동적 시스템 명세를 위한 시제논리언어와 그 검증
(A Temporal Logic for Specification of Dynamic Systems and Its Verification)

조 승 모 † 김 형 호 †† 차 성 덕 ††† 배 두 환 †††
(Seung Mo Cho) (Hyung Ho Kim) (Sung Deok Cha) (Doo Hwan Bae)

요 약 대부분의 객체지향 시스템을 비롯한, 복잡한 시스템들은 그 구성요소들이 시스템의 수행시간 중에 변하는 동적인 특성을 가지고 있다.하지만 대부분의 기존 분석기법들은 이러한 측면을 무시하고 있다. 이 논문에서는 이러한 동적 시스템을 명세하고 분석하기 위한 기법을 제안하고자 한다.

이를 위해, 동적 시스템의 명세를 기술하기 위한 새로운 시제논리인 HDTL을 제안하고, 기존의 시제논리를 위한 분석기법을 수정하여 새로운 tableau 기법을 제안하였다. HDTL은 변수와 한정자를 사용하여 동적 시스템의 자동적 분석을 가능하게 하였다. 이 기법을 사용하여 우리는 시스템의 요구사항 명세를 기술하고, 시스템의 수행이 그 명세를 만족하는지를 살펴 볼 수 있다. 실험을 통해 HDTL과 분석기법의 적용성을 보였다.

키워드 : 정형적 명세, 시제 논리, 동적 시스템, 테스팅

Abstract Many modern complex systems, including most object-oriented systems, have dynamic characteristics that their components are dynamically configured during run-time. However, few analysis techniques are available that consider the dynamic nature of systems explicitly. We propose a specification and analysis method for these dynamic systems.

We design a new temporal logic, called HDTL, to specify the properties of dynamically evolving systems, and tune up the tableau method for this logic. HDTL incorporates variables and quantifiers that enable the automatic analysis. Using HDTL and the analysis method, we can specify the correctness requirements of systems and check whether the system actually agree with the requirements or not. An experiment shows that HDTL is suitable for specifying dynamic properties and the analysis technique works well.

Key words : Formal Specification, Temporal Logic, Dynamic system, Testing

1. 서 론

소프트웨어의 개발 단계에서 시스템의 정형적 명세(formal specification)의 가치에 대해서는 이제 폭 넓은 합의가 이루어져 있다고 할 수 있다[1]. 개발 단계의 초기에서, 이것은 중요한 요구사항을 명확히 나타냄으로써 불분명함으로 인해 생길 수 있는 추후의 문제를 예방할 수 있다. 이렇게 만들어진 명세는 그 자체로서도 의미를 가지지만, 그것을 가지고 검증(verification or validation)을 수행함으로써 시스템이 정확하게 동작하리라는 데 대한 확신(confidence)을 높여줄 수 있다는 더 큰 장점을 가진다.

이러한 시스템의 명세를 위해 사용되는 명세 언어들은 크게 두 가지 부류로 나누어 질 수 있다. 하나는 시스템의 기능을 명세하는 것이고, 다른 하나는 시스템의 특성을 명세하는 것이다[2]. 시스템의 기능을 명세하는 언어들은 많은 종류가 있고, 그 대다수는 오토마타[3], 페트리넷[4], 프로세스 대수[5] 등의 의미모델에 기반한 것들이다. 이들은 시스템이 시간의 흐름에 따라 어떤 식으로 변해가는지를 나타내는 언어이다.

이와 다르게 시스템의 특성(property)을 명세하는 언

† 비 회 원 : 한국과학기술원 첨단정보기술연구센터
seung@salmosa.kaist.ac.kr
†† 비 회 원 : 한국과학기술원 전산학과
hhkim@salmosa.kaist.ac.kr
††† 종신회원 : 한국과학기술원 전산학과 교수
cha@salmosa.kaist.ac.kr
bae@salmosa.kaist.ac.kr
논문접수 : 2000년 11월 10일
심사완료 : 2002년 4월 23일

어들도 필요하다. 이는 좀더 높은 수준에서 시스템을 명세한다. 위의 기능 명세가 "how"에 해당한다면, 특성 명세는 "what"에 해당하는 경우가 많다. 따라서 특성 명세로는 시스템이 꼭 만족해야 할 특성을 기술하여, 이를 기능 명세, 혹은 구현된 소프트웨어와 비교하여 그 정확성을 검사하는 것이 일반적이다.

병렬 시스템(concurrent systems)이나 반응 시스템(reactive systems) 등의 경우, 특성 명세 언어로서 일반적으로 많이 사용되는 언어는 시제논리(temporal logic)[6]이다. 이는 전통적인 논리에서 사용되는 연산자들에 시간을 다루는 시제 연산자들을 추가한 것으로, 시스템에 관계된 사건이나 상태들이 어떤 순서로 일어나야 하는지를 기술하는데 사용된다. 예를 들어, "엘리베이터의 문이 열리면 그 후에 벨이 울려야 한다"라던가, "학회 참가자는 한번 한 세션에 참가한 후에는, 그 세션이 끝나기 전에는 다른 세션으로 옮길 수 없다" 등등의 성질을 기술할 수 있다.

이러한 시제논리를 복잡한 시스템의 명세에 사용하려 할 때의 문제점 중 하나는, 동적 시스템(dynamic systems)을 명세할 경우이다. 동적 시스템은 시스템을 이루는 구성요소들이 수행시간중에 추가, 삭제 되거나, 그들간의 연결이 변화되는 시스템을 가리킨다. 이러한 시스템은 수행 중의 특정 시점, 혹은 특정 시간 구간에 대해, **그때** 시스템에 속하는 구성요소들의 집합이 어떤 것인지를 **수행 중**(run time)에야 알 수 있다는 특징이 있다. 문제는 우리가 명세를 작성하는 것은 프로그램을 수행하기 전(static time) 이라는 점이다. 따라서 이러한 시스템에 시제논리를 사용하려고 할 경우, 논리의 기본이 되는 행동의 대상이나 주체를 기술할 수 없는 문제가 발생한다. 이는 일반적인 시제논리가 그 기반을 propositional logic에 두고 있기 때문이다.

예를 들어, 간단한 시제논리식인 $\Box(p \rightarrow \Diamond q)$ 와 같은 식은 사건 p가 일어나면 그 후에 사건 q가 일어나야 함을 의미한다. 이때 "사건 p"와 "사건 q"는 시스템의 모든 수행시간에 걸쳐 유일한 의미를 가지는 것으로, '엘리베이터의 문이 열린다'라던가, '경보음이 울린다'던가 하는 특정한 사건을 나타내는 것으로 해석할 수 있다. 일반적인 시제논리에서 이러한 사건의 집합은 유한하며 고정되어 있는 것으로 가정된다.

하지만 동적시스템의 경우, 이러한 가정이 항상 만족되지 않는 경우도 있다. 예를 들어 객체지향 시스템에서 객체 A가 객체 B에게 메시지를 보내는 경우를 생각해 보자. 일반적으로 객체지향 시스템에서의 객체는 클래스를 기반으로 수행시간 중에 동적으로 생성되므로, 시스템의 한 순간에 몇 개의 객체들이 존재하고 있을지, 미리 아는 것은 불가능하다. 메시지의 이름이 같다고 하더라도 메시지를 주고받는 객체가 다르다면, 그때 각 메시지 전달 사건들은 다른 것으로 보아야 한다. 따라서 $Send_{A,B}$와 같은 식으로, 각각의 사건에 별개의 이름을 주어야 한다. 그런데 위에서 본 바와 같이 객체의 집합이 미리 정해져 있지 않으므로, 이러한 사건들의 집합은 고정되어 있지 않고, 무한할 수도 있다. 즉, 수행하기 전에는 이 집합을 알 수 없다.

시제논리 명세는 시스템을 수행하기 이전에, 수행과는 무관하게 정적으로 기술되므로, 논리식을 기술하는데 필요한 기본 단위들이 정적으로 알려져야 한다. 그런데 위와 같은 이유로 해서 동적 시스템에서는 이러한 일이 쉽지 않게 된다.

이를 해결하기 위한 직관적인 방법은 단순한 propositional 시제논리가 아닌 first-order 시제논리를 사용하는 것이다. 일반적인 propositional 시제논리에서는 논리의 기본으로써 propositional logic을 사용했지만, first-order 시제논리는 first-order logic에 기반하게 된다. 따라서 변수(variable)와 한정자(quantifier)들을 사용해서 시스템을 이루는 구성요소들을 기술하게 된다. 하지만, 이 방법은 기술하기에는 좋지만 그것을 이용해서 검증을 수행하는 것, 예를 들면 모델체킹이나 테스팅 등에 사용하기는 힘들다는 문제가 있다. 따라서 이런 분석을 수행할 때는 propositional 시제논리만을 사용하는 경우가 대부분이고, first-order 시제논리를 사용할 경우에는 완전히 자동화된 분석을 제공하지는 못하게 된다[7].

이러한 문제를 해결하기 위해, 우리는 새로운 시제논리인 HDTL (Half-order Dynamic Temporal Logic)을 제안하고자 한다. 이것은 기존의 실시간 시제논리와 관련된 연구에서 제안된 동결한정자 (freeze quantifier)[8]를 사용한 새로운 시제논리이다. 이는 시제논리에 바인딩(binding) 개념을 추가함으로써 자연스럽게 상태공간이 확장되는 것을 나타낼 수 있게 하여, 동적 시스템의 자연스러운 명세를 가능하게 한다.

이렇게 제안된 명세를 이용한 검증 방법으로서 우리는 동적 분석(dynamic analysis) 을 제시하고자 한다. 동적 시스템은 본질적으로 무한한 상태공간을 가지므로, 모델체킹 (model checking) 등의 완전한 정형적 검증 (formal verification) 은 힘들다. 동적 분석은 테스팅과 런타임 모니터링 등을 통칭하는 것으로, 대상 시스템, 혹은 그 모델을 수행시키며 분석하는 것을 말한다. 매 수행마다 다른 동작을 보일 수 있으므로, 이 방법은 시스템의 정확성을 증명하는 것보다 시스템의 오류를 검

출하는 것을 주목적으로 한다. 우리는 제안된 명세언어를 이용해 이 분석을 수행하는 기법을 제시한다. 이는 자동적인 방법으로 시스템을 분석하여, 시스템에 대한 신뢰성을 높이는데 기여할 수 있다.

본 논문의 구성은 다음과 같다. 2절에서는 본 연구의 배경이 되는 연구들에 대해 다루어 본다. 이는 시제논리 등의 명세언어에 대한 부분과 명세기반 테스팅 등을 포함한다. 3절에서는 대상이 되는 동적 시스템을 정의하고 그 안에서 정의된 언어가 어떻게 해석되는지를 보인다. 4절에서는 이러한 의미를 바탕으로 동적 분석을 수행하는 기법에 대해 설명한다. 5절에서는 HDTL을 사용하여 실제 시스템의 특성을 명세하는 실험에 대해 기술한다. 6절에서 결론을 맺고 향후 연구방향에 대해 살펴본다.

