http://dx.doi.org/10.3745/KTSDE.2013.2.8.543

Automatic Extraction of Abstract Components for supporting Model-driven Development of Components

Yun Sang Kwon† · Park Min Gyu†† · Choi Yunja†††

ABSTRACT

Model-Driven Development(MDD) helps developers verify requirements and design issues of a software system in the early stage of development process by taking advantage of a software model which is the most highly abstracted form of a software system. In practice, however, many software systems have been developed through a code-centric method that builds a software system bottom-up rather than top-down. So, without support of appropriate tools, it is not easy to introduce MDD to real development process. Although there are many researches about extracting a model from code to help developers introduce MDD to code-centrically developed system, most of them only extracted base-level models. However, using concept of abstract component one can continuously extract higher level model from base-level model. In this paper we propose a practical method for automatic extraction of base level abstract component from source code, which is the first stage of continuous extraction process of abstract component, and validate the method by implementing an extraction tool based on the method. Target code chosen is the source code of TinyOS, an operating system for wireless sensor networks. The tool is applied to the source code of TinyOS, written in nesC language.

Keywords : Component Model, Reverse Engineering, nesC, MARMOT Framework

모델기반 컴포넌트 개발방법론의 지원을 위한 추상컴포넌트 자동 추출기법

윤 상 권† · 박 민 규†† · 최 윤 자†††

요 약

모델 중심 개발 방법론은 시스템 개발의 추상화 수준을 높임으로써 구현 세부 사항과는 독립적으로 중요한 요구사항과 설계 문제 등을 개발 단계 초기에 점검할 수 있도록 해준다. 그러나 현재까지 많은 소프트웨어가 코드중심, 상향식 개발방식을 통해 개발되어 왔고, 따라서 적절한 도구의 지원 없이는 이러한 모델 중심 개발 방법을 도입하는 것이 쉽지 않다. 현재 코드 중심으로 개발된 시스템에 모델 중심 개발 방법론을 도입할 수 있도록 코드로부터 모델을 생성하는 역공학적인 접근방법이 연구되고 있으나 대부분 코드에서 일차적인 모델을 추출하는 데 그치고 있다. 하지만 추상컴포넌트 개념을 이용하면 이러한 모델을 일차적으로 추출에 그치지 않고, 추출된 일차 모델을 상위수준의 추상화 단계로 연속적으로 추출할 수 있다. 본 연구에서는 이러한 추상컴포넌트의 연속적인 추출 과정 중 첫 번째 단계인 코드로부터 최하위 기반(base) 추상컴포넌트를 추출하는 과정을 자동화할 수 있는 기법을 제안하고, 실제 도구 구현을 통해 그 기법의 타당성을 평가한다. 실험 대상으로 선택된 코드는 무선센서 네트워크 운영체제인 TinyOS의 소스 코드이며, 해당 소스 코드는 nesC 언어로 작성되었다.

키워드 : 컴포넌트 모델, 역공학, nesC, MARMOT Framework

1. 서 론

모델 중심 개발(Model Driven Development, MDD)[1]은 모델 중심 아키텍처(Model Driven Architecture, MDA)를 확장한 것으로, 프로그램의 핵심적인 처리과정과 논리를 구현 기술로부터 분리시킴으로써 개발자가 구현 세부사항에 앞서 문제 해결 방법에 더 집중할 수 있게 한다[6]. 모델 중심 개발을 이용했을 때 얻을 수 있는 대표적인 이점들은 다음과 같다. 첫째, 시스템의 각 부분을 모듈화함으로써 시스

※ 이 논문은 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(2011-0003758).
† 준 회 원 : 경북대학교 전자전기컴퓨터학부 석사
†† 준 회 원 : 경북대학교 IT 대학 컴퓨터학부 박사과정
††† 정 회 원 : 경북대학교 IT 대학 컴퓨터학부 부교수
논문접수 : 2012년 8월 17일
수 정 일 : 1차 2013년 3월 8일
심사완료 : 2013년 4월 15일
* Corresponding Author : Choi Yunja(yuchoi76@knu.ac.kr)

템의 이해도를 높이고 변경 사항 발생시 국지적인 변경만으로 해결이 가능하여 빠른 대처와 높은 신뢰도를 유지할 수 있다[7]. 둘째, 개발 초기단계에서부터 모델중심의 검증이 가능해져 초기오류를 방지하고 시스템의 품질을 향상시킬 수 있다.

