온톨로지 기반의 상황정보관리에서 추론 성능 향상을 위한 어플리케이션 지향적 상황정보 선인출 기법

(Application-Oriented Context Pre-fetch Method for Enhancing Inference Performance in Ontology-based Context Management)

이 재 호 † 박 인 석 †† 이 동 만 †† 현 순 주 †††
(Jaeho Lee) (Insuk Park) (Dongman Lee) (Soon Joo Hyun)

요 약 유비쿼터스 컴퓨팅 환경에서 온톨로지 기반의 상황정보 모델은 추론을 통한 개념적 상황정보의 획득, 상황정보의 공유와 재사용의 이점을 제공하기 때문에 널리 사용되고 있다. 이 중에서 추론은 상황인지 어플리케이션이 센서로부터 직접 얻을 수 없는 개념적 상황정보를 이용할 수 있도록 해준다. 하지만 추론의 경우, 그 처리 시간이 대상이 되는 상황정보의 크기가 커질수록 증가하게 되며, 이때 야기되는 시간 지연은 상황인지 어플리케이션의 실제적인 동작을 방해한다. 본 논문에서는, 추론 속도를 향상시키기 위해 작업 메모리에서 처리 되는 상황정보의 크기를 줄이는 상황정보 선인출 기법을 제안한다. 우리는 상황인지 어플리케이션의 질의와 관련이 있는 상황정보를 결정하기 위해 기존의 쿼리트리를 이용한 방법을 확장한다. 제안한 기법을 이용하여 선인출된 상황정보만 작업 메모리에 유지함으로써, 온톨로지 기반의 상황정보가 제공하는 이점을 유지하면서 추론에 의해 야기되는 시간 지연을 줄일 수 있다. 우리는 제안한 기법을 기존의 유비쿼터스 컴퓨팅 미들웨어, Active Surroundings에 적용시키고 실험을 통해 성능 향상을 보였다.

키워드 : 온톨로지 기반의 상황정보 관리, 추론 성능 향상

Abstract Ontology-based context models are widely used in ubiquitous computing environment because they have advantages in the acquisition of conceptual context through inferencing, context sharing, and context reusing. Among the benefits, inferencing enables context-aware applications to use conceptual contexts which cannot be acquired by sensors. However, inferencing causes processing delay and thus becomes the major obstacle to the implementation of context-aware applications. The delay becomes longer as the amount of contexts increases. In this paper, we propose a context pre-fetching method to reduce the size of contexts to be processed in a working memory in attempt to speed up inferencing. For this, we extend the query-tree method to identify contexts relevant to the queries of a context-aware application. Maintaining the pre-fetched contexts optimal in a working memory, the processing delay of inference reduces without the loss of the benefits of ontology-based context model. We apply the proposed scheme to our ubiquitous computing middleware, Active Surroundings, and demonstrate the performance enhancement by experiments.

Key words : ontology-based context management, enhancing inference performance

1. 서 론

상황인지 어플리케이션은 그것의 동작이 사용자의 위치나 행동, 성향, 주변기기의 상태 등과 같은 상황정보에 따라 결정되고 실행이 된다. 예를 들어, Teleport System[1]은 사용자의 기존 데스크탑 환경을 그 사용자가 현재 위치하고 있는 장소의 컴퓨터로 옮겨주는 상황

† 정 회 원 : 삼성전자 정보통신 사업부
leejaeho@icu.ac.kr
†† 학생회원 : 한국정보통신대학교 공학부
ispark@icu.ac.kr
dlee@icu.ac.kr
††† 정 회 원 : 한국정보통신대학교 공학부 교수
shyun@icu.ac.kr
논문접수 : 2005년 7월 29일
심사완료 : 2006년 5월 9일

인지 어플리케이션이고, Navigation System[2]은 사용자의 현재 이동 속도에 따라 네비게이션에 보여지는 정보의 형태(메시지 크기)를 동적으로 변화시켜 주는 상황인지 어플리케이션이다. 위와 같이 상황정보에 기반하여 동작하는 상황인지 어플리케이션의 개발을 용이하게 지원하기 위해서는 동적으로 변화하는 상황정보의 저장과 관리를 위한 상황정보 모델이 필요하다. 이에 상황정보 모델링에 관련한 연구가 다양하게 진행되고 있으며, 기존 연구 방향은 크게 어플리케이션에 종속적인 모델(Application-oriented model), 모델링 기법 기반의 모델(Model-oriented model), 그리고 온톨로지 기반 모델(Ontology-based model)로 나눠질 수가 있다. 어플리케이션에 종속적인 상황정보 모델을 이용한 Cooltown[3]은 상황정보를 특정 URL을 통해서 접근하도록 웹 기반의 모델링 방법을 이용했으며, Context toolkit[1]에서는 상황정보를 어플리케이션에서 요구하는 [상황정보 이름-값]의 형태로 표현하고 이용하였다. 하지만 이러한 모델링의 경우, 정형적인 형태의 모델이 존재하지 않기 때문에 다른 시스템간의 상황정보 공유와 재사용이 불가능하다는 단점을 가지고 있다. 모델링 기법 기반의 모델은 기존의 정보 시스템에서 데이타를 모델링하기 위해 사용한 기법들을 이용하여 상황정보를 모델링한 시도들이며, Henricksen[4]은 ER 모델과 UML을 상황정보 모델링에 이용하였다. 그러나, 이러한 기법의 모델링도 이 기종간의 상황정보 공유의 문제를 해결하지 못하였다. 이에 최근에는, 위에서 언급한 연구들의 문제를 해결하기 위한 방법으로 온톨로지를 이용한 상황정보 모델들이 제시되고 있다. 온톨로지 기반의 상황정보 모델은 상황정보의 공유와 재사용, 그리고 추론을 통해 센서로부터 알아낼 수 없는 개념적인 상황정보의 획득을 가능하게 한다는 이점을 가지고 있다.

이러한 이점으로 인하여, 온톨로지 기반의 상황정보 모델을 이용하는 상황정보 관리 시스템이 최근에 많이 개발되었다[5-8]. 이러한 상황정보 관리 시스템들의 역할은 크게 다양한 센서로부터 상황정보를 수집하는 과정, 추론의 과정을 통하여 개념적 상황정보를 획득하는 과정, 그리고 적절한 상황정보를 그것을 이용하고자 하는 어플리케이션에게 전달하는 과정으로 나눌 수가 있다. 하지만 추론의 경우, 그 처리 시간이 대상이 되는 상황정보의 크기와 적용시켜야 하는 규칙의 개수가 많아짐에 따라서 증가하게 되는 문제점을 가지고 있다. 유비쿼터스 컴퓨팅 환경이 더욱 더 확장되고 지능화될수록 관리해야 하는 상황정보의 크기도 증가하게 되고, 결과적으로 추론의 처리 시간도 증가하게 된다. 온톨로지를 상황정보 모델로 사용하고 있는 기존의 다른 시스템에서도 이 문제점에 대해서 인지를 하고 있으나, 적극적인 해결책은 아직 제시되지 않고 있는 상태이다[5-8].

