실시간 객체 모델의 다중 스레드 구현으로의 스케줄링을 고려한 자동화된 변환

(Automated Schedulability-Aware Mapping of Real-Time Object-Oriented Models to Multi-Threaded Implementations)

홍 성 수 †

(Sung Soo Hong)

요 약 실시간 시스템이 복잡해짐에 따라 이를 개발하는 과정에서 객체 지향 설계 방법론과 이를 지원하는 CASE 도구들이 널리 사용되고 있다. 그러나 이런 객체 지향 CASE 도구를 사용할 경우, 설계자들은 별도의 과정으로 객체 중심으로 설계된 모델을 실제 수행되는 주체인 태스크로 변환시켜야 한다. 불행하게도 객체 모델과 태스크는 특성이 근본적으로 다르고, 스케줄 가능성을 분석하기가 어렵기 때문에 이러한 과정을 자동화하기는 매우 어렵다. 이 문제를 해결하기 위하여 많은 CASE 도구에서는 개발자가 직접 수동으로 객체를 태스크로 변환시키도록 요구하고 있다. 결과적으로 개발자들은 자신의 경험을 바탕으로 하여 임시 변통적인 방법에 의존하여 태스크를 유도하고 있다. 유도된 태스크 집합은 결과 시스템의 스케줄 가능성에 직접적으로 중요한 영향을 미친다.

본 논문에서는 실시간 객체 지향 설계 모델을 스케줄 가능성을 고려해 다중 스레드 구현으로 자동적으로 변환하는 방법을 제안한다. 본 논문에서 태스크는 다른 주기와 종료시한을 갖는 상호 배타적인 트랜잭션들로 이루어진다. 이러한 새로운 태스크 모델에 대하여 스케줄 가능성 분석 알고리즘을 제시한다. 또한 제안된 방법을 지원하기 위하여 런 타임 시스템과 코드 생성이 어떻게 지원되어야 하는지에 대하여 설명한다. 사례 연구는 단일 태스크 매핑의 부적절성을 보여줌과 함께, 다중 태스크를 수동으로 유도하는 것이 매우 어렵고, 제안된 방법이 실질적으로 유용하다는 것을 명백하게 보여준다.

키워드 : 실시간 객체지향 모델링, UML(Unified Modeling Language), 실시간 스케줄링, 선점임계, 다중스레딩

Abstract The object-oriented design methods and their CASE tools are widely used in practice by many real-time software developers. However, object-oriented CASE tools require an additional step of identifying tasks from a given design model. Unfortunately, it is difficult to automate this step for a couple of reasons: (1) there are inherent discrepancies between objects and tasks; and (2) it is hard to derive tasks while maximizing real-time schedulability since this problem makes a non-trivial optimization problem. As a result, in practical object-oriented CASE tools, task identification is usually performed in an ad-hoc manner using hints provided by human designers. In this paper, we present a systematic, schedulability-aware approach that can help mapping real-time object-oriented models to multi-threaded implementations.

In our approach, a task contains a group of mutually exclusive transactions that may possess different periods and deadline. For this new task model, we provide a new schedulability analysis algorithm. We also show how the run-time system is implemented and how executable code is generated in our frame work. We have performed a case study. It shows the difficulty of task derivation problem and the utility of the automated synthesis of implementations as well as the inappropriateness of the single-threaded implementations.

Key words : real-time object-oriented modeling, unified modeling language(UML), real-time *scheduling, preemption threshold, multi-threading*

† 종신회원 : 서울대학교 전기컴퓨터공학부 교수
sshong@redwood.snu.ac.kr

논문접수 : 2001년 2월 19일
심사완료 : 2001년 12월 27일

1. 서 론

실시간 내장형 시스템은 더 높은 안정성과 신뢰성 및 성능을 요구받으면서, 점점 더 복잡하고 정교해지고 있다. 이에 따라 실시간 시스템을 개발하는 과정에서 설계자들이 체계적인 소프트웨어 설계 기법을 사용하는 것이 불가피해졌다[1]. 소프트웨어 설계 방법론은 1960년대부터 발전되어 왔으며, 현재 다양한 방법론이 존재하고 있다. 소프트웨어 설계 방법론에 채택되는 정책에 따라 실시간 시스템 설계를 구성하는 소프트웨어 구성 요소(component)의 구조(structure)와 행동 양식(behavior)이 결정된다[2]. 이러한 정책은 (1) 태스크 기반 설계 정책과 (2) 객체 지향 설계 정책의 두 가지로 분류할 수 있다.

DARTS(Design Approach for Real-Time Systems)는 잘 알려진 태스크 기반 소프트웨어 설계 방법론 중의 하나이다. DARTS에서는 주어진 요구 사항 명세의 구조화된 분석(Structured analysis)으로부터 태스크를 얻어내는 것에 가장 큰 중점을 둔다[2]. 결과적으로 DARTS에서 태스크 기반의 설계를 구현으로 변환하는 것은 매우 쉬운 일이다.

태스크 기반 설계 방법론과 달리, ROOM(Real-Time Object-Oriented Modeling)[3]과 UML-RT[4]와 같은 객체 지향 설계 방법론에서는 태스크의 식별을 시스템 설계에서 더 나중의 단계로 미루어 둔다. 객체 지향 설계 방법론에서는 실시간 시스템을 메시지를 통하여 통신하는 동시적이고 능동적인 객체들의 집합으로 본다. 객체 지향 설계 방법론은 태스크 기반 설계 방법론에 비하여 탁월한 장점을 가지고 있다. 우선, 설계자들이 시스템의 상위 수준의 추상화된 특성을 뽑아내는데 집중할 수 있게 해준다. 이는 상위 수준 시스템 특성을 나타낼 수 있는 객체 지향 모델링 언어를 사용하여 가능하다. 또한 객체 지향 설계 기법은 실시간 시스템의 설계 모델을 구체적인 구현으로부터 분리할 수 있게 해준다. 이는 시간에 따라 발전할 수 있는 복잡한 실시간 시스템에 대한 소프트웨어 유지 및 보수를 쉽게 해주고, 소프트웨어 재사용, 컴포넌트 기반 코딩을 가능하게 해준다. 객체 지향 설계 방법론이 인기를 얻음에 따라 이를 지원하는 CASE 도구들도 또한 나타났다. 설계자는 CASE 도구를 통해 실시간 시스템을 설계하고 실행 가능한 모델을 통해 분석하고, 시스템을 위한 실행 코드를 생성할 수 있다.

