논리 프로그램의 병렬 수행에서 재초기화 알고리즘의 정확성 증명을 위한 참조 집합: 직관적 접근

(A Reference Set for the Correctness Proof of Resetting Algorithms in Parallel Execution of Logic Programs: An Intuitive Approach)

김 도 형 † 최 광 무 ††

(Do-Hyung Kim) (Kwang-Moo Choe)

요 약 논리 프로그램(logic program)의 병렬 수행 시, 중요한 작업 중의 하나인 재초기화(resetting)를 수행하기 위한 알고리즘들의 정확성(correctness) 증명을 위한 기준을 제시한다. 많은 연구자들이 논리 프로그램의 병렬 수행을 위한 수행 모형을 제안한 바 있고, 그것과 관련하여 재초기화를 효율적으로 처리하기 위한 알고리즘들을 제시하였다. 그러나 그 알고리즘들이 올바르게 작동하는 지에 대해서는 분명한 언급이 없는 경우가 많았다. 본 논문에서는 정확성에 대한 판단을 내릴 수 있는 기준을 도출한다. 이 도출 과정은 직관적으로 보아 명백한 사실에 대한 관찰로부터 유도되는 식으로 이루어진다. 도출된 정확성 판단 기준에 근거하여 기존의 재초기화 방법들을 점검하고, 보다 효율적이라고 판단되는 재초기화 방법도 서술한다.

Abstract A criterion for the correctness proof of resetting algorithms in parallel execution of logic programs is suggested in the paper. Many researchers have proposed various execution models to exploit parallelism in logic programs, thus various algorithms to handle resetting efficiently. It is, however, hardly mentioned whether those algorithms are "really" correct in every case. In this paper, we devise a yardstick which may serve for probing the correctness of resetting algorithms. This criterion is derived from observations that are intuitively clear. According to the derived criteria, we analytically review existing schemes for resetting, and describe another new resetting method which seems to be more efficient than others.

1. 서 론

현재까지 제안된 새로운 프로그래밍 패러다임(paradigm) 가운데에서도, *논리 프로그래밍(logic programming)*은 특히 그 프로그램에 자연스럽게 내재하고 있는 높은 병렬성 때문에 많은 연구자들의 주목을 받았다. 주지하다시피 논리 프로그램 내의 병렬성은 여러 종류(예컨대 AND, OR, 흐름(stream), 탐색(search) 병렬성 등)가 있으나[3], 특히 AND와 OR 병렬성이 가장 두드러지고 명백한 것이어서 집중적인 연구의 대상이 되었다.

우리는 하나의 논리 프로그램에 대해서, 그 프로그램을 수행하는데 필요한 탐색 공간(search space)을 나타낸다고 볼 수 있는 AND/OR 증명 트리(proof tree)를 쉽게 구성할 수 있다[9]. 달리 표현하자면, "논리 프로그램을 수행하는 것"과 "상응하는 AND/OR 증명 트리 내에서 탐색하는 것" 간에는 유사성이 있음을 쉽게 알 수 있다. AND 병렬성은 AND/OR 증명 트리에서 AND 노드(node)로부터 OR 노드로의 병렬 탐색이고, OR 병렬성은 OR 노드로부터 AND 노드로의 병렬 탐색인 것이다. AND/OR 증명 트리의 성질에 따라, AND 노드에서

· 이 논문은 1994년도 성신여자대학교 학술연구조성비 지원에 의하여 연구되었음

† 종신회원 · 성신여자대학교 전산학과 교수

†† 종신회원 · 한국과학기술원 전산학과 교수

논문접수 1995년 7월 12일

심사완료 · 1995년 11월 18일

OR 노드를 완전히 병렬로 탐색하는 것은 변수의 결합 충돌(binding conflict)을 야기할 우려가 있어서 달성되지 않을 수 있다. 반면에 OR 병렬성은 완전하게 달성할 수 있다(물론 효율적인 구현의 문제는 별도로 하는 것이다).

그 결과, AND 병렬성을 이용하기 위한 많은 계산 모형이 제안되었다[1-3, 6, 12-15]. 그 중에서도 *AND/OR 프로세스 모델*[3]이 논리 프로그램의 계산 구조(즉, AND/OR 증명 트리)를 매우 잘 반영하고 있다는 점으로 인하여 널리 알려져 있다. (사실 많은 다른 모형들은 이 모형의 개선이거나 수정, 확장 혹은 변형이라고 볼 수도 있다.)

AND/OR 프로세스 모델에서는 AND/OR 증명 트리의 각 노드에 대해서 하나의 프로세스(process)가 만들어진다. 프로세스는 노드의 종류처럼 AND와 OR 프로세스, 두 가지가 있다. 하나의 AND와 OR 프로세스는 각각 하나의 클로즈(clause)와 하나의 리터럴(literal)을 해결하는 책임을 맡고 있다. AND/OR 프로세스 모델의 수행적 의미(operational semantics)는 개념적으로 간단하다. OR 프로세스가 활성화되면 이 OR 프로세스가 맡고 있는 리터럴과 단일화(unification)가 가능한 리터럴을 헤드(head)로 가지고 있는 모든 클로즈에 대해서 하나씩의 AND 프로세스가 동시에 만들어진다. OR 프로세스는 자식(child)인 AND 프로세스들의 수행을 시작시키고 그들로부터 (성공 혹은 실패의) 메시지를 기다린다. AND 프로세스의 경우는 상황이 좀 더 복잡하다. AND/OR 프로세스 모델은 AND 병렬성을 처리하기 위해서 세 개의 알고리즘을 사용하는데 *리터럴 정렬*(*literal ordering*), *전방 수행*(*forward execution*), *후방 수행*(*backward execution*) 알고리즘들이 그것이다. AND 프로세스가 활성화되면 이 AND 프로세스가 맡고 있는 클로즈 내의 바디(body) 리터럴 중 자신의 *생산자*(*producer* 혹은 *generator*) 리터럴이 모두 해결되었거나 아예 없는 것들에 대해 OR 프로세스를 동시에 만든다. 바디 리터럴들 간의 생산자-*소비자*(*consumer*) 관계(relation)를 결정하는 것이 리터럴 정렬 알고리즘의 역할이고, 바디 리터럴들 간의 *자료 종속성*(*data dependency*)을 이용해 이루어진다. 그리고 나서 AND 프로세스는 자식 OR 프로세스들의 수행을 시작시키고 그들로부터 (성공 혹은 실패의) 메시지를 기다린다. 이것이 전방 수행의 첫 단계이다. 전방 수행 알고리즘의 목적은 결합되지 않은(unbound) 변수들에 항(term)을 결합시키는 것이다. 만약 어떤 자식 OR 프로세스가 실패를 보고하면 후방 수행 알고리즘이 시작되고, 그 알고리즘에 의해 백트랙킹(backtracking)이 수행된다. (두 개 이상의 리터럴이 실패를 보고하는 상황을 *다중 실패*(*multiple failure*)라고 한다[15].)

앞에서 서술되었다시피 논리 프로그램의 AND-병렬 수행에서 가장 복잡한 부분은 후방 수행이다. 따라서, 효율적인 후방 수행에 대한 많은 연구가 수행되었고 많은 후방 수행 알고리즘들이 그 우열을 다퉈왔다. (여기서 어떤 후방 수행 알고리즘이 다른 것보다 효율적이라고 말하는 것은, 불필요하게 하는 일의 양이 적다든가 병행적인 일을 불필요하게 순차적으로 처리하는 등으로 지체하지 않는다는 뜻이다.) 그런데 후방 수행 알고리즘의 어떤 부분은 이러한 비교의 근거가 빈약하여 논쟁을 무의미하게 만드는 측면이 있다. 이에 대한 보다 자세한 설명은 2절에 나올 것인데, 이 논문에서는 이러한 비교의 보다 분명한 기준을 제시하고자 한다.