본 논문에서는, 추론에 걸리는 시간을 줄이기 위해 작업 메모리[1]에 적재되는 상황정보의 크기를 줄이는 상황정보 선인출 기법을 제안한다. 우리는 여러 종류의 상황인지 어플리케이션의 개발을 통해 실제 추론 과정에 이용되는 상황정보는 전체 상황정보에서 일부분에 지나지 않음을 알게 되었다. 예를 들어, 사용자의 현재 위치와 상태에 따라서 전구의 밝기를 조절하는 어플리케이션의 경우는 사용자의 위치와 상태, 그리고 전구의 상태정보만을 이용하게 된다. 반면에, 사용자의 의도에 기반해서 디스플레이 화면의 크기를 조절하는 어플리케이션의 경우에는 위에서 소개한 어플리케이션이 이용을 했던 전구의 상태정보 대신에 디스플레이의 상태정보에 기반해서 동작을 한다. 그러므로, 우리는 각 상황인지 어플리케이션이 실행하는데 있어 필요하지 않은 상황정보는 작업 메모리에 위치할 필요가 없으며, 그 정보를 미리 알아서 제거하면 추론의 속도를 향상시킬 수 있다고 생각하게 되었다. 우리는 상황인지 어플리케이션의 실행에 필요한 상황정보를 결정하기 위해 쿼리트리 방법[9]을 확장한다. 첫 단계에서는, 어플리케이션의 쿼리를 이용하여 쿼리트리를 구성하고, 기존의 쿼리트리 방법에서 제안한 제약속성(Constraint predicate)을 이용하여 쿼리와 관련 있는 상황정보를 결정한다. 그 후, 상황정보의 타입을 쿼리트리에 적용시켜 실제 어플리케이션 동작하는 실행시간 이전에는 필요 없는 상황정보들을 추가적으로 제거(filtering)한다. 이를 위해, 우리는 상황정보를 센서에 의한 상황정보, 추론에 의한 상황정보, 그리고 미리 정의된 상황정보로 분류를 하였다. 또한, 우리는 제안한 상황정보 선인출 기법을 유비쿼터스 컴퓨팅 미들웨어, Active Surroundings[6]에 적용을 시켰으며, 실험을 통해 제안한 기법이 작업 메모리에 적재되는 상황정보의 크기를 줄이고, 추론에 걸리는 시간을 단축시킴을 보였다. 이것은 제한된 자원의 컴퓨팅 기기가 산재하는 유비쿼터스 컴퓨팅 환경에서 온톨로지 기반의 상황정보 관리 시스템이 대규모의 상황정보를 다루는데 도움을 줄 수 있을 것이다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구들에 대한 소개를 하고, 3장에서는 제안하는 상황정보 선인출 기법에 대해서 자세히 설명을 한다. 4장에서는 실제 온톨로지 기반의 상황정보 관리시스템에 적용시킨 구현에 대해서 설명을 하고, 5장에서는 실험을 통한 성능 향상을 보인다. 마지막으로 6장에서는 요약과 향후 과제에 대한 소개로 결론을 맺는다.

1) 추론엔진에 의해 사용되는 메모리

2. 관련 연구

2.1 온톨로지 기반 상황정보 관리 시스템

Context Broker Architecture(CoBrA)은 분산 시스템간의 상황정보 공유를 용이하게 지원하기 위해 온톨로지 기반의 상황정보 모델(SOUPA)을 개발하였다[10]. 이 연구에서는 개념적 상황정보를 획득을 위해 F-OWL 추론엔진을 이용한다[11]. F-OWL 추론엔진의 경우 추론의 성능향상을 위해 Tabling 기법을 엔진 내부에서 사용하고 있다. Tabling 기법은 Jena2[12]를 포함한 여러 추론엔진에서 사용하는 일종의 캐싱(Caching) 기법이다. 이 방법의 경우, 주어진 하나의 triple(subject-predicate-object)이 검사의 대상이 되는 온톨로지 모델에 존재하게 되면, 테이블에 그 triple과 결과값을 저장한다. 나중에 같은 triple이 또 다시 검사되는 경우에는 전체 모델을 검색하지 않고, 그 테이블을 조사해서 결과값을 돌려줄 수 있다. 이 경우, 최초 쿼리의 경우 그 응답시간이 길어질 수 있으나, 테이블에 저장된 이후부터는 응답시간을 많이 줄일 수 있다. 하지만, 대상이 되는 모델에 새로운 데이타가 입력, 삭제되는 등의 갱신이 발생하게 되면 만들어진 테이블은 무효화되어 버린다. 그러므로, 빈번하게 변화하는 상황정보 모델의 경우 Tabling 기법의 이점을 누릴 수가 없다.

Semantic Space은 정형화된 상황정보의 표현, 이 기종간의 상황정보 공유, 그리고 추론을 통해 개념적 상황정보 획득이라는 이점을 누리기 위해 온톨로지를 이용하여 상황정보를 모델링(CONON) 하였다[5]. 이 시스템은 상황정보의 크기가 커짐에 따라 발생하게 되는 추론의 응답시간 지연을 문제점으로 인식하고 해결책을 다음과 같이 제시하였다. 첫 번째는, 신속한 실행시간이 보장되어야 하는 어플리케이션의 경우 추론의 과정 없이 실행되도록 하여 추론 때문에 발생하는 문제점을 회피하는 해결책을 내어놓았다. 두 번째로, 추론은 성능이 좋은 컴퓨터에서 처리할 수 있도록 하여 좀 더 빠른 응답시간을 기대하였다. 위의 두 가지 해결책에 비해 좀 더 적극적인 노력은 상황정보 모델을 상위 레벨(High level)과 도메인에 한정된 레벨(Domain specific level)로 나눈 것이다. 이렇게 나눔으로써, 어플리케이션이 동작하는 도메인에 해당하는 상황정보만 작업 메모리에 적재시킬 수 있게 된다. 결과적으로 작업 메모리의 크기를 줄일 수 있기 때문에 추론의 속도를 빠르게 할 수 있다. 하지만 이런 경우에도, 그 도메인에서 동작하는 어플리케이션이 그 도메인에 한정된 상황정보를 모두 다 사용하는 것은 아니다. 뿐만 아니라 도메인의 크기가 커지게 되면 작업 메모리에 적재되는 상황정보의 크기도 커지게 됨으로 앞에서 언급한 문제점들이 다시 발생하게 된다. 우리의 상황정보 모델도 Semantic Space의 2-레벨 구조로 설계하였다. 하지만 우리는 도메인에 한정된 상황정보에서 어플리케이션에 의해서 실제 사용되는 상황정보만을 작업 메모리에 적재하는 기법을 제안함으로써, 도메인 크기 증가의 결과로 발생하는 추론의 속도 저하 문제를 해결할 수 있다.