그러나 현재의 실시간 시스템을 위한 객체 지향 CASE 도구들은 중요한 결점을 하나 가지고 있다. 객체 지향 설계 방법론에서는 기반 런 타임 시스템이 능동적인 객체를 직접적으로 지원하지 않을 때, 객체 지향 설계 모델을 구현으로 변환하는 방법이 명확하지 못하다. 이는 대부분의 실시간 운영체제에 해당한다. 왜냐하면 객체 기반의 커널은 간접 포인터와 객체 참조들에 의하여 매우 느려지는 경향이 있기 때문이다. 따라서 객체 지향 CASE 도구에서는 태스크를 식별하는 별도의 과정이 필요하다. 불행하게도, 이 과정을 자동화하는 것은 몇 가지 이유로 인하여 매우 어렵다: (1) 객체와 태스크는 그 특성에 있어서 근본적으로 차이가 있으며, (2) 실시간 스케줄 가능성을 최대화하면서 태스크들을 유도하는 것은 간단하지 않은 최적화 문제이다. 한편 유도된 태스크 집합은 결과 시스템의 스케줄 가능성에 직접적으로 중요한 영향을 미친다. 이 문제를 해결하기 위하여 많은 객체 지향 CASE 도구에서는 개발자가 직접 수동으로 객체를 태스크로 변환시키도록 요구하고 있다. 결과적으로 개발자들은 자신의 경험을 바탕으로 하여 임시 변통적인 방법에 의존하여 태스크를 유도하고 있다.

실시간 객체 지향 모델의 자동화된 구현과 스케줄 가능성 테스트에 관한 연구는 [5, 6, 7]과 같은 연구에서 이미 다루어졌다. 그러나 이러한 연구들은 객체에서 태스크로의 변환이 이루어진 상태에서만 스케줄 가능성을 분석하며, 스케줄 가능성을 고려한 상태에서 객체를 태스크로 변환하는 방법을 제시하지는 못했다.

본 논문에서는 실시간 객체 지향 모델로부터 스케줄 가능성을 고려해서 자동적으로 구현을 생성하는 방법을 제시한다. 또한 이 방법을 현재 널리 사용되고 있는 CASE 도구에 어떻게 적용할 수 있는지 설명한다. 설계 모델로부터 태스크를 추출함에 있어서 트랜잭션(transaction)의 개념을 사용한다. 트랜잭션은 외부 입력으로부터 하나의 외부 출력을 내기까지의 양 끝단 수행을 나타낸다. 실시간 객체 모델에서 트랜잭션은 수행 과정에서 발생하는 이벤트의 연결로 표현할 수 있다. 본 논문에서는 실시간 객체 모델로부터 트랜잭션을 추출하고, 스케줄 가능성을 고려하여 트랜잭션을 묶어서 태스크로 변환시키는 방법을 제안한다. 또한 이렇게 생성된 태스크의 수를 줄이기 위하여 [8, 9]에서 제시된 선점 임계(preemption threshold) 스케줄링을 적용하였다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문의 구현 모델의 재료가 되는 UML-RT 설계 모델과, 이를 지원하는 CASE 도구에 관해 간략히 살펴본다. 3장에서는 본 논문에서 제시하는 접근 방법에 관해 자세히 다룬다. 4장에서는 제안된 방법을 지원하기 위한 구현의 구조와 함께, 실제 CASE 도구가 제안된 방법을 지원하기 위하여 어떻게 구현될 수 있는지를 설명한다. 5장에

서는 제안된 방법의 응용을 보여주는 사례 연구를 제시하고, 6장에서 논문의 결론을 맺는다.

2. UML-RT와 CASE 도구의 개관

이 장에서는 제안된 방법의 소스 프로그래밍 언어로 선택한 UML-RT와 UML-RT를 지원하는 CASE 도구인 RoseRT에 관해 간략히 설명한다.

2.1 UML-RT 개관

UML(Unified Modeling Language)[10]은 소프트웨어 시스템을 명세하고, 시각화시키며, 구성하고, 문서화하며 또한 수행까지 시키기 위하여 사용될 수 있는 그래픽 언어이다. UML은 실제 대형 시스템에서도 성공적으로 적용되어 유용성을 입증 받았으며, 모델링 언어의 산업 표준으로 인정받고 있다. UML-RT(UML Real-Time)는 UML에 이벤트 중심이며 분산처리의 가능성이 있는 시스템에 필요한 새로운 개념을 추가한 것이다. UML-RT의 가장 기본적인 모델링 요소는 캡슐(capsule)이다. 캡슐은 동시적이고 독립적인 제어권을 갖는 수행 가능한 능동적인 객체이다. 캡슐들은 포트(port)라 불리는 인터페이스 객체를 통해서 메시지를 주고받음으로써만 다른 캡슐들과 통신할 수 있다. 각 캡슐의 구조 모델은 자신이 포함하는 캡슐들과 그 캡슐들의 포트들 간의 연결로 이루어진다. 각 캡슐의 행동 모델은 FSM(Finite State Machine)으로 기술된다. 캡슐의 포트에 메시지가 도착하면 FSM에 따라 상태 천이가 발생하며 상태 천이에 해당하는 액션(action)이 실행된다. 이하 여기에서 설명한 UML-RT의 용어를 사용한다.

2.2 RoseRT CASE 도구

RoseRT는 객체 지향 실시간 시스템 모델을 설계할 수 있는 시각적인 모델링 환경을 제공한다. 또한 RoseRT는 설계된 모델로부터 자동적으로 실행 가능한 프로그램을 생성할 수 있도록 해준다. 생성된 프로그램은 응용 프로그램 코드와 런 타임 시스템(RTS, Run-Time System) 라이브러리로 구성된다. 런 타임 시스템 라이브러리의 핵심 구성요소는 controller 객체로 스레드마다 존재하며, 각 캡슐 개체(instance)에 전달되는 메시지들을 관리하는 역할을 한다. controller 객체는 메시지 큐에서 가장 높은 우선 순위의 메시지를 뽑아 내어 해당 캡슐 개체에 전달한다.