하지만 코드 중심의 개발방식이 만연한 개발현장에서 모델 중심 개발 방법을 적용하기에는 현실적인 어려움이 있으며, 추가비용을 최소화하면서 개발방식을 점진적으로 전환하기 위해서는 지원도구의 개발이 필수적이다. 그 일환으로 이미 작성된 코드로부터 추상컴포넌트[3]를 추출하여 컴포넌트 모델을 생성하는 역공학적인 접근방법이 연구되고 있다[2].

본 연구에서는 이러한 역공학적 접근방법의 일부로서, 무선 센서 네트워크 운영체제로 널리 사용되고 있는 TinyOS[5]의 프로그램 소스 코드로부터 UML클래스 다이어그램과 상태차트로 구성된 추상컴포넌트를 자동 추출하는 방법과 도구를 개발하였다.

TinyOS는 컴포넌트 개념을 반영한 nesC[4,5] 프로그래밍 언어로 작성되었다. 따라서 본 연구에서는 nesC파서를 구현하였으며, 그 과정에서 파서가 생성할 AST(Abstract Syntax Tree, 추상구문트리)의 형태와 생성된 nesC AST를 순회하면서 추상컴포넌트 모델을 자동추출하기 위한 nesC 문법 요소와 UML 다이어그램 요소 사이의 대응 규칙을 정의하였다. 모델 추출기는 nesC 파일을 입력 받아 XML 형식의 UML컴포넌트 모델을 출력하며, 그 결과는 MARMOT 프레임워크 지원도구를 통하여 가시화되고 검증되었다.

본 논문의 나머지 부분은 다음과 같이 구성된다. 제 2 장에서는 본 연구의 배경이 되는 MARMOT 프레임워크와 nesC 프로그래밍 언어에 대해 간략히 설명하며, 제 3 장에서 관련 연구를 소개한다. 제 4장은 nesC 코드로부터 컴포넌트 모델을 추출하는 방법을, 제 5장에서는 추출기의 설계 방식에 대해 설명한다. 제 6 장에서는 도구의 적용결과를 통하여 도구의 효용성을 보이며, 제 7 장에서 결론과 향후 과제에 대한 토의로 마무리한다.

2. 연구 배경

2.1 MARMOT 개발 방법론과 역공학적 적용

MARMOT 개발 방법론[3]은 모델 중심·컴포넌트 기반의 하향식 개발 방법론으로서, 시스템 전체를 하나의 추상컴포넌트로 나타내고 이를 점진적으로 세분화해 나가면서 개발하는 방법이다. 각 세분화 과정에서 모델검증 기법을 사용하여 세분화 전후 컴포넌트 사이의 행위 일관성 및 다양한 특성에 대한 검증을 수행할 수 있다.

MARMOT 개발 방법론에서 정의하는 추상컴포넌트는 크게 외부로 보여지는 명세 부분과 내부의 구현 부분으로 이루어진다(Fig. 1). 외부 명세 부분은 컴포넌트가 외부로 제공하는 행위를 정의하며, 내부 구현 부분은 외부에 제공되는 기능들이 실제로 구현되는 방식을 나타낸다.

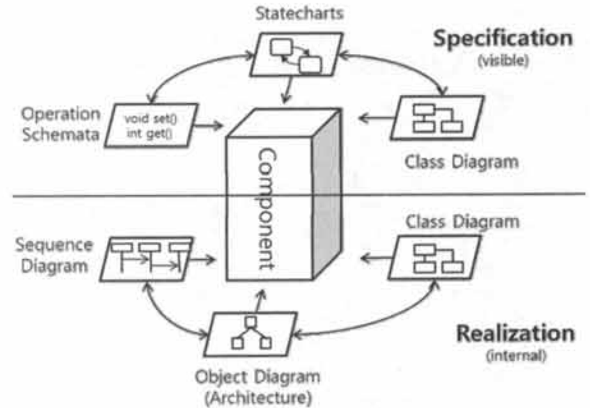


Fig. 1. Outline of abstract component

MARMOT 에서는 시스템 전체를 하나의 추상컴포넌트로 정의하고, 내부명세에서 하향식으로 분화하여 컴포넌트 나무 형태로 개발이 진행되며, 제일 마지막으로 분화된 추상컴포넌트가 실제 물리적 컴포넌트와 대응되는 구조를 갖는다(Fig. 2).