논문은 다음과 같이 구성되어 있다. 2절에서는 이 논문에서 다루는 문제의 보다 자세한 규정과 그 동기를 서술한다. 3절에서는 후방 수행 알고리즘의 비교 기준으로 제시하는 것과, 그것에 바탕을 둔 보다 효율적이라고 생각되는 후방 수행 알고리즘을 제안하고 설명한다. 4절에서는 다른 관련 연구와의 비교를 기술하고 토론한다. 5절은 결론으로서 논문을 끝맺는다.

2. 배경, 문제 인식 및 동기

이 절에서는 논문이 다루고 있는 문제를 보다 분명히 규정짓기 위해, 후방 수행을 보다 자세히 서술한다.

앞 절에서 언급했다시피, 후방 수행은 리터럴(이 논문에서는 '리터럴'과 'OR 프로세스'를 오해의 소지가 없을 때는 혼용할 것이다)이 실패를 보고했을 때 시작된다. (일단은 *단독 실패*(*single failure*)[15]의 경우에 대해서만 생각하겠다.) 만약 그 리터럴이 자료 종속 관계 상의 선행자(predecessor) 리터럴을 가지고 있지 않다면, 이 AND 프로세스의 전체 수행은 실패로 끝나게 된다. 그렇지 않은 경우에는 이 실패를 고치기 위해 *백트랙 리터럴*(*backtrack literal*)을 *재수행*(*redo*)시켜야만 한다. 일반적으로 그 실패를 고칠 수 있는 리터럴은 여러 개 있다. 즉, 두 개 이상의 '후보(candidate) 백트랙 리터럴'이 존재하는 것이다. 그런 경우 어느 리터럴을 먼저 재수행시켜서 그 변수의 값이 재결합(rebinding)되게 할 것인지를 정해야 한다. 달리 표현하자면, 어느 변수의 값을 먼저 변화시켜서 새로운 해(solution) 튜플(tuple)을 만들어 그 만족 여부를 검사받게 할 것인지를 정해야 한다. 어떤 가능한 해 튜플도 빠뜨리지 않기 위해서는 후보들 중에서 백트랙 리터럴을 선택하는 일을 어떤 정해

진 규칙을 가지고 체계적으로 해야 함은 명백하다. 이 문제를 해결하기 위해서, *내포 루프 모형(nested loop model)*[3]이라고 부르는 단순하면서도 자연스러운 방법이 제안되어 보편적으로 사용되고 있다. 이 방법에서는 바디 리터럴에 대해 선형 순서를 부여한다. 즉, 생산자-소비자(혹은 같은 뜻으로, 자료 종속) 관계를 나타내는 *자료 종속 그래프(data dependency graph*: DDG)를 구성한 뒤 너비-우선(breadth-first)으로 그 그래프를 순회하면 순차적으로 정렬된 리터럴들의 리스트를 얻을 수 있는 것이다. 이 리스트의 이름은 *순차적 정렬 리터럴 리스트(linearly ordered literals list*: LOLL)이고 AND/OR 프로세스 모델에서 중요한 자료 구조의 하나이다.

후방 수행과 관련해서는 해결해야 될 두 개의 중요한 문제가 있다. 첫번째는 앞 단락에서 서술한 소위 *백트랙 리터럴 선택 문제(backtrack literal selection problem)*이다. 내포 루프 모형에 따라 백트랙 리터럴을 선택하여 그것의 변수가 새로운 결합을 가지도록 재수행한 뒤에는, LOLL에서 백트랙 리터럴 뒤에 오는 리터럴들의 수행을 처음부터 다시 시작시킬 필요가 있을 수 있다. 이것은 가능한 값들의 조합을 건너뛰지 않기 위해서이다. 따라서 우리는 어느 리터럴들을 처음부터 다시 시작, 즉 *재초기화(reset)*시켜야 하는지를 결정해야 한다. 이것이 *재초기화 문제(reset problem)*라고 알려진 두번째 과제이다. (여기까지의 서술은 우리로 하여금 명령형 언어(imperative language)에서의 반복 구조(repetitive structure)를 연상하게 한다. 내포 루프로 이루어진 반복 구조에서는 바깥 루프에 있는 인덱스(index) 변수의 값이 바뀌면 모든 안쪽 루프의 인덱스 변수는 재초기화가 되기 때문이다. *실제로* 내포 루프 모형이라는 이름도 그 유사성 때문에 붙여진 것이다.) 재수행되거나 재초기화된 리터럴로부터 전달되는 값을 소비하는 리터럴들은, 자신의 생산자 리터럴들의 변화로 인해 과거에 전달받았던 값들은 무의미하므로 자연히 *취소(cancel)*된다[3, 5].

말할 것도 없이 재초기화되는 리터럴의 수가 적을수록 후방 수행은 효율적으로 된다. 이런 맥락에서, LOLL의 백트랙 리터럴 뒤에 오는 모든 리터럴들을 다 재초기화시키지 않고도 바른 수행을 할 수가 있다. 왜냐하면 LOLL이라는 이름을 보면 그 리스트가 선형 순서(혹은 *단순 순서(simple order)*)임을 뜻하는 듯 하지만, 사실 자료 종속 관계는 *부분 순서(partial order)*이기 때문이다. 이렇게 일부의 리터럴만 재초기화하려는 시도[1, 2, 12, 13]를 *선택적 리터럴 재초기화(literal-level selective resetting)*라고 부르겠다 반면에 *선택적 해 재초기화(solution-level selective resetting)*라고 부를 만한 연구도 있었다[6, 14]. 이 연구들은 재초기화되는 리터럴이 만든 과거의 해 중에서 '일부'만 다시 생성되도록 하여 재초기화의 효율성을 제고하려고 한다. 이 논문에서는 선택적 해 재초기화는 다루지 않는다. (리터럴이냐 해냐에 관계없이, 논리 프로그램의 효율적인 병렬 수행을 위해서는 선택적 재초기화가 상당히 중요하다고 우리는 생각한다. 그것은 *지능적 백트래킹(intelligent backtracking)*과 짝을 이루는 동작이라고 할 수 있는데, 그 이유는 둘 다 효율을 위해서 AND/OR 증명 트리의 전지(pruning) 작업을 수행하는데 다만 다른 위치에서 한다는 차이가 있을 뿐이기 때문이다.)

이러한 관찰의 결과, 여러 선택적 리터럴 재초기화 알고리즘이 등장했다. 앞서 서술한 대로, 어떤 리터럴이 재수행됨으로 인해서 재초기화가 필요하게 된 리터럴들을 결정하는 것이 재초기화 알고리즘의 목적이고, 그 재초기화될 리터럴들의 집합을 가능한 한 작게 만드는 것이 선택적 리터럴 재초기화 알고리즘의 목표이다. 물론 부주의하게 이 집합의 크기를 줄이면 재초기화가 필요한 리터럴까지 이 집합에서 누락됨으로 인해서, 해당 논리 프로그램 내에 존재하는 해를 발견하지 못하는 상황이 발생할 수도 있다. 따라서, 만약 어떤 알고리즘이 재초기화가 불가피한 리터럴들만 재초기화시킨다면 그것이 곧 가장 바람직한 선택적 리터럴 재초기화 알고리즘이 될 것이다.

현재까지는 스스로 '가장 바람직한' 선택적 리터럴 재초기화 알고리즘임을 보인 연구는 없었고, 다만 선행 연구들보다 효율적이라고 주장한 것들뿐이었다. 더우기 그 주장이 예를 통해 진술되는 식이어서, AND-병렬 수행의 임의 상황에서도 그 주장이 옳은지, 최악의 경우 많은 상황에서는 그 주장이 옳으나 소수의 경우에는 오히려 존재하는 해를 놓친다든가 하는 일은 없는지(효율성을 위해 재초기화되는 리터럴들의 집합을 지나치게 줄이려다가 발생하는 경우) 조차 확고한 근거 위에서 주장된다고 볼 수 없었다. 따라서 어떤 선택적 재초기화 알고리즘이 존재하는 해를 놓치는 일이 발생하지 않으면서도, 기존의 방법에 비해 어떤 상황에서나 그 효율성이 높거나 적어도 같음을 보이는 데 사용될 수 있는 기준의 제시는 가치있는 일이 되리라고 본다. 이 논문에서는 그러한 기준으로서, '어떤 리터럴의 현재 결합이 바뀜으로 인해서, 직관적으로 보아 재초기화의 필요성이 명백한 리터럴들의 최소 집합'을 정의하고, 그 유용성을 선행 연구의 분석을 통해 보이려고 한다. (그 집합을 재초기화의 정확성(correctness) 증명을 위한 *참조 집합*

(*reference set*)이라고 부르겠다)

3. 동적 영향 집합

이 절에서는 먼저 재초기화의 정확성 증명을 위한 참조 집합의 바탕 생각을 서술한 뒤, 관련 정의 등을 기술하겠다.