2. 배경 연구

2.1 시제 논리와 tableau를 이용한 검증

시스템을 명세할 때는, 그 시스템이 어떤 분류에 속하는가, 혹은 시스템의 어떤 측면에 중점을 두고 명세하려고 하는가, 등의 고려사항에 따라 다른 명세언어를 사용하게 된다. 변형 시스템(transformational systems)의 경우, 시스템의 특성을 명세하기 위해서는 주로 전후조건(pre/post condition)에 기반한 Hoare 논리[9]와, 그 변형들이 주로 사용되어 왔다. 이는 시스템에 들어가는 입력과 나오는 출력이 어떤 관계를 가지는지를 정의하는 것이다. 반면 상호작용 시스템(interactive systems)이나 반응 시스템(reactive systems)과 같은 병렬 시스템의 경우에는 시제논리(temporal logic)을 사용하여 특성을 명세하는 것이 일반적이다. 이는 Pnueli에 의해서 처음 제안된 방법이다[10].

전후조건에 기반한 명세에서는, 한 프로그램의 수행이 명세에 맞는지를 검사하기 위해서 그 프로그램의 입력과 출력이, 명세에서 정의된 predicate을 만족시키는지의 여부를 조사한다. 반면에 병렬 시스템에서는 프로그램의 수행이 입력과 출력으로 정의되는 것이 아니라, 입력과 출력을 포함하는 상태의 순차(sequence)로 정의된다. 이는 상태 순차가 외부와의 상호작용의 기록(history)을 나타내기 때문이다. 따라서 시제논리 명세는 시스템의 상태가 시간에 따라 어떻게 변화되는지를 기술하게 된다.

시제논리는 기본적인 논리식에 사용되는 연산자들(AND, OR, NOT)에 시제논리 연산자를 추가한 형태로 구성된다. 논리식의 기본은 proposition으로, 이는 한 시점에서의 사건의 발생이나 조건의 성립 등을 의미한다. 많이 사용되는 시제논리 연산자에는 ○(next), ◇(eventually), □(always), U(until), P(precedes)[1] 등이 있다.

시제논리로 명세를 기술한 후 검증하기 위해서는 여러 가지 방법이 사용된다. 그 중 많이 이용되는 것은, 시제논리에 해당하는 오토마타를 만드는 것이다. 이때 그 오토마타에 의해서 받아들여지는 상태순차(state sequence)의 집합, 즉 그 오토마타의 언어가 그 시제논리를 만족시키는 순차들이 된다.

예를 들어, 다음과 같은 간단한 시제논리식을 보자.

$$\Box (p \rightarrow \Diamond q)$$

이는 "모든 상태에서, p가 발생하면, 언젠가는 q가 발생한다"는 의미이다. 이 식을 만족하는 상태순차의 집합은, Figure 1의 오토마타의 언어가 된다. 여기서 T0는 초기 상태이고, 두 겹의 원으로 나타난 accept는 최종상태(final state)가 된다. T1은 일단 p가 발생한 상태, T2는 그 후에 q가 발생한 상태를 나타낸다. 따라서 상태가 T2에서 끝나는 경우, 그 상태순차는 명세 □(p→◇q)를 만족시키게 된다.

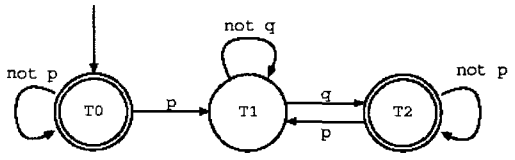


그림 1 LTL식 □(p→◇q)에 대한 오토마타

이렇게 선형 시제 논리에서 오토마타로 변환하는 과정을 간단히 설명하면, 다음과 같다. 우선 검증해야 할 식을 몇 가지 규칙에 따라 변형시킨다. 이 규칙들은 기본적으로, 한 시제논리식은, 현재의 상태에서 검증할 수 있는 식과, 다음 상태로 검증을 미뤄야 할 식, 이렇게 두 가지로 분리될 수 있다는 점을 이용한 것이다.

위와 같은 식이라면, 식의 최 외곽에 □ 연산자가 있으므로 규칙 [r □] 를 적용시킨다. 각 연산자들에 대한 규칙은 표 1과 같다[2][11].

표 1의 의미는 다음과 같다. 이것은 기본적으로, 검증해야 할 복잡한 시제논리식을, 그보다 간단한 식으로 바

1) 이 연산자는 단독으로 유용하다기 보다는 until 연산자의 상대개념(dual)이 필요하기 때문에 정의된다.
¬(p U q)=¬p P q

2) 여기서 ○와 ⊙, U와, P와등을 구별하는 것은 시제연산자의 강약(strong and weak operators)을 구분하기 위함이다. 자세한 사항은 뒤에 각 연산자의 의미(semantics)를 정의할 때 다시 설명한다.

[r ∧] [r ∨] [r →]	$f_1 \wedge f_2 \Rightarrow <\{f_1, f_2\}>$ $f_1 \vee f_2 \Rightarrow <\{f_1\}, \{f_2\}>$ $f_1 \rightarrow f_2 \Rightarrow <\{\neg f_1\}, \{f_2\}>$
[r □] [r ◇]	$\Box f \Rightarrow <\{f, \bigcirc \Box f\}>$ $\Diamond f \Rightarrow <\{f\}, \{\odot \Diamond f\}>$
[r *U*] [r P]	$f_1 U f_2 \Rightarrow <\{f_2\}, \{f_1, \odot(f_1 U f_2)\}>$ $f_1 P f_2 \Rightarrow <\{f_1, \neg f_2\}, \{\neg f_2, \bigcirc(f_1 P f_2)\}>$
[r U] [r *P*]	$f_1 U f_2 \Rightarrow <\{f_2\}, \{f_1, \bigcirc(f_1 U f_2)\}>$ $f_1 P f_2 \Rightarrow <\{f_1, \neg f_2\}, \{\neg f_2, \odot(f_1 P f_2)\}>$

표 1 시제논리의 각 연산자들에 대한 변환규칙

꾸어 가는데 사용된다. 시제논리식의 최외곽에 있는 연산자의 종류에 따라 각각 다른 방법으로 식을 분해해 나가게 된다. 표의 맨 좌측 열은 규칙의 이름을, 그리고 화살표의 왼쪽은 규칙이 적용될 식의 형식을, 오른쪽은 규칙의 적용 결과를 나타낸다. 규칙이 적용된 결과는 식들의 집합(예- $\{f_1, f_2\}$)이나, 식들의 집합의 집합(예 - $<\{f_1\}, \{f_2\}>$)이 된다. 전자의 경우는 집합을 이루는 식들의 AND(conjunction)이란 의미가 되고, 후자의 경우는 집합들의 OR (disjunction of conjuntion) 이란 의미가 된다.

이를 이용해서 시제논리를 검사하는 과정은 다음과 같다. 우선 처음에 검증하려는 시제논리식을 유일한 원소로 가지는 논리식의 집합을 만들고, 이를 규칙에 따라 변환해 가면서 생기는 집합들을 상태로 가지는 오토마타를 만들면 된다. 이때 상태간의 전이는 논리식 중에서 하나의 상태만으로 검증가능한 논리식이 된다. 이렇게 만들어지는 오토마타는 그 크기가 유한하므로 시제논리식이 주어지면 알고리즘적으로 생성가능하다[11,12].

이렇게 오토마타가 만들어지면 이것을 이용해서 검증이 수행된다. 검증 대상 시스템의 동작모델이 되는 오토마타를 A_1, 시제논리식에서 생성된 오토마타를 A_2라고 하면 모델체킹은 다음과 같은 식을 증명하는 것으로 정의될 수 있다. (L(A)는 오토마타 A의 언어.)

$$L(A_1) \cap L(\neg A_2) = \phi$$

2.2 실시간 시제논리와 동결 한정자

시제논리를 크게 둘로 나누면 실시간 시제논리와, 그 외의 고전적 시제논리로 구분될 수 있다. 우리가 제안하는 HDTL은 실시간 시제논리는 아니다. 하지만, HDTL은 실시간 시제논리에 대한 연구에서 많은 영향을 받았다. 이 절에서는 그 점에 관해 살펴본다.

실시간 시제논리는 시간을 정량적으로 나타내기 위한 시제논리의 확장이다. 이런 확장을 통해 우리는 "모든 요구 p는 5 단위시간 안에 답변 q를 받는다"와 같은 특성을 명세할 수 있게 된다. 여러 가지 실시간 시제논리에서는 주로 시간을 변수로 설정하여, 그 변수에 대한 first-order 수식으로 시제논리식을 정의한다. 위에서 든 것을 이런 실시간 시제논리로 기술하면 다음과 같은 식이 된다. 이때 변수 T는 현재 시간을 나타내는 변수이다.

$$\Box\ \forall x.((p \wedge x=T) \rightarrow \Diamond(q \wedge T \leq x+5))$$

Henzinger와 Alur는 이러한 first-order 논리의 도입이 일반적으로 지나치게 높은 복잡도를 야기하여 이해나 검증에 도움이 되지 못한다고 주장했다[8]. 실제로, first-order 실시간 논리들을 사용하는 기존의 검증 방법론들은 많은 부분 사람의 개입을 필요로 해서, propositional 시제 논리들이 제공하고 있는 것 같은 높은 수준의 자동화된 분석 방법을 제공하지 못하고 있다.

그들은 대안으로 동결한정자(freeze quantifier)를 이용하여 시간 변수에 대한 제한적인 접근만을 허용하는 실시간 시제논리인 TPTL(Timed Propositional Temporal Logic)을 제안했다. 하나의 동결연산자 "x."는 변수 x를 현재의 상태의 시간에 바인딩시킨다. 이를 이용하면 위의 성질은 다음과 같이 명세할 수 있다.

$$\Box\ x.(p \rightarrow \Diamond\ y.(q \wedge y \leq x+5))$$

(이것은 "요구 p가 있을 때마다, 변수 x는 그때의 시간으로 바인딩되고, 그에 해당하는 답변 q는 시간 y에 나오고, 그때 y는 최대 x+5이다"라고 읽는다.)

TPTL의 장점으로 크게 두 가지를 들 수 있다. 하나는 기존의 first-order 실시간 논리가 정형적이고 기계적인 검증에 사용하기 힘들었던 단점을 보안, 모델 체킹을 가능하게 했다는 점이다. 하지만 이 논문의 주제와 관련되어 TPTL이 가지는 중요한 점은, 이러한 분석 방법의 측면이 아니라, 이 언어가 first-order 시제논리와 유사한 표현력을 제공하면서도, 사용하기 쉽고 정형적인 의미를 가지는 표기법을 제공했다는 점이다. 이는 특히 동적 시스템의 명세에서 많은 의미를 가진다고 할 수 있다. 그 특성상, 동적 시스템의 명세에서는 first-order 논리와 유사한 시제논리가 사용되어야 하는데, 이는 실제 검증 시에 지나친 복잡도를 야기하게 된다. 따라서 그보다 단순하면서도 명료한 의미구조를 가진 시제논리로서 TPTL은 의미를 가진다.

3. 시스템 및 언어 정의

3.1 동적 시스템

HDTL은 일반적인 시제논리에 동적 시스템 명세를 위한 기능을 확장한 것이다. 관련 연구에서 살펴보았듯이, 동적 시스템이라 함은 시스템을 이루는 구성요소들의 생성, 삭제가 실행시간중에 일어나는 시스템을 가리킨다. 이러한 시스템은 여러 가지 방식으로 구현 될 수

있다. 여기서는 객체지향 방식으로 구현된 동적 시스템을 주로 염두에 두고 기술하였다. 하지만 제안된 방법은 에이전트 시스템 등으로 구현된 기타 동적 시스템의 명세에도 별다른 수정 없이 쓰일 수 있다.