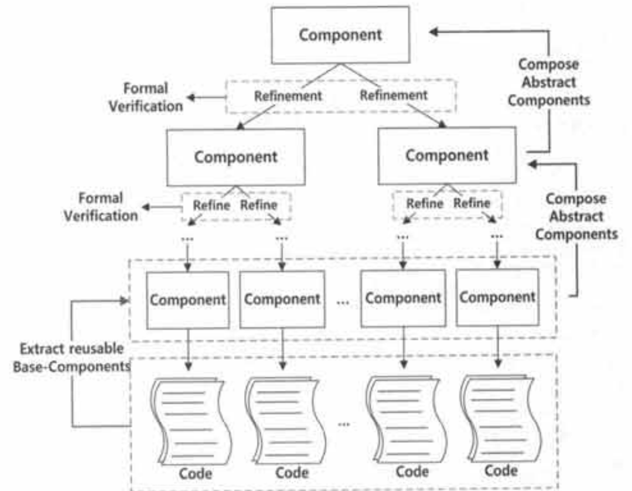


Fig. 2. Top-down refinement of abstract component

참고 문헌[2]에서는 이러한 추상컴포넌트의 개념을 이용하여 역공학적 접근 방법을 제안하였는데, 이 접근 방법에서는 물리적 코드로부터 시작해서 시스템 전체를 나타내는 응용프로그램 단계의 추상컴포넌트까지 컴포넌트 모델을 상향식으로 구성하게 된다. 가장 먼저 코드로부터 기반 추상컴포넌트를 추출하게 되며, 추출된 기반 추상컴포넌트들의 상태차트를 합성하여 점진적으로 더 높은 단계로 추상화하게 된다. 최종적으로 얻어진 컴포넌트 모델은 코드 중심으로 개발된 시스템에 MDD를 도입하는 데 이용될 수 있다.

본 연구에서는 코드로부터 추상컴포넌트 모델을 생성하는 이러한 단계 중 코드에서 최하위기반 추상컴포넌트를 추출하는 첫 번째 단계를 자동화하는 데 초점을 맞추었으며, 이 단계를 자동화할 수 있는 기법을 제안하고 실제 추출 도구 구현을 통해 그 기법의 타당성을 평가하였다. 기반 컴포넌

트의 추출 이후 단계적 추상컴포넌트 합성을 통한 상위 모델 합성은 참고 문헌[15]에서 다루고 있으며, 지면관계상 논문의 범위에 포함하지 않았다. 추출의 대상이 되는 코드는 프로그래밍 언어에 따라서 종류가 다양하므로 실제 추출 도구 구현을 위해서는 특정 언어를 실험 대상으로 선택하여야 한다. 본 연구에서 선택한 언어는 무선센서 네트워크 운영체제인 TinyOS의 작성에 사용된 nesC이다.

2.2 nesC 프로그래밍 언어

nesC는 TinyOS를 효과적으로 구현하기 위해 고안된 언어로[4], 컴포넌트기반, 이벤트기반이라는 TinyOS의 특성을 구문에 반영하며 TinyOS 응용프로그램 개발에 사용된다. nesC는 C 언어의 기본 구문들을 모두 지원할 뿐 아니라, 컴포넌트와 인터페이스의 명세를 지원하는 특징을 지닌다.

인터페이스 파일에는 인터페이스 제공자가 구현해야 할 커맨드와 인터페이스 사용자가 구현해야 할 이벤트(핸들러)들이 함수 원형의 형태로 정의되어 있으며(Fig. 3), 인터페이스 사용자는 커맨드를 호출하고 인터페이스 제공자는 이벤트를 발생시킨다. 즉, 커맨드는 인터페이스 사용자가 사용할 수 있는 기능(operation)을 나타내며, 이벤트는 인터페이스 제공자가 발생시킬 수 있는 특정 작업의 완료 혹은 하드웨어 인터럽트 등의 신호 (signal)를 나타낸다. 따라서 인터페이스 제공자는 인터페이스의 모든 커맨드를 구현하여야 하며, 인터페이스 사용자는 인터페이스의 모든 이벤트(핸들러)를 구현하여야 한다.

```
Interface Timer {
        command result_t stop();
        event result_t fired();
}
```

Fig. 3. Interface file of nesC

```
module LedsP @safe() {
  provides {
    interface Init;
  }
  uses {
    interface GeneralIO as Led0;
  }
}
implementation {...}
```