3.1 원리

그림 1의 논리 프로그램 *목표문*(*goal statement* 혹은 *query*)과 그것에 대한 DDG(그림에서 점선으로 된 가지는 일단 무시한 경우)에서, 리터럴 #5(이후 바디 리터럴은 이렇게 바디 내에서 자신의 순서 번호에 의해 표시되기도 한다)가 만들어 낸 어떤 해들, 예컨대 e_0, e_1, . . . , e_n 등을 가지고는 #8이 성공할 수가 없어서 #8에 의해 이 해들이 거부되었다고 가정하겠다. 그리고 #4가 실패했다. 그 결과 #2가 백트랙 리터럴로 선택되어 재수행되었다. 이제 재초기화될 리터럴들을 결정해야 한다.

← p1(A,B), p2(A,C), p3(A,D), p4(B,C), p5(B,E), p6(D), p7(C,E), p8(E).

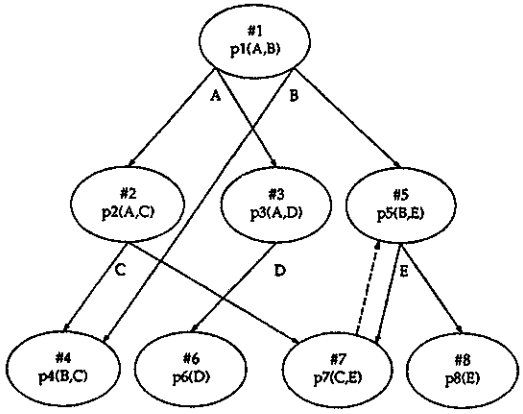

그림 1 목표문의 예와 그것을 위한 DDG 및 AG

이 (재초기화) 문제와 관련하여 Conery는 내포 루프 모형을 그대로 따르는 매우 단순한 접근 방법을 택했다. 즉 LOLL에서 재수행되는 리터럴의 뒤에 오는 모든 생산자 리터럴(그림 1의 예에서는 #3과 #5)들을 재초기화시킨다[3]. 그러나 그림의 DDG를 보면 명백하다시피, #3을 재초기화시키는 것은 이 상황에서는 필요없는 일이다.

이후 Conery의 방식이 너무 안이했으며, 재수행되는 리터럴의 뒤에 있는 어떤 리터럴은 재초기화될 필요가 없음을 약간의 분석을 통해 알 수 있다는 점을 여러 연구자들이 지적했다. 곧 선택적 리터럴 재초기화가 가능함을 지적한 것이다. 선택적 리터럴 재초기화라는 생각의 출발은 Chang 등의 *백트랙 경로*(*backtrack path*)[1]부터이다. Chang 등과 동일한 생각에 근거하여, Choe 등[2]과 Park 등[13]도 또한 독자적으로 선택적 (리터럴) 재초기화 방법을 개발했다. 이 모든 방법들은 클로즈 내의 어떤 리터럴들 간에는 '독립성(independence)'이 있을 수 있음을 주목했다. 그래서 재수행되는 리터럴과 '독립적인' 리터럴들은 재초기화시키지 않는 것이다.

우리는 리터럴들 간의 독립성을, 우리가 *영향*(*affection*)[5]이라고 부르는 매우 간단한 개념을 사용해 분명하게 설명할 수 있다고 생각한다. 영향은 단어 그대로 클로즈 내의 리터럴들이 서로 끼치는 효과를 일반화시킨 것이다. 영향은 두 가지 종류로 나뉠 수 있는데, '값 전달(forwarding values)'과 '재수행 야기(causing redoing)'이다. DDG로 표현되는 자료 종속성(즉 값을 전달하고 받는 것)은 영향의 특별한 경우라고 생각할 수 있다. DDG는 자식 리터럴에 대한 부모 리터럴의 영향은 설명해 주나, 부모 리터럴에 대한 자식 리터럴의 영향은 전혀 표현하지 않는다. 그렇지만 자식 리터럴은 또한 그 부모에게 영향을 줄 수 있다—자식 리터럴의 실패는 그 부모 리터럴의 재수행을 야기시키는 것이다.

따라서 만약 우리가 리터럴들 간에 주고받은 영향의 경력(affection history)에 관한 정보를 유지한다면, 실패가 일어났을 때 후방 수행을 자연스럽고 효율적으로 수행할 수 있다. 이것은 다음과 같은 생각에 근거한 것이다: "실패한 리터럴에 영향을 끼쳐 온 리터럴들이 백트랙 리터럴의 후보가 될 것이고(실패한 리터럴에 영향을 끼치지 않는 리터럴들은 재수행된다 하더라도 그 실패를 고칠 수 없을 것이기 때문에), 그 중 LOLL에서의 순서로 보아 가장 오른쪽의 것이 내포 루프 모형에 따라 백트랙 리터럴로 선택되어야만 한다는 것은 직관적으로 명백하다. 그 백트랙 리터럴을 재수행시킨 다음에는, 백트랙 리터럴에 영향을 받아온 생산자 리터럴들을 재초기화시키고(자신에게 영향을 미치는 리터럴이 재수행에 의해 상태가 변했기 때문에), 재수행되거나 재초기화되는 부모를 가진 자손 리터럴들을 취소시켜야만 한다(자식에게 값을 전달해 준 부모가 재수행이나 재초기화를 통해 그 값을 바꿨으므로)는 것 또한 명백하다. 그러므로, 두 리터럴이 서로 영향을 끼치지 않으면(즉, 서로 영향을 주지도 받지도 않으면) 서로 *독립적*(*independent*)이다."

DDG를 사용해 자료 종속성을 나타내듯이, 리터럴들 간의 이러한 영향 관계를 *영향 그래프*(*affection graph*: AG)[5]라고 부르는 또 다른 방향성(directed) 그래프로 가시화시킬 수 있을 것도 같다. (그림 1에서 점선으로

된 가지를 포함하는 그래프가 AG이다. 이렇게 되면 DDG는 AG의 부분 그래프(subgraph)가 된다.) AG의 노드는 DDG와 마찬가지로 리터럴들을 나타낸다. AG에는 *자료 가지(data arc)*와 *재수행 가지(redo arc)*, 이렇게 두 가지 종류의 가지가 존재한다. 자료 가지와 재수행 가지는 각각 자료 종속과 재수행 야기를 나타낸다. (따라서 DDG의 가지는 자료 가지이다.) 어떤 실패한 리터럴로 인하여 그것에 대한 백트랙 리터럴이 재수행될 때마다, 실패한 리터럴(을 표시하는 노드)로부터 백트랙 리터럴(을 표시하는 노드)로 새로운 재수행 가지가 만들어진다. 따라서, AG 상에서 어떤 리터럴 P로부터 다른 리터럴 Q까지 (방향성) 경로(path)가 존재하면 P는 Q에 영향을 미친다는 것을 알 수 있다. 예를 들어서 위의 그림 1과 함께 서술한 상태가 현재의 실패 상황이라면, 우리는 그림의 DDG에서 #8로부터 #5로 가상의 (재수행) 가지를 상정할 수 있을 것이다. 그 결과 #8로부터 #5까지 새로운 경로가 개설되고, 곧 #8은 #5에 영향을 미치게 된다. 이렇게 되면 #2가 재수행된 다음에는 #2에 의해 영향을 받는 리터럴들(#4와 #7)에 대해 무엇인가의 작업을 해야만 할 것이다. 이 예의 경우에는 그 리터럴들이 #2가 생산한 값들을 소비하기 때문에 취소되어야만 한다. #3과 #5는 #2에 독립적이므로 그들에 대해서는 아무런 처치가 필요없다. 그 반면에 만약 #8 대신에 #7이 실패해서 #5의 재수행을 야기했었다면, #5가 #2의 영향을 받는 리터럴이 되었을 것이다(#2로부터 #7까지의 자료 가지를 거치고, #7로부터 #5까지의 재수행 가지로 이루어지는 경로가 생김을 알 수 있다). 따라서 #2를 재수행한 다음에는 #5는 재초기화되어야만 한다. 왜냐하면 #5는 #2의 영향을 받는 리터럴들(#4, #5, #7, #8) 중 재수행되거나 재초기화되는 어떤 리터럴들의 자식도 아니기 때문이다. (그림 1을 참조하라. 실선으로 된 것은 자료 가지, 점선으로 표시된 것은 재수행 가지들이다.)