명세언어를 사용하여 시스템을 기술할 때는 관찰의 수준을 결정하는 일이 먼저 선행되어야 한다. 이는 명세의 추상성 정도를 결정하는 일이다. 본 논문에서 고려된 동적 시스템의 예인 객체지향 시스템의 경우, 객체간의 메시지 전달을 관찰하는 것이 일반적으로 받아들여지고 있는 추상성 수준이다. 따라서 이 논문에서는 객체들간의 메시지 전달에 관심을 집중하기로 한다. 즉 시스템의 구성요소 각각의 내부 상태, 예를 들면 객체들이 가지는 국지적 상태에 대해서는 고려하지 않는다. 또한 일반적인 객체지향 시스템에서와 같이 동기적 메시지 전달의 경우에 대해서만 고려한다.[3)]

이러한 가정하에서 본 논문에서 다루고자 하는 동적 시스템을 다음과 같이 정의하고자 한다.

정의 3.1 (동적 시스템) 하나의 동적 시스템 $S=<M,O,C,conf_0,R>$은 다음과 같이 정의된다.

- M은 시스템 내에서 전송될 수 있는 메시지들의 유한한 집합이다. 여기에는 시스템을 구성하는 객체들의 생성, 삭제를 위한 두개의 특별한 메시지가 포함된다. {create, remove} ⊆ M.
- O은 시스템에 존재할 수 있는 구성요소(객체)들의 무한한 집합이다.
- C는 시스템이 가질 수 있는 형상(configuration)[4)]의 집합이다.
- $conf_0 \subseteq C$는 시스템이 가질 수 있는 초기 형상들의 집합이다.
- $R \subseteq C \times C$은 시스템의 형상의 변화를 나타내는 관계이다.

여기서 우리는 시스템 구성요소 각각의 내부상태에 대해서는 고려하지 않으므로, 형상의 변화는 비결정적(nondeterministic)이다. 따라서 R 은 함수가 아니라 관계(relation)로 정의된다. 매 순간의 시스템의 형상은 다음과 같이 정의할 수 있다.

정의 3.2 (형상) 매 순간에 있어서의 시스템의 형상 $c_i=<O_i,\sigma_i>$는 다음과 같이 정의된다.

- $O_i \subseteq O$은 그 순간에서 시스템을 구성하고 있는 객체들의 집합이다.
- $\sigma_i=<o_1,o_2,m>$는 그때 행해지는 메시지 전달을 나타낸다. o_1은 메시지를 보내는 객체, o_2는 받는 객체, 그리고 $m \in M$은 전달되는 메시지를 가리킨다. 이때 $o_1 \in O_i$이다. o_2의 경우는 다음과 같다.
  - 만일 m = create이면, $o_2 \notin O_i$.
  - 그 외의 경우는 $o_2 \in O_i$.

이렇게 시스템을 정의한다면 시스템이 수행될 때 어떻게 변해가는지를 형상의 순차(sequence)로 정의할 수 있다.

정의 3.3 (형상순차) 하나의 형상순차 $\bar{c}=c_0c_1...c_n$은 형상의 유한한 순차이다. 이때 $c_0 \in conf_0$이고, $0 \le i \le n$에 대해 $(c_i, c_{i+1}) \in R$이다.

그런데 위에서, 우리는 시스템의 변화를 메시지 전송을 기본으로 해서 외부에서 관찰하는 것으로 간주하였다. 이럴 경우, 각 형상이 가지고 있는 두 가지 정보 중에서 시스템의 구성요소에 대한 정보(O_i)는 일반적으로 불필요하고 알기 힘든 것이 된다. 즉 시스템이 만족해야 할 조건을 명세하려고 할 때, 실행시간 중의 시스템의 구성요소들에 대한 자세한 정보는 불필요한 경우가 많다. 따라서 우리는 관심을 제한하여 형상 중에서 메시지에 대한 부분(σ_i)만을 관찰의 대상으로 삼기로 한다. 이 부분을 시스템의 상태(states)라고 명명하고, 형상에서 구성요소 정보 부분을 떼어낸 것으로 정의한다. 즉 하나의 형상 $c_i=<O_i,\sigma_i>$에 대응하는 상태는 σ_i가 된다.

이렇게 정의하면 시스템의 상태순차는 다음과 같이 정의될 수 있다.

정의 3.4. (상태순차) 하나의 형상순차 $\bar{c}=c_0c_1...c_n$가 주어질 때, $c_i=<O_i,\sigma_i>$이면, 이 형상순차에 대응되는 상태순차는 $\sigma_i=\sigma_0\sigma_1...\sigma_n$으로 정의된다.

여기서 $\sigma_i=<o_1,o_2,m>$이고, o_1,o_2가 가질 수 있는 값의 집합이 무한할 수 있으므로 - 즉 시스템의 구성요소가 무한할 수 있으므로 - 이 상태순차에 대한 명세는 기존의 propositional 시제논리로는 기술될 수 없다.

3.2 구문과 의미

전 절에서 우리는 상태순차를, 메시지를 주고 받는 두 객체와 메시지 이름의 튜플의 순차로 정의하였다. 이렇게 상태순차를 정의할 때, 이를 바탕으로 다음과 같이 HDTL의 구문을 정의할 수 있다.

정의 3.5 (구문) HDTL의 term π과 formula φ를 다음과 같이 정의한다.

- π := snd(x) | rcv(y) | msg(x) | m

3) 분산 시스템과 같은 상황에서의 비동기적 메시지 전달을 고려할 때는 아래에서 정의되는 '상태' 개념에, 송신과 수신 여부가 추가되면 된다. 즉 하나의 메시지 전달 사건이 두개의 사건으로 분리된다.

4) 여기서 형상이라는 말은 상태(state)와 유의어이다. 상태라는 말을 뒤에서 다른 의미로 쓰기 때문에 구분하였다

- $\phi := \pi_1=\pi_2 \mid \pi_1\neq \mid$ false $\mid$ true $\mid \phi_1\rightarrow\phi_2 \mid \phi_1\wedge\phi_2$
  $\mid \phi_1\vee\phi_2 \mid \neg\phi \mid \bigcirc\phi \mid \odot\phi \mid \Box\phi \mid \Diamond\phi \mid$
  $\phi_1 U \phi_2 \mid \phi_1 U_{\phi_2} \mid \phi_1 P \phi_2 \mid \phi_1 P_{\phi_2} \mid x.\phi$

이때, x∈V, m∈M이다. (V는 변수들의 집합.)

여기서 변수들은 각 상태에 바인딩 된다. 그리고 그 벡터의 각 요소를 읽는 함수들이 snd(x), rcv(x), msg(x) 이다. (각각 송신객체, 수신객체, 메시지이름을 의미한다.) 이 함수의 값이 다시, 논리의 기본이 되는 term들로 정의된다. 거기에 메시지들은 미리 정의되어 있으므로, 그 자체로 상수로서 term의 역할을 한다(m).

한 HDTL formula ϕ는 한 상태에 대한 조건이거나, 한 상태에서 시작되는 상태순차에 대한 조건이 된다. 한 상태에 대한 조건은 일반적인 논리와 마찬가지로 equality, and, or, not 등으로 기술된다. 상태순차에 대한 조건은 일반적인 시제논리연산자들(○, □, ◇, U 등)과 동결한정자에 의해 기술된다.

동결한정자는 하나의 변수를 상태순차 중의 한 시점에서, 그 상태의 값으로 바인딩해 주는 역할을 한다. $\phi(x)$를, 변수 x를 포함하는 수식이라 할때 $x.\phi(x)$ 가 상태순차 $\sigma=\sigma_0\sigma_1\ldots\sigma_n$에 의해 만족된다는 말은, $\phi(\sigma_0)$이 참이라는 말이다. 이때 $\phi(\sigma_0)$는 $\phi(x)$에서 x를 상태순차 σ의 처음 상태의 값(σ_0)으로 치환하여 얻어진다. 예를 들어,

$$x.(msg(x) = borrow)$$

이것은 현재 상태(< n_1, n_2, m>)의 값을 변수 x에 바인딩 시켜서, 그때 식 msg(x) = borrow 의 만족여부를 묻는 식(f_1)이 된다. 따라서 이 식은 하나의 상태를 가지고 만족 여부를 판단할 수 있는 식이 된다. 여기에 시제 연산자를 붙이면,

$$\Diamond\, x.(msg(x) = borrow)$$

이는 언젠가 f_1이 일어나는지를 묻는 식이 된다. 즉 언젠가, 그 상태의 메시지 값이 borrow 인지를 묻는 HDTL 식이 된다.

이런 의미는 다음과 같은 식으로 정형적으로 기술될 수 있다.

정의 3.6 (의미) σ를 하나의 상태순차라고 하자. 그리고 E:V→O×O×M을 각 변수들에 대한 환경(environment)이라고 하자. 하나의 쌍 (σ, E)가 formula ϕ를 만족시킨다는 것은 $\sigma \models_E \phi$라고 쓸 수 있다. 이것은 다음과 같이 귀납적으로 정의된다.

- $\sigma \models_E \pi_1=\pi_2$ iff $E(\pi_1)=E(\pi_2)$
- $\sigma \models_E \pi_1=\pi_2$ iff $E(\pi_1)=E(\pi_2)$
- $\sigma \models_E \pi_1\neq\pi_2$ iff $E(\pi_1)\neq E(\pi_2)$
- $\sigma \models_E$ true
- $\sigma \models_E \phi_1\rightarrow\phi_2$ iff $\sigma \models_E \phi_1 \rightarrow \sigma \models_E \phi_2$
- $\sigma \models_E \phi_1\wedge\phi_2$ iff $\models_E \phi_1 \wedge \models_E \phi_2$
- $\sigma \models_E \phi_1\vee\phi_2$ iff $\models_E \phi_1 \vee \models_E \phi_2$
- $\sigma \models_E \neg\phi$ iff $\neg(\models_E \phi)$
- $\sigma \models_E \bigcirc\phi$ iff $\|>1^1 \rightarrow {}^1\models_E \phi$
- $\sigma \models_E \odot\phi$ iff $\|>1\wedge^1 \models_E \phi$
- $\sigma \models_E \Box\phi$ iff $^i\models_E \phi$ for all i≥0
- $\sigma \models_E \Diamond\phi$ iff $^i\models_E \phi$ for some i≥0
- $\sigma \models_E \Box\phi$ iff $^i\models_E \phi$ for all i≥0
- $\sigma \models_E \phi_1 U \phi_2$ iff
  * $\sigma^i \models_E \phi_2$ for some i≥0,
    and $\sigma^j \models_E \phi_1$ for all j, 0≤j<i
- $\sigma \models_E \phi_1 \mathrm{U} \phi_2$ iff
  * $\sigma^j \models_E \phi_1$ for all j≥0, or
  * $\sigma^1 \models_E \phi_2$ for some i≥0,
    and $\sigma^j \models_E \phi_1$ for all j, 0≤j<i
- $\sigma \models_E \phi_1 P \phi_2$ iff
  * for some i≥0, $\sigma^i \models_E \phi_2 \rightarrow \sigma^j \models_E \phi_1$,
    for some j, 0≤j<i and,
  * $\sigma^j \models_E \phi_1$ for some j≥0
- $\sigma \models_E \phi_1 \mathrm{P} \phi_2$ iff
  * for some i≥0, $\sigma^i \models_E \phi_2 \rightarrow \sigma^j \models_E \phi_1$,
    for some j, 0≤j<i
- $\sigma \models_E x.\phi$ iff $\sigma \models_{E[x:=\sigma 0]} \phi$

이때 $E(f(x)) = f(E(x))$, $E(m) = m$이다. (f∈{snd, rcv, msg}) 또 σ^i는 σ에서 처음 i개의 상태를 제거한 상태순차이다. 그리고 $|\sigma|$는 σ의 길이를 의미한다.