Fig. 4. Interface specification of nesC component

nesC 응용프로그램은 컴포넌트들을 연결·조립함으로써 만들어진다. 각각의 컴포넌트는 인터페이스를 제공하거나 사용하게 되며, 이 인터페이스를 통해서만 컴포넌트에 접근 가능하다. Fig. 4에서와 같이 모든 컴포넌트는 자신이 사용하거나 제공하는 인터페이스를 명세하게 된다.

nesC의 컴포넌트에는 단위컴포넌트(module)와 조합컴포넌트(configuration) 두 종류가 있다. 단위컴포넌트는 실제 코드를 통해 인터페이스를 구현하는 반면(Fig. 5), 조합컴포넌트는 이미 구현되어 있는 다른 컴포넌트들을 연결, 조합하여 간접적으로 인터페이스를 구현하게 된다(Fig. 6).

```
module SurgeM {…}
implementation {
        uint16_t sensorReading;
        command result_t StdControl.init() {
        return call Timer.start(2, 1000);
        }
...
}
```

Fig. 5. Implementation of nesC component (module component)

```
configuration TimerC {
        provides {
                interface StdControl;
                interface Timer;
        }
}
implementation {
        components TimerM, HWClock;
        StdControl = TimerM.StdControl;
        TimerM.Clk -> HWClock.Clock;
}
```

Fig. 6. Implementation of nesC component (configuration component)

3. 관련 연구

코드로부터의 모델 추출 기법은 다양한 방식으로 연구되어 왔다. 참고문헌[14]에서는 모델 기반의 재공학 방법론을 제시하고 있다. 과거부터 사용되어 온 레거시 시스템을 확장하거나 현대적인 시스템으로 재공학할 경우 모델 기반의 방법과 코드 기반의 방법을 사용할 수 있는데, 코드 기반의 방법을 사용할 경우 모델 기반 방법에 비해 코드의 길이가 더 길어지며, 시스템의 결합도 및 복잡도가 더 커지게 됨을 보여주고 있다. 이를 통해 코드에서 모델을 추출하여 모델 기반으로 옮겨 오는 것이 소프트웨어 유지 보수 측면에서 많은 이득을 가져옴을 보였다.

참고 문헌[9]에서는 C++ 소스 코드로부터 UML 클래스, 시퀀스, 액티비티 다이어그램을 추출하여 가시화 해주는 도구를 개발하였다. 참고 문헌[10]에서는 철도 제어 시스템으로부터 역공학을 통해 유스케이스 다이어그램, 클래스 다이어그램, 시퀀스 다이어그램, 상태 차트 등을 추출하여 명세와 코드 사이의 일치성에 관한 검증 및 코드에 대한 리팩터링 작업을 수행하였다. 참고문헌[12]에서는 기호 실행 (symbolic execution) 기법을 통해 TinyOS 프로그램으로부터 상태기계를 생성하는 도구를 소개하고 있다. 이 논문에서는 기호 실행 기법을 통해 생성된 기호 상태(symbolic state)를 일반적인 상태기계로 변환하는 방법을 소개하고 있으며, 이를 통해 다양한 TinyOS 응용프로그램에 대해 상태기계를 생성하였

다. 또한 생성된 상태기계를 분석하여 코드 분석에서 발견하지 못했던 잠재적 오류 가능성을 발견하였다.

하지만 기존 연구들은 코드에서 다이어그램 형태의 일차적인 행위모델만을 추출하는데 그치고 있어, 조합컴포넌트를 통해 컴포넌트의 연동이 복잡하게 이루어지는 TinyOS와 같은 모델을 표현하기에는 적합하지 않다. 또한 기존 모델들은 소스코드의 분석이라는 측면에서 유의미하지만, MARMOT과 같은 하향식 개발방식에 적용하기 위한 상위 수준까지의 추상화를 지원하고 있지는 못하다. 본 연구는 추상컴포넌트라는 개념을 적용해 추출된 일차 모델을 상위 수준의 추상컴포넌트로 연속적으로 끌어 올릴 수 있도록 기반을 제공한다는 점에서 기존연구들과 차별화된다.