우리는 앞의 두 단락에서 서술한 생각이 재수행, 재초기화 그리고 취소의 근저에 있는 원리를 직관적으로 자연스럽게 반영하고 있다고 본다.

그래서, 어떤 리터럴 P의 *동적 영향 집합(dynamic affecting set)*이란 수행 중에 P에 영향을 받은 리터럴들의 집합을 나타내기로 한다. 이 집합은 영향의 경력을 이용해 동적으로 구성된다. 동적 영향 집합은 공통 부분이 없는(disjoint) 두 개의 집합으로 더 쪼갤 수 있는데, 곧 *동적 취소 리터럴 집합(dynamic canceled literals set)*과 *동적 재초기화 리터럴 집합(dynamic reset literals set)*이다. 이 논문의 목표는 앞에서 기술한 원리에 따라 이들 집합을 구하는 방법을 제안하고 그것들을 후방 수행에 이용하는 것이다.

3.2 알고리즘

AG는 다소 추상적인 자료 구조이므로, 이제 동적 영향 집합과 관련 연산의 실제적인 정의를 내리도록 하겠다. 그에 앞서 약간의 예비 정의가 나올 것이다. 다음 정의는 내포 루프 모형을 따르기 위해 필요하다

정의 3.1: 축소 자료 종속 그래프(reduced DDG)[2] 리터럴 P를 위한 "축소 자료 종속 그래프" G_P는 DDG G의 부분 그래프이고 다음을 만족한다.

$$\mathrm{VERTEX}(G_P) = \{\, Q \mid Q \in \mathrm{VERTEX}(G),\ \mathrm{ORD}(P) \le \mathrm{ORD}(Q) \,\}$$ 이고

$$\mathrm{EDGE}(G_P) = \{\, Q \to R \mid Q \to R \in \mathrm{EDGE}(G),\ \mathrm{ORD}(P) \le \mathrm{ORD}(Q) < \mathrm{ORD}(R) \,\}.$$

$\mathrm{VERTEX}(G_P)$와 $\mathrm{EDGE}(G_P)$는 G_P 내의 노드와 가지의 집합을 각각 나타낸다. $\mathrm{ORD}(P)$는 LOLL에서의 P의 순서 번호를 뜻한다. □

다음 정의들은 재수행으로 인해 형성되는 새로운 영향 관계를 설명하고 있다.

정의 3.2: 재수행 야기(caused redoing; REDO)
P와 Q를 두 개의 상이한 리터럴들이라고 하자. 우리는 이 리터럴들이 다음 조건을 만족할 때 그리고 그 때만(if and only if, iff) "P REDO Q"라고 말한다:

(1) $P \in \mathrm{VERTEX}(G_Q)$,
(2) 다음 조건 중 하나가 성립한다:
  (2.1) P의 실패가 직접적으로(directly) Q의 재수행을 야기한다.
  (2.2) 다음과 같은 리터럴 R이 존재한다: $R \neq P$, $R \neq Q$, $R \in \mathrm{VERTEX}(G_Q)$, P의 실패가 직접적으로 R의 재수행을 야기하고 R REDO Q. □

정의 3.3: 동적 영향 집합(dynamic affecting set; D_AFFECTING)
P와 Q를 두 개의 상이한 리터럴들이라고 하자. 우리는 이 리터럴들이 다음 조건을 만족할 때 그리고 그 때만 "P DIRECTLY_D_AFFECT Q"라고 말한다:

(1) $Q \in \mathrm{VERTEX}(G_P)$,
(2) 다음 조건 중 하나가 성립한다:
  (2.1) $Q \in \mathrm{CHILDREN}(P)$($\mathrm{CHILDREN}(P)$는 DDG 상에서 P의 자식들을 나타낸다),
  (2.2) 다음과 같은 리터럴 R이 존재한다: $R \neq P$, $R \neq Q$, $R \in \mathrm{VERTEX}(G_P)$, $R \in$

CHILDREN(P)이고 R REDO Q.

우리는 DIRECTLY_D_AFFECT의 이행적 종결(transitive closure), 즉 DIRECTLY_D_AFFECT$^+$를 D_AFFECT로 나타낸다. 그러면 D_AFFECTING(P) = { Q | P D_AFFECT Q }. □

이제 리터럴들간의 독립성을 그들의 영향 집합을 사용해 정의할 수 있게 되었다.

정의 3.4: 동적 독립성(dynamic independence)
P와 Q를 두 개의 상이한 리터럴들이라고 하자. 우리는 이 리터럴들이 다음 조건을 만족할 때 그리고 그 때만 "P와 Q는 동적으로 서로 독립적이다"라고 말한다:

$P \notin$ D_AFFECTING(Q)이고 $Q \notin$ D_AFFECTING(P) □

만약 어떤 리터럴이 재수행되거나 재초기화되면, 그 리터럴의 동적 영향 집합에 있는 리터럴들은 그 리터럴의 변화에 의해 영향을 받게 된다. 따라서 동적 영향 집합 내의 어떤 리터럴이 재수행되거나 재초기화된 리터럴들이 만든 값을 소비하고 있다면, 그 값은 이미 변했을 수 있으므로 자연히 그 리터럴은 취소되어야만 한다. 그 외의 경우는 재초기화될 것이다.

정의 3.5: 동적 취소/재초기화 리터럴 집합(dynamic canceled/reset literals set)

(1) D_CANCEL(P) = { Q | $Q \in$ CHILDREN(R), $R \in$ ({P} ∪ D_AFFECTING(P)) }

(2) D_RESET(P) = D_AFFECTING(P) - D_CANCEL(P)

정의 3.5에서 구해지는 D_RESET(P)가 P가 재수행되거나 혹은 재초기화되어 그 결합이 바뀌었을 때 후방 수행의 정확성을 유지하기 위하여 추가로 재초기화 작업이 필요한 리터럴들의 모임, 즉 "재초기화에 대한 참조 집합"이 된다.

다음 두 정리는 동적 영향 집합에 대해 분명하게 관찰되는 점들을 요약한 것이다.

보조 정리 3.6: 동적 전이성(dynamic transitivity)
만약 P D_AFFECT Q이고 Q D_AFFECT R이면, P D_AFFECT R이다.

증명: 정의 3.3으로부터 명백하다. □

정리 3.7: 동적 취소(dynamic cancellation)
만약 P D_AFFECT Q이고 $R \in$ D_CANCEL(Q)이면, $R \in$ D_CANCEL(P)이다.

증명. 보조 정리 3.6에 의해서, 만약 P D_AFFECT Q이면 ({Q} ∪ D_AFFECTING(Q)) ⊆ D_AFFECTING (P)이다. 또 정의 3.5에 따라 $R \in$ D_CANCEL(Q)라는 것은 $R \in$ { L | $L \in$ CHILDREN(S), $S \in$ ({Q} ∪ D_AFFECTING(Q)) }임을 뜻한다. 따라서 $R \in$ D_CANCEL(P)이다 □

이제 여기까지 설명한 개념과 정의에 근거한 후방 수행 알고리즘을 서술하겠다.