$E[x:=s]$은 E와, x를 제외한 모든 부분에서 일치하고, x에 대해서는 s∈O×O×M을 매핑하는 환경이다. 이렇게 함으로써 동결 한정자에 의해 변수에 새로운 값이 바인딩 되는 것을 나타낼 수 있다.

위의 정의에서 기존의 시제논리와 다른 부분은 환경(environment, E)을 사용한 것과, 동결한정자의 정의가 추가된 부분이다.

HDTL 식은 변수를 포함한다. 식 안에서 그 변수가 참조될때 식의 진위여부를 가리기 위해서는 변수가 어떤 값을 가지는지를 알고 있어야 한다. 환경은 그것을 위해, 각 변수에 대해 바인딩되어 있는 상태의 값을 저장하는 역할을 한다. 이 변수들은, 우리가 시제논리식을 파싱(parsing)하여 트리구조를 만든다고 하면, 그 맨 아래 노드에 term간의 equality(혹은 inequality)의 형태로 사용되게 된다. 이것을 정의한 것이, 정의 3.6의 처음 두줄이다.

환경에 포함된 변수들의 값을 정하는 것은 동결한정자이다. 정의 3.6의 마지막 부분을 보면 환경이 어떻게

변하는지 알 수 있다. 즉 변수는 상태순차의 현재값(σ_0)으로 바인딩되어 환경에 추가되게 된다. 따라서 동결한 정자를 포함한 식의 진리값은 그 논리식이 평가되는 시점의 상태에 의존적이다.

또한 주의할 것은, HDTL이 유한한 상태순차에 대해서 해석된다는 점이다. 무한한 상태순차에 해석하는 경우와의 차이는, 상태순차의 마지막을 어떻게 해석하는가 하는 점이다. 이점 때문에 각 시제논리연산자에는 강약(strong and weak operators)이 정해지게 된다. strong next($\odot$)의 경우는, 다음 상태가 **존재하고** ($|\sigma|>1$), 그때 그 다음 상태에서 식이 만족되는지를 살핀다. 반면 weak next($\bigcirc$)의 경우는 다음 상태가 **존재한다면**, 그때 그 다음 상태에서 식이 만족되는지를 살핀다. 따라서 두 식은 상태순차의 마지막에서 다른 값을 가진다. *U*와 U, *P*와 P도 각각 이런 강약의 차이를 나타낸다. 예를 들어 precedence(*P*, P)의 의미는 'ϕ_2가 나온다면 그 전에 ϕ_1이 나와야 한다'인데, strong precedence(*P*)의 경우에는 ϕ_1이 적어도 한번은 발생해야 하는 것을 강제하고 있는 반면, weak precedence (P)의 경우에는 ϕ_2가 나오지 않을때는 ϕ_1도 발생할 필요가 없다고 간주하는 것이 다르다. until의 경우에는 ϕ_2의 발생을 강제하느냐 아니냐의 차이가 있다.

4. 동적 분석에의 이용

전 절에서 우리는 HDTL의 구문과 그 의미를 살펴보았다. 이 장에서는 HDTL을 가지고 동적 시스템의 특성을 명세하고 분석하기 위한 알고리듬을 제시한다.

여기서 제시하는 분석 기법은 HDTL로 작성된 명세를 이용해 시스템의 수행을 모니터링하는 방법이다. 즉 시스템을 수행시킬 때, 생성되는 메시지들을 모니터링하여, 그 메시지 순차가 명세된 것에 위배되는 동작을 보이지 않는지를 검사한다. 이를 위해서는 주어진 명세에 대응하는 오토마타, 혹은 그와 유사한 수행가능한 표현(executable representation)이 필요하다.

[11]에서는 플로우그래프(*flow graph*)라는 표현을 사용하여 분석을 수행한다. 이는 기존의 tableau-based 방법에서 만들어지는 오토마타와 같은 역할을 한다. 오토마타에서는 상태와 전이를 각각 그래프의 노드(nodes)와 에지(edges)로 나타내지만, 플로우 트리에서는 그것들을 둘다 노드로 나타내고 단지 표기를 달리한다는 점이 다르다. 상태노드는 타원으로, 전이노드는 사각형으로 나타낸다. 하나의 전이노드는 따라서, 여러개의 입력 상태노드를 가질 수 있지만, 출력 상태노드는 하나만을 가진다.

우리는 이 방법을 확장하여, 그래프가 아닌 트리(*tree*) 형식의 자료구조인 플로우트리를 사용한다. 위에서 제시한 표기상의 차이점과 함께, 플로우 트리는 다음과 같은 점에서 기존의 오토마타와 다르다.

- 기존의 오타마타는 그 크기가 유한하다. 하지만 여기서 만들어지는 트리는 크기가 무한할 수 있다. 이는 객체가 무한히 생성될 수 있기 때문이다. 물론 실질적으로는 검증에 사용가능한 자원의 양에 따라 제한된다.
- 기존의 오토마타 생성은 검증이 수행되기 전에 완성된다. 하지만 여기서는 위와 같은 이유로 그런 것이 불가능하므로, 검증[5]의 수행시에 트리가 만들어진다.
- 따라서 기존에는 하나의 시제논리식에 대해, 하나의 오토마타가 만들어지는데 반해, 여기서는 입력으로 들어오는 (혹은 관찰되는) 상태순차에 따라 다른 트리가 만들어진다.
- 기존의 오토마타는 보는 관점에 따라 언어 생성기(generator)와 언어 판독기(acceptor), 두 가지로 볼 수 있었다. 하지만 여기서는 판독기로만 정의되고 사용된다.

이러한 특징들로 인해, 우리가 제시하는 플로우 트리 모델은 기존의 오토마타의 단순한 시각적 변종이 아니다. 이는 [11]에서와 차이점인데, 그 논문에서는 플로우 그래프를 사용하여 오토마타를 다르게 나타내고 있다. 위와 같은 특징으로 인해 우리는 그래프 대신 트리표현을 사용하고자 한다.

이 절에서 우리는 이러한 특징을 가진 플로우 트리의 생성 방법에 대해 살펴본다. 그리고 그것을 가지고 동적 분석을 수행하는 방법에 대해 설명하고자 한다.

4.1 플로우 트리

예를 들어 다음과 같은 HDTL 명세가 있다고 하자.

$f_0 = \Box x.(msg(x)=borrow$

$\rightarrow \Diamond y.(msg(y)=return \wedge rcv(x)=snd(y)))$ (식1)

이것은 가상의 도서대출 시스템에서 생각할 수 있는 특성이다. 한 사람이 어떤 책을 빌리면, 언젠가는 그 책이 반환되어야 한다는 것을 의미한다. 책을 빌리고 반환하는 행위는 사람과 책간의 메시지 전달로 정의된다. 이

5) 이하에서 검증이란 verification보다는, validation이란 의미로 사용된다. 즉 서론에서 말한 테스팅이나 런타임 모니터링 등의 동적분석을 수행하는 것을 의미한다. 이 분석의 결과로 시스템이 주어진 명세를 만족한다는 것에 대한 완전한 확신을 얻을 수는 없지만, 시스템을 더 신뢰할 만한 것으로 볼 수 있게 한다

시스템이 동적 시스템인 이유는 사람과 책 간의 '대출' 이라는 관계가 동적으로 변하기 때문이다. 이러한 경우, 기존의 단순한 시제논리로 위와 같은 명세를 표기하는 것은 쉽지 않다.[6]

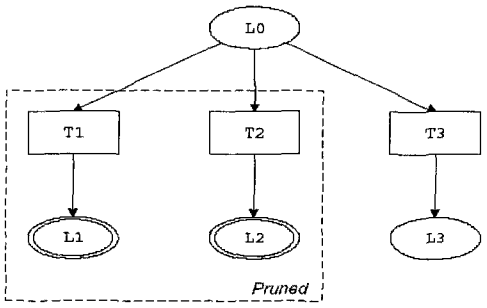
우리가 제시하는 방법은, 오토마타를 미리 만들어 놓는 대신에, 관찰되는 상태순차에 따라 필요한 부분만을 만들어 가는 것이다. 예를 들어 위와 같은 명세에 대해, 다음과 같은 상태순차가 입력으로 들어온다고 하자.

$\sigma_i=\sigma_0, \sigma_1, \sigma_2, \sigma_3, \cdots$

$=(<p_1, b_1, \text{borrow}>, <p_2, b_2, \text{borrow}>,$

$<b_1, p_1, \text{return}>, <b_2, p_2,, \text{borrow}>, \cdots)$

이 상태순차는 두 명의 사람(p_1, p_2)이 두 권의 책(b_1, b_2)를 각각 빌리고 반납하는 과정을 나타낸 것이다. 이런 상황에서 검증이 어떻게 수행되는지를 간단히 따라가 보기로 한다. 초기 상태에서 표 1과 유사한[7] 방법으로 트리를 만들어 보면 그림 2와 같다.



$L0 := L1 := L2 := \{f_0\}$
$L3 := \{f_0, f_1\}$
$\quad := \{f_0, \Diamond y.(msg(y) = \text{return} \wedge rcv(\sigma_0) = snd(y))\}$
$T1 := msg(\sigma_0) \neq \text{borrow}$
$T2 := msg(\sigma_0) = \text{return} \wedge rcv(\sigma_0) = snd(\sigma_0)$
$T3 := true$

그림 2 식 1을 위한 플로우 트리의 첫단계

여기서 타원으로 나타난 부분이 검증시 각 상태에서 검사해야 할 시제논리식을 나타내고, 사각형으로 나타난 부분이 상태가 넘어갈 때 검사해야 할 조건이 된다. 따라서 사각형으로 나타난 부분에는 현재상태의 정보만 가지고 참 거짓이 결정될 수 있는 식이 기술되고, 타원으로 나타난 부분에는 다음상태에서 만족될 시제논리식이 기술된다. 두겹의 선으로 나타난 타원은 그 상태가 논리식을 만족시키는 최종상태(accepting states)임을 나타낸다. 각 노드에는, 그것에 관계된 시제논리식의 집합이 연관되어 있다. L0, L1, L2, L3는 타원으로 나타난 부분을, 그리고 T1, T2, T3 는 사각형으로 나타난 부분에 관련된 시제논리식이다.