그러나 RoseRT는 불필요한 블록킹 원인을 가지는 응용 프로그램을 생성한다. Saksena 등은 [6]에서 UML-RT로 설계된 응용 프로그램에서 블록킹 시간의 원인이 되는 우선 순위 역전의 원인들을 (1) 메시지와 스레드의 이중 스케줄링, (2) 스레드 간 메시지 전달, (3) 완전 수행(run-to-completion)의 세 가지로 열거하였다. 또한 이들 블록킹 원인들을 피하거나 최소화하기 위한 안내 지침을 제안하였다. 본 논문에서 제안된 방법에서는 스레드 간의 메시지 전달이 발생하지 않는다. 따라서 본 논문의 제안된 방법은 두 번째 안내 지침을 제외한 나머지 안내 지침을 따른다.

3. 제안된 변환 방법

이 장에서는 임의의 주어진 실시간 객체 지향 설계 모델로부터 스케줄링 가능한 다중 스레드 구현을 자동적으로 생성하는 방법을 구체적으로 설명한다. 그림 1에 3 단계로 구성된 제안된 방법의 개관을 나타내었다. 각 단계에서는 (1) 트랜잭션, (2) 논리적 스레드, (3) 물리적 스레드를 추출한다. 첫 번째 단계에서는 설계 모델로부터 트랜잭션을 추출한다. 두 번째 단계에서는 상호 배타적인 트랜잭션을 묶어서 태스크로 병합하고, 각 태스크에 우선 순위와 선점 임계값을 스케줄링 가능성을 고려해 할당한다. 이 단계에서의 태스크를 최종 구현을 구성하는 "물리적 스레드"와 구별하기 위하여 "논리적 스레드"라고 부른다. 맨 마지막 단계에서 논리적 스레드를 우선 순위와 선점 임계값에 따라서 비선점 그룹으로 묶는다. 각 비선점 그룹이 물리적 스레드가 된다.

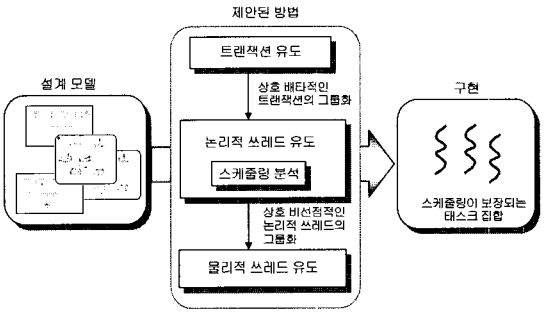

그림 1 제안된 방법의 개관

3.1 설계 모델

설계 모델에는 입력 이벤트와 출력 이벤트의 집합이 있다. 각 외부 입력은 이벤트를 연속적으로 발생시키고, 이에 따라 일련의 작업이 수행되게 되는데, 본 논문에서 이 일련의 작업을 트랜잭션(transaction)이라고 정의한다. UML-RT에서 하나의 외부 입력 이벤트는 여러 개의 트랜잭션을 발생시킬 수 있다. 설계 모델에 대하여 다음과 같은 몇 가지 가정이 있다: (1) 모든 외부 입력은 주기적이다. (2) 트랜잭션의 종료 시한은 주기와 같

다. 이는 UML-RT에서 양 끝단 종료 시한을 명세하는 방법이 없기 때문이다. (3) 메시지의 우선 순위는 설계 시점에서 아니라 구현 시점에 결정된다. 본 제안 방법의 목적은 설계 모델을 반복적으로 세밀하게 조율하지 않고, 제안된 방법을 통해 모든 메시지들을 스케줄 가능하게 만들고자 하는 것이다. (4) 같은 캡슐에서 시작한 트랜잭션들은 상호 배타적으로 수행된다. 대부분의 시스템이 실질적으로 이 가정을 따른다. 예를 들어 조작자 계기판의 버튼을 통해 조작자로부터 명령을 받아들이는 로봇 명령 해석기 시스템을 생각해 보자. 만약 명령이 "수동 모드 시작"이면 이 시스템은 로봇을 수동 모드로 설정한다. 만약 명령이 "자동 모드 시작"이면 로봇을 자동모드로 설정하는 다른 트랜잭션을 시작하게 된다. 즉 같은 능동적 객체에서 시작한 트랜잭션이 상호 배타적으로 수행되는 것을 볼 수 있다.

본 논문에서 사용하는 표기법은 표 1과 같다.

표 1 표기법

표기법	설명
τ_i	트랜잭션
A_j^i	트랜잭션 τ_i를 이루는 액션
$\langle A_1^i, A_2^i, \ldots, A_{n_i}^i \rangle$	액션의 연속으로서의 트랜잭션 τ_i
$O(A_j^i)$	액션 A_j^i를 포함하는 능동적 객체
$C(A_j^i)$	액션 A_j^i의 최악 수행 시간
$C(\tau_i)$	트랜잭션 τ_i의 최악 수행 시간, $C(\tau_i)=\sum_j C(A_j^i)$로 가정한다.
$T(\tau_i)$	트랜잭션 τ_i의 주기
$\pi(\tau_i)$	트랜잭션 τ_i의 우선순위
$\gamma(\tau_i)$	트랜잭션 τ_i의 선점 임계값

3.2 1단계: 트랜잭션의 유도

제안된 방법은 트랜잭션의 유도를 위하여 "메시지 연속 트리"라고 명명한 중간단계의 트리 구조를 생성한다. 메시지 연속 트리에서 각 노드는 액션을 나타낼 수 있으며 각 모서리는 메시지의 흐름을 나타낸다. 구체적으로, 한 노드는 액션이나 메시지의 동시 발생(conjunction) 또는 선택적 발생(disjunction)을 나타낸다. 액션 노드는 AND-action, OR-action과 LEAF-action 노드로 분류된다. AND-action 노드는 왼쪽에서 오른쪽으로 모든 밖으로 나가는 메시지를 차례대로 보낸다. OR-action 노드는 액션의 조건에 따라 밖으로 나가는 메시지 중 단지 한 메시지만을 보낸다. LEAF-action 노드는 밖으로 나가는 메시지를 가지고 있지 않다. 액션이 메시지들을 중첩된 형태로 보낼 경우 교량(bridge) 노드가 사용된다. 액션 노드와 마찬가지로 교량 노드는 AND-bridge, OR-bridge, LEAF-bridge로 분류된다. 그림 2는 메시지 연속 트리의 예를 보여준다.