알고리즘 3.8: 후방 수행

(1) /* 백트랙 리터럴의 선택 */

(1.1) 실패한 리터럴 L_f에 대해서 후보 백트랙 리터럴 집합 L_B를 구성한다.
/* 후보 백트랙 리터럴 집합을 구하는 알고리즘은 [1, 6, 7, 11, 12, 14, 15]을 참고 할 수 있다. 이 집합은 관련 문헌에 여러 상이한 이름으로 나타나는데, B-LIST나 REDO SET 등이다. */

(1.2) L_B의 원소 중에서 백트랙 리터럴(실제로는 선형 순서로 보아 가장 오른쪽의 리터 럴) L_b를 선택한다.

(2) /* 영향 집합의 수정; 재수행, 재초기화, 취소 */

(2.1) (L_B - {L_b})에 있는 원소들의 영향 집합을 갱신한다.
/* 이 절의 정의들로부터 이 연산이 어떠해야 하는지는 분명하다고 본다. 일종의 영향 집합이라고 볼 수 있는 것을 갱신하는 예를 [6-8]에서 볼 수 있다. */

(2.2) L_b를 위한 OR 프로세스에게 재수행 메시지를 보낸다.

(2.3) D_RESET(L_b)에 포함되어 있는 각 리터럴에 대해 재초기화 연산을 수행한다. 즉 그 리터럴들은 지금까지 생산했던 해를 처음부터 다시 생성하게 된다.

(2.4) D_RESET(L_b)에 포함되어 있는 각 리터럴, 예컨대 L_r을 재초기화시킨 다음, L_r을 D_RESET(L_b)로부터 삭제한다.
/* AG 상에서 얘기하자면, L_b로부터 L_r까지의 경로에서 동적으로 구성된 재수행 가지가 재초기화에 의해 삭제되는 것이다. */

(2.5) D_CANCEL(L_b)에 포함되어 있는 각 리터럴에 대응하는 OR 프로세스에게 취소 메시지를 보낸다. □

3.3 다중 실패의 처리

앞 절까지는 단독 실패가 일어난 경우를 다루었으나, 실제로는 여러 OR 프로세스의 병행 수행(concurrent

execution)으로 인해 다중 실패인 경우가 훨씬 빈번하게 일어날 듯 싶다.

다중 실패를 처리하는 알고리즘은 Woo와 Choe가 처음 제안했다[15]. 후속 연구가 진행되어 리터럴들 간에 독립성이 존재한다는 사실을 발견함에 따라, 다중 실패를 병렬로 처리하려는 알고리즘들이 등장하기 시작했다[2, 13]. 이러한 알고리즘들의 원리는 단순하면서도 명백하다. 즉, 두 개의 상이한 실패 리터럴에 대한 두 개의 상이한 백트랙 리터럴이 서로 독립적이라면, 그들의 재수행은 병행하여 처리될 수 있다는 것이다.

다중 실패를 처리하는 우리의 알고리즘은 [2]에 있는 알고리즘의 동적 변형판(dynamic version)이다.

알고리즘 3.9: 동적 다중 백트래킹(dynamic multiple backtracking)

(1) 각 실패한 리터럴 $L_{fi}(i = 1, 2, \ldots, n)$에 대해서,
  (1.1) L_{fi}에 대한 백트랙 리터럴 L_{bi}를 찾는 알고리즘을 호출한다.
  (1.2) 만약 (1.1)의 결과가 "AND 프로세스의 실패"라면 역시 "AND 프로세스의 실패"라는 결과를 가지고 이 알고리즘을 벗어난다. 그렇지 않다면 계속한다.
(2) $L_{bi}(i = 1, 2, \ldots, n)$를 ORD($L_{bi}$)에 따라 오름 차순으로 정리하여 BACKTRACK_LIST를 만든다.
(3) BACKTRACK_LIST가 빌 때까지 다음 소단계들을 반복한다.
  (3.1) 가장 왼쪽의 리터럴, 예컨대 L_{bj}를 선택한다.
  (3.2) L_{bj}와 D_AFFECTING(L_{bj})에 포함되어 있는 리터럴들을 BACKTRACK_LIST로부터 삭제한다.
  (3.3) L_{fj}에 대한 후방 수행 알고리즘을 수행한다. □

4. 비교와 토의

이 절에서는 다른 선택적 리터럴 재초기화 방법을 우리 방법과 비교한다.

4.1 Choe 등의 방법

"영향"이라는 개념은, 비록 분명하게 언급되지는 않았지만 Choe 등[2]에 의해서 처음 제안된 것이다. 그들의 "영향 집합"은 개념적인 면에서 우리의 동적 영향 집합과 근본적으로 동일하다. 허나 그들의 분석은 정적(static)이기 때문에 리터럴들 간의 어떤 독립성을 발견하지 못할 수 있다는 뜻에서, 결과적으로 보다 안이한 방법이다 따라서 그들의 방법은 불필요한 재초기화와 취소를 행할 가능성이 있다. 또한 그들의 방법은 백트래킹의 병렬성도 우리 방법에 비해 보다 낮을 수 있다.

4.4절에서의 참조와 비교를 위해, 그들 방법에서의 핵심 정의와 정리들을 약간 수정된 형태로 아래에 기술한다

정의 4.1: 정적 영향 집합(static affecting set)[2]
P와 Q를 두 개의 상이한 리터럴들이라고 하자. 우리는 이 리터럴들이 다음 조건을 만족할 때 그리고 그 때만 "P S_AFFECT Q"라고 말한다:

(1) $Q \in$ VERTEX(G_P),
(2) 두 개의 노드 P와 Q가 G_P 상에서 서로 "연결되어(connected) 있다".

DDG 상에 있는 두 개의 노드는, 가지의 방향성을 무시하고 둘 사이에 경로가 존재할 때 *연결되어* 있다고 말한다. 그러면 S_AFFECTING(P) = { Q | P S_AFFECT Q }. □

정의 4.2: 정적 독립성(static independence)
P와 Q를 두 개의 상이한 리터럴들이라고 하자. 우리는 이 리터럴들이 다음 조건을 만족할 때 그리고 그 때만 "P와 Q는 정적으로 서로 독립적이다"라고 말한다:

$P \notin$ S_AFFECTING(Q)이고 $Q \notin$ S_AFFECTING(P)

혹은 동일한 의미로서

S_AFFECTING(P) $\cap$ S_AFFECTING(Q) = $\emptyset$ □

정의 4.3: 정적 취소/재초기화 리터럴 집합(static canceled/reset literals set)[2]

(1) S_CANCEL(P) = { Q | $Q \in$ CHILDREN(R), $R \in$ ({P} $\cup$ S_AFFECTING(P)) }.
(2) S_RESET(P) = S_AFFECTING(P) - S_CANCEL(P) □

보조 정리 4.4: 정적 전이성(static transitivity)[2]
만약 P S_AFFECT Q이고 Q S_AFFECT R이면, P S_AFFECT R이다. □

정리 4.5: 정적 포함(static inclusion)[2]
만약 P S_AFFECT R, Q S_AFFECT R이고 ORD(P) < ORD(Q)이면, P S_AFFECT Q이고 따라서 ({Q} $\cup$ S_AFFECTING(Q)) $\subset$ S_AFFECTING(P)이다. □

정리 4.6: 정적 취소(static cancellation)[2]
만약 P S_AFFECT Q이고 $R \in$ S_CANCEL(Q)이면, $R \in$ S_CANCEL(P)이다. □

4.2 Park 등의 방법

Park 등은 또다른 선택적 리터럴 재초기화 방법을 제안한 바 있다[13]. 이 방법 역시 정적 분석(static analysis)을 이용하고, Choe 등의 방법과 완전히 똑같은 결과를 보인다. 그들의 방법은 Choe 등의 방법에 비해 서술 방식에 있어서 약간의 차이가 있지만, 3.1절에서 언급했다시피 바탕을 두고 있는 생각은 동일하다.

역시 4.4절에서의 참조와 비교를 위해, Park 등이 제안한 방법의 핵심 정의를 아래에 포함시켰다.

정의 4.7 Park 등의 방법[13]

(1) CON은 모든 소비자 리터럴들의 집합을 나타낸다. 같은 의미로 DDG 상의 모든 단말 노드(leaf node)들의 집합이라고 할 수도 있다.

