최소방문 기록을 이용한 병행 시스템의 상태 공간 순회 기법

(State Space Exploration of Concurrent Systems with Minimal Visit History)

이 정 선 [†] 최 윤 자 [††] 이 우 진 [†††]
(Jung Sun Lee) (Yunja Choi) (Woo Jin Lee)

요 약 이른 시스템 개발 단계에서 요구사항 에러를 찾기 위해서는 시스템의 행위가 정형 언어로 표현되어야 하고, 도달성 분석이나 싸이클 탐색과 같은 분석 기술로 분석해야 한다. 하지만 이 기술들은 시스템의 상태 공간 순회를 기반으로 하기 때문에 시스템이 복잡해지면 상태 폭발 문제가 발생할 수 있다. 즉, 순회를 위한 메모리와 수행 시간이 큰 상태 공간 때문에 기하 급수적으로 증가한다. 본 논문에서는 병행 시스템에서 이러한 문제가 나타나는 원인을 지적하고 순회에 필요한 메모리를 줄이기 위해서 병행적 상태 공간을 합성하지 않고 순회한다. 또한 수행 시간을 줄이기 위해서 방문 기록을 최소한으로 유지하는 새로운 기술을 제시한다. 마지막으로 이 기법이 효과적임을 실험 결과를 통해 보인다.
키워드 : 상태 공간 순회, 상태폭발, on-the-fly

Abstract For detecting requirement errors in early system development phase, the behaviors of a system should be described in formal methods and be analyzed with analysis techniques such as reachability analysis and cycle detection. However, since they are usually based on explicit exploration of system state space, state explosion problem may be occurred when a system becomes complex. That is, the memory and execution time for exploration exponentially increase due to a huge state space. In this paper, we analyze the fundamental causes of this problem in concurrent systems and explore the state space without composing concurrent state spaces for reducing the memory requirement for exploration. Also our new technique keeps a visited history minimally for reducing execution time. Finally we represent experimental results which show the efficiency of our technique.
Key words : State Space Exploration, State Explosion, on-the-fly

· 본 연구는 방위사업청과 국방과학연구소의 지원으로 수행되었습니다. (UD060048AD)

† 학생회원 : 경북대학교 전자전기컴퓨터학부
rubyindark@naver.com
†† 정 회 원 : 경북대학교 컴퓨터학부 교수
yuchoi76@knu.ac.kr
††† 종신회원 : 경북대학교 컴퓨터학부 교수
woojin@knu.ac.kr
(Corresponding author임)

논문접수 : 2010년 5월 18일
심사완료 : 2010년 6월 28일

Copyright©2010 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.

정보과학회논문지: 소프트웨어 및 응용 제37권 제9호(2010.9)

1. 서 론

정형적 검증 방법은 시스템의 버그를 찾는 방법으로서 테스트와는 다르게 이른 개발 단계에서 적용할 수 있어서 상대적으로 비용이 적게 든다는 장점이 있다. 이러한 장점 때문에 많이 사용되고 있는 정형적 검증 방법 중에서 모델 체킹[1]은 시스템의 행동을 정형적 언어로 모델링하고, 그 모델이 시스템이 만족해야 할 속성을 만족하는지를 검사하는 방법이다.

모델 체킹은 정형언어로 모델링된 상태 모델과 검사를 원하는 속성을 입력으로 받고, 그 상태 공간을 순회하면서 시스템이 그 속성을 만족하는지 분석한다. 이 때 사용하는 상태 공간 순회 기법은 모델 체킹에서 기본이 되는 아주 중요한 부분이다. 상태 공간 순회 기법은 그 간단한 응용만으로도 시스템의 간단한 속성을 검사할 수 있다. 예를 들어 상태 공간을 너비 우선 탐색(breath

first search)으로 탐색하면 특정 상태까지의 가장 짧은 경로를 구하거나 도달성 분석을 할 수 있고, 깊이 우선 탐색(depth first search)으로 탐색하면 싸이클을 찾을 수 있다.

이러한 모델 기반 분석을 하기 위해서는 먼저 시스템의 행동을 UML 상태머신 다이어그램[2]이나 Petri-net [3] 등의 정형적 언어로 표현하게 되는데, 이를 분석할 때는 분석이 용이한 유한상태머신[4]으로 변환하여 사용하는 경우가 많다. 하지만 이 변환과정에서 정형 언어가 가지고 있던 구조적인 속성이 없어지거나 여러 유한상태머신이 하나로 합쳐지면서 상태의 수가 기하급수적으로 늘어나는 상태폭발 문제가 발생한다.