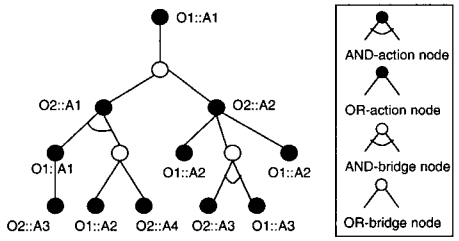

그림 2 메시지 연속 트리

제안된 방법은 트랜잭션을 두 단계로 유도한다: (1) 외부 이벤트에 의하여 야기된 각 액션으로부터 트리를 구성한다. (2) 각 트리로부터 트랜잭션을 유도한다.

(1) 외부 이벤트에 의하여 야기된 각 액션으로부터 시작하는 트리의 생성: 트리를 생성하는 규칙이 다음과 같다.

- 루트 노드는 반드시 액션 노드이어야 한다.
- 액션의 경로 표현(path expression)이 생성되도록 액션의 코드 구조를 분석한다. 액션의 생성된 경로 표현 P에 대하여
  - 만약 P가 보내는 메시지를 포함하고 있지 않다며, 그 노드는 LEAF 노드가 된다.
  - 그렇지 않다면,
    - * 만약 $P=\wedge P_i\ (i=1,\ldots,n)$ 이면, 그 노드는 각 P_i에 대하여 n개의 자식 노드를 가진 AND-action 노드가 된다.
    - * 만약 $P=\vee P_i\ (i=1,\ldots,n)$ 이면, 그 노드는 각 P_i에 대하여 n개의 자식 노드를 가진 OR-action 노드가 된다.
    - * 각 P_i의 자식 노드에 대하여
      - 만약 $P_i=\wedge P_{i,j}\ (i=1,\ldots,n)$ 이면, 그 노드는 AND-bridge 노드가 된다.
      - 만약 $P_i=\vee P_{i,j}\ (i=1,\ldots,n)$ 이면, 그 노드는 OR- bridge 노드가 된다.
    - * 위의 규칙들은 중첩된 형태인 $P_{i,j,k,\ldots}$에 같은 방식으로 적용된다.
- 만약 메시지가 (받는 객체의 상태에 따라) 다른 액션들을 일으킬 수 있다면, 받는 객체의 상태에 따라서 OR-bridge 노드가 생성되고 메시지를 보내는

노드에 연결된다.

(2) 각 트리로부터 트랜잭션을 유도하기: 각 메시지 연속 트리에 대하여 루트 노드로부터 시작하여 노드들을 추적함으로써 액션의 연속을 유도한다. 각 액션의 연속이 트랜잭션이 된다. 노드를 추적함에 있어서,

- AND-node를 만나면 노드의 각 모서리에 대하여 왼쪽에서 오른쪽으로 "깊이 우선 순서"로 추적한다.
- OR-node를 만나면 노드로의 각 모서리에 대하여 다른 연속을 생성한다.

그림 2에서는 5개의 트랜잭션이 유도된다: <O1::A1, O2::A1, O3::A1, O2::A3, O1::A2:>, <O1::A1, O2::A1, O3::A1, O2::A3, O2::A4>, <O1::A1, O2::A2, O3::A2>, <O1::A1, O2::A2, O2::A3, O1::A3>, <O1::A1, O2::A2, O1::A2>.

3.3 2 단계: 논리적 스레드의 추출

논리적 스레드의 추출은 3 단계로 이루어진다. 각 논리적 스레드에 대해서 (1) 구성하는 트랜잭션들을 찾아내고, (2) 우선 순위를 결정하며, (3) 최대 선점 임계값을 할당한다.

(1) 트랜잭션의 그룹화: 논리적 스레드의 구성원을 찾는 것은 동시에 실행이 불가능한 트랜잭션들을 찾는 것으로 이루어진다. 3.1절의 가정에 의하여 외부 이벤트를 처음으로 받는 객체를 공유하는 트랜잭션들을 하나의 논리적 스레드로 묶을 수 있다. 논리적 스레드는 $L_i=\{\tau_j^i \mid j=1,2,\ldots\}$로 나타낸다. 논리적 스레드는 여러 개의 트랜잭션의 집합이므로 여러 개의 주기, 최악 수행시간, 완전 수행 블록킹 시간을 가질 수 있다.

(2) 스케줄 가능한 우선 순위 할당: 우선 순위 할당에는 Audsley의 알고리즘[11]을, 스케줄 가능성 테스트에는 응답 시간 분석[12]을 채용하여 사용한다. 논리적 스레드가 여러 개의 스케줄링 특성을 가지고 있는 점을 반영하여 기존의 알고리즘을 확장하였다. 논리적 스레드 L_i의 응답시간 $R(L_i)$는 다음의 식을 사용하여 계산할 수 있다.

$$\beta(\tau_i) = \max_{L:\pi(\tau_i)<\pi(\tau_j),\ k:\pi(\tau_k)\le\pi(\tau_i)} \left\{ \max_{k,m} \{C(A_k^i : O(A_k^i)=O(A_m^k)\} \right\}$$

$$C^R(L_i) = C(\tau_j^i) :: \max_j \{C(\tau_j^i)+\beta(\tau_j^i)\}$$

$$\beta(L_i) = \beta(\tau_j^i) :: \max_j \{C(\tau_j^i)+\beta(\tau_j^i)\}$$

$$C^I(L_i) = C(\tau_j^i) :: \max_j \{C(\tau_j^i)/T(\tau_j^i)\}$$

$$T(L_i) = T(\tau_j^i) :: \max_j \{C(\tau_j^i)/T(\tau_j^i)\}$$

$$R(L_i) = \beta(L_i) + C^R(L_i) + \sum_{\pi(L_j)>\pi(L_i)} \lceil \frac{R(L_i)}{T(L_j)} \rceil \cdot C^I(L_j)$$

여기에서 $\beta(\tau_i)$는 트랜잭션 τ_i의 블록킹 시간을 나타낸다. 트랜잭션 기반의 구현 구조는 각 능동 객체가 완전 수행(run-to-completion) 의미 체계를 지킬 수 있도록 하기 위하여 뮤텍스(mutex)를 사용하여야 한다. 합리적으로 완전 수행 블록킹 시간을 제한시키기 위하여, 제안된 방법은 PCPE(PCP Emulation) 프로토콜[13]을 채용하여 사용한다. PCPE에서는 태스크가 임계 영역 안에서 수행할 때 그 태스크의 우선 순위를 무조건적으로 그 태스크가 소유하고 있는 뮤텍스의 priority ceiling 값 중 최대값으로 증가시킨다. 각 뮤텍스의 priority ceiling 값은 그 뮤텍스가 속한 능동 객체에 보내지는 메시지들의 선점 임계값의 값 중에서 최고의 값으로 할당한다. 이 단계에서는 각 논리 스레드의 선점 임계값이 우선 순위와 같은 값을 가진다. PCPE 프로토콜을 사용하면 트랜잭션은 수행을 시작하려 할 때에 단 한 번만 블록킹 당할 수 있다.