(2) "리터럴 P의 부모 집합"은 PARENTS(P) = { Q | DDG 상에서 Q로부터 P로 오는 방향성 가지가 존재한다 }로 정의된다.

(3) "리터럴 P로부터 직접적으로 도달 가능한 리터럴 집합(directly reachable literal set)"은 DRLS(P) = { Q | DDG 상에서 P로부터 Q에 이르는 방향성 경로가 존재한다 }로 정의된다.

(4) "리터럴 P로부터 간접적으로 도달 가능한 리터럴 집합(indirectly reachable literal set)"은 IDRLS(P) = ID(P) - DRLS(P)로 정의된다. 여기서 ID(P)는 다음과 같이 정의된다. ID(P) = { Q | DDG 상에서 P로부터 Q에 이르는 방향을 무시한 경로가 존재하고, 그 경로 상에 있는 모든 리터럴들과 Q가 선형 순서로 보아 P보다 뒤에 있다 }.

(5) "리터럴 P와 관련한 재초기화 리터럴 집합(related reset literal set)"은 RRLS(P) = { Q | $Q \in$ IDRLS(P)이고 PARENTS(Q) $\cap$ IDRLS(P) = $\emptyset$ }로 정의된다.

(6) "리터럴 P의 취소 리터럴 집합(cancel literal set)"은 CLS(P) = $\bigcup_{Q \in (RRLS(P) \cup P)} DRLS(Q)$ 로 정의된다.

(7) 두 개의 생산자 리터럴 P와 Q는 다음 조건을 만족할 때 그리고 그 때만 "독립적"이라고 말한다:
(RRLS(P) $\cup$ CLS(P)) $\cap$ (RRLS(Q) $\cup$ CLS(Q)) = $\emptyset$. □

4.3 Ng과 Leung의 방법

Ng과 Leung은 또다른 선택적 리터럴 재초기화 방법을 제안했다[12]. 이 방법은 수행 시(run-time)의 정보를 이용한다. 허나 여전히 우리 방법보다는 덜 효율적이다. (보다 자세한 비교는 [8]에 서술될 것이다.)

더우기 문제인 점은 그들의 방법이 "완결성(completeness)을 결여"하고 있다는 것이다. 즉, 그들의 방법은 어떤 상황에서는 꼭 수행해야만 하는 재초기화를 빠뜨림으로 인해 적법한 해를 놓칠 수 있다[8]. 이 방법은 아예 잘못된 것이기 때문에, 다음 절에서 우리 방법과의 비교는 하지 않을 것이다.

4.4 비교

4.1절과 4.2절에서 기술된 두 방법은 모두 클로즈를 정적으로 분석하여 각 바디 리터럴에 대한 재초기화/취소 리터럴 집합을 결정해 버린다. 따라서 그 방법들은 수행 시 정보(즉 영향 경력)를 이용하는 우리 방법보다 더 보수적이고 안이하다. (물론 우리 방법은 개선된 동작에 대한 수행 시 부담을 져야만 한다.)

예를 들어 그림 1을 다시 생각해보자. 3.1절에서 묘사된 첫번째 실패 상황에서 두 방법 모두 재수행된 리터럴 #2의 뒤에 있는 생산자 리터럴들 중 #5를 재초기화시킨다. 그러나 #5를 재초기화시키는 것은 전혀 무익하다. #5가 과거에 만든 해들은 모두 #8에 의해 거부되었고, 또한 #8은 #5가 만들어내는 변수 E의 값 외에 다른 변수들의 값은 소비하고 있지 않기 때문이다. 반면에 만약 #5의 해가 #8이 아닌 #7에 의해 거부되었다고 한다면, 변수 C와 E의 값들(#7이 소비하는 값들)의 모든 가능한 조합을 빠뜨리지 않기 위해서 리터럴 #2의 재수행 후 #5는 꼭 재초기화되어야만 한다. 두 방법들은 실제 수행 시에 #7이 실패하지 않을 것이라는 것을 컴파일 시에 미리 판단할 수 없는 일이기에, 어쩔 수 없이 #5를 재초기화하지 않을 수 없는 것이다.

수행 시의 정보를 이용하는 덕분에, 우리 방법이 [2, 13]의 방법보다 적은 재초기화와 취소를 수행한다는 점은 아주 분명해 보인다. 게다가 후방 수행의 경우도 (리터럴들 간의 독립성을 세분하여 보기 때문에) 보다 병행처리가 가능하다.

지금까지의 토의를 정리하여, [2, 13]과 이 논문에서 정의된 여러 집합들 간의 관계를 요약하면 다음과 같다.

정리 4.8: 정의된 집합들 간의 관계

(1) RRLS(P) $\cap$ CON = $\emptyset$[13].
(1') S_RESET(P) $\cap$ CON = $\emptyset$.
(1'') D_RESET(P) $\cap$ CON = $\emptyset$.
(2) RRLS(P) $\cap$ CLS(P) = $\emptyset$[13].
(2') S_RESET(P) $\cap$ S_CANCEL(P) = $\emptyset$
(2'') D_RESET(P) $\cap$ D_CANCEL(P) = $\emptyset$.
(3) DRLS(P) $\cup$ IDRLS(P) = RRLS(P) $\cup$ CLS(P)[13].
(3') DRLS(P) $\cup$ IDRLS(P) = S_RESET(P) $\cup$ S_CANCEL(P)
(3'') D_RESET(P) $\cup$ D_CANCEL(P) $\subseteq$ DRLS(P) $\cup$ IDRLS(P).

(4) S_RESET(P) = RRLS(P)
(4′) D_RESET(P) ⊆ RRLS(P).
(5) S_CANCEL(P) = CLS(P).
(5′) D_CANCEL(P) ⊆ CLS(P)

증명: 정적 분석 혹은 동적 분석을 채택한 여러 방법들의 의미있는 비교를 위해서, 여기서부터는 DDG가 처음 만들어진 후에 변화하지 않는다고 가정한다. (동적으로 DDG가 변화하는[3] 상황에서는 동적 분석을 채택한 방법이 나은 것은 너무나 당연한 일이기 때문에, 그 점만을 가지고 우리 방법이 기존의 방법에 비해 낫다고 주장하는 것은 불공정한 비교라고 판단되어서이다.)

(1′) 정의 4.1에 의해서.
(1″) 정의 3.3에 의해서.
(2′) 정의 4.3에 의해서.
(2″) 정의 3.5에 의해서.
(3′) (3), (4), 그리고 (5)에 의해서.
(3″) (3), (4′), 그리고 (5′)에 의해서.
(4) 식의 좌변인,
S_RESET(P)
= S_AFFECTING(P) − S_CANCEL(P)(정의 4.3에 의해서)
= ID(P) − S_CANCEL(P)(정의 4.1과 정의 4.7의 (4)에 의해서)
= ID(P) − CLS(P)((5)에 의해서)
= RRLS(P)((2), (3), 그리고 정의 4.7의 (4)에 의해서).
이것은 식의 우변이다.
(5) 식의 우변인,
CLS(P)
= ID(P) − RRLS(P)((2), (3), 그리고 정의 4.7의 (4)에 의해서)
= DRLS(P) ∪ IDRLS(P) − RRLS(P)(정의 4.7의 (4)에 의해서)
= DRLS(P) ∪ { Q | Q ∈ IDRLS(P)이고 PARENTS(Q) ∩ IDRLS(P) ≠ ∅ }(정의 4.7의 (5)에 의해서. DRLS(P) ∩ RRLS(P) = ∅임을 주의하라.)
= DRLS(P) ∪ ({ Q | Q ∈ CHILDREN(R), R ∈ IDRLS(P) } − { Q | Q ∈ CHILDREN(R), Q ∈ DRLS(P), R ∈ IDRLS(P) })(만약 Q가 IDRLS(P)에 속하면 PARENTS(Q)도 또한 IDRLS(P)에 속한다는 사실로부터)
= DRLS(P) ∪ { Q | Q ∈ CHILDREN(R), R ∈ IDRLS(P) }(집합 연산의 성질에 의해)
= { Q | Q ∈ CHILDREN(R), R ∈ {P} ∪ DRLS(P) } ∪ { Q | Q ∈ CHILDREN(R), R ∈ IDRLS(P) }(정의 4.7의 (3)에 의해서)
= { Q | Q ∈ CHILDREN(R), R ∈ {P} ∪ DRLS(P) ∪ IDRLS(P) }(집합 연산의 성질에의해서)
= { Q | Q ∈ CHILDREN(R), R ∈ {P} ∪ ID(P) }(정의 4.7의 (4)에 의해서)
= { Q | Q ∈ CHILDREN(R), R ∈ {P} ∪ S_AFFECTING(P) }(정의 4.1과 정의 4.7의 (4)에 의해서)
= S_CANCEL(P)(정의 4.3에 의해서).
이것은 식의 우변이다.
(4′)과 (5′)
D_AFFECT 관계가 S_AFFECT 관계보다 더 성립되기 어렵다는 사실로부터 D_RESET(P) ⊆ S_RESET(P)이고 D_CANCEL(P) ⊆ S_CANCEL(P)임은 자명하다. (예를들어서, 어떤 리터럴 P의 임의의 부모 리터럴 Q는 P와 "연결되어" 있다(정의 4.1을 참조하라). 그러나 수행 시간에 P가 실제로 Q의 재수행을 초래할 때 그리고 그 때만 P REDO Q이다(정의 3.2를 참조하라).) 따라서 (4)와 (5)로부터, D_RESET(P) ⊆ RRLS(P)이고 D_CANCEL(P) ⊆ CLS(P)임은 바로 도출된다 □