2.2 추론의 속도를 향상시키기 위한 방법론

우리는 기존의 추론엔진에서 성능을 향상시키기 위한 노력들에 대해서 알아 보았다. 온톨로지 추론을 위해 사용되는 추론엔진에는 Jena2[12], Racer[13], FaCT[14], Hoolet[15], and Triple[16] 등이 있다. Jena2는 RDF와 OWL에서 제공하는 의미를 이해하는 추론엔진을 제공한다. 하지만 Jena2의 메모리 기반 추론 모델은 triple들이 저장되어 있는 그래프를 반복적으로 순회하면서 triple패턴 매치를 하도록 구현이 되었기 때문에, 검사해야 하는 대상 triple의 개수가 증가함에 따라 추론의 응답시간이 지연되는 문제점을 안고 있다. 그리고 Racer, FaCT, Hoolet, 그리고 Triple과 같은 여러 추론엔진들이 좋은 성능을 보이면서 널리 사용되고 있으나, OWL로 기술된 대규모의 wine 온톨로지(http:// www.w3.org/TR/owl-guide/wine.owl)를 이용한 테스트에서 좋은 성능을 보여주지 못하고 있다[11].

Alon Y. Levy은 쿼리트리를 이용하여 추론의 대상이 되는 데이타의 크기를 줄이고, 그 결과 추론의 속도를 향상시키는 방법을 제안하였다[9]. 쿼리트리는 검사를 해야 하는 전체 데이타에서 주어진 쿼리가 검사를 해야 하는 부분을 한정시켜주는 분석 도구이다. 그러므로 쿼리트리를 이용하면 특정 쿼리를 위해 필요한 데이타를 결정할 수가 있게 된다. 우리는 상황인지 어플리케이션의 실행에 관련 있는 상황정보를 알아내기 위해 기존의 쿼리트리 방법을 확장해서 적용하였다.

3. 상황정보 선인출 기법

3.1 설계시 고려사항

온톨로지 기반의 상황정보 모델에서 추론의 속도는 두 가지 요소를 줄임으로써 향상될 수 있다. 그 중 하나는 적용시켜야 하는 규칙의 개수이고, 또 다른 하나는 추론의 대상이 되는 상황정보의 크기이다[17]. 규칙은 또 다시 두 종류로 나눠질 수가 있는데, 하나는 표 1에 보인 subClassOf, Symmetric, 그리고 Transitive와 같이 온톨로지의 의미정보를 유지하기 위해 필요한 규칙이다. 이러한 규칙들은 클래스 요소들간의 관계 표현과 클래스 요소가 취해야 하는 데이타의 형식과 값을 제약하는데 사용된다. 따라서 온톨로지의 의미정보를 위한 규칙은 그 수를 줄일 수가 없다. 또 다른 규칙은 사용자가 개념적 상황정보를 생성하기 위해 미리 정의해 놓은 규칙이다. 센서를 통해 직접적으로 알아낼 수 없는 사용

자의 현재 상태와 같은 상황정보는 미리 규칙으로 정의해놓고 추론을 통해서 알아낼 수 있다. 이 같은 종류의 규칙들은 어플리케이션이 그 개념적 상황정보를 필요로 할 때만 사용된다. 그러므로 위에서 언급한 두 종류의 규칙은 어플리케이션의 정확한 실행을 위해서 그 수를 줄일 수가 없다. 그러므로 우리는 추론의 속도를 향상시키기 위한 방법으로 상황정보의 크기를 줄이는 것에 초점을 맞추었다.

표 1 OWL의 속성과 관련한 규칙(Rule)의 예

Transitive -Property	(?P rdf:type owl:TransitiveProperty)∧ (?A ?P ?B)∧(?B ?P ?C)->(?A ?P ?C)
Symmetric Property	(?P rdf:type owl:SymmetricProperty)∧ (?X ?P ?Y) -> (?Y ?P ?X)
inverseOf Property	(?P owl:inverseOf ?Q)∧(?X ?P ?Y) -> (?Y ?Q ?X)
Equivalent Property	(?P owl:equivalentClass ?Q) -> (?P rdfs:subClassOf ?Q), (?Q rdfs:subClassOf ?P)
subClassOf Property	(?A rdfs:subClassOf ?B)∧ (?B rdfs:subClassOf ?C) -> (?A rdfs:subClassOf ?C)

추론의 대상이 되는 상황정보의 크기를 줄이는 것은 어플리케이션의 실행에 필요한 상황정보만을 작업 메모리에 유지함으로써 가능해진다. 여기서 어플리케이션의 실행에 필요한 상황정보란 그 어플리케이션이 가지고 있는 쿼리가 결과값을 얻기 위해 접근하는 상황정보라고 할 수 있다. 쿼리를 이용해서 검색 대상을 한정시키는 방법으로는 관련 연구에서 살펴본 쿼리트리를 이용한 기법이 있다[9].

쿼리트리를 이용한 방법은, 우선 주어진 쿼리를 이용하여 쿼리트리를 만들고 그 트리를 순회하면서 제약속성을 이용해서 각 쿼리에 불필요한 fact들을 제거한다. 여기서 제약속성이란 knowledgebase에 있는 fact들을 한정시키기 위해 사용하는 제약 조건이다. 예를 들어 AgeOf(x, y)라는 쿼리와 y<=150 이라는 제약속성이 있다면, y>150인 fact들은 주어진 쿼리의 결과값을 위해서는 불필요한 fact라고 할 수 있다. 이러한 과정은 실제 쿼리가 요청되기 전에 행해지고, 쿼리 요청 시에는 y<=150을 만족하는 fact들만 작업 메모리에 적재가 되어 검색의 대상이 된다. 이와 같이 쿼리트리를 이용해 검색 대상의 크기를 줄인 후 추론을 하게 되면 전체 검색 대상에서 쿼리를 하는 경우보다는 속도를 증가시킬 수 있다. 하지만 상황인지 어플리케이션 쿼리의 경우에는 이 쿼리트리 기법을 적용시키기 전에 한가지 고려할 사항이 있다. 상황인지 어플리케이션의 쿼리에는 센서로부터 입력 받는 상황정보의 값을 요구하는 경우가 많이 있으며, 이 같은 종류의 상황정보는 그 값이 실제 센싱되기 전에는 의미 없는 값을 가진다. 상황정보의 이러한 특징으로 인하여 기존의 쿼리트리 기법을 수정할 필요가 있게 되었으며, 수정한 방법을 통하면 더 짧은 시간에 최적의 상황정보를 구하여 추론의 검색 대상으로 사용할 수 있게 된다. 우리는 상황정보가 언제 의미 있는 값이 되는지 구분하기 위해서 상황정보를 분류하였으며 다음 장에서 자세히 설명 하겠다.

3.2 상황정보의 표현과 분류

우리는 OWL을 이용하여 가정 환경을 위한 상황정보를 설계하였다. 설계는 CONON[17]에서 제시한 2-레벨 구조의 이점을 누리고자 상위레벨과 하위레벨로 나누어서 정의하였다. 상위레벨 상황정보는 모든 도메인에서 공통적으로 사용될 수 있는 클래스와 그것의 속성을 정의하고, 온톨로지의 의미정보를 이용하여 각 속성들간의 관계나·제약 조건들을 기술 하였다. 하위레벨 상황정보는 상위레벨에서 정의한 일반적인 개념과 속성을 이용하여 각 도메인에 한정된 객체들에 대해서 정의 하였다. 예를 들어, 가정이라는 도메인의 하위레벨 상황정보인 'Bedroom'과 비즈니스 도메인의 하위레벨 상황정보인 'Office'의 정의는 모두 상위레벨 상황정보 'Room'의 정의에 따라 만들어진 객체들이다. 정의된 상황정보의 일부분을 그림 1에 나타내었다.