$R(L_i)$는 $C^R(L_i)$와 $\beta(L_i)$가 증가함에 따라 단조 증가하므로 $C^R(L_i)$와 $\beta(L_i)$의 값을 논리적 스레드의 트랜잭션 중 수행 시간과 완전 수행 블록킹 시간을 최대로 하는 트랜잭션의 수행 시간과 블록킹 시간으로 한다. 반면에 L_i에 대한 높은 우선 순위 작업에 의한 간섭 시간은 $C^I(L_j)/T(L_j)$에 따라 단조 증가하므로 $C^I(L_j)$와 $T(L_j)$의 값을 주기 대 수행시간이 최대가 되는 트랜잭션의 수행 시간과 주기로 한다.

(3) 최대 선점 임계값 할당: 높은 선점 임계값을 갖는 태스크일수록 더 적은 문맥 교환 횟수를 갖게 되므로 각 논리적 스레드에 가능한 최대의 선점 임계값을 할당하도록 한다. 제안된 방법에서는 최대 선점 임계값 할당 알고리즘으로 [9]에서 제시된 방법을 채용하여 사용한다. 또한 선점 임계값을 고려하여 [9, 12]의 스케줄 테스트 알고리즘을 변형하여 사용하며, 본 논문에서 제시한 논리적 스레드의 특성을 고려하여 [9]에서 제시된 분석 방법에 변경을 가하였다. 선점 임계값 $\gamma(L_i)$를 가진 논리적 스레드 L_i의 응답시간 $R(L_i)$는 다음과 같은 식에 의해 계산될 수 있다.

$$B(L_i) = \max_j \{ C^I(L_i) :: \gamma(L_j) \ge \pi(L_i) > \pi(L_j) \}$$

$$S(L_i) = B(L_i) + \sum_{\pi(L_j)\ge\pi(L_i)} \left(1 + \lfloor \frac{S_i}{T(L_j)} \rfloor \right) \cdot C^I(L_j)$$

$$R(L_i) = F(L_i) = S(L_i) + C^R(L_i) + \sum_{\forall L_j \pi(L_j)>\gamma(L_i)} \left\{ \lceil \frac{F(L_i)}{T(L_j)} \rceil - \left(1 + \lfloor \frac{S(L_i)}{T(L_j)} \rfloor \right) \right\} \cdot C^I(L_j)$$

이 수식에서 $B(L_i)$는 낮은 우선 순위를 가졌지만, 높은 선점 임계값을 가진 태스크에 의해 블록된 시간을

나타낸다. $S(L_i)$와 $F(L_i)$는 각각 논리적 스레드의 최악 시작 시간과 종료 시간을 나타낸다. 완전 수행에 의한 블록킹($\beta(L_i)$)과 선점 임계값에 의한 블록킹($B(L_i)$)은 항상 논리적 스레드(트랜잭션)가 수행을 시작할 때만 발생한다. 또한 둘 중의 한 가지만 발생하며, $B(L_i)$가 $\beta(L_i)$보다 항상 크기 때문에 위의 수식에서와 같이 블록킹 시간은 $B(L_i)$만 사용된다.

트랜잭션과 트랜잭션 내의 각 액션은 자신을 포함하는 논리적 스레드의 우선 순위와 선점 임계값을 그대로 이어 받게 된다. 결과적으로 실제 구현에 있어서는 액션을 발생시키는 메시지가 논리적 스레드의 우선 순위와 선점 임계값을 할당받게 된다.

3.4 3단계: 물리적 스레드 추출

두 개의 논리적 스레드 L_i와 L_j는, 만약 $\pi(L_i) \geq \gamma(L_j)$이고 $\pi(L_j) \geq \gamma(L_i)$이면 서로 선점 불가능하다[9]. 이러한 관계를 이용하여 각 쌍이 모두 서로 선점 불가능한 비선점 그룹을 쉽게 만들 수 있다. 제안된 방법에서는 각 비선점 그룹을 하나의 물리적 스레드로 사상(mapping)한다. 이는 스레드의 개수를 크게 줄일 수 있으며, 이에 따라 실행 시 문맥 교환 시간과 정적인 메모리 요구량을 줄일 수 있다.

4. 구현 구조와 도구 지원

RoseRT가 캡슐 단위로 스레드에 사상(mapping)하는 반면, 본 논문의 제안된 방법에서는 각 메시지를 스레드에 사상시킨다. 제안된 방법은 RoseRT 런 타임 시스템과 코드 생성을 최소한으로 변형시킴으로써 구현될 수 있으므로, RoseRT에 제안된 방법을 구현하는 것은 매우 쉬운 일이다.