4.5 구체적인 예를 통한 비교

이 절에서는 하나의 구체적인 예를 가지고 우리들의 알고리즘이 다른 알고리즘들에 비해 효율적임을 보일려고 한다.

그림 2의 논리 프로그램을 생각해 보자. 이 프로그램의 목표문에 대한 DDG는 그림 3에 나와 있다.

```
← p1(A,B), p2(A,C), p3(A,D), p4(B,E), p5(B,C), p6(C,D), p7(D), p8(C,E), p9(D,E)

p1(a0,b0).   p2(a0,c0).   p2(a0,c1)   p3(a0,d0).   p4(b0,e0).
p4(b0,e1)    p5(b0,c1)    p6(c0,d0)   p6(c1,d0)    p7(d0)
p8(c0,e0)    p8(c0,e1)    p8(c1,e0)   p8(c1,e1)    p9(d0,e1).
```

그림 2 예제 논리 프로그램

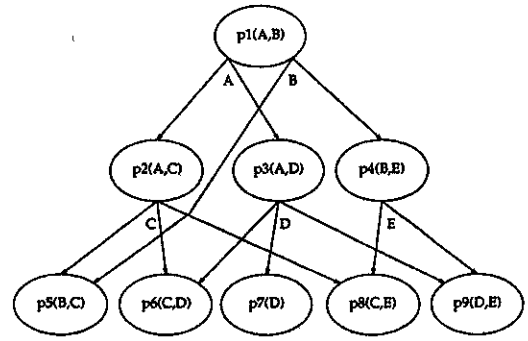

그림 3 그림 2의 목표문을 위한 DDG

이 경우, AND/OR 프로세스 모델 하에서는 다음과 같은 일련의 일들이 이 목표문을 위한 AND 프로세스 내에서 일어난다:

(1) *p1*은 { *A/a0*, *B/b0* }으로 결합되어 성공한다. (이

이후부터 혼동의 염려가 없을 때는 리터럴을 그 술어 이름(predicate name)만으로 간단히 나타내겠다.)

(2) *p2*, *p3*, *p4*는 각각 { *A/a0*, *C/c0* }, { *A/a0*, *D/d0* }, { *B/b0*, *E/e0* }으로 성공한다.

(3) *p6*, *p7*, *p8*은 각각 { *C/c0*, *D/d0* }, { *D/d0* }, { *C/c0*, *E/e0* }으로 성공한다.

(4) *p5*와 *p9*는 각각 { *B/b0*, *C/c0* }과 { *D/d0*, *E/e0* }에 의해 실패한다.

이제 실패가 일어났고, 또한 다중 실패 경우이다(두 리터럴 #5와 #9가 실패를 보고했다). 이 상황을 개선하기 위해서는 후방 수행 알고리즘이 수행되어야만 한다.

여기서 Choe 등이나 Park 등의 후방 수행 알고리즘을 채택했다고 가정하자. 또한 어떤 백트랙 리터럴 선택 알고리즘에 의해 #2와 #4가 각각 #5와 #9에 대한 백트랙 리터럴로 결정되었다고 가정하자. 그런데 이 두 개의 백트랙 리터럴의 재수행은 병렬로 이루어질 수가 없다. 왜냐하면 Choe 등이나 Park 등의 정의에 의하면 이 두 백트랙 리터럴은 서로 독립적이지 않기 때문이다 ($S_AFFECTING(p2) \cap S_AFFECTING(p4) = \{ p8, p9 \} \neq \emptyset$이고 $(RRLS(p2) \cup CLS(p2)) \cap (RRLS(p4) \cup CLS(p4)) = \{ p8, p9 \} \neq \emptyset$인 것을 유의하라.) 따라서, 가장 왼쪽의 백트랙 리터럴, 즉 *p2*만 재수행된다. 그 다음에는 $S\_RESET(p2)$(혹은 $RRLS(p2)$)에 있는 리터럴들은 재초기화되고, $S\_CANCEL(p2)$(혹은 $CLS(p2)$)에 있는 리터럴들은 취소된다.

그러므로, 이 시점부터는 다음과 같은 일련의 일들(아래에 있는 (5)부터 (8)까지)이 일어난다. (Choe 등의 알고리즘과 Park 등의 알고리즘은 완전히 같은 동작을 보인다는 점을 지적하고 싶다.)

(5) *p2*가 *p5*를 위한 백트랙 리터럴로 선택된다.

(6) *p2*가 재수행되어 { *A/a0*, *C/c1* }로 성공한다.

(7) *p3*과 *p4*는 재초기화된다.

(8) *p5*, *p6*, *p7*, *p8*, *p9*는 취소된다.

반면에 우리가 이 논문에서 제안한 후방 수행 알고리즘에 따르면, 동일한 실패 상황에서의 두 백트랙 리터럴 *p2*와 *p4*는 서로 독립적이다. 왜냐하면 $p2 \notin D_AFFECTING(p4)$이고 $p4 \notin D_AFFECTING(p2)$이기 때문이다 따라서 이 두 백트랙 리터럴들은 병렬로 재수행될 수 있다. 그러므로 이제는 다음과 같은 일련의 일들((5′)부터 (6′)까지)이 일어나게 된다:

(5′) *p2*와 *p4*는 각각 *p5*와 *p9*를 위한 백트랙 리터럴로서 선택된다.

(6′) *p2*와 *p4*는 재수행되고 각각 { *A/a0*, *C/c1* }과 { *B/b0*, *E/e1* }로 성공한다

*p2*와 *p4*로 백트래킹한 뒤, 이제 *p5*와 *p2*, *p9*와 *p4* 간에 새로운 REDO 관계가 형성되었으므로 관련된 리터럴들의 영향 집합들을 갱신해야만 한다. (이 예의 경우에서는 관련 리터럴들이 *p1*과 *p3*이다.)

영향 집합을 갱신한 뒤에는 일부 리터럴들을 재초기화시키고 취소하는 것이 필요하다. 다음과 같은 일들((7′)부터 (8′)까지)이 일어난다:

(7′) 어떤 리터럴도 재초기화되지 않는다.

(8′) *p5*, *p6*, *p8*, *p9*는 취소되나 *p7*은 취소되지 않는다.

앞에서 보다시피 우리 알고리즘에서는 재초기화 작업이 필요가 없으나, Choe 등이나 Park 등의 후방 수행에서는 *p3*과 *p4*가 재초기화되었다. 또한 우리 방법에서는 *p7*이 취소되지 않았으나 그들의 방법들에서는 취소되었다.