OWL로 기술된 상황정보는 추론엔진에서 RDF triple <subject, predicate, object>의 형태로 변환하여 다루게 된다. 예를 들어, 그림 1(b)에서 보인 Bedroom의 Brightness속성은 <Bedroom, Brightness, Bedroom-Brightness>의 형태로 표현이 된다. 이때 predicate에

```
<owl:Class rdf:ID="Room">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#doorState"/>...
<owl:ObjectProperty rdf:about="&room;Brightness">
  <rdfs:domain rdf:resource="#Room"/>
  <rdfs:range rdf:resource="&state;Brightness"/>
  <owl:classifiedAs rdf:resource="&icu;Sensed"/>
</owl:ObjectProperty> ...
```

(a) 상위레벨 상황정보 온톨로지

```
<Room rdf:ID="Bedroom">
 <hasDevice rdf:resource="&device;Bed"/> ......
 <Brightness rdf:resource="&state;BedroomBrightness"/>
</Room>
```

(b) 가정 도메인의 상황정보 온톨로지

```
<Room rdf:ID="Office">
<hasDevice rdf:resource="&device;Desk"/> ...
<hasDevice rdf:resource="&device;Fax"/>
</Room>
```

(c) 비즈니스 도메인의 상황정보 온톨로지

그림 1 'Room' 상위레벨 상황정보와 'Bedroom', 'Office' 하위레벨 상황정보의 정의

해당하는 Brightness는 Bedroom과 BedroomBrightness사이의 관계를 표현해주는 역할을 하며, 그것의 subject와 object자리에 위치할 수 있는 상황정보는 각각 Room과 Brightness 타입이어야 한다는 제약을 가지고 있다(그림 1(a)). 우리는 이것을 제약속성이라고 정의하였으며, 다음 장에서 이것의 사용에 대해서 자세히 설명을 하겠다.

우리는 상황정보를 다음과 같이 세 가지 타입으로 분류를 하였다: (1)센서에 의한 상황정보(Sensed context), (2)추론에 의한 상황정보(Deduced context), (3)미리 정의된 상황정보(Defined context). 그리고 위와 같은 상황정보의 타입을 표현하기 위해, 그림 1(a)에 나타낸 것과 같이 각 속성에 'owl:classifiedAs'라는 제약을 정의하였다. 이와 같이 분류한 상황정보의 타입은 쿼리트리를 구성하는 단계에서 실제 필요한 쿼리인지 그렇지 않은 쿼리인지를 결정하기 위해 사용된다. 물론 상황정보 모델링에 관련한 다른 연구에서도 위와 같이 분류된 상황정보 타입을 사용하고 있으나, 상황정보의 불확실성 문제를 해결하기 위한 목적으로 사용한다는 점에서 차이가 있다[4,17]. 각각의 상황정보의 타입에 대해서 살펴보면 다음과 같다.

첫 번째, 'person; locatedAt'과 같은 센서에 의한 상황정보는 실행시간에 센서로부터 입력 받는 상황정보이다. 그러므로 센서가 동작하지 않는 쿼리트리 구성 단계에서는 아무런 의미가 없다. 따라서 '<?p person; locatedAt Bedroom>'와 같은 쿼리는 실행시간 전에는 아무런 값도 받을 수가 없게 된다. 두 번째, 'person; hasStatus'와 같은 추론에 의한 상황정보는 일반적으로 센서에 의한 상황정보가 입력이 된 후에 추론의 과정을 거쳐 얻어낼 수 있다. 추론에 의한 상황정보는 3.1장에서 언급한 사용자 정의 규칙을 만족하는 경우에 자동적으로 생성되는데, 이 규칙은 다른 타입의 상황정보 값을 얻기 위한 쿼리로 구성되어 있다. 예를 들어, 현재 사람의 상태가 '취침 중(sleeping)' 인가 하는 상황정보는 현재의 상황정보들이 sleep-rule[2]을 만족하는 경우에 구할 수가 있다. 이 규칙에는 '<?p person;locatedAt Bedroom>', '<?d device;hasState ON>'와 같이 사람의 현재 위치, 침대의 현재 상태 등을 얻고자 하는 쿼리가 포함되어 있다. 이런 종류의 쿼리는 실행시간에 센서로부터 그 값을 받는 상황정보에 대한 쿼리이므로, 사람의 현재 상태가 '취침 중'이라는 상황정보는 실행시간 전에는 얻을 수가 없다. 마지막으로, '<TV device; locatedIn Bedroom>'와 같이 미리 정의된 상황정보 같은 경우에는 사용자에 의해 정의가 되고 정의 된 후에는 그 값의 변화가 없는 상황정보이다. 그러므로 미리 정의된 상황정보 값을 얻기 위한 쿼리는 언제나 그 결과가 동일하다. 결과적으로 실행시간 전에 정확한 쿼리의 결과를 받을 수 있는 상황정보는 미리 정의된 상황정보가 유일하고, 이에 우리는 상황정보 선인출 과정에서 이 상황정보만 고려를 한다.

3.3 상황정보 선인출

제안한 기법은 두 개의 메모리 공간을 필요로 한다. 첫 번째, 작업 메모리는 실제 실행시간에 추론엔진에 의해 사용이 되는 메모리로써, 선인출 결과로 나온 상황정보만을 담게 되는 장소이다. 두 번째, 전처리 메모리는 실행시간 전에 상황인지 어플리케이션에 필요한 상황정보를 선인출하는 과정에서 사용되는 메모리이다. 우리는 상황인지 어플리케이션의 쿼리를 이용해서 쿼리트리를 구성하고 전처리 메모리에서 필요한 상황정보를 선택한 후, 작업 메모리로 전달하는 일련의 과정을 선인출 이라고 정의 내린다.

선인출 과정 전에, 우리는 상위레벨 상황정보를 작업 메모리에 적재 한다. 상위레벨에 정의되어 있는 상황정보들간의 관계나 제약조건 등은 실행시간에 동적으로 반영되는 상황정보의 불일치성 검사와 온톨로지 의미정보를 이용한 추가적인 상황정보의 획득을 가능하게 해준다. 예를 들어, 상위레벨 상황정보 'Person'에 정의되어 있는 속성 'locatedAt'은 또 다른 상위레벨 상황정보 'Room'에 정의되어 있는 속성 'hasPerson'과 'owl: inverseOf' 관계로 연결되어 있다. 이런 경우, 실행시간에 센서로부터 '<Mr. Lee locatedAt Bedroom>'이라는 상황정보가 작업 메모리에 반영이 되면 온톨로지 규칙에 의해 자동적으로 '<Bedroom hasPerson Mr.Lee>'라는 상황정보가 추가적으로 생성이 된다. 그러므로 상위레벨의 상황정보는 실행시간 동안 작업 메모리에 상주를 해야 한다. 전처리 메모리에는 모든 상황정보 하에서의 쿼리 결과를 얻기 위해 상위레벨 상황정보뿐만 아니라 모든 하위레벨 상황정보까지 적재 시킨다.