본 연구와 같이 컴포넌트의 개념을 활용해 모델을 추출하는 연구에서는 참고문헌[13]이 유사하다. 이 연구에서는 클래스들과 인터페이스들 간의 구현, 사용 관계를 분석해 초기 컴포넌트 집합을 얻은 뒤, 모든 컴포넌트 쌍에 대해 이름의 유사도, 의존성 등의 메트릭을 사용하여 가장 유사한 컴포넌트 쌍을 합성하여 더 높은 단계로 추상화하는 방식을 사용하였다. 하지만 이 연구는 추출된 다수의 컴포넌트의 합성에 집중하고 있으며, 컴포넌트 모델의 역공학적 추출의 관점은 자세히 설명하지 않는다. 이는 목표 코드의 모든 구조를 온전히 컴포넌트로 추출하고자 하는 본 연구와는 접근 방식 및 목적이 다르다.

본 연구에서는 역공학적 접근방법을 통해 목표 코드로부터 UML클래스 다이어그램과 상태차트로 구성된 추상컴포넌트를 자동 추출하는 기법을 제시하고, 이를 구현하는 도구를 개발하였다. 본 연구는 MARMOT개발 방법론[3]의 가장 기반적인 컴포넌트 추출에 대해 다루고 있으며, 제안한 기법들은 참고문헌[8, 15]에 활용될 수 있다.

4. 컴포넌트 모델 추출 기법

nesC의 컴포넌트 구성방식은 MARMOT 의 추상컴포넌트와 자연스럽게 일치한다. nesC 조합컴포넌트는 MARMOT 추상컴포넌트의 외부구조와 외부행위양식과 대응되며, nesC 단위컴포넌트는 MARMOT 추상컴포넌트의 내부구조와 내부행위양식에 대응될 수 있다. 본 연구에서 코드로부터 컴포넌트 모델을 추출하는 과정은 크게 두 단계로 이루어져 있다. 첫 번째 단계에서는 nesC 코드에 대한 구문 분석을 수행하여 AST (Abstract Syntax Tree, 추상 구문 트리)를 생성하며, 두 번째 단계에서는 생성된 AST를 순회하며 만나는 노드 별로 그에 해당하는 컴포넌트 모델을 생성한다.이번 장에서는 nesC 언어의 각 구문 구조와 UML 메타모델 요소 사이의 대응 규칙에 대해 설명하고, nesC 프로그래밍 언어에 대한 AST 설계에 대해 설명한다.

4.1 nesC 컴포넌트와 추상컴포넌트의 대응관계

Table 1은 nesC의 주요 구성요소가 추상컴포넌트와UML 에서는 어느 구성요소로 대응되는지에 관한 연관관계를 보이고 있다. nesC 단위컴포넌트는 구체적인 구현내용을 명세하고 있으므로 MARMOT 추상컴포넌트의 내부 행위양식에 해당되며 UML 상태다이어그램으로 대응하였다. 반면, 조합컴포넌트는 실제 구현이 아닌 컴포넌트들 사이의 연결관계를 명세하고 있으므로, MARMOT 추상컴포넌트의 외부명세 부분에 대응된다.

Table 1. Correspondence rule for extracting abstract components

nesC elements		Abstract component element	UML element
module component	Name	Component name	Class name
	Interface	Component connection	Interface
	Implementation	Component internal behaviour	State Diagram
Configuration Component	Name	Component name	Class name
	Interface	Component connection	Interface
	Implementation	Component internal structure	Class and class relationship

Fig. 7은 하나의 단위컴포넌트를 하나의 클래스로 대응한 예이다. 단위컴포넌트의 인터페이스 명세 부분은 UML클래스의 포트로, 구현 부분은 클래스의 상태차트와 내부 변수 (Attribute)로 대응하였다.

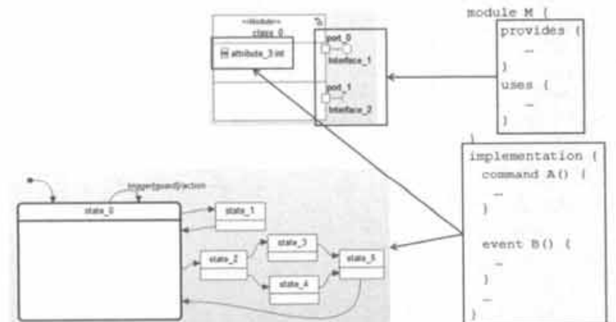


Fig. 7. Abstract component model generation based on module component

컴포넌트의 행위를 나타내는 상태도는 기본적으로 하나의 초기 상태를 가지며, 각 함수에 대한 호출 이벤트를 초기 상태에서 나와서 다시 초기 상태로 돌아가는 전이 고리 (Loop) 형태로 표현되었다. 무선 센서 네트워크는 특정한 종료 시점이 없이 계속적으로 작동하는 반응형(reactive) 시스템이므로 상태도는 별도의 종료 상태를 갖지 않는다.