4.1 구현 구조

그림 3은 메시지의 데이타 구조를 나타낸다. 그림에서 나타낸 바와 같이, 메시지는 목적 캡슐 개체에 대한 참조와 함께 자신을 처리할 스레드에 대한 참조를 가진다. 그림 4는 각 스레드에 대한 기본 코드 구조를 보여준다. 각 스레드는 외부 이벤트(메시지)가 전달되기를 기다린다. 외부 이벤트가 전달되면, 스레드는 새로운 트랜잭션을 시작한다. 그림 4 (a)의 내부 while 루프는 트랜잭션의 액션을 호출함으로써 이벤트를 처리한다. 그림 4 (b)는 각 액션의 수행이 그 액션을 수행하는 능동 객체를 보호하는 뮤택스에 의하여 연속적으로 이루어지는 것을 보여준다. 이는 각 능동 객체의 상태 변화에 대한, UML-RT의 완전 수행 의미 체계에 의해서이다. 그림 4 (c)는 FSM으로 표현되는 능동 객체의 행동을 구현하는 함수(Behavior())의 의사 코드를 보여준다. 이 함수는 자신이 속한 능동 객체로부터 적절한 액션을 선택한다. 이는 능동 객체의 현재의 상태에 따라서, 각 액션에 해당하는 포트와 시그널 종류를 전달된 메시지의 것과 매칭시킴으로써 이루어진다. 만약 매칭되는 액션이 발견되지 않으면 전달된 메시지는 버려진다. 이러한 각 능동 객체를 위한 코드는 코드 생성기가 합성한다.

```
class Message {
    Message*    next;

    Thread      target_thread;
    Capsule     target_object;
    Port        port;

    Short       signal;
    void*       data;

    int         priority
}
```

그림 3 메시지 데이타 구조

```
Thread::mainLoop() {
    while(1) {
        wait_for_external_event();
        while(message queues are not empty) {
            foreach(Message M)
                dispatch(M);
        }
    }
}
```

(a)

```
Thread::dispatch(Message M) {
    M.target_object.mutex->enter();
    M.target_object->Behavior( M.signal, M.port );
    M.target_object.mutex->leave();
}
```

(b)

```
An_object::Behavior(int _signal, int _port ) {
    switch ( currentState ) {
    case 0:
        switch( _port ) {
        case 0:
            switch ( _signal ) {
            case 0:
                chain1_t0();
                ...
            }
            ...
        }
        ...
    }
}
```

(c)

그림 4 각 스레드의 메인 루프의 의사 코드

4.2 도구 지원

(1) 런 타임 시스템 라이브러리: UML-RT 메시지 데이타 구조를 확장하여 메시지가 속한 논리적 스레드의 정보를 포함할 수 있도록 하였다. 이는 메시지의 우선 순위, 선점 임계값, 논리적 스레드가 속한 물리적 스레드에 대한 포인터를 포함한다. 이와 함께 메시지 전송 함수의 인자와 내부 루틴에서 대상 controller 객체를 찾는 방법을 변경하였다. 각 능동 객체에 대하여서는 뮤택스를 정의하고 메시지 디스패칭 루틴에서 이 뮤택스를 사용하도록 수정하였다. 그리고 Controller 객체를 수정하여 전송된 메시지의 우선 순위와 선점 임계값에 따라 스레드의 우선 순위가 동적으로 변할 수 있도록 하였다.

(2) 코드 생성기: 제안된 방법에서 유도된 물리 스레드를 생성할 수 있도록 코드를 생성하며, 각 호출되는 메시지 전달 함수에 인자로서 제안된 방법에서 얻어진 논리적 스레드의 정보를 채워 넣는다. 또한 각 능동 객체의 개체의 뮤택스에 priority ceiling 값을 초기화시킨다.

표 2 축구 로봇 시스템에서의 액션의 특성 (시간 단위: ms)

캡슐	상태 천이	액션	액션에서 보내진 메시지	주기	WCET
Communication	Ready → Ready	timeout	start, stop	500	4.0
Vision	Ready → Ready	timeout	visionData	400	90.0
Location	Ready → Ready	visionData	ballLost, ballFound	-	5.0
	Ready → Ready	requestLocation	location	-	3.0
Motor	Ready → Ready	timeout	movement	5	0.5
	Ready → Ready	setSpeed	-	-	0.5
PathTracker	Ready → Ready	movement	setSpeed	-	0.5
	Ready → Ready	setPath	-	-	1.0
RobotControl	Standing → Search:Ready	start	-	-	0.1
	Search, Shoot → Standing	stop	setPath	-	0.1
	Search → Shoot:Ready	ballFound	-	-	0.1
	Shoot → Search:Ready	ballLost	-	-	0.1
	(Any state) → Search:Ready	entry action	requestLocation	-	0.1
	Search:WaitTimer→ Search:Ready	Search:timeout	-	200	0.2
	Search:Ready→ Search:WaitTimer	location	setPath	-	2.0
	(Any state) → Shoot:Ready	entry action	requestLocation	-	0.1
	Shoot:WaitTimer → Shoot:Ready	Shoot:timeout	-	100	0.2
	Shoot:Ready → Shoot:WaitTimer	location	setPath	-	15.0

5. 사례 연구

사례 연구로서 UML-RT로 축구 로봇을 설계하고, 제안된 방법을 사용하여 자동으로 스케줄링이 보장되는 구현을 만들었다. 또한 현재 널리 사용되는 CASE 도구에서 설계자들이 구현을 만들어내는 방식을 사용하여 두 개의 다른 버전의 로봇 축구 시스템을 구현하였으며, 이를 제안된 접근 방법과 비교하였다.

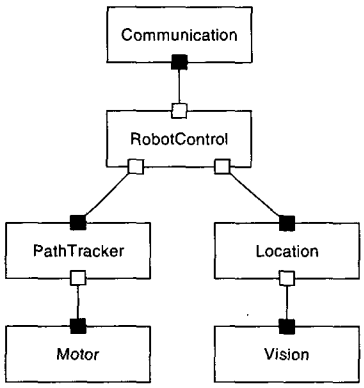

그림 5 축구 로봇 시스템의 구조 다이어그램

5.1 축구 로봇에 대한 설명

로봇 축구는 자율적인 로봇들로 구성된 두 개의 팀이 골을 얻기 위하여 경쟁하는 게임이다. 로봇 축구 게임에는 다양한 종류가 있으며 각기 다른 규칙과 여러 종류의 로봇을 사용한다[14]. 본 사례 연구에서의 각 축구 로봇은 한 개의 카메라, 두 개의 모터, 마이크로 제어기, 무선 통신 모듈을 가지고 있다. 축구 로봇은 무선 통신 기기를 통하여 stop과 start의 두 개의 명령을 받을 수 있다. 일단 로봇이 start 명령을 받으면, 카메라로부터 얻은 정보를 사용하여 공을 찾으며, 공을 몰기 위한 최적 경로를 계산한다. 그리고 그 계산된 경로로 자신이 갈 수 있도록 모터를 제어한다. 본 사례 연구에서는 그림 5와 같은 구조 다이어그램을 가진 축구 로봇 시스템을 사용한다. 즉, Communication, Vision, Location, Motor, PathTracker와 RobotControl의 6개의 캡슐 개체(instance)로 구성된다. 표 2는 각 캡슐의 액션과 이들 액션들에서 보내지는 메시지들, 각 액션의 주기 및 최악 수행시간을 보여준다. 표 2에서 ":"은 하부 상태를 나타내기 위하여 사용되었다. RobotControl 캡슐의 타임아웃에 의한 액션은 상태에 따라 100ms나 200ms의 주기를 가지는 것을 볼 수 있다.