우리는 선인출 과정에서 어플리케이션의 실행에 불필요한 상황정보를 전체 상황정보에서 제거를 한다. 그 과정은 두 단계에 걸쳐서 진행이 된다: a) 쿼리트리 기법으로부터 차용한 제약속성을 이용한 제거, b) 상황정보의 타입을 이용한 제거.

첫 단계에서는, 제약속성을 이용해서 불필요한 상황정보를 제거한다. 우리가 개발한 상황인지 어플리케이션의 쿼리는 온톨로지 쿼리 언어인 triple match의 형태로 기술되어 있다. Triple match는 <subject, predicate, object> 형태의 쿼리로써 그 조건을 만족하는 모든 결

---

2) sleep-rule:(?p rdf:type Person)(?d rdf:type Device)(?p locatedAt Bedroom)(?d locatedIn Bedroom)(?d hasState ON) -> (?p hasStatus sleeping). 사람이 위치하고 있는 곳의 침대가 그 사람에 의해서 사용되고 있을 때를 취침 중이라고 정의하였다.

과값을 돌려준다[18]. OWL을 이용하여 기술한 상황정보 역시 triple 형태로 표현이 되며, 각 predicate에는 제약속성이 정의되어 있다. 따라서, 상황정보 쿼리가 주어졌을 때 기술된 제약속성으로부터 쿼리가 검사를 해야 하는 검색 대상을 알아 낼 수 있다. 예를 들어, '<?p, locatedAt, ?r>'와 같은 상황정보 쿼리는 상위레벨 상황정보 'Person'에 기술된 제약속성에 따라 '?p'에는 Person 타입의 객체가 올 수 있으며, ?r에는 Room 타입의 객체만이 위치할 수 있다. 우리는 상황인지 어플리케이션의 쿼리를 이용해서 쿼리트리를 구성한다. 그림 2에서 전등 어플리케이션[3]의 쿼리에 기반한 쿼리트리를 보인다. 그림 2에서 쿼리 '<?p person;hasStatus Sleeping>'는 추론에 의한 상황정보를 위한 쿼리이다. 추론에 의한 상황정보를 구하고자 하는 쿼리는 다른 타입의 상황정보를 위한 쿼리로 나눠진다. 그러므로 주어진 쿼리는 '<?p rdf;type Person>', '<?d rdf;type Device>', '<?p person;locatedAtBedroom>', '<?d device; locatedIn Bedroom>', 그리고 '<?d device;hasState ON>'의 쿼리로 나눠진다. 그림 2에 보인 각 노드의 label은 제약속성으로 한정된 상황정보를 나타낸다. 위와 같이 쿼리트리를 구성한 후 제약속성을 만족하는 상황정보만 구하면 작업 메모리에 적재되는 상황정보의 크기를 줄일 수가 있다. 예를 들어, 위에서 언급한 전등 어플리케이션 쿼리의 경우, 전체 RDF triple 6000개의 상황정보에서 실험한 결과 50% 정도의 상황정보가 제거 되었다.

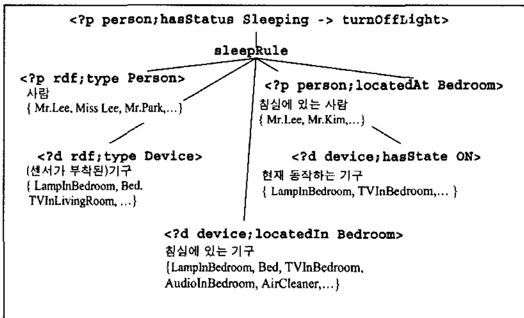


그림 2 전등 어플리케이션 쿼리 기반의 쿼리트리

첫 번째 단계 이후, 우리는 센서로부터 입력되는 상황정보를 위한 쿼리를 쿼리트리로부터 제거하기 위해 상황정보 타입을 이용한다. 3.2 장에서 살펴본 바와 같이 센서에 의한 상황정보를 얻고자 하는 쿼리는 쿼리트리를 구성하는 시간에는 어떠한 결과도 받을 수 없다. 그러므로 그러한 쿼리들은 쿼리트리로부터 제거시킬 수

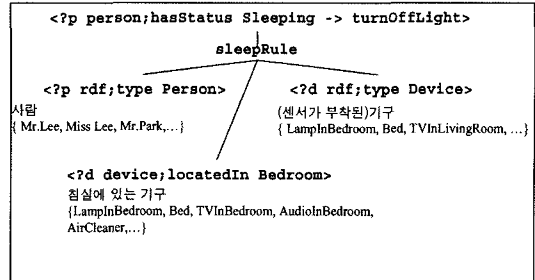


그림 3 센서에 의한 상황정보 쿼리를 제거한 전등 어플리케이션 쿼리 기반의 쿼리트리

있으며, 결과적으로 선인출을 해야 하는 쿼리의 수를 줄일 수 있다. 따라서 선인출에 걸리는 처리시간은 줄어들며 불필요한 상황정보는 제거될 수 있다. 그림 3은 두 단계를 모두 마친 후의 전등 어플리케이션의 쿼리트리를 나타낸다.

마지막으로, 쿼리트리에 남아있는 쿼리를 전처리 메모리를 대상으로 검사를 한다. 그리고 그 결과값을 작업 메모리에 적재함으로써 상황인지 어플리케이션의 실행에 필요한 상황정보만을 유지시킬 수 있다. 그림 4에서 상황정보 선인출의 모든 과정을 보인다.

1. 전처리 메모리와 작업 메모리의 초기화.
   1.1. 상위, 하위레벨 상황정보 온톨로지 전체를 전처리 메모리에 적재.
   1.2. 상위레벨 상황정보 온톨로지를 작업 메모리에 적재.
2. 상황인지 어플리케이션 초기화 시간에, 그 어플리케이션 쿼리 기반의 쿼리트리 구성.
3. 구성된 쿼리트리에서 추론에 의한 상황정보 쿼리를 다른 타입의 상황정보 쿼리로 변환.
4. 쿼리트리에서 센서에 의한 상황정보에 대한 쿼리를 추가적으로 제거.
5. 쿼리트리에 남아있는 쿼리를 전처리 메모리에 적재되어 있는 상황정보를 대상으로 검사.
6. 5번의 결과 상황정보들 작업 메모리에 적재.

그림 4 상황정보 선인출 과정

4. 구 현

우리는 제안한 상황정보 선인출 기법을 기존의 유비쿼터스 컴퓨팅 미들웨어 Active Surroundings[6]에 적용을 시켰다. 우리는 가정 환경을 위한 상황정보 온톨로지를 OWL을 이용해서 설계했으며, Jena2를 온톨로지

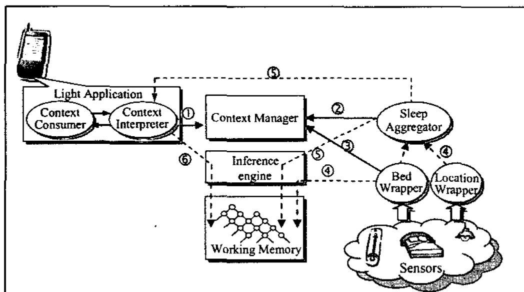
3) 사용자의 상태가 취침 중이면 그 장소의 전등을 꺼주는 상황인지 어플리케이션

쿼리와 추론을 위해 사용하였다. 우선, 우리는 상황정보 선인출 기법을 적용하지 않은 Active Surroundings에서 개발되어 동작하고 있는 전등 어플리케이션의 동작 절차에 대해서 보인다. 미들웨어에서 동작하는 상황인지 어플리케이션의 실행 절차를 설명함으로써, 상황정보 선인출 기법이 어떻게 적용되어 동작하는 지에 대한 이해를 도울 수 있다.