그림 2가 만들어진 과정을 좀더 자세히 따라가 보기로 한다. 이 과정은 2.1절에서 설명한 시제논리에 해당하는 오토마타를 생성하는 과정과 유사하다. 오토마타의 각 상태(플로우트리에서 타원으로 나타나는 부분)는 그 상태에서부터 입력상태순차를 관찰하여 검증하여야 할 부분이고, 오토마타의 전이(플로우 트리에서 사각형으로 나타나는 부분)는 현재의 입력상태정보를 가지고 판단할 수 있는 전이정보를 가진다. 다른 점은 플로우트리의 생성은 입력 상태순차에 대한 정보를 가지고 동적으로 수행된다는 점이다.

우선 처음에 트리의 루트노드인 L0를 만든다. 이 노드에 붙은 식의 집합은 $\{f_0\}$로, 검증해야 할 HDTL 명세 자체가 된다. 이것은 앞으로 들어올 상태순차가 f_0를 만족하는지를 검사하겠다는 의미가 된다.

그리고 상태순차의 처음 상태인 $\sigma_0=<p_1, b_1, \text{borrow}>$를 읽는다. 이 정보를 이용하여 우리는 트리를 확장하게 된다. 우선 여기서는 표 1을 참고하여 설명한다. 식 $\{f_0\}$를 σ_0를 가지고 변환하는 과정은 다음과 같다.

Step 1: $\{f_0\}=\{\Box x.(msg(x)=\text{borrow} \rightarrow$
$\Diamond y.(msg(y)=\text{return} \wedge rcv(x)=snd(y)))\ \}$
$\Rightarrow \{x.(msg(x)=\text{borrow} \rightarrow$
$\Diamond y.(msg(y)=\text{return} \wedge rcv(x)=snd(y))), \bigcirc f_0\}$

첫 번환은 □에 대한 변환규칙인 [r□]를 적용한 것이다. 즉 식□f에 대한 검증은 f와(AND) ○□f에 대한 검증으로 나눠 생각할 수 있다. 이 규칙을 적용한 결과, 원래 하나의 식의 집합이던 것이 두개의 식을 가지는 집합으로 변경되었다. 이때 두 번째 식인 $\bigcirc f_0$는 next 연산자로 싸여 있으므로 현재 상태에서는 검증할 수 없고, 다음 상태로 검증을 미뤄야 하는 부분이다. 반면 앞의 식에는 최외곽에 동결한정자가 있으므로, 현재 상태의 정보를 가지고 변수를 바인딩해 주어야 한다.

Step 2: $\Rightarrow \{msg(\sigma_0)=\text{borrow} \rightarrow$
$\Diamond y.(msg(y)=\text{return} \wedge rcv(\sigma_0)=snd(y)), \bigcirc f_0\}$

두 번째 변환이 이러한 동결한정자를 처리하여 변수를 바인딩해주는 변환이다[8]. 여기서는 상태순차의 처음

6) 불가능한 것은 아니다. 모든 사람과 모든 책간의 대출, 반납 사건을 정적으로 정의하여 이름을 붙이고, 그 사건들을 이용하여 명세를 기술하면 된다. 하지만, 이렇게 하는 경우 검증을 위한 오토마타의 크기는 비현실적으로 커지게 된다. 이는 변수라는 개념이 없기 때문이다

7) 동결연산자로 인한 확장을 포함, 자세한 사항은 뒤에 설명한다.

8) 이 부분에 대한 정확한 규칙의 정의는 뒤의 표 2를 보라.

값인 σ_0를 변수 x에 바인딩하여, 동결한정자 내의 수식을 변환하게 된다. 이 결과로, 집합내의 처음 식에서는 변수 x가 사라지게 된다. 두번째 식($\bigcirc f_0$)은 그대로 유지된다.

Step 3: $\Rightarrow$ {$msg(\sigma_0) \neq borrow$, $\bigcirc f_0$},
{$\Diamond y.(msg(y)=return \wedge rcv(\sigma_0)=snd(y)), \bigcirc f_0$}

세 번째 변환은 implication에 대한 규칙인 [r→]를 적용한 것이다. 이것은 본질적으로 implication이 negation과 or로 표현되는 것을 나타낸다. 여기서 주의할 것은 지금까지 하나였던 식의 집합이 두개로 분리된 것이다. 표 1을 설명할 때 말한 것처럼, 집합 내의 수식은 AND로 연결되고, 집합 간에는 OR로 연결된다.

Step 4: $\Rightarrow$ {$msg(\sigma_0) \neq borrow$, $\bigcirc f_0$},
{$y.(msg(y)=return \wedge rcv(\sigma_0)=snd(y)), \bigcirc f_0$},
{$\odot\Diamond y.(msg(y)=return \wedge rcv(\sigma_0)=snd(y)), \bigcirc f_0$}

Step 5: $\Rightarrow$ {$msg(\sigma_0) \neq borrow$, $\bigcirc f_0$},
{$msg(\sigma_0)=return \wedge rcv(\sigma_0)=snd(\sigma_0)), \bigcirc f_0$},
{$\odot\Diamond y.(msg(y)=return \wedge rcv(\sigma_0)=snd(y)), \bigcirc f_0$}

네 번째 변환은, ◇에 대한 변환규칙 [r◇]를 적용한 것이다. 이것은 식 ◇f에 대한 검증은 f, 또는(OR) ⊙◇f를 검증함으로써 수행될 수 있다는 의미이다. 역시 OR의 의미이므로, 집합은 다시 한번 나누어져서, 3개의 집합이 된다. 주의할 것은, ◇의 경우에는 ○가 아니라 ⊙를 사용한다는 점이다. ◇f의 경우, f가 한번은 나타나야 한다는 것을 의미한다. 따라서 현재 상태가 마지막 상태이면서 아직 f가 한번도 발생하지 않은 경우는 식을 만족시키는 것으로 볼 수 없다. 따라서 strong next 연산자로 변환된다. 마지막 변환은 다시 변수 y에 대한 동결한정자에 대한 것으로, y를 σ_0에 바인딩하는 것이다.

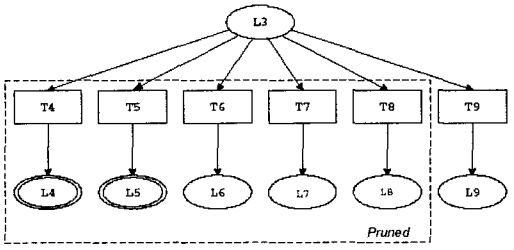
이상의 과정의 결과로 3개의 식의 집합이 만들어졌다. 이제 식들은 더 이상 변환될 수 없는데, 이것은 식들이 전부 기본적인 형태, 즉 현재 상태에서 만족여부가 판단되는 것(e.g. $msg(\sigma_0) \neq borrow$)들이거나, 다음 상태가 되어야 더 분석할 수 있는 식(즉 최외곽에 ○이나 ⊙가 있는 경우)이기 때문이다. 전자의 경우를 basic formula라고 하고, 후자의 경우를 next formula 라고 한다.

이 각각의 집합을 가지고 플로우트리를 확장한다. 각 집합내의 next formula에서 ○ 혹은 ⊙를 떼어낸 부분이 다음 상태에서 검증할 부분이 되고, 플로우트리에서 타원으로 나타나는 부분이 된다(L1, L2, L3). 이때 ○의 경우에는, 현재 상태가 마지막 상태인 경우에는 진리값이 참이 되므로 그 노드는 최종상태(accepting states)가 된다(e.g. L1, L2). 그림으로는 최종상태에 해당되는 노드를 두겹의 선으로 된 타원으로 나타낸다. 반면 ⊙의 경우에는 마지막 상태에서 거짓이 되므로, 최종상태가 아니다. 각 집합내의 basic formula 부분은 상태가 이동할때 검증해야 하는 식이므로, 플로우트리에서 사각형으로 나타나는 부분이 된다(T1, T2, T3). 즉 (T1, L1), (T2, L2), (T3, L3)가 각각의 집합에 대응되게 된다. 이때 basic formula 부분이 없는 집합의 경우에는 true가 된다(T3).

그림 2는 이런 식으로 만들어진 것이다. 의미적으로 보면, T1과 T2의 경우는 책을 빌리는 메시지가 아니거나, 반환하는 메시지인 경우이다. 따라서 그러한 하나의 상태만을 봤을때는 명세 f_0를 만족하고 있으므로, 두 상태 L1과 L2는 최종상태가 된다. 그리고 거기서부터 만족해야 할 논리식은 원래의 식인 f_0와 같게 된다.

하지만 여기서 예를 들고 있는 상태(σ_0)의 경우, T1과 T2는 둘다 만족시키지 않고, T3만을 만족시킨다. 따라서 우리는 플로우트리를 수행시켜 상태를 L3에 이르게 한다. 그리고 L1과 L2에 대한 정보를 버린다. 이렇게 함으로써 트리는 다시 L3에서 시작되게 된다. 이 노드는 f_0과 f_1으로 구성되어 있다. f_0는 원래의 식과 같은 것으로, 이후에도 원래의 시제논리식이 계속 만족되어야 한다는 것을 뜻하고, f_1은 한번 책을 빌렸으므로, 추후에 그 책이 반납되어야 함을 뜻한다. 따라서 L3는 최종상태가 아니다.

유사한 방법으로 L3를 다시 전개해 보면 트리는 그림 3과 같이 된다.



$T4 := msg(\sigma_1) = \text{return} \wedge rcv(\sigma_0) = snd(\sigma_1)$
$L4 := \{f_0\}$
$T5 := msg(\sigma_1) = \text{return} \wedge rcv(\sigma_0) = snd(\sigma_1) \wedge rcv(\sigma_1) = snd(\sigma_1)$
$L5 := \{f_0\}$
$T6 := msg(\sigma_1) = \text{return} \wedge rcv(\sigma_0) = snd(\sigma_1)$
$L6 := \{f_0, \Diamond y.(msg(y) = \text{return} \wedge rcv(\sigma_1) = snd(y))\}$
$T7 := msg(\sigma_1) \neq \text{borrow}$
$L7 := \{f_0, f_1\}$
$T8 := msg(\sigma_1) = \text{return} \wedge rcv(\sigma_1) = snd(\sigma_1)$
$L8 := \{f_0, f_1\}$
$T9 := true$
$L9 := \{f_0, f_1, f_2\}$
$:= \{f_0, f_1, \Diamond y.(msg(y) = \text{return} \wedge rcv(\sigma_1) = snd(y))\}$

그림 3 플로우 트리 (계속)

여기서 다시 주어진 상태순차를 이용하면, T9만이 가능함을 알 수 있고, 따라서 L9를 계속 전개하게 된다. 이때 L9와 L3를 비교해 보면, 검증해야 할 조건이 더 강화되었음을 알 수 있다. 이는 책의 대출이 증가되었음을 의미한다. 따라서 L9에서 추가된 식은, 두 번째 빌린 책에 대한 반납이 있어야 함을 나타내는 것이 된다. 이런 식으로 트리를 전개해 나가면서 검증을 수행한다.

이런 과정을 정형화하기 위해서, 우리는 정의 3.6과 같이 환경(environment)를 포함하는 새로운 변환 규칙을 정의해야 한다. 이때 동결 한정자는 환경에 정보를 추가하는 역할을 하게 된다. 이를 고려해서 변환규칙을 정의하면 표 2와 같다.