5.2 다른 스레드 변환 방법간의 비교

본 사례 연구에서는 제안된 방법과 함께, 기존의 CASE 도구에서 설계자들이 사용하고 있는 방법을 사용하여 축구 로봇 시스템을 구현하였고, 본 절에서 이들을 비교한다.

5.2.1 단일 스레드 변환

이 변환 방법은 모든 캡슐 객체를 하나의 스레드로 사상시킨다. 이는 RoseRT와 같은 CASE 도구에서 기본적으로 구현을 만들어내는 방식이다. 이 변환 방식을 사용할 경우, 생성된 축구 로봇이 정확하게 동작하지 않을 수 있는 것을 쉽게 볼 수 있다. 구체적인 예로,

Motor 캡슐 개체에서 처리되는 메시지를 생각해 보자. 단일 스레드 구현에서는 모든 상태 전이가 비선점적으로 이루어지기 때문에 이 메시지의 처리는 현재 처리되고 있는 상태 전이가 완전히 끝날 때까지 이루어 질 수 없다. 만약 Motor 캡슐 개체의 타임아웃 간격이 시스템의 다른 어떤 액션의 수행 시간보다 작다면, Motor 캡슐 개체는 다음 타임아웃까지 액션을 수행할 수 없을 수 있다. 이러한 문제를 피하기 위하여서는 설계 모델을 수정하여 모든 긴 액션을 작은 단위로 쪼개어야 한다. 그러나 이런 수정을 항상 할 수 있는 것도 아니다.

표 3 캡슐 기반의 스레드 사상과 우선 순위 할당

캡슐	스레드	우선 순위	주기
Communication	th_{com}	1	500 ms
Vision	th_{vis}	2	400 ms
Location	th_{vis}	2	400 ms
Motor	th_{mot}	4	5 ms
PathTracker	th_{mot}	4	5 ms
RobotControl	th_{rot}	3	100 ms

5.2.2 캡슐 기반의 다중 스레드 변환

이 변환 방법은 캡슐 개체들을 병합하여 몇 개의 그룹을 만들고 이들 각 그룹을 하나의 스레드로 만든다. 이 방법은 RoseRT CASE 도구에서 다중 스레드를 지원하기 위하여 사용되는 방식이다. 불행하게도, 현재까지 이 방법을 기반으로 하여 자동으로 태스크를 식별하여 주는 방법을 개발한 연구 성과는 없었다. 일단 설계자가 임시 변통적인 방법으로 캡슐 기반으로 태스크를 식별해 낸 후, 스케줄링이 가능한지를 판단할 수 있는 [5, 7]과 같은 연구 결과가 있을 뿐이다. 이 변환 방법을 사용하기 위하여서는 얼마나 많은 스레드를 생성할지, 어떤 캡슐 개체를 합쳐서 스레드로 사상할 지를 설계자가 임시 변통적으로 결정해야 한다.

표 3의 스레드 사상은 본 사례 연구에서 사용한 것으로, Saksena 등이 제시한 [6]의 안내 지침을 따른 것이다. 구체적으로, 스레드 간 메시지 전달과 문맥 전환에 의한 부하를 줄이기 위하여 서로 연결이 있는 캡슐 객체들을 그룹지어 만들었다. 각 스레드의 주기는 사상된 캡슐 개체에 발생하는 타임아웃 이벤트의 주기를 계승하도록 하였다. 스레드의 우선 순위는 비율 단조(rate-monotonic) 우선 순위 할당 방식[15]을 사용하여 할당하였다. 숫자가 클수록 높은 우선 순위를 나타낸다. 스레드 사상과 우선 순위 할당을 마친 후에 [5, 7]의 응답 시간 분석을 사용하여 추출된 스레드 집합에 대하여 스케줄 가능성 분석을 하였다. RobotControl 캡슐 개체의 타임아웃 이벤트에 의하여 야기된 트랜잭션의 응답 시간 계산 결과가 다음과 같다.

$$
\begin{aligned}
R_{Robot} &= (C^{Robot}_{Shoot\ location} + C^{Vision}_{timeout} + C^{Robot}_{shoot\ location} + C^{Path}_{setPath}) \\
&\quad + (C^{Robot}_{Shoot\ timeout} + C^{Robot}_{Shoot\ Readyentry} + C^{Location}_{requestLocation} \\
&\quad + C^{Robot}_{Shoot\ location} + C^{Path}_{setPath}) \\
&\quad + \lceil \frac{R_{Robot}}{T_{mot}} \rceil \cdot (C^{Motor}_{timeout} + C^{Path}_{movement} + C^{Motor}_{setPath}) \\
&= 201.8
\end{aligned}
$$

여기에서 C^{A}_{B}는 캡슐 개체 A에서의 액션 B의 최악 수행시간을 나타낸다. Vision 캡슐 개체와 Location 캡슐 개체가 같은 스레드에 사상되었기 때문에 첫 번째 항에 $C^{Vision}_{timeout}$이 포함되는 것을 볼 수 있다. 이 트랜잭션의 최대 응답 시간은 타임아웃 간격 (100ms)보다 크다. 따라서 이 응답 시간을 줄이기 위하여 다른 스레드 변환을 다시 시도하거나 모델을 재 수정하여야 한다.