그림 5는 전등 어플리케이션이 동작하는 과정을 보인다. 이 어플리케이션이 동작하기 위해서는 우선적으로 상황정보 관리 시스템에 등록이 되어야 한다. 그림 5에 표현된 각 Wrapper는 Context Toolkit[1]에서 제안된 개념으로 센서로부터의 신호를 상황정보의 형태로 변환하고, 작업메모리에 반영을 하는 Toolkit이다. 그리고 Context Aggregator는 개념적 상황정보를 생성하기 위한 규칙을 가지고 있으며, 현재의 상황정보가 그 규칙을 만족하면 상황정보를 생성해 내는 역할을 한다. Sleep Aggregator는 사람의 현재 상태가 수면 중인지를 추론을 통해 생성해내는 역할을 한다. 사람의 현재 상태가 수면 중이면 불을 끄는 전등 어플리케이션의 경우 Sleep Aggregator가 생성해 내는 상황정보가 필요하다. 그러므로 어플리케이션이 등록이 될 때 Sleep Aggregator 역시 자동으로 함께 등록이 된다. 그리고 Sleep Aggregator가 개념적 상황정보를 생성해 내기 위해 참조하는 Location Wrapper와 Bed Wrapper 역시 관리 시스템에 등록이 된다. 등록된 Aggregator와 Wrapper는 자신들이 생성해 내는 상황정보를 subscribe하고 있는 어플리케이션이나 Aggregator의 목록을 가지고 있다가, 상황정보의 변화가 생겼을 때 그 변화를 알려준다. 실제 동작 시간에, Location Wrapper와 Bed Wrapper는 센서로부터 입력 받은 사용자의 현재 위치 정보와 침대의 상태 정보를 각각 가지고 있으며, 그것을 작업 메모리에 반영을 한다. 그 때, Sleep Aggregator는 반영된 위치 정보와 침대의 상태 정보를 검사한 후, 그것이 가지고 있는 규칙(sleep-rule)을 만족하면 '수면 중' 이라는 상황정보를 생성해내고 작업 메모리에 반영을 시킨다. 마지막으로, 전등 어플리케이션은 사람의 현재 상태가 '수면 중'인가를 검사해서 그 결과가 참이면 전등을 꺼주는 역할을 한다.

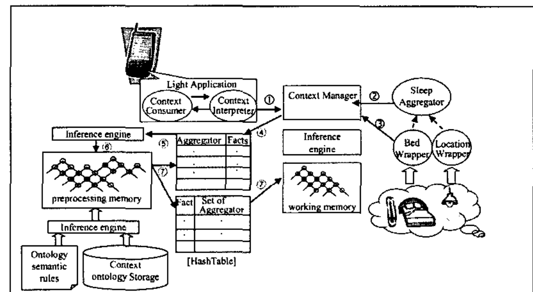
그림 6은 상황정보 선인출 기법을 적용한 미들웨어에서의 전등 어플리케이션의 동작 절차를 나타낸다. 상황정보 관리 시스템은 Context Aggregator와 Wrapper들이 필요에 의해 등록이 될 때 그것들의 쿼리를 상황정보 선인출을 위한 컴포넌트로 전달한다.

우리는 어플리케이션 실행시간에 작업 메모리에 적재되는 상황정보를 정확하게 유지하기 위해 두 개의 해쉬 테이블을 사용한다. 그 중 하나는, [Context Aggre-



①: 어플리케이션은 상황정보 관리시스템에 등록이 되면서, 필요한 Context Aggregator와 Wrapper를 요청한다.
②: 요청된 Context Aggregator가 등록이 되면서, 필요한 Context Wrapper를 요청한다.
③: 요청된 Context Wrapper가 등록이 된다.
④: Context Wrapper가 상황정보의 변화를 Context Aggregator에게 알려주고 그 변화를 작업 메모리에 반영한다.
⑤: Context Aggregator가 상황정보의 변화를 어플리케이션에게 알려주고 그 변화를 작업 메모리에 반영한다.
⑥: 어플리케이션은 동작하기 위한 조건인지를 검사한다.

그림 5 상황정보 선인출 기법을 적용하지 않은 미들웨어에서의 전등 어플리케이션 동작 절차



①: 어플리케이션은 상황정보 관리시스템에 등록이 되면서, 필요한 Context Aggregator와 Wrapper를 요청한다.
②: 요청된 Context Aggregator가 등록이 되면서, 필요한 Context Wrapper를 요청한다.
③: 요청된 Context Wrapper 가 등록이 된다.
④: 등록된 Context Aggregator 가 이미 선인출이 되었는지를 검사한다.
⑤: Aggregator 가 [aggregator-triples] 해쉬 테이블에 존재하지 않으면 쿼리트리를 구성한다.
⑥: 구성된 쿼리트리를 이용해서 필요한 상황정보를 인출한다.
⑦: 선인출된 결과값을 [aggregator-triples], [triple-set of aggregator] 해쉬 테이블에 저장하고 작업 메모리에 적재한다.

그림 6 상황정보 선인출 기법을 적용한 미들웨어에서의 전등 어플리케이션 동작 절차

gator, RDF triples]의 형태로 키(key) 역할을 하는 Context Aggregator가 이미 선인출 되어 그것의 결과값인 triple들이 존재하는지를 체크하는 역할을 한다. 다른 하나는 [RDF triple, Context Aggregator의 집합]형태로 triple이 키의 역할을 한다. 이것은 실행시간에 작업 메모리에 적재되어 있는 triple의 사용여부를 체크해서 메모리에서 제거할 수 있도록 해준다. 여기서 Context Aggregator의 집합이란, 키인 triple을 실제 사용하는 Aggregator의 집합을 의미 한다. 위와 같이 두 개의 해쉬 테이블을 유지함으로써, 선인출된 상황정보를 저장하고 동적으로 작업 메모리에서 추가 또는 제거를 할 수 있다. 예를 들어, 새로운 어플리케이션이 동적으로 상황정보 관리 시스템에 등록이 되면, 그 어플리케이션의 요구에 의해 함께 등록이 되는 Context Aggregator는 첫 번째 해쉬 테이블을 이용해서 이미 선인출된 결과값이 있는지를 검사한다. 만약 그 Aggregator의 선인출 검사 결과가 있다면 선인출의 과정은 생략 된다. 반대로, 더 이상 사용되지 않는 Context Aggregator가 시스템에서 제거될 때, 첫 번째 해쉬 테이블을 통해서 제거되는 Aggregator의 선인출 결과 triple들을 확인하고, 두 번째 해쉬 테이블을 이용해서 제거하려는 triple이 다른 Aggregator에 의해서 사용 중인지를 확인한다. 사용 중인 Aggregator가 없다면, 그 triple을 작업 메모리에서 제거를 시킨다. 상황정보 선인출 기법을 적용을 시킨 후의 전등 어플리케이션 동작을 그림 6에서 보인다.