본 논문에서는 상태폭발 문제가 발생하는 원인을 지적하고, 이 문제를 해결하기 위해 새로운 상태 공간 순회 기법을 제안한다. 제시되는 순회 기법은 on-the-fly 방식으로 상태 공간을 합성하지 않고, 방문 기록도 필요한 부분만 유지하는 방법으로 상태 폭발을 피하게 된다. 그리고 실험 결과를 통해 제안된 기법이 일반적인 기법보다 시간적, 공간적으로 더 효율적임을 보인다.

논문의 구성은 다음과 같다. 먼저 2장에서는 이와 연관된 기존의 연구에 대하여 소개한다. 이어서 3장에서는 기존의 유한상태머신을 순회하는데 나타날 수 있는 문제점과 그 원인을 설명한다. 4장에서는 이 문제를 해결할 수 있는 새로운 기법을 제안한다. 5장에서는 실험 결과를 제시하고 이를 통해서 제안된 기법의 효율성을 보인다. 마지막으로 6장에서는 결론을 맺고 향후 연구 방향에 대하여 소개한다.

2. 관련연구

상태폭발 문제는 상태의 수가 기하급수적으로 늘어나서 상태 공간을 저장할 공간이 모자라는 경우나 상태 공간을 저장할 공간이 있더라도 상태 공간을 순회하는데 너무 오랜 시간이 걸리는 경우를 말한다. 이 문제를 해결하기 위해서 지금까지 다양한 연구가 이루어져왔다.

SPIN[1]은 모든 상태를 나열하고 순회하는 모델 체킹 도구이다. 이 도구에서는 상태들의 저장공간을 줄이기 위해 상태를 축약해서 표현하거나 방문한 상태를 비트열에 기록하는 비트-상태 해싱 기법을 이용한다. 하지만 이 방법은 메모리를 절약할 수는 있으나 상태 공간을 순회하는 시간은 줄이지 못한다.

λ-트랜지션 추상화 방법[4]과 부분 순서 축약 방법[5]은 여러 개의 트랜지션을 하나로 합성하는 방식으로 상태 공간을 줄이는 방식이다. 이러한 방법들은 상태 공간의 크기나 순회 시간을 줄일 수 있지만, 시스템이 가질 수 있는 모든 상태를 검사하는 방법은 아니다.

on-the-fly 모델 체킹[6]은 여러 상태 공간을 병행적으로 다룰 때, 상태 공간을 합성하지 않고 필요할 때 다음 상태를 계산해서 사용하는 방식이다. 이 방식은 상태 공간을 합성하여 이용하지 않으므로 합성된 상태 공간을 저장할 메모리를 아낄 수 있다. 하지만 상태 공간을 순회하는 시간은 줄이지는 못한다. 이 순회 시간을 줄이기 위해서 on-the-fly 방식에 부분 순서 축약 방법을 함께 사용하는 방법[7]도 연구되었다. 하지만 부분 순서 축약 방법으로 수행 시간을 줄이는 접근 방식은 시스템이 가질 수 있는 모든 상태를 검사할 수 없는 문제가 있다.

본 논문에서는 시스템의 전체 상태를 순회하면서 저장 공간과 시간을 모두 줄일 수 있는 방법을 소개한다. 그러기 위해서 기존의 연구들과는 달리 상태의 수를 줄이지 않으면서 필요한 저장 공간을 줄이기 위해 on-the-fly 방식을 이용하고, 상태 공간을 순회하는데 걸리는 시간을 줄일 수 있는 새로운 기법을 제시한다.

3. 유한상태머신의 문제점

유한상태머신은 현재의 상태에서 수행될 수 있는 트랜지션을 통해 다음 상태로 천이하는 방법으로 시스템의 행동을 기술하는 정형언어이다. 아래는 유한상태머신의 정의이다.

정의 1. 유한상태머신

유한상태머신 FSM은 다음과 같이 정의된다.

FSM = (S, s_0, δ, F)

S: 상태들의 집합

s_0: 초기 상태

δ: 트랜지션의 집합

F: 종료 상태의 집합

δ에 속한 하나의 트랜지션 t와 이에 관계된 함수들은 다음과 같이 정의된다.

$t = (name, s_s \rightarrow s_d),\ s_s \in S,\ s_d \in S$

name(t) = name

source(t) = s_s

destination(t) = s_t

유한상태머신이 합성될 때 나타나는 상태폭발 문제의 원인과 큰 상태공간을 순회할 때 나타나는 상태폭발 문제의 원인은 서로 다른 유형을 가진다. 이 장에서는 유한상태머신을 다루면서 나타날 수 있는 상태폭발 문제의 원인에 대해 설명한다.