Fig. 8은 추상컴포넌트 구성요소와 조합컴포넌트의 대응을 도식화한 것이다. 우선 조합컴포넌트 전체를 하나의 클래스로 대응하고 인터페이스 명세는 클래스의 포트로 대응하였다. 또한 조합컴포넌트 구현 부분의 정보는 추상컴포넌

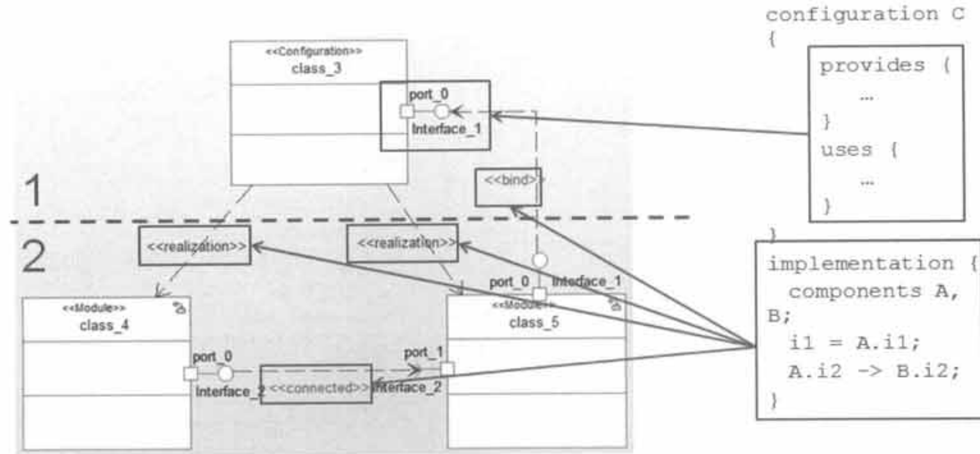


Fig. 8. Abstract component model generation based on configuration component

트가 세분화되는 하위 컴포넌트들과 그들 간의 연결 정보를 제공한다. 따라서 이 정보는 클래스 사이의 의존관계를 나타내는 UML 요소인 Dependency와 포트 사이의 정보흐름을 나타내는 UML 요소인 InformationFlow로 표현되도록 하였다. Fig. 8에 나타나 있는 UML 모델 전체가 하나의 추상컴포넌트 모델을 나타내며, 이를 기반으로 상향식 합성이 일어나게 된다. Fig. 8에서 1로 표시한 부분은 외부 모델, 2로 표시한 부분은 내부 모델을 나타낸다.

4.2 세부 대응 규칙

Fig. 9와 Fig. 10은 각각 nesC의 문법 구조와 nesC 모델링에 사용된 UML 메타모델 요소를 나타내며, 대응되는 요소끼리 같은 번호로 표시하였다. Fig. 9의 (+) 표시는 요소가 1회 이상, (*) 표시는 요소가 0회 이상 반복될 수 있음을 의미한다.

먼저 nesC파일 전체를 나타내는 심벌인 nesC_file은 UML의 Package로 대응하였으며, nesC 파일 중 인터페이스 파일을 나타내는 interface_definition은 Interface로, 컴포넌트 파일을 나타내는 component는 Class로 대응하였다. 또한 컴포넌트의 인터페이스 사용/제공 관계를 나타내는 comp_spec부분은 Port로, 컴포넌트 중 단위컴포넌트의 구현부분을 나타내는 module_impl은 StateMachine으로, 조합컴포넌트의 구현부분을 나타내는 config_impl은 Information Flow와 Dependency로 대응하였다.

Fig. 9에서 module_impl은 func_def(함수 정의) 들로 구성되어 있으며, 각 func_def의 하위에는 comp_stmt(함수 몸체)가 있음을 알 수 있다. 이와 같은 comp_stmt의 하위에는 다양한 종류의 구문이 올 수 있으며(함수호출, 대입 등), 그 구문들의 의미가 상대적으로 구체적이기 때문에 추상적 의미를 갖는 상위 요소들에 대한 대응 관계인 Fig. 9, Fig. 10과 같이 단순히 일대일 대응 관계로 설명하기에는 어려움이 있다. 따라서 func_def이하 nesC 요소들의 UML 요소에 대한 대응관계는 의사코드(pseudocode)를 통해 상세히 설명하고자 한다.