5. 평 가

우리는 제안한 상황정보 선인출 기법의 적용으로 향상된 추론의 실행속도를 비교해 보았다. 실험은 1GB 메모리, 3.0GHz의 window xp기반의 PC에서 수행을 하였다. 기존에 사용하던 우리의 상황정보 모델은 약 2000개의 RDF triple로 구성이 되어 있는 작은 규모이다. 이 것은 대규모 상황정보에서의 추론 속도 향상을 보이기에는 작은 개수이기 때문에, 우리는 새로운 도메인에 대한 상황정보 정의를 통해 triple의 개수를 약 8000개로 증가시켰다. 또한, 우리는 간단한 상황정보 쿼리에서부터 복잡한 쿼리, 그리고 실제 동작하는 상황인지 어플리케이션에서 사용되는 쿼리를 준비하고 실험에 사용하였다. 표 2는 실험에서 사용된 쿼리를 나타낸다.

그림 7은 간단한 쿼리 Q1에 대한 실험 결과를 보인다. 표 2에 보인 것처럼 Q1은 단순한 형태의 쿼리이다. 그러므로 그 쿼리의 응답시간은 작업 메모리에 적재되어 있는 상황정보의 크기에 영향을 받으며, 그 결과 제안한 기법을 적용하지 않은 경우에는 상황정보 개수의 증가에 따라 응답시간이 증가함을 볼 수 있다. 반면에, 제안한 기법을 적용한 경우에는 상황정보가 증가하더라

표 2 실험에 사용한 온톨로지 쿼리

Query	Description
Q1(Simple query)	<?p rdf:type Person>∧ <?p locatedAt Bedroom>
Q2 (Complex query)	<?p rdf:type Person>∧ <?p gender ?pg>∧ <?p birthDate ?pbr>∧ <?p name ?pn>∧ <?p locatedAt ?pr>∧ <?p hasStatus ?ps>∧ <?u rdf:type UserPreference>∧ <?u onPerson ?p>∧ <?u hasWeight ?uw>∧ <?u hasService ?us>∧ <?d rdf:type Device>∧ <?d used ?du>∧ <?d hasService ?ds>∧ <?d hasState ?dst>∧ <?d hasDimLevel ?ddl>∧ <?d locatedIn ?r>∧ <?r rdf:type Room>∧ <?r hasPerson ?p>∧ <?r hasDevice ?d>∧ <?r SoundLevel ?rs>∧ <?r DoorState ?rd>∧ <?r Brightness ?rb>
Q3 (WatchTV rule)	<?p rdf:type Person>∧ <?d rdf:type Device>∧ <?p locatedAt ?d>∧ <?d hasState xsd:true>
Q4 (Sleep rule)	<?p rdf:type Person>∧ <?d rdf:type Device>∧ <?p locatedAt Bedroom>∧ <?d locatedIn Bedroom>∧ <?d hasState xsd:true>∧ <?l hasDimLevel xsd:0>
Q5 (EnterBed room rule)	<?p rdf:type Person>∧ <?p locatedAt BedroomDoor>

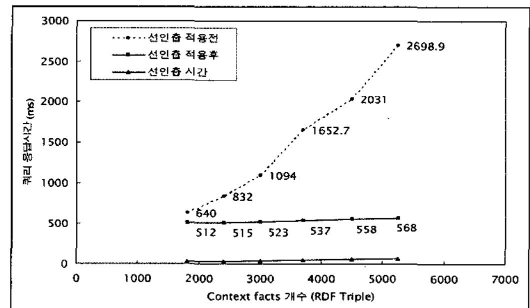


그림 7 Q1 쿼리의 응답시간

도 Q1의 실행에 관련된 상황정보만 유지를 시킬 수 있기 때문에 일정한 응답시간을 보이고 있다. 또한, Q1의 선인출 시간은 무시할 정도로 작음을 알 수 있다.

그림 8은 표 2에서 나타낸 Q2 쿼리의 응답시간을 보인다. Q2는 결과값을 얻기 위해 전체 상황정보의 대부분을 이용하는 복잡한 형태의 쿼리이다. 그러므로 제안된 기법을 적용한 경우와 그렇지 않은 경우 모두에서 상황정보의 증가에 따라 쿼리 응답시간이 증가하는 것을 볼 수 있다. 위와 같이 매우 복잡한 형태의 쿼리 경우에는, 상황정보 선인출을 적용하더라도 작업 메모리의 크기를 많이 줄일 수가 없다. 하지만 이런 경우라 하더라도 선인출의 결과가 전체 상황정보의 크기보다 더 커

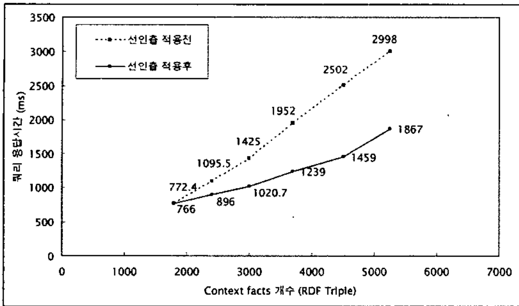


그림 8 Q2 쿼리의 응답시간

질 수는 없기 때문에, 더 짧은 쿼리 응답시간을 유지할 수 있게 된다.

마지막으로, 우리는 실제 사용자의 생활에서 동작하는 상황인지 어플리케이션을 이용하여 실험을 하였다. 표 2 에서 보인 Q3, Q4, 그리고 Q5는 실제 동작하는 어플리케이션에서 사용하는 쿼리들이다. 그림 9의 그래프는 상황정보 선인출 기법을 적용하지 않은 경우에 상황정보의 증가에 따른 쿼리 응답 시간을 보인다.

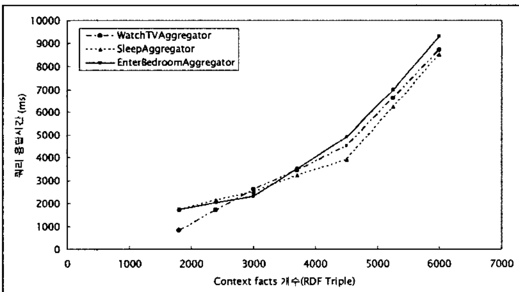


그림 9 상황정보 선인출 기법을 적용하지 않은 Active Surroundings에서의 Q3, Q4, Q5 쿼리의 응답시간

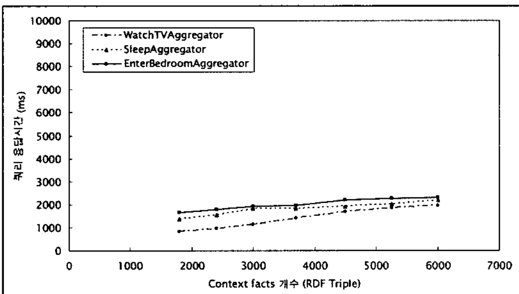


그림 10 상황정보 선인출 기법을 적용한 Active Surroundings에서의 Q3, Q4, Q5 쿼리의 응답시간

그림 10은 제안한 상황정보 선인출 기법을 적용한 경우의 쿼리 응답 시간을 나타낸다. 각 그래프는 쿼리의 응답시간과 실제 선인출에 사용된 처리 시간을 합한 값을 나타낸다. 그림 9와 비교를 했을 때, 쿼리의 응답시간이 일정하게 유지되는 것을 볼 수 있다.