3.1 유한상태머신의 합성과 문제점

시스템이 크고 복잡해지면 그 행동을 하나의 유한상태머신으로 나타내기 보다는 여러 개의 서브 시스템을 병행적으로 나타내는 것이 더 자연스럽다. 이 때 전체 시스템의 상태는 각각의 유한상태머신이 가진 상태들의 조합으로 나타낼 수 있다. 조합에 사용된 부분 상태들은

자신이 가진 트랜지션을 병행적으로 수행할 수 있으므로, 조합된 상태는 부분 상태들이 가지는 모든 트랜지션을 가지게 된다. 여러 유한상태머신에서 같은 이름을 사용하는 트랜지션들의 경우에는 이를 동시에 수행해야 하는 합성적 트랜지션이라고 가정한다. 그러므로 어떤 조합 상태에서 합성적 트랜지션이 수행되기 위해서는 같은 이름을 가진 트랜지션을 수행할 수 있는 상태들이 모두 포함되어야 한다. 따라서 여러 유한상태머신들은 다음과 같이 병행적으로 합성하여 하나의 유한상태머신으로 나타낼 수 있다.

정의 2. 유한상태머신의 병행적 합성

유한상태머신 $FSM_1=(S_1, s_{10}, \delta_1, F_1)$, $FSM_2=(S_2, s_{20}, \delta_2, F_2)$, ..., $FSM_n=(S_n, s_{n0}, \delta_n, F_n)$을 합성한 유한상태머신 FSM_r은 다음과 같이 나타낼 수 있다.

$$FSM_r = (S_r, s_{r0}, \delta_r, F_r)$$

$S_r = S_1 \times S_2 \times \ldots \times S_n$

$s_{r0} = (s_{10}, s_{20}, \ldots, s_{n0})$

$\delta_r = \{t \mid t = (name, (s_{s1}, s_{s2}, \ldots, s_{sn}) \rightarrow (s_{d1}, s_{d2}, \ldots, s_{dn}))$

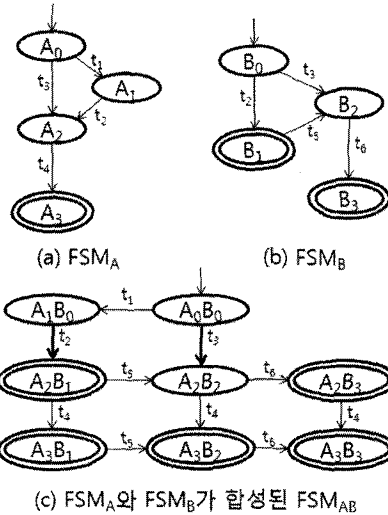
where $\forall u,\ u \in \delta_k,\ 1 \le k \le n,\ name(t)=name(u)$

$(source(u)=s_{sk}$ and $destination(u)=s_{dk})\ \}$

$Fn = \{(s_1, s_2, \cdots, s_n) \mid \vee_{k=1 \text{ to } n}\ s_k \in F_1 \cup F_2 \cup \cdots \cup F_n\}$

그림 1은 유한상태머신 합성의 간단한 예를 보이고 있다. 그림 1(a)와 (b)는 각각 유한상태머신 FSM_A와 FSM_B를 나타내고 있다. 다이어그램에서 타원은 상태를 나타내고 화살표는 그 이름과 함께 트랜지션을 나타낸다. 상태 중에서도 시작점이 없는 화살표를 가진 것은 초기 상태를 나타내고, 두개의 타원을 겹쳐서 나타낸 상태는 종료 상태를 나타낸다. 그림 1(c)는 이 유한상태머신들을 병행적 합성 정의에 따라 합친 FSM_{AB}를 나타낸다. FSM_{AB}의 각 상태는 FSM_A와 FSM_B가 가진 상태의 조합으로 나타내고, 그 상태들은 합성 되기 전의 상태들이 가진 모든 트랜지션을 가지고 있다. 여기에서 굵은 화살표로 표시된 t_2와 t_3는 FSM_A와 FSM_B에서 같은 이름으로 나타난 트랜지션이다. 그래서 FSM_A와 FSM_B에서 모두 나타나는 트랜지션이지만 FSM_{AB}에서 하나의 합성적 트랜지션으로 표현되는 것을 볼 수 있다. 또한 병행적 합성의 정의에 따라 하나 이상의 종료 상태가 포함되어 조합된 상태는 종료 상태가 됨을 볼 수 있다.

여러 개의 유한상태머신이 병행적으로 합성되어 하나의 유한상태머신이 만들어질 때, 합성된 유한상태머신이 가질 수 있는 상태의 최대수는 합성되기 전의 유한상태머신들이 가진 상태수를 모두 곱한 것과 같다. 예를 들어 상태의 수가 N개인 유한상태머신 M개를 병행적으로 합성하면, 합성된 유한상태머신이 가지는 상태의 수는 최대 N^M개가 된다. 이처럼 합성되는 유한상태머신이 많아질수록 합성된 결과가 기하급수적으로 커지게 되고, 결국에는 이를 저장할 공간이 모자라게 되는 상태폭발 문제가 발생한다.