4.6 토의

제안된 방법의 다른 관련 연구에 대한 상대적 효율성은 4.4절과 4.5절의 비교를 통해서 거의 명백하다고 생각한다.

잘 알려져 있다시피, AND/OR 프로세스 모델에서는 DDG의 각 노드에 대응하는 자식 OR 프로세스(그리고 그것의 자식 AND 프로세스, 또 그것의 자식 OR 프로세스 등등)가 존재한다. 4.5절에서 예로 든 논리 프로그램은 하나의 간단한 예에 불과하다. 실제적인 프로그램 중에는 그것보다 복잡한 것들도 얼마든지 존재한다. 이럴 경우, AND/OR 프로세스 모델에 의한 병렬 수행은 대단히 많은 AND, OR 프로세스들이 계층을 이루고 있는 트리(tree) 구조이다. 이러한 구조 상에서 어떤 하나의 프로세스를 재초기화하거나 취소하는 것은 그 자손 프로세스들 모두에 영향을 미치게 된다. DDG와 영향 집합은 AND 프로세스에 의해 관리되는 내부 자료 구조들이다. 따라서 선택적 재초기화 및 취소 메시지 수의 감소를 목적으로 그 자료 구조들을 갱신하는 것이 불필요한 재초기화와 취소를 위해 수많은 프로세스들을 종료(termination)시키고 다시 생성(re-spawning)하는 것보다는 분명히 비용이 덜 들 것으로 보인다. (물론 프로세스도 하나의 자료 구조이지만, 우리의 알고리즘에서 관

리되는 자료 구조들보다는 크고 복잡한 것이 일반적이다.)

구현을 통해 다른 방법보다 효율적임을 보이는 것도 의미있다고 생각된다. 그러나, 그 정량적인(quantitative) 효율성의 정도는 구현의 세부적인 사항과 환경(예컨대, 병렬 프로그램 처리기가 수행되는 컴퓨터의 구조, 그 운영 체제에서 관리하는 프로세스 자료 구조의 형태, 병렬 프로그램 처리기 내부의 자료 구조 형태 등)에 큰 영향을 받을 것이다.

여러 가지 형태의 실제적인 구현을 생각할 수 있을 것 같다. 이 논문이나 [6]처럼, 백트랙 리터럴이 결정된 뒤 그것을 나머지 후보 백트랙 리터럴들의 영향 집합에 각각 추가시키는 방법이 있을 수 있다. 혹은 [7, 8, 12]에서 다룬 방식으로, AND 프로세스의 수행 동안에 필요한 정보를 수집한 뒤 실제로 영향 집합이 필요한 시점에 구성할 수도 있다.

이 논문에서 다룬 선택적 재초기화의 문제는, 인공 지능 분야에서 널리 다루는 *제약 만족 문제(constraint-satisfaction problems)*[10]의 한 특수한 형태로 볼 수도 있다. 특히, "백트래킹과 관련하여 과거에 했던 무익한 일을 거듭하는 것(thrashing)"을 막기 위한 Gaschnig의 "백마킹(backmarking) 알고리즘"[4]과 우리의 재초기화 알고리즘은 근본적인 생각에서는 동일하다.

5. 결 론

이 논문에서 우리는 논리 프로그램의 병렬 수행 시, 재초기화 작업의 정확성 증명을 위한 참조 집합을 제시했다. 이 집합은 영향이라고 부르는 직관적으로 간단한 개념으로부터 자연스럽게 도출되었다. 우리는 이 개념이 후방 수행의 문제점들을 정확히 파악하고 후방 수행 알고리즘의 배후에 있는 원리를 명백하게 묘사하고 있다고 생각한다.

제시된 참조 집합에 근거하여 기존의 재초기화 방법들을 정형적으로(formally) 비교했다. 또한, 보다 효율적이라고 판단되는 재초기화 방법도 제안했다. 이 방법은 수행 시의 정보를 이용하여 다른 방법에 비해 보다 적은 수의 취소 메시지를 만들고 불필요한 재초기화를 피한다. 이러한 비교도 역시 정형적으로 수행되었다.

참 고 문 헌

[1] J.-H Chang, A. M Despain, and D DeGroot, "AND Parallelism of Logic Programs Based on a Static Data Dependency Analysis," *Proceedings of the CompCon Spring*, pp. 218-225, 1985.

[2] K.-M Choe, M. J. Lee, and N. S. Woo, "Multiple Backtracking in the AND Process of Logic Programming," *Proceedings of TENCON 87 - IEEE Region 10 Conference*, pp 826-829, 1987

[3] J. S. Conery, "The AND/OR Process Model for Parallel Interpretation of Logic Programs," *Ph.D. Dissertation*, Department of Information and Computer Science, University of California at Irvine, 1983.

[4] J. Gaschnig, "A General Backtrack Algorithm that Eliminates Most Redundant Tests," *Proceedings of the 5th IJCAI*, pp. 457-457, 1977.

[5] D.-H. Kim and K.-M Choe, "The AND Process Configuration: A Unified Framework for AND-Parallel Evaluation of Logic Programs," Unpublished manuscript.

[6] D.-H. Kim, K.-M. Choe, B.-M. Chang, and S.-H. Lee, "The Scorer Method: A Scheme for Selective Resetting without Termwise Unification Failure Cause Analysis in the AND-Parallel Evaluation of Logic Programs," *Technical Report CS-TR-90-49*, Department of Computer Science, KAIST, 1990.

[7] D.-H. Kim, K.-M. Choe, and T Han, "Refined Mark(s)-Set-Based Backtrack Literal Selection for AND Parallelism in Logic Programs," *Parallel Processing Letters*, Vol. 2, No. 1, pp. 61-69, World Scientific Publishing Co., 1992.

[8] D.-H. Kim, K. M. Choe, and I.-S. Yun, "Analyses of the Recent Mark-Set-Based Backward Execution Algorithms for AND Parallelism in Logic Programs: Flaws, Corrections, and Proofs," Unpublished manuscript.

[9] R. Kowalski, *Logic for Problem Solving*, p287, Elsevier North Holland, New York, 1979.

[10] V. Kumar, "Algorithms for Constraint-Satisfaction Problems: A Survey," *AI Magazine*, Vol 13, No. 1, pp. 32-44, 1992.

[11] Y.-J Lin, V. Kumar, and C Leung, "An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs," *Proceedings of the 3rd ICLP*, pp. 55-68, 1986.

[12] K. W. Ng and H F Leung, "Competition: A Model of AND-Parallel Execution of Logic Programs," *The Computer Journal*, Vol. 33, No. 3, pp. 215-218, 1990.

[13] C.-I Park, K.-H Park, and M. Kim, "Efficient Backward Execution in AND/OR Process Model," *Information Processing Letters*, Vol 29, No. 4, pp. 191-198, 1988

[14] W Winsborough, "Semantically Transparent Selective Reset for AND Parallel Interpreters Based on the Origin of Failures," *Proceedings of*

the 4th SLP, pp. 134-152, 1987.
[15] N. S. Woo and K.-M. Choe, "Selecting the Backtrack Literal in the AND/OR Process Model," *Proceedings of the 3rd SLP*, pp. 200-210, 1986.

김 도 형

1985년 2월 서울대학교 공과대학 컴퓨터 공학과 졸업(학사). 1987년 2월 한국과학기술원 전산학과 졸업(석사). 1992년 2월 한국과학기술원 전산학과 졸업(박사). 1992년 3월부터 8월까지 한국과학기술원 정보전자연구소에서 연수연구원으로 근무. 1992년 9월부터 성신여자대학교 전산학과에 재직중. 관심분야는 프로그래밍 언어, 컴파일러, 논리 프로그래밍 등임.

최 광 무

1976년 서울대학교 전자공학과 졸업(학사). 1978년 한국과학기술원 전산학과 졸업(석사). 1984년 한국과학기술원 전산학과 졸업(박사). 현재 한국과학기술원 전산학과 부교수. 관심분야는 프로그래밍 언어론, 논리 프로그램의 병렬 수행 및 컴파일러 구성임.