표 4 논리적 스레드의 물리적 스레드로의 사상

물리적 스레드	논리적 스레드	우선 순위	preemption threshold
Ph 1	L_{Motor}	4	4
Ph 2	$L_{RobotControl}$	3	3
	$L_{Communication}$	1	3
Ph 3	L_{Vision}	2	2

5.2.3 제안된 방법을 사용한 다중 스레드 변환

제안된 방법을 적용시킨 결과 8 개의 트랜잭션이 유도되었으며 이들은 4 개의 논리적 스레드로 사상되었고, 최종적으로 3 개의 물리적 스레드로 사상되었다. 표 4는 최종 물리적 스레드의 사상 결과를 보여준다. 두 개의 논리적 스레드 $L_{RobotControl}$과 $L_{Communication}$은 같은 선점 임계값을 가지기 때문에 서로 선점할 수 없으므로 하나의 스레드로 안전하게 사상될 수 있다. 이 구현은 제안된 방법을 통하여 스케줄 가능성을 보장받는다.

이와 같이 제안된 방법은 타이밍 분석을 쉽게 자동화시켜주고, 사람의 개입 없이 UML-RT와 같은 실시간 객체 지향 모델에 직접 적용되어 태스크를 식별하여 준다.

6. 결 론

본 논문에서는 객체 지향 설계 모델로부터 스케줄 가능성을 보장받는 다중 스레드 구현을 생성하는 자동화된 방법을 제시하였다. 제안된 방법은 (1) 트랜잭션, (2) 논리적 스레드, (3) 물리적 스레드를 유도하는 세 단계로

구성된다. 트랜잭션은 메시지 연속 트리를 사용하여 유도한다. 논리적 스레드는 상호 동시 실행이 불가능한 트랜잭션들을 그룹지어 만든다. 논리적 스레드는 여러 트랜잭션을 포함할 수 있으므로 하나 이상의 주기, 블록킹 시간, 최악 수행 시간 등의 스케줄 특성을 가지게 된다. 제안된 방법에서는 이러한 새로운 태스크 모델에 대한 스케줄 분석 알고리즘을 제시하였다. 이를 이용하여 논리적 스레드들에 전체 시스템이 스케줄 가능하도록 우선순위를 할당하였다. 또한 문맥 전환 부하를 줄이기 위하여 선점 임계(preemption threshold) 스케줄링을 채용하여, 각 논리 스레드에게 가능한 최대의 선점 임계값을 할당하였다. 마지막 단계에서는 논리적 스레드를 비선점 그룹으로 묶어서 최종 물리적 스레드로 사상하였다. 이는 스레드의 숫자를 크게 줄일 수 있게 하고, 그 결과 수행시 문맥 전환 부하와 정적 메모리 요구량을 줄이게 한다.

본 논문에서는 RoseRT와 같은 CASE 도구에서 제안된 방법을 어떻게 지원될 수 있는지에 대하여서도 기술하였다. 제안된 방법은 기존의 CASE 도구에 쉽게 적용될 수 있다. 또한 본 논문에서 제시한 사례 연구는 임시 방편적으로 태스크를 유도하는 것이 얼마나 어려운지를 보였으며, 제안된 방법의 유용성을 명백하게 보여주었다.

참 고 문 헌

[1] M. Boasso, *"Control systems software,"* IEEE Transactions on Automatic Control, pp. 1094-1106, January 1993.

[2] H. Gomaa, *"Software design methods for concurrent and real-time systems,"* Adisson-Wesley Publishing Company, 1993.

[3] B. Selic, G. Gullekson, and P. T. Ward, *"Real-time object-oriented modeling,"* John-Wiley & Sons, Inc, 1994.

[4] B. Selic and J. Rumbaugh, *"Using UML for modeling complex real-time systems,"* White Paper, Publicated by ObjecTime, and available from http://www.objectime.com, March 1998.

[5] D. Gaudrean and P. Freedman, *"Temporal analysis and object-oriented real-time software development: a case study with ROOM/ObjecTime,"* Proceedings of IEEE Real-Time Systems Symposium, pp. 110-118, May 1996.

[6] M. Saksena, P. Freedman and P. Rodziewicz, *"Guidelines for automated implementation of executable object oriented models for real-time embedded control systems,"* Proceedings of IEEE Real-Time Systems Symposium, pp. 240-251, June 1997.

[7] M. Saksena, A. Ptak, P. Freedman, and P. Rodziewicz, *"Schedulability analysis for automated implementations of real-time object-oriented models,"* Proceedings of IEEE Real-Time Systems Symposium, pp. 92-102, December 1998.

[8] Y. Wang and M. Saksena, *"Scheduling fixed-priority tasks with preemption threshold,"* Proceedings of IEEE Real-Time Computing Systems and Applications Symposium, pp. 328- 335, 1999.

[9] M. Saksena and Y. Wang, *"Scalable real-time system design using preemption thresholds,"* Proceedings of IEEE Real-Time Systems Symposium, pp. 25 -34, 2000.

[10] Object Management Group, *"OMG Unified Modeling Language Specification, Version 1.4,"* September 2001.

[11] N. Audsley, *"Optimal priority assignment and feasibility of static priority tasks with arbitrary start times,"* Technical Report YCS 164, Department of Computer Science, University of York, England, December 1991.

[12] K. Tindell and A. Burns and A. Wellings, *"An extendible approach for analyzing fixed priority hard real-time tasks,"* Real-Time Systems Journal, pp. 133-151, 1994.

[13] Institute for Electrical and Electronic Engineers, *"IEEE Std. 1003.1c-1995 POSIX Part 1: System Application Program Interface-Amendment 2: Threads Extension,"* 1995.

[14] Federation of International Robot-soccer Association. *http://www.fira.net.*

[15] J. P. Lehoczky, L. Sha, and Y. Ding, *"The rate monotonic scheduling algorithm: exact characterization and average case behavior,"* Proceedings of IEEE Real-Time Systems Symposium, pp. 166-171, 1989.

홍 성 수

1986년 서울대학교 컴퓨터공학과 졸업(B.S.). 1988년 서울대학교 컴퓨터공학과 졸업(M.S.). 1988년 ~ 1989년 한국전자통신연구소(연구원). 1994년 University of Maryland, Department of Computer Science(Ph.D.). 1994년 ~ 1995년 University of Maryland, Department of Computer Science(Faculty Research Associate). 1995년 ~ 1995년 Silicon Graphics Inc. (Member of Technical Staff) 1995년 ~ 1997년 서울대학교 전기공학부 전임강사. 1997년 ~ 2001년 서울대학교 전기공학부 조교수. 2001년 ~ 현재 서울대학교 전기컴퓨터공학부 부교수