software testing: a craftsman's approach*. CRC press, 2002.