표 2 새로 정의한 변환규칙

[r ∧] [r ∨] [r →]	$(f_1 \wedge f_2)E \Rightarrow \langle\{f_1E, f_2E\}\rangle$ $(f_1 \vee f_2)E \Rightarrow \langle\{f_1E\}, \{f_2E\}\rangle$ $(f_1 \rightarrow f_2)E \Rightarrow \langle\{\neg f_1E\}, \{f_2E\}\rangle$
[r □] [r ◇]	$(\Box f)E \Rightarrow \langle\{fE, (\bigcirc\Box f)E\}\rangle$ $(\Diamond f)E \Rightarrow \langle\{fE\}, \{(\odot\Diamond f)E\}\rangle$
[r U] [r P]	$(f_1 U f_2)E \Rightarrow \langle\{f_2E\}, \{f_1E, (\odot(f_1 U f_2))E\}\rangle$ $(f_1 P f_2)E \Rightarrow \langle\{f_1E, \neg f_2E\}, \{\neg f_2E, (\bigcirc(f_1 P f_2))E\}\rangle$
[r U] [r P]	$(f_1 U f_2)E \Rightarrow \langle\{f_2E\}, \{f_1E, (\bigcirc(f_1 U f_2))E\}\rangle$ $(f_1 P f_2)E \Rightarrow \langle\{f_1E, \neg f_2E\}, \{\neg f_2E, (\odot(f_1 P f_2))E\}\rangle$
[r frz]	$(x.f)E \Rightarrow \langle\{fE[x:=\sigma_i]\}\rangle$

이 변화규칙은 상태정보인 σ_i를 파라메터로 가진다. 따라서 검증이 수행되는 동안, i번째 상태에서 이 규칙을 적용할 때는 σ_i를 가지고 적용하게 된다.

이 변환규칙을 가지고 만들어 나가게 되는 플로우 트리에 대해서 정의하면 다음과 같다.

정의 4.1 (플로우 트리) 한 플로우 트리 G는 다음과 같은 6개의 요소로 정의된다.

G=< Locs, Trans, Lab, Flow, Init, Fin >

이때,

- Locs는 location의 집합이다.
- Trans는 transition의 집합이다.
- Lab은 각 노드의 labeling 함수이다.
- Flow는 flow relation 이다.
- Init는 초기노드이다.
- Fin은 최종노드이다.

Locs는 플로우트리에서 타원으로 나타나고, Trans는 사각형으로 나타난다. Flow는 플로우트리 내의 에지를 나타내고, 따라서 한 쪽이 location이면 다른 쪽은 transition이 된다.

Lab은 각 노드에 관련된 레이블링 함수가 된다. 각 노드에는 $\{f_1E_1, f_2E_2, f_3E_3, \cdots\}$과 같은 식으로 시제논리식과 환경의 페어의 집합이 할당되게 된다. cl(f)는 원래 검사하려는 HDTL 식 f의 모든 부분식(subformula)과 그것의 부정(negation)의 집합이다. 그리고 Env는 식이 포함하는 변수들에 대한 가능한 모든 바인딩의 집합이다.

분석과정은 표 2를 가지고 상태순차를 읽어나가면서 위와 같은 플로우트리를 만들어 나가게 된다. 현재의 Locs들에 붙은 레벨들이 포함하고 있는 fE 페어에 대해 변환규칙들을 적용한다. 그 결과로는 위에서 든 예와 같이, fE 들의 집합의 집합이 생성된다. 이 각 집합들에서 basic formula는 transition가 되고, next formula에서는 next 연산자를 떼어내고 다시 location 들이 된다. 이때, weak next의 경우에는 location은 최종상태가 되고, stong next의 경우에는 그렇지 않게 된다.

4.2 동적 분석

이상과 같이 플로우 트리를 생성하는 과정을 정의하였다면 이것을 이용해서 동적 분석, 즉 테스팅이나 런타임 모니터링을 수행할 수 있다. 두 과정 모두 하는 일은 유사하다. 프로그램을 수행하면서 외부에서 결과를 관찰하여 명세에 기술된 대로 동작하는지를 관찰하는 것이다. 차이점은 테스팅은 대상이 되는 시스템이 개발과정 중의 것이고, 런타임 모니터링은 최종 산출물이라는 점뿐이다.[9]

이 과정은 다음과 같이 진행된다. 우선 시스템을 초기상태로 설정하고 실행시킨다. 그리고 우리가 관찰하려는 일이 발생하는 매 순간마다, 즉 우리의 가정에 따르면 메시지 전달이 일어나는 순간마다, 외부에서 상태를 관찰한다. 그리고 관찰된 상태에 따라 검증 프로그램이 유지하는 내부표현, 즉 플로우 트리를 수정한다. 그 과정에서 프로그램을 더 수행하는 것이 의미가 없으면 종료하고, 아니면 계속 수행시키면서 관찰한다.

여기서 중요한 것은 플로우 트리의 수정이다. 전 절에서 우리는 플로우트리가 프로그램이 수행되어 감에 따라, 그 관찰 결과를 이용해서 확장되어 나감을 보였다. 따라서 매 순간, 플로우 트리에서 중요한 정보는 상태를 나타내는 트리의 마지막 노드들(leaf nodes)의 집합이 된다. 그 외의 노드들이 가지는 정보는 과거에 대한 것으로, 계속 보존될 필요는 없다.

이때 마지막 노드들의 집합이 어떤 상태인지가 지금까지의 입력상태순차에 대한 관찰결과가 된다. 이에 따

9) 이 논문에서는 닫힌 시스템(closed systems)을 대상으로 한다. 이는 테스트케이스의 생성 및 입력을 고려하지 않는다는 의미이다. 열린 시스템(open systems)의 경우, 테스트 드라이버를 포함시켜 닫힌 시스템으로 간주할 수 있다

라 관찰을 계속해야 하는지의 여부와, 시스템이 기술된 명세에 맞게 수행되는지의 여부를 판단할 수 있다. 플로우 트리가 비결정적 오토마타(nondeterministic automata)와 유사하다는 것을 고려하면, 플로우 트리가 받아들이는 순차와, 최종 상태의 관계에 대해 쉽게 알 수 있다. 한 스트링(string)이 비결정적 오토마타에 의해 받아들여진다는 말은, 그 스트링으로 오토마타를 수행했을 때, 최종상태에 이르는 경로(path)가 존재한다는 것을 의미한다. 이를 염두에 두고 가능한 경우를 생각해 보면, 다음과 같은 경우가 있음을 알 수 있다.

1. 마지막 노드들의 집합이 공집합인 경우 :

전 절에서 예를 들며 설명했듯이, 한 상태에서 그려진 플로우 트리중에서, 현재 입력상태가 만족시키는 전이만이 수행되게 되고, 나머지는 버려지게 된다. 따라서 노드들의 집합이 공집합이라는 말은 플로우 트리를 수행하려고 할때 가능한 전이가 하나도 없는 경우이다. 따라서, 이것은 검증이 실패했음을 뜻한다. 즉 테스팅으로 말하면 명세에 어긋나는 수행을 발견한 것이다. 예를 들어 □x.msg(x)=borrow 라는 명세가 있을때, 입력 상태로 borrow 가 아닌 메시지전달이 들어왔다면, 이는 플로우 트리의 전이에 연관된 조건(msg(σ)=borrow)를 만족시키지 못하게 된다. 따라서 플로우 트리를 수행시키면 마지막 노드들의 집합은 공집합이 되고, 이는 □x.msg(x)=borrow라는 명세가 이 상태에서 지켜지지 않았음을 의미한다.

2. 마지막 노드의 집합에 최종상태(accepting states)인 것이 하나 이상 있는 경우 :

이는 현재까지의 관찰에서는 명세가 만족되고 있음을 의미한다. 따라서 여기서 관찰이 끝날 경우, 시스템은 명세를 만족하는 수행을 했다고 결론을 내릴 수 있다. 위와 같은 명세의 예에서, 입력 상태로 borrow 메시지를 전달하는 것이 들어왔다면, 이는 조건 msg(σ)=borrow을 만족시키고, 플로우 트리의 마지막 노드에는 다시 { □x.msg(x)=borrow, φ }이라는 레벨이 붙게 된다. 그리고, 그 상태가 최종 상태로 표시되게 되므로, 현재 상태에서는 명세를 만족하고 있음을 알 수 있게 된다.

3. 마지막 노드의 집합에 최종상태인 것이 하나도 없는 경우 :

미래의 상태순차 입력을 이용하여 검증해야 할 것이 남은 경우이다. 따라서 이 상태에서 관찰이 종료될 경우, 시스템은 명세를 만족하지 못한 것으로 볼 수 있다. 예를 들어 ◇x.msg(x)=borrow 라는 명세가 있을 때, borrow 가 아닌 메시지 전달만이 입력으로 들어오는 한에는, 아직 명세를 만족시키지 못한 것이 된다. 이때 플로우 트리의 마지막 노드들에는 최종상태인 것이 포함되지 않게 된다.

4. 마지막 노드들이 가지는 시제논리식들, 즉 다음 상태에서 만족시켜야 할 식이 true인 경우 :

이후의 상태 관찰에 관계없이 시스템은 주어진 명세를 만족하는 것으로 판정된다. 예를 들어 ◇x.msg(x)=borrow와 같은 식의 경우, 맨 처음 상태에서의 메시지 전달이 borrow이기만 하면 이후의 상태순차 입력은 관계없는 것이 된다. 이럴때 플로우 트리를 그리면 transition 부분에는 msg(σ)=borrow 라는 조건이 들어가지만, 거기서 연결된 location에는 해당하는 formula가 없게 된다. 따라서 검증은 그 상태에서 종료할 수 있다.

이러한 구분은 대상이 되는 명세의 종류와 관계된다. 한 명세가 반드시 이러한 4가지 경우를 다 가지는 것은 아니다. 일반적으로 시제논리로 기술된 명세는 크게 safety와 liveness로 나눠진다고 간주된다. 간단히 말하면, safety는 어떤 나쁜 일이 일어나지 않는다는 것을 의미하는 성질이고, liveness는 어떤 좋은 일이 결국 일어날 것이라는 성질이다[13]. 이 분류를 좀더 엄밀하게 정의하면 다음과 같이 나타낼 수 있다[14]. 식 Φ가 safety formula인지 liveness formula인지를 가르는 기준은 다음과 같다.

- Safety

식 Φ를 만족시키지 않는 (즉 ¬Φ를 만족시키는) 모든 순차 σ는 하나의 prefix σ[0..k]를 가지고, 그때 그 prefix에 어떤 순차를 붙여 확장시키던지 간에 그런 확장된 순차들도 모두 Φ를 만족시키지 않는다.

- Liveness

모든 유한한 순차는 그것에 특정한 순차를 붙여 확장함으로써, 식 Φ를 만족하는 순차로 확장될 수 있다.

동적분석 과정에서 이러한 구분이 가지는 의미를 생각해 보자. 동적분석은 각 관찰 단계에서 그때까지 관찰된 유한한 순차를 가지고 시스템이 제대로 동작하고 있는지를 확인하는 것이다. 이때 중요한 정보는 시스템이 **제대로 동작하지 않는** 시점에서 문제의 발생을 알리는 것이다. 따라서 이를 위해서는 유한한 관찰을 통해 시스템과 명세의 불일치 여부를 판단해 낼 수 있어야 한다.