(a) FSM_A (b) FSM_B

(c) FSM_A와 FSM_B가 합성된 FSM_{AB}

그림 1 유한상태머신의 병행적 합성의 예

3.2 유한상태머신의 순회와 문제점

유한상태머신의 상태 공간 전체를 순회하는 문제는 그래프 탐색 문제와 동일하다. 따라서 유한상태머신 역시 깊이 우선 탐색 방식과 너비 우선 탐색 방식을 적용할 수 있다. 그래프 탐색 방법들은 공통적으로 현재 상태에서 천이될 수 있는 다음 상태를 계산해야 하는데, 이를 후임 함수(successor function) *succ*(s)로 나타낸다. 만약 정의 1과 같은 하나의 유한상태머신을 대상으로 한다면 *succ*(s)는 다음과 같이 정의된다.

정의 3. 단일 유한상태머신에서의 후임 함수

$succ(s)=\{s_n \mid s_n \in S$ where $\forall t \in \delta,\ source(t)= s$

$(destination(t) = s_n)\}$

그림 2는 후임 함수를 이용하여 깊이 우선 탐색 알고리즘을 재귀적으로 표현한 것이다. 여기에서 V는 이미 방문한 상태를 저장하고, 현재 상태를 이미 방문했었는지를 알아보기 위해 사용하는 집합이다.

```
Function DFS(s : state)
    if(s is already added in V)
        return
    /* do something here with s */
    add s at V
    next = succ(s)
    for all s_n in next
```

그림 2 깊이 우선 탐색 알고리즘

이러한 유한상태머신 순회 알고리즘은 병행적으로 합성된 유한상태머신에서도 사용할 수 있다. 차이점은 합성된 유한상태머신의 상태가 합성전의 유한상태머신들의 상태인 부분 상태들의 조합으로 표현된다는 것과 합성적 트랜지션 수행 시 여러 개의 부분 상태들이 한번에 변할 수 있다 것이다. 합성된 유한상태머신의 이러한 특성을 고려하면 다음과 같은 후임 함수를 정의할 수 있다.

정의 4. 합성된 유한상태머신의 후임 함수

$succ((s_{s1}, s_{s2}, \ldots, s_{sn}))=\{(s_{d1}, s_{d2}, \ldots, s_{dn}) \mid$
$(s_{d1}, s_{d2}, \ldots, s_{dn}) \in S_r$
where $\forall t \in \delta_r$, $source(t)=(s_{s1}, s_{s2}, \cdots, s_{sn})$
$(destination(t)=(s_{d1}, s_{d2}, \cdots, s_{dn}))\}$

유한상태머신의 순회에서는 방문 기록 집합에 새로운 상태를 방문할 때마다 그 상태를 저장하고, 다음에 다시 그 상태에 도달했을 때 이미 방문했던 상태인지를 검사하게 된다. 그렇기 때문에 방문 기록 집합은 순회하는 상태 공간이 커지면 기록하는 상태의 수도 많아진다. 3.1절에서 다룬 것과 같이 여러 개의 유한상태머신을 합성하게 되면, 합성된 유한상태머신은 그 상태 수가 기하급수적으로 늘어나게 된다. 이 상태 공간을 저장할 수 있다고 하더라도, 방문 상태 집합도 기하급수적으로 커지면서 현재 상태가 방문 상태 집합에 존재하는지 검색하는 시간도 길어지게 된다. 예를 들어 상태 공간의 크기가 N^M인 경우에 모든 상태를 한번씩만 방문한다고 가정하면 검색 시간은 $O(N^M(O(1)+O(N^M))/2)=O(N^{2M})$이 된다. 이처럼 상태 공간을 순회하는데 시간이 기하급수적으로 길어지는 상태폭발 문제가 발생하게 된다.

4. 부분 방문 기록을 이용한 병행 시스템의 상태 공간 순회 기법

상태폭발 문제는 상태 공간이 너무 커서 이를 저장할 수 없는 문제와 상태 공간을 저장할 수는 있지만 이를 순회하는데 걸리는 시간이 너무 길어지는 문제로 분류할 수 있다. 이제부터 전자의 경우를 공간적 상태폭발이라 칭하고, 후자의 경우를 시간적 상태폭발이라 칭한다.

공간적 상태폭발은 3.1절에서 살펴본 것과 같이 유한상태머신을 합성할 때 발생한다. 따라서 이 문제는 유한상태머신을 합성하지 않고 병행적으로 순회하는 on-the-fly 방식을 적용함으로써 해결할 수 있다. 이 방식을 이용하기 위해서는 유한상태머신들을 합성하지 않고도 합성된 유한상태머신의 후임 함수와 같은 출력을 갖은 후임 함수가 정의되어야 한다. 이러한 후임 함수는 정의 2와 정의 4를 이용해 쉽게 정의할 수 있다. 병행적으로 실행되는 유한상태머신의 전체 상태는 병행적 합성에서와 같이 각 부분 상태의 조합으로 표현되고, 그 상태는 각 부분 상태의 모든 트랜지션을 가지게 된다. 여기에서 합성적 트랜지션은 병행적 합성 때와 마찬가지로 하나의 트랜지션으로 취급된다. 아래는 이를 나타낸 병행적 유한상태머신의 후임 함수의 정의이다.