우리는 위에서 보인 실험결과를 통해, 작업 메모리에 적재되어 있는 상황정보의 개수가 2,000 triple 미만일 경우에는, 선인출 기법을 적용한 경우와 그렇지 않은 경우 모두 비슷한 쿼리 응답 시간을 보임을 알 수 있었다. 하지만 그 개수가 2,000 triple 보다 큰 경우에는, 상황정보 선인출 기법을 적용한 경우가 그렇지 않은 경우보다 좋은 성능을 보임을 알 수 있었다.

6. 요약 및 향후 연구 방향

최근, 상황정보의 공유와 재사용, 그리고 추론을 통해 센서로부터 얻을 수 없는 상황정보를 얻을 수 있다는 이점으로 온톨로지 기반의 상황정보 모델이 많이 사용되고 있다. 하지만 추론의 경우, 그 처리 시간이 상황정보의 크기에 따라 함께 증가한다는 단점을 가지고 있다. 그러므로, 온톨로지 기반의 상황정보 모델을 사용하는 상황정보 관리 시스템에서는 추론의 속도를 향상시키는 모듈 또는 기법이 필요하다. 본 논문에서는, 추론 속도를 향상시키기 위하여 상황인지 어플리케이션 쿼리를 기반으로 실행에 필요한 상황정보를 선인출하고 유지하는 방법으로 작업 메모리에 적재되는 상황정보의 크기를 줄였다. 그리고 실제 온톨로지 기반의 상황정보 모델을 사용하는 유비쿼터스 컴퓨팅 미들웨어, Active Surrounding에 제안한 기법을 적용하여 기존의 경우와 비교해서 추론에 걸리는 시간이 많이 줄어든 것을 실험을 통해 증명하였다. 현재 우리는 제안한 상황정보 선인출 기법에 쿼리 최적화 기법을 적용하여 선인출에 걸리는 시간을 줄이는 노력을 함께 진행 중이다.

참 고 문 헌

[1] Dey, A.K. et al., "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications," Human-Computer Interaction Journal. Vol. 16, No. 2-4, pp. 97-166, 2001.

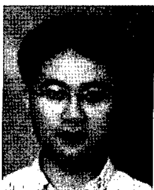
[2] Baus, J. et al., "A resource-adaptive mobile navigation system," In Proceedings of Intl. Conf. on Intelligent User Interfaces, San Francisco, 2002.

[3] Kindberg, T. et al., "People, places, things: Web presence for the real world." In Proceedings of IEEE Workshop on Mobile Computing Systems and Applications, 2000.

[4] Henricksen, K. et al., "Modeling Context Information in Pervasive Computing System," Pervasive 2002, LNCS 2414, pp. 167-180, 2002.

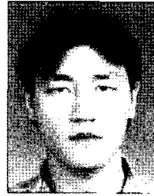
[5] Wang, X.; Dong, J.S.; Chin, C.Y.; Hettiarachchi,

S.R.; and Zhang, D., "Semantic Space: an infrastructure for smart spaces," Pervasive Computing, IEEE. Vol. 3, No. 3, pp. 32-39, 2004.
[6] Lee, D. et al., "A Group-Aware Middleware for Ubiquitous Computing Environments," In Proceedings of the 14th International Conference on Artificial Reality and Telexis-tence (ICAT), 2004.
[7] Ranganathan, A. and Campbell, R.H., "An Infrastructure for Context-Awareness based on First Order Logic," Journal of Personal and Ubiquitous Computing. Vol. 7, No. 6, pp. 353-364, 2003.
[8] Khedr, M. and Karmouch, A., "ACAI: Agent-Based Context-aware Infrastructure for Spontaneous Applications," Journal of Network & Computer Application, 2004.
[9] Levy, A.Y. et al., "Speeding up inferences using relevance reasoning: a formalism and algorithms," Artificial Intelligence. Vol. 97, No. 1-2, pp. 83-136, 1997.
[10] Chen, H.; Finin, T.; and Joshi, A., "An Ontology for Context-Aware Pervasive Computing Environments," Special Issue on Ontologies for Distributed Systems, Knowledge Engineering Review, 2004.
[11] Zou, Y. et al., "F-OWL: an Inference Engine for the Semantic Web," In Proceedings of the 3rd International Workshop on Formal Approaches to Agent-Based Systems, 2004.
[12] Carroll, J.J. et al., "Jena: Implementing the Semantic Web Recommendations," WWW2004, 2004.
[13] Racer. Available online at http://www.racer-systems.com/.
[14] FaCT. Description Logic (DL) classifier. Available online at http://www.cs.man.ac.uk/~horrocks/FaCT/.
[15] Hoolet. OWL-DL Reasoner. http://owl.man.ac.uk/hoolet
[16] Triple, http://triple.semanticweb.org/
[17] Gu, T.; Wang, X.H.; Pung, H.K.; and Zhang,D.Q., "An Ontology-based Context Model in Intelligent Environments," In Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference, 2004.
[18] Wilkinson, K.; Sayers, C.; Kuno, H.A.; Reynolds, D.; and Ding ,L., "Supporting Scalable, Persistent Semantic Web Applications," IEEE Data Eng. Bull. Vol. 26, No. 4, pp. 33-39. 2003.



이 재 호

2002년 영남대학교 컴퓨터공학과(학사) 2005년 한국정보통신대학교 공학부(석사). 2005년 9월~현재 삼성전자 정보통신 사업부. 관심분야는 Context Modeling, Context-aware Computing



박 인 석

2000년 경북대학교 컴퓨터공학과(학사) 2002년 한국정보통신대학교 공학부(석사) 2002년 9월~현재 한국정보통신대학교 공학부 박사과정. 관심분야는 Context and Data Management in Ubiquitous Computing

이 동 만

1982년 서울대학교 컴퓨터공학과(학사) 1984년 한국과학기술원 전산학과(석사) 1987년 한국과학기술원 전산학과(박사) 1988년~1991년 미국 HP Information Technology Operation, Senior MTS 1991년 1월~1992년 7월 한국과학기술원 Senior 연구원. 1992년~1997년 9월 미국 HP Workstation System Division, Technical Contributor. 1998년~현재 한국정보통신대학교 교수. 관심분야는 Distributed Systems, Computer Networks, Mobile Computing, Pervasive Computing

현 순 주

1981년 경북대학교 전자공학과(학사) 1984년~1986년 벨기에 Bell Telephone/ITT 초빙연구원. 1987년 벨기에 Katholike Universitate 전자공학과(석사). 1995년 미국 Florida 대학교 전자전기 및 컴퓨터 공학과(박사). 1983년~1997년 전자통신연구원(ETRI) 연구원. 1998년~현재 한국정보통신대학교 부교수. 관심분야는 Context Management, Ubiquitous Computing, Temporal Database, Multimedia Database