이러한 관점에서 보면, 명세 중에서 safety 명세가 liveness 명세에 비해 동적분석에 더 적합함을 알 수 있다. 위의 정의에 따라서 safety 명세는 유한한 순차의 관찰만으로 그 불일치를 검출할 수 있다. 반면 liveness의 경우는 모든 유한한 순차가 미래의 동작에 따라 명세를 만족하는 순차로 확장될 수 있으므로, 유한한 관찰에 의거한 판단은 거의 의미가 없게 된다.[10] 따라서 동적분석이 의미를 가지기 위해서는 대상이 되는 명세를

safety 명세로 한정하는 것이 좋다.

문제는 한 명세가 어느 쪽에 속하는지를 구분하는 것이 쉽지 않다는 점이다[15]. 따라서 우리는 식의 구분에 대해서는 사용자의 책임으로 남겨두기로 한다.

위에서 정의된 4가지 경우는, 이런 명세의 구분 관점에서 보면 각각 다음과 같은 경우임을 알 수 있다. 1번은 safety 명세의 불일치를 발견하는 경우이다. 2번은 safety 명세와의 일치, 혹은 liveness 명세를 만족하는 경우이다. 3번은 liveness 의 경우, 아직 명세를 만족하는지 확신할 수 없는 경우이다. 미래에 명세의 만족을 보장할 수 있는 상태입력이 들어올 경우, 2번과 같은 상태가 되어 만족함을 알리게 된다.

4번의 경우는 liveness의 일종인 guarantee 명세에 해당하는 경우이다[2]. 이런 명세는 기본적으로 (canonical form) ◇p와 같은 형식을 가진다. 따라서 한번 p가 발생하면 그 후에는 시스템이 어떻게 동작하건 그 명세를 만족시킨 것으로 간주할 수 있다. 따라서 관찰을 종료하고 분석 결과를 낼 수 있게 된다.

이상과 같은 상황을 고려하여 동적 분석을 수행하는 모니터를 구현하였다. 그 알고리즘은 다음과 같다.

```
function monitor(f: Formula, s : array of State)
fun make tree(f: Formula, s : State)
begin
  var flow-tree=nil;
  reduced-formula :=reduce(f, s);
  forall disjunct in reduced-formula begin
    a :=atomic part(disjunct);
    n :=next part(disjunct);
    flow-tree :=(atomic=a, next=n) :: flow-tree;
       (* list concatenation *)
  end;
  return flow-tree;
end;

begin
  (* initialize *)
  flow-tree :=make tree(f, s[0]);
  (* loop *)
  for i=1 to length(s) begin
    before-flow-tree :=flow-tree;
    flow-tree :=φ; ;
    forall branch in before-flow-tree begin
      if evaluate(atomic(branch), s[i]) then
      begin
        flow-tree :=make tree(next(branch), s[i]);
        break (* from forall, since it deterministic *)
      end;
    end; (*forall*)
    if flow-tree=φ; then return False
  end; (* for *)
  return flow-tree
end
```

10) 모델체킹 등의 정형적 검증의 경우는 무한한 상태순차를 고려하므로 두가지 성질 모두에 대해서 검증이 수행되고, 의미를 가진다.

제시된 알고리즘 monitor는 속성을 명세하고 있는 HDTL 식과 검증하고자 하는 상태순차를 받아들여, 그 상태순차가 명세를 위반하는지 검증한다. 만약, 상태순차가 명세를 위반할 경우 false를 되돌리고, 그렇지 않을 경우 현재의 플로우 트리를 되돌린다.

본 알고리즘의 동작을 설명하기에 앞서서 알고리즘의 핵심부분인 make_tree함수를 설명하도록 하자. 이 함수는 임의의 식 f와 상태 s를 받아들여 제시된 변환 규칙에 따라서 s를 사용하여 f를 플로우 트리로 변환한다. 즉, 식을 각 clause가 basic이거나 next인 disjunctive normal form으로 변환하는 것이다. 구성되는 플로우 트리와 tableau에 의해서 생성되는 오토마타와의 차이는 next식을 더 이상 변환하지 않는다는 것이다. 즉, 항상 트리의 높이를 1로 유지하고 그 이상의 변환은 다음 입력 때까지 연기한다. 이는 다음의 상태 순차에 의해서 어떠한 next식이 선택될지 모르며, 동결 한정자를 변환하기 위해서는 새로운 상태가 필요하기 때문이다.

알고리즘은 우선 상태 순차의 처음 상태를 사용하여 플로우 트리를 구성함으로써 시작한다. 이 작업은 make_tree함수를 사용하여 이루어진다. 이렇게 구성된 플로우 트리에 대해서 그리고 상태 순차의 두 번째 상태부터 그 끝에 이르기까지 다음에 진행할 branch를 결정하고 이에 대해서 새로운 플로우 트리를 구성하는 작업을 반복한다. 진행할 branch를 결정하는 것은 플로우 트리를 구성하는 branch 중에서 basic 식의 값이 현재 상태에서 참인 것을 선택함으로써 이루어진다.

만약, basic식이 참인 branch가 없다면, 현재의 상태 순차는 주어진 식을 어기는 것이다. 이 경우 알고리즘은 false를 되돌린다. 주어진 모든 상태 순차에 대해서 알고리즘이 수행되었을 경우에는 현재 유지하고 있는 플로우 트리를 되돌린다. 이 플로우 트리는 상태 순차를 진행한 시스템의 현재 상태를 나타낸다.

5. 실 험

이 장에서는 지금까지 제안된 방법의 타당성을 조사하기 위해, 실제 시스템의 수행결과를 가지고 실험을 수행한 결과를 제시한다.

실험에 사용한 시스템은 현재 개발중인 디지탈 TV에 들어가는 소프트웨어이다. 이 시스템은 여러 태스크로 구성되어 있으며, 그들간의 상호작용을 통해 디지탈 TV가 제공해야할 여러 기능을 구현하고 있다. 이 시스템의 통합 테스팅의 자동화를 위해서 시스템 구성요소간의 메시지 전달을 모니터링하여, 그것이 제대로 명세대로 동작

하는지를 확인하는 작업이 수행되고 있다[16]. 이때 이 시스템에 존재하는 동적 구성요소의 존재가 문제가 된다.

디지탈 TV는 오디오와 비디오는 물론, 컴퓨터가 가지는 다양한 기능을 하나의 기기로 통합, 제공하려는 목표를 가지고 있다. 그러한 의도에 따라, 디지탈 TV에는 여러가지 응용 프로그램들이 수행될 수 있다. 여기에는 게임, 증권정보제공 등이 포함된다. 이러한 프로그램은 방송국과 연결된 라인을 따라, 비디오 정보와 함께 인코딩되어 가정까지 전송되어 오게 된다. 디지탈 TV 소프트웨어는 그런 정보를 디코딩하여 사용자에게 전달해 주게 된다. 따라서 이런 과정에서 여러가지 응용프로그램들이 디지탈 TV 상에서 생성, 소멸하게 된다.

시스템 상에서는 이런 동적 응용 프로그램들을 관리하는 하나의 클래스가 존재한다. 이 클래스의 이름을 임의로 Viewer 라고 하자. 이 클래스의 객체는 하나의 응용 프로그램이 새로 시작될때마다 하나씩 생성되어 수행된다. 따라서 이 클래스의 객체가 여럿 존재하는 상황에 대하여, 이 시스템이 잘 동작중인지의 여부를 나타내는 명세는, 기존의 방법으로는 제대로 나타내기 힘들다. Viewer 클래스의 객체들은 시스템에 하나 존재하는 Manager 객체와 통신하면서 작업을 수행한다. Manager 객체는 Viewer 클래스의 상태를 바꾸도록 메시지를 보낼 수 있고, Viewer 클래스는 자신의 상태 변화를 Manager에게 알리기 위해 메시지를 보낼 수 있다. 이런 동작과 관련된 메시지들을 정리하면 표 3과 같다.

이러한 시스템에 대해 우리는 HDTL로 명세를 기술할 수 있었다. 이는 Viewer 클래스에 대한 자연어 명세서에서 추출한 것이다. 그 문서에서는 Viewer 클래스, 혹은 그 클래스와 연결되어 동작하는 응용 프로그램들의 생명 주기를 정의하고 있다. 즉, 언제 초기화되고, 시작, 정지, 소멸되는지 등에 대한 사항이다. 이런 사항을 HDTL로 명세하면 다음과 같다.

표 3. DTV에서 사용되는 메시지들 (부분)

메시지 이름	메시지송신 클래스	메시지수신 클래스	의미
sendShutdown	Manager	Viewer	로드되었음을 알림
sendInit	Manager	Viewer	초기화하라는 지시
sendStart	Manager	Viewer	수행시작 지시
sendStop	Manager	Viewer	수행 정지 지시
sendShutdown	Manager	Viewer	프로그램 제거 지시
Paused	Viewer	Manager	프로그램이 정지되었음을 알림
Destroyed	Viewer	Manager	프로그램이 정지되었음을 알림

- 응용 프로그램이 init 된 후에는 start 되기 전에 stop 될 수 없다. (F1)

□ x.(msg(x)=sendInit →
○(¬y.(msg(y)=sendStop∧rcv(x)=rcv(y))
Uz.(msg(z)=sendStart∧rcv(x)=rcv(z))))

- start와 pause는 교대로 나타나야 함. (F2)

□x.(msg(x)=sendStart →
○(¬y.(msg(y)=sendStart∧rcv(x)=rcv(y))
U z.((msg(z)=sendStop∧rcv(x)=rcv(z))
∨(msg(z)=Paused∧rcv(x)=snd(z)))

□x.(msg(x)=sendStop →
○(¬y.((msg(y)=sendStop∧rcv(x)=rcv(y))
∨(msg(y)=Paused∧rcv(x)=snd(y))))
U z.(msg(z)=sendStart∧rcv(x)=rcv(z)))))

□x.(msg(x)=Paused →
○((¬y.((msg(y)=sendStop∧snd(x)=rcv(y))
∨ (msg(y)=Paused∧snd(x)=snd(y))))
U z.(msg(z)=sendStart∧snd(x)=rcv(z)))))

- destroy 명령은 바로 집행되어야 한다. (F3)

□x.(msg(x)=sendShutdown →
○□y.(¬(rcv(x)=snd(y)∨rcv(x)=rcv(y))))

첫 번째 요구사항과 두 번째 요구사항은 유사한 형식을 가진다. 이들은 모두, a라는 이벤트가 일어난 후에는 b라는 이벤트가 일어나기 전까지 c라는 이벤트의 발생을 금지하는 형태를 띈다. 즉 기존의 시제논리로 표현하면 □(a → ○(¬c U b))와 같은 형태이다. 하지만, 이 시스템과 같이 같은 클래스의 여러 인스턴스가 존재할 수 있을 경우에는, 메시지의 이름뿐이 아니라 메시지 교환에 참가하는 객체들간에도 관계가 있음이 명시되어야 한다.