정의 5. 병행적 유한상태머신의 후임 함수

유한상태머신 $FSM_1=(S_1, s_{10}, \delta_1, F_1)$, $FSM_2=(S_2, s_{20}, \delta_2, F_2)$, …, $FSM_n=(S_n, s_{n0}, \delta_n, F_n)$이 병행적으로 실행되는 경우 후임 함수는 다음과 같다.

$succ((s_{s1}, s_{s2}, \ldots, s_{sn}))=\{(s_{d1}, s_{d2}, \ldots, s_{dn}) \mid$
$(s_{d1}, s_{d2}, \ldots, s_{dn}) \in S_1 \times S_2 \times \ldots \times S_n$
where $\forall t \in \delta_k$, $1 \le k \le n$, $source(t) = s_{sk}$,
$\forall u \in \delta_j$, $1 \le j \le n$, $name(t)=name(u)$
$(source(u)= s_{sj}$ and $destination(u)= s_{dj})\}$

시간적 상태폭발은 3.2절에서 살펴본 것과 같이 현재 방문중인 상태가 방문 상태를 기록하는 집합 V에 존재하는지 검사할 때, V에 속한 상태가 기하급수적으로 늘어나면서 발생한다. 따라서 이 문제를 해결하기 위해서는 V가 가진 상태의 수를 최소로 줄이는 방법이 필요하다.

상태 공간을 순회할 때는 위에서 정의한 후임 함수를 이용해서 현재 상태가 가진 트랜지션을 따라 다음 상태로 천이해가게 된다. 이때 다음 상태는 그 상태로 천이할 수 있는 트랜지션 개수만큼 방문을 시도하고, 그 상태를 트랜지션의 개수만큼 방문한 뒤에는 더 이상 방문하지 않는다. 그림 1(c)를 예로 들면, A_1B_0나A_2B_1, A_3B_1의 경우에는 한번만 방문을 시도하게 되고, 그 후에는 방문을 시도하지 않는다. A_2B_2의 경우에는 A_0B_0나 A_2B_1에서 각각 한번씩 방문을 시도하게 되고, 그 후에는 방문을 시도하지 않는다. 이를 이용하여 더 이상 방문을 시도하지 않을 상태를 방문 기록에서 지운다면, 방문 기록의 일부만 가지고도 상태 공간을 순회할 수 있게 된다. 이렇게 상태 공간 순회에 필요한 상태만 기록하고 있는 방문 기록을 부분 방문 기록이라 부르고, 이는 현재 상태가 있는지 검색하는 시간을 단축시킬 수 있다.

이러한 방법을 적용하기 위해서 상태를 입력으로 받고, 그 상태를 목적지로 하는 트랜지션의 개수를 출력하는 집계 함수(counting function) *counting*(s)을 도입한다. 다음은 집계 함수의 정의이다.

정의 6. 집계 함수

$counting(s) = |\{t \mid t \in \delta_r \text{ where } destination(t) = s\}|$

그림 3은 부분 방문 기록 기법을 적용한 새로운 깊이 우선 탐색 알고리즘이다. 이 알고리즘은 그림 2의 알고리즘에서 방문 기록을 이용한 검색 부분이 변경되었다. 변경 전의 알고리즘에서는 집합 V에 상태가 있는지 확인하기만 한다. 하지만 변경된 알고리즘에서는 집합 V에 상태가 있는지 확인할 뿐만 아니라 방문을 시도한 횟수가 집계 함수의 결과와 같다면 그 상태에는 더 이

```
Function partial_history_DFS(s : state)
    if(s is already added in V)
        increase count of record in V
        if(count of record == counting(s))
            remove record at V
        return
    /* do something here with s */
    add (s, count = 1) at V
    next = succ(s)
    for all sn in next
```

그림 3 부분 방문 상태 기록을 이용한 깊이 우선 탐색 알고리즘

상 방문을 시도하지 않을 것이므로 집합 V에서 제거한다. 이를 수행하기 위해서 집합 V에는 상태와 그 상태에 방문을 시도한 횟수도 함께 기록해야 한다.