예를 들어 첫 번째 명세를 보자. 두 번째 줄에 나타난 rcv(x)=rcv(y) 조건은 처음에 sendInit 메시지를 받은 객체(Viewer 클래스의 한 인스탄스)와 sendStop 메시지를 받는 객체가 같은 인스탄스여야 함을 의미한다. 다른 명세에서의 조건들도 마찬가지로 이해될 수 있다. 이상과 같은 명세들은 기존의 propositional 시제논리로는 표현할 수 없었던 것들이다. 변수를 사용하는 HDTL이기 때문에 이렇게 동적으로 변하는 시스템의 구성요소들을 이용하여 명세할 수 있게 되었다.

실험을 위해, 4장에서 제시한 알고리듬을 ML[17]로 구현하였다. ML은 강력한 타입 처리 기능과 함수적 프로그래밍을 지원하는 언어이다. 따라서 제시한 알고리듬과 거의 같은 구조를 가지는 프로토타입 분석기를 구현할 수 있었다. 이 분석기는 검사할 명세(HDTL 식)와, 시스템의 수행에서 생성된 이벤트 순차를 입력으로 받아, 그 순차

가 명세를 만족시키는지를 검사한다. 출력으로는 n번째의 이벤트를 받은 시점에서 그때까지의 순차가 명세를 만족시키는지를 참, 거짓으로 출력한다. 따라서 최종 출력은 이벤트 순차의 길이만큼의 bool 값의 리스트가 된다.

디지탈 TV 소프트웨어에서 이벤트들을 추출하기 위해서는 여러가지 방법이 있을 수 있다. 현 단계에서 우리는 [16]에서와 같이 모니터링하려는 프로그램을 직접 수정하여 수행시키는 방법을 사용했으나, 최종적으로는 자동적으로 모니터링할 수 있는 이벤트를 발생시킬 수 있어야 할 것이다. 최근까지는 별도의 프로그램을 이용하여야 가능했으나[18], JDK 1.3부터는 모니터링과 관련된 기능이 기본적인 라이브러리로 제공되기고 있다[19]. 본 논문에서 제시하는 방법은 이벤트 추출의 다음 단계에 해당되므로, 어떤 방법으로 이벤트 순차가 출력되었는지에 관계없이 적용될 수 있다. 우리는 생성된 이벤트 순차를 가지고 실험하여, 그것이 위에서 기술한 명세들을 만족함을 확인하였다.

이때 걸린 메모리 소비량을 제시하기 위해, 우리는 성능의 척도로 n번째 이벤트를 검사하는 시점에서 플로우 트리의 leaf nodes에 있는 conjunct 들의 수를 사용한다. 직관적으로, 각 conjunct 들은 n번째 이후의 이벤트들이 만족해야 할 요구사항을 나타내게 된다.

예를 들어 위의 F1에 대한 분석을 수행한다고 하면, 초기에는 하나의 conjunct, 즉 F1자체만이 플로우 트리에 존재하게 된다. 그러다가 x.(msg(x)=sendInit)를 만족시키는 이벤트, 즉 sendInit라는 메시지 ID를 가지는 이벤트가 들어오면, 새로이 다음과 같은 conjunct가 추가되게 된다.

$$(\neg y.(msg(y)=sendStop \wedge rcv(x)=rcv(y)) \; U \; z.(msg(z)=sendStart \wedge rcv(x)=rcv(z)))$$

즉 sendInit 이벤트의 발생이, 그 뒤에 시스템이 만족해야 할 추가적 요구사항을 생성하는 것으로 볼 수 있다. 다시 이 conjunct를 살펴보면, 의미상으로 이 요구사항은 해당하는 sendStart 이벤트가 발생하면 더 이상 검사하지 않아도 되는 것을 알 수 있다. 따라서 z.(msg(z)=sendStart∧rcv(x)=rcv(z)))를 만족시키는 이벤트가 발생하면 이 conjunct 는 사라지게 된다. 그림 4는 이러한 상황을 나타내고 있다. 그림에서 x축은 이벤트 순차 중에서의 순서를 나타내고, y축은 그때의 conjunct의 갯수를 나타낸다. 즉 식 F1과 같은 형식의 명세를 검사할때는 메모리 소비량은 늘었다 줄었다 하면서 어느 정도 일정한 경향을 보인다.

반면 F3와 같은 식은 다른 경향을 보인다. F3은 한 프로그램이 종료될 때, 즉 sendShutdown 메시지를 받은 후에는, 그 프로그램은 이벤트를 송수신하지 않는다는 것을 검사하는 것이다. 따라서, 한 프로그램이 종료할 때마다 검사해야 할 조건은 늘어나고, 이는 conjunct 수의 단조 증가로 이어진다. 따라서 이런 경우에는 시간과 연관된 휴리스틱의 도입을 고려할 필요가 있다. 이는 추후에 더 연구되어야 할 것이다.

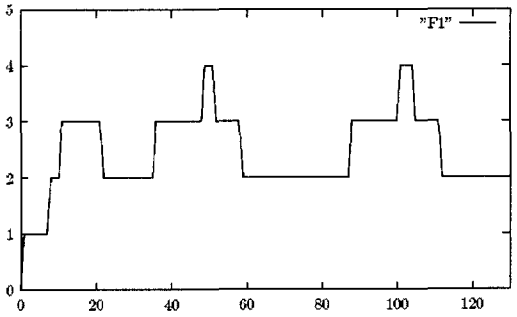


그림 4 Number of conjunct(F1)

6. 결 론

동적 시스템은 시스템을 이루는 구성요소들이 수행시간 중에 추가 혹은 삭제될 수 있는 시스템을 말한다. 이런 시스템에 대한 연구들은 주로 구현수준이나 동작 명세(operational specification)에 대해서 집중되어 있는 반면, 시스템의 특성(property)을 명세하는 쪽에 관련된 연구는 거의 없는 형편이다. 시스템의 특성 명세는 테스팅 등의 동적 분석에서 유용하게 쓰일 수 있으므로 꼭 필요하다고 할 수 있다.

우리는 이 논문에서 동적 시스템의 특성을 명세할 수 있는 새로운 시제논리언어인 HDTL을 제안했다. 이는 기존의 propositional logic 이나 first-order logic, 둘 중 어느 것과도 다른 표현력을 가진다. 기존에 실시간 시스템 명세를 위해 제안되었던 동결한 정자를 사용함으로써, 사용하기 쉽고 이해하기 쉬운 표현을 제안하고자 했다.

이렇게 제안된 명세를 이용하기 위해 우리는 이를 이용한 검증기법을 제안했다. 하지만 동적 시스템의 특성상 모델체킹과 같은 완전한 형태의 정형적 검증은 힘들기 때문에, 테스팅과 런타임 모니터링으로 응용될 수 있는 방법을 제안했다. 이는 상태순차가 입력으로 들어오는 경우, 입력을 받으면서 해석하여 오토마타를 확장하는 방식이다. 이 방식은 물론 기존의 모델체킹에 비해 시스템의 정확성을 증명할 수 없다는 문제가 있다. 하지만 이는 상태 공간이 무한하다는, 대상 시스템의 본질적 특징으로, 방법론의 문제라고 할 수는 없다.

우리는 제시한 알고리듬을 구현한 프로토타입 인터프

리터를 구현했다. ML[17]의 강력한 타입 처리 기능과 선언적 프로그래밍 스타일 덕분에 직관적인 프로그래밍이 가능했다. 이를 가지고 실험을 통해, 제시한 방법이 실제로 동작함을 보였다.

향후 가능한 연구방향으로는 다음과 같은 것들이 있다. 우선 명세 언어의 정교화가 필요하다. 현 단계에서 동적 분석에 사용되는 정보는 시스템 구성요소간에 전달되는 메시지만이다. 많은 경우 이것만으로도 시스템의 성질을 기술할 수 있으나 그렇지 못한 경우도 있을 수 있다. 이럴때 추가로 필요한 부분을 관측할 수 있는 장치가 필요하다. 그리고 플로우 트리의 규모를 제한하기 위한 휴리스틱의 개발이 필요할 수 있다. 마지막으로, 테스팅에 이 방법을 사용하려면, 테스트케이스의 생성이 중요한 문제가 될 수 있다. 자동으로 테스트케이스를 생성하는 연구가 필요할 것으로 보인다.

참 고 문 헌

[1] M.G. Hinchey and J.P. Bowen(eds.), *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, 1995.

[2] Zohar Manna and Amir Pnueli, *The temporal logic of reactive and concurrent systems - Specification*, Springer-Verlag, 1992.

[3] D. Harel, "On Visual Formalisms," *Communications of ACM*, pp. 514-530, May 1988.

[4] J. Peterson, *Petri-net theory and the modeling of systems*, Prentice Hall, 1981.

[5] C. A. R. Hoare, *Communicating sequential processes*, Prentice Hall, 1985.

[6] A. Emerson, "Temporal and Model Logic," *Handbook of Theoretical Computer Science*, Chap 16, edited by J. van Leeuwen, Elesevier Science Publishers, 1990.

[7] J. Bohn, W. Damn, O. Grumberg, H. Hungar, K. Laster, "First-order-CTL model checking," *FSTTCS 98, LNCS 1530*, 1998.

[8] Rajeev Alur and Thomas A. Henzinger, "A Really Temporal Logic," *The Journal of the ACM*, Vol 41, pp. 181-204, 1994.

[9] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of ACM*, vol. 12, no. 10, Oct. 1969.

[10] A. Pnueli, "The temporal logic of programs," In *Proc. 18th IEEE Symposium on Foundation of Computer Science*, 1977.

[11] Laura K. Dillon and Y. S. Ramakrishna, "Generating Oracles from Your Favorite Temporal Logic Specifications," *Proc. 4th ACM SIGSOFT Symp. Foundations of Software Engineering*, San Francisco, pp. 106-117, October 1996.

[12] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple On-The-Fly Automatic Verification of Linear Temporal Logic," *Pro. of Symposium on Protocol Specification, Testing, and Verification (PSTV95)*, pp. 3-18, June 1995.

[13] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. 3, no. 2, Nov. 1977.

[14] B. Alpern and F. B. Schneider, "Defining liveness," *Information processing letters*, vol. 21, pp. 181-185, 1985.

[15] A. P. Sistla, "Safety, liveness and fairness in temporal logic," *Formal Aspect of Computing*, vol. 6, pp. 495-511, 1994.

[16] "SEC DTV 소프트웨어의 통합 테스팅 자동화 도구 개발, 연구결과보고서", 한국과학기술원 전자전산학과 소프트웨어공학 연구실, 2000.

[17] L. C. Paulson, *ML for the working programmer*, Cambridge University Press, 1996.

[18] Moonjoo Kim, Mahesh Viswanathan, Hanene Ben-Abdallah, Sampath Kannan, Insup Lee and Oleg Sokolsky, "Formally Specified Monitoring of Temporal Properties", *Proc. of European Conference on Real-Time Systems*, York, UK, June 9-11, 1999.

[19] Java Platform Debugger Architecture, http://java.sun.com/j2se/1.3/docs/guide/jpda/index.html

조 승 모
정보과학회논문지 : 소프트웨어 및 응용
제 29 권 제 5 호 참조

김 형 호
정보과학회논문지 : 소프트웨어 및 응용
제 29 권 제 5 호 참조

차 성 덕
정보과학회논문지 : 소프트웨어 및 응용
제 29 권 제 5 호 참조

배 두 환
정보과학회논문지 : 소프트웨어 및 응용
제 29 권 제 5 호 참조