여러 개의 유한상태머신을 병행적으로 순회하는 경우에는 트랜지션의 수가 상태의 수처럼 기하급수적으로 증가한다. 따라서 정의 6의 집계 함수를 사용하면 모든 트랜지션을 대상으로 계산해야 하기 때문에 계산에 많은 시간이 소요된다. 이러한 성능상의 이유로 엄격한 집계 함수 보다 좀더 느슨한 집계 함수를 사용할 필요가 있다. 그래서 본 논문에서는 느슨한 집계 함수를 제안한다. 만약 어떤 상태에서 느슨한 집계 함수의 결과가 엄격한 집계 함수의 결과보다 작다면 방문 기록에 이 상태를 방문 했다는 기록이 삭제된 후에 다시 방문할 수 있으므로 방문했던 상태를 반복해서 계속 방문하는 무한 루프에 빠질 수 있다. 하지만 본 논문에서 제시하는 느슨한 집계 함수는 각 부분 상태만을 고려하고 다른 부분 상태와의 관계를 고려하지 않기 때문에 그 결과가 엄격한 집계 함수의 결과보다 크거나 같아서 이러한 문제가 발생하지 않는다. 반대로 느슨한 집계 함수의 결과가 엄격한 집계 함수의 결과보다 큰 상태의 경우에는 순회가 종료될 때까지 방문 기록에 남아있게 된다. 하지만 실제로 이렇게 계산되는 상태의 수가 미치는 영향이 매우 적다는 것을 다음 장의 실험 결과를 통해 볼 수 있다. 다음은 본 논문에서 사용한 느슨한 집계 함수의 정의이다.

정의 7. 느슨한 집계 함수

$counting((s_1, s_2, \ldots, s_n)) = |\{t \mid t \in \delta_k, 1 \le k \le n \text{ where } destination(t) = s_k\}|$

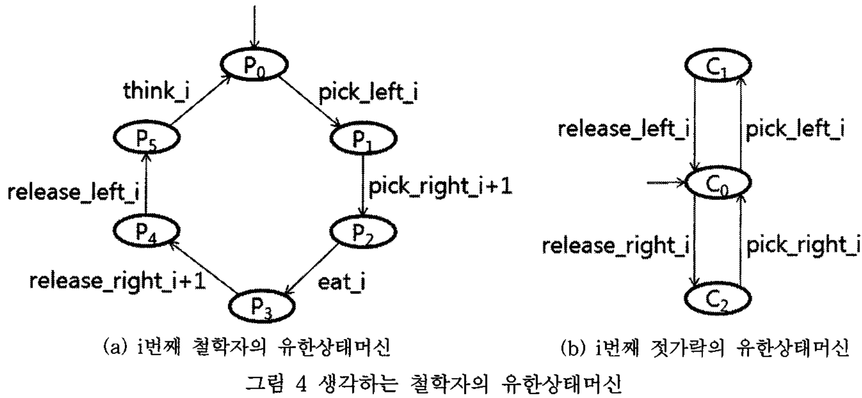
제시된 부분 방문 기록 기법은 방문 기록에서 필요 없는 정보를 삭제함으로써 상태 공간을 순회하는데 필요한 검색 시간을 줄인다. 이 때, 줄어드는 방문 기록의 크기는 순회하려는 상태 공간의 형태와 순회에 사용하는 알고리즘에 따라 조금씩 달라질 수 있지만 부분 방문 기록의 최대 크기가 상태 공간의 크기의 1/k이라면, 3.2절에서의 예와 같이 상태의 수가 N^M인 상태 공간을 순회하는데 필요한 검색 시간은 $O(N^{2M})$ 보다 작은 $O(N^M(O(1)+O(N^{M/k}))/2)=O(N^{2M/k})$가 된다. 그러므로 부분 방문 기록 기법을 적용하면 상태 공간 순회가 더 빨라 질 수 있다.

5. 기존 방식과의 효율성 비교 실험

이 장에서는 기존의 on-the-fly 상태 공간 순회 방식과 본 논문에서 부분 방문 기록 기법을 적용한 상태 공간 순회 방식의 효율성을 비교하는 실험 결과를 보인다. 먼저 부분 방문 기록 기법을 적용한 그림 3의 알고리즘을 이용하여 총 상태의 수와 순회 중 부분 방문 기록의 최대 크기를 비교하여 상태 공간 순회 과정에서 저장이 필요한 상태의 수가 얼마나 차이가 나는지 살펴본다. 그리고 부분 방문 기록 기법을 적용하지 않고 on-the-fly 방식을 이용하여 그림 2의 알고리즘으로 상태 공간을 순회한 시간과 on-the-fly 방식과 부분 방문 기록 기법을 모두 이용한 그림 3의 알고리즘으로 상태 공간을 순회한 시간을 비교하여 상태 공간 순회에 부분 방문 기록 기법이 얼마나 시간을 단축시키는지 살펴본다.

그림 4는 생각하는 철학자 문제를 유한상태머신으로 나타낸 모델이다. i번째 철학자 모델에서는 초기 상태 P_0에서 트랜지션 pick_left_i를 수행하여 P_1상태로 천이할 수 있다. 이 트랜지션 pick_left_i는 i번째 젓가락 모델에서 C_0상태에서 C_1상태로 천이하기 위한 트랜지션 pick_left_i와 이름이 같은 합성적 트랜지션이다. 그러므로 이 두 개의 트랜지션은 동시에 수행되어야 한다. 마찬가지로 i번째 철학자의 트랜지션 pick_right_i+1은 i+1번째 젓가락 모델의 트랜지션 pick_right_i와 같은 이름을 가지는 합성적 트랜지션이다. 따라서 i번째 철학자 모델의 상태가 P_1에서 P_2로 천이할 때 i+1번째 젓가락 모델의 상태도 함께 C_0에서 C_2로 천이한다. i번째 철학자의 상태가 P_0일 때 P_{i0}, i번째 젓가락의 상태가 C_0일 때 C_{i0}라 표기하면 3명이 참여한 생각하는 철학자 문제의 초기 상태는 $(P_{10}, P_{20}, P_{30}, C_{10}, C_{20}, C_{30})$로 나타낼 수 있다.

이 예제에서 엄격한 집계 함수와 느슨한 집계 함수의 차이를 볼 수 있다. 상태 $(P_{14}, P_{24}, P_{35}, C_{11}, C_{21}, C_{30})$의 경우 엄격한 집계 함수의 결과는 1로 상태 $(P_{14}, P_{23}, P_{35}, C_{11}, C_{21}, C_{31})$만이 계산된다. 하지만 느슨한 집계 함수의 결과는 2로 상태$(P_{14}, P_{24}, P_{34}, C_{11}, C_{21}, C_{31})$도 계산된다. 하지만 후자의 경우에는 실제로는 도달하지 못한 상태이다. 이 상태가 계산되는 이유는 느슨한 집계 함수가 부분 상태 P_{35}의 이전 상태로 P_{34}가 있다는 것만 고려하고 다른 부분 상태를 고려하지 않아 이 상태에 도달할 수 없다는 것을 판단하지 못하기 때문이다. 이처



(a) i번째 철학자의 유한상태머신 (b) i번째 젓가락의 유한상태머신

그림 4 생각하는 철학자의 유한상태머신

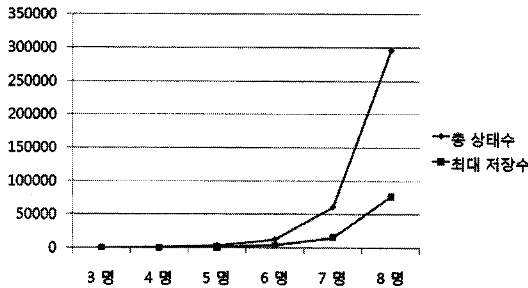


그림 5 상태의 총 개수와 방문 기록의 최대 크기

럼 느슨한 집계 함수는 엄격한 집계 함수보다 큰 값을 계산하는 경우가 발생할 수 있다.

그림 5는 생각하는 철학자 모델 순회에 느슨한 집계 함수를 사용했을 때 순회한 상태의 총 개수와 순회 중 방문 기록의 최대 크기를 비교한 그래프이다. 철학자의 수가 적을 때는 큰 차이를 보이지 않다가, 철학자의 수가 늘어나면 상태의 수가 폭발적으로 증가하면서 그 차이가 점점 더 커지는 것을 볼 수 있다. 기존의 상태 공간 순회에서는 방문했던 모든 상태를 저장해야 하기 때문에 검색 대상이 기하급수적으로 증가했지만, 부분 방문 기록을 이용하는 경우 검색 대상을 확연히 줄일 수 있다. 게다가 기존의 방법은 합성한 상태 공간을 순회하기 위해서 그 상태 공간 전체를 생성하고 저장해두어야 하지만, 제시된 기법은 필요할 때에만 다음 상태를 계산하기 때문에 합성한 상태 공간을 저장할 공간이 필요하지 않다. 따라서 제시된 기법은 공간적 상태 폭발을 크게 완화시킬 수 있음을 알 수 있다.

표 1은 생각하는 철학자 모델을 on-the-fly 방식을 적용한 기존의 깊이 우선 탐색 방식으로 순회하는데 걸린 시간과 부분 방문 기록 기법을 이용한 깊이 우선 탐색 방식으로 순회하는데 걸린 시간을 비교한 것이다. 가장 왼쪽 열은 생각하는 철학자 모델에 참여한 철학자 모델의 수를 나타내고, 다음 두 개의 열은 각각 기존 방

표 1 기존 방식과 제시된 기법의 수행시간 비교

n	기존 방식 (sec)	제시된 기법 (sec)	제시된 기법 / 기존 방식
3	0.062	0.016	0.258
4	2.059	0.405	0.196
5	115.799	14.305	0.123
6	9490.246	542.655	0.057
7	-	18416.917	-

식과 제안된 기법을 사용했을 때 순회에 걸리는 시간을 초로 나타낸다. 오른쪽 마지막 열은 제시된 기법과 기존 방식의 비율을 나타냈다. 이 값이 1보다 작을수록 제시된 기법의 효율이 좋다는 의미이고, 1보다 클수록 효율이 나쁘다는 의미가 된다.

표 1의 내용을 살펴보면 철학자 모델의 수가 증가할수록 순회에 걸리는 시간이 기하급수적으로 늘어남을 알 수 있다. 제시된 기법 역시 순회에 걸리는 시간이 기하급수적으로 늘어나지만 두 순회 시간의 비율은 철학자 모델의 수가 증가할수록 오히려 줄어드는 것을 볼 수 있다. 따라서 제시된 기법은 시간적 상태 폭발도 크게 완화시키는 것을 알 수 있다.

6. 결 론

본 논문에서는 여러 개의 유한상태머신을 병행적으로 다룰 때 나타나는 상태 폭발 문제의 원인을 지적하고 이를 극복하는 새로운 상태 공간 순회 기법을 제시했다. 여러 개의 유한상태머신은 병행적 합성으로 하나의 유한상태머신으로 만들 때 상태의 수가 폭발적으로 늘어나는 공간적 상태폭발 문제가 발생하는데, 이 문제는 on-the-fly 방식을 이용해 유한상태머신을 합성하지 않고 병행적으로 다루어 해결할 수 있다. 하지만 상태의 수가 기하급수적으로 늘어나면 방문 기록에서 특정 상태를 방문 했었는지 검색하는 시간도 기하급수적으로 늘어나는 시간적 상태폭발 문제도 발생한다. 본 논문에

서는 시간적 상태폭발 문제를 완화하기 위해서 더 이상 방문을 시도하지 않는 상태는 방문 기록에서 삭제하는 부분 방문 기록 기법을 제시했다. 그리고 제시된 기법에서 사용하는 부분 방문 기록의 최대 크기가 기존의 방문 기록보다 훨씬 작다는 것과 상태 공간 순회 시간이 크게 줄어든다는 것을 실험을 통해 보였다.

이 기법은 기존의 상태 공간 순회 방식을 크게 변경하지 않으므로 큰 문제없이 병렬적 상태 공간 순회에서도 다룰 수 있을 것으로 보고 있다. 향후 연구에서는 이 기법을 병렬적 상태 공간 순회에 적용하고, 다른 병렬적 상태 공간 순회 방식과 비교할 수 있을 것으로 기대한다.

참 고 문 헌

[1] G.J. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, vol.23, Issue 5, pp.279-275, May 1997.

[2] OMG, UML 2.0, http://www.omg.org/spec/UML/2.2/Superstructure

[3] W. Reisig, *Petri Nets : An Introduction*, Springer-Verlag, 1985.

[4] P.J. Denning, et al., *Machines, Languages, and Computation*, Prentice Hall, 1978.

[5] A. Valmari, "Stubborn sets for reduced state space generation," *Proceedings of Advanced in Petri nets*, pp.491-515, 1990.

[6] C. Courcoubetis, M. Vardi, P. Woloper, M. Yannakakis, "Memory-Efficient Algorithms for the Verification of Temporal Properties," Lecture Notes in Computer Science, vol.531, pp.233-242, 1991.

[7] D. Pelde, "Combining partial order reductions with on-the-fly model-checking," *Formal Methods in System Design*, vol.8, no.1, pp.39-64, Jan. 1996.

이 정 선

2008년 경북대학교 전자전기컴퓨터학부 졸업(학사). 2010년 경북대학교 전자전기컴퓨터학부 졸업(석사). 관심분야는 임베디드 실시간 시스템 모델링 및 분석, 모델 기반 개발 방법론 등

최 윤 자

1991년 연세대학교 수학과(이학사). 1993년 연세대학교 수학과(이학석사). 1993년~1996년 삼성데이터시스템. 1999년 미네소타대학 전산과(이학석사). 2003년 미네소타대학 전산과(박사). 2003년~2006년 프라운호퍼연구소 연구원. 2006년~2008년 경북대학교 전자전기컴퓨터학부 전임강사. 2008년~현재 경북대학교 IT대학 컴퓨터학부 조교수. 관심분야는 소프트웨어 분석, 모델기반개발방법론

이 우 진

1992년 경북대학교 컴퓨터과학과(학사). 1994년 한국과학기술원 전산학과(공학석사). 1999년 한국과학기술원 전산학과(공학박사). 1999년~2002년 한국전자통신연구원 S/W공학연구부 선임연구원. 2002년~현재 경북대학교 IT대학 컴퓨터학부 부교수. 관심분야는 임베디드 실시간 시스템 모델링 및 분석, 요구 공학, Petri nets, 컴포넌트 개발 기술 등