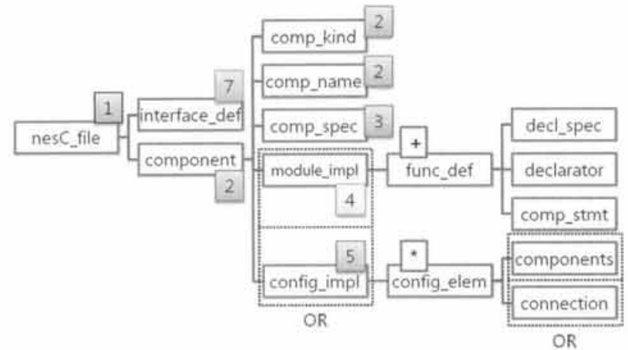


Fig. 9. Grammar structure of nesC

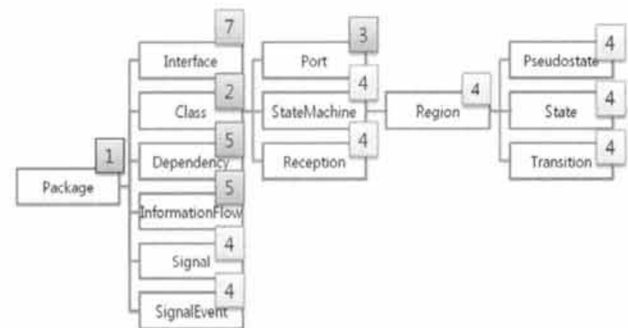


Fig. 10. UML Meta model elements

Fig. 11은 module_impl구조를 상태기계로 변환하기 위한 일반적인 대응 규칙이다. module_impl은 하나의 상태기계로 변환되며, 상태기계를 생성하기 위해 상태기계의 시작점인 초기상태를 생성한다. 다음으로 func_def는 module_impl에 선언되어 있는 함수를 나타내며, func_def을 표현하기 위해 초기상태에서 출발하는 하나의 상태전이를 생성한다. 해당 상태전이는 func_def에 해당하는 함수의 호출을 응대하는 상태전이이다. 함수의 행위는 func_def하위의 comp _stmt에 정의되어 있으며, 따라서 해당 comp_stmt를 순회하며 새로 추가된 상태전이에 행위를 작성한다. func_def하위의 모든

```
Region Reg;
Transition TrPointer;
MakeStateMachine(Node node) {
   NodeType nodeType = node.GetNodeType();
   Switch(nodeType) {
   Case module_impl:
      // initialize
      State initState = Reg.createState();
      // recursively traverse func_def under module_impl
      Node ChildNode = GetChildNode(node);
      foreach(func_def in ChildNode){
         MakeStateMachine(func_def);
      }
      break;
   Case func_def:
      // generate transition from initial state
      TrPointer =Reg.CreateTransition(From: initState);
      // make trigger event responding function calls
      TrPointer.addTrigger(createTrigger(node));
      // recursively traverse comp_stmt under func_def
      Node ChildNode = GetChildNode(node);
      foreach(cNode in ChildNode){
         if(cNode.GetNodeType().equals("comp_stmt")){
            CreateFunctionBehavior(cNode);
         }else{ ... }
      }
      // set target state of latest transition to initial state
      TrPointer.SetTarget(initState);
      break;
   }
}
```

Fig. 11. Correspondence rule between module_impl and state machine

행위가 작성된 이후 마지막으로 생성된 상태전이의 도달점을 다시 초기상태로 설정해 함수 호출을 하나의 전이 고리로 표현하였다.

Fig. 12은 상태전이의 행위 생성에 대한 일반적인 대응규칙이다. 상태전이의 행위는 node가 표현하고 있는 내용을 토대로 생성하며, 깊이 우선 탐색방법(DFS)을 통해 하위 노드가 있을 경우 하위 노드를 우선적으로 처리한다. node가 담고 있는 행위는 참고문헌[8]에서 제안한 Fig. 13의 행위문법을 통해 표현된다. 예를 들어 Fig. 14의 original state에서 waiting state로의 상태전이가 가지고 있는 행위 globalTimer ! timerTick(); 는 globalTimer.timerTick(); 와 같은 call 행위구문을 Fig. 13의 문법을 통해 표현한 것이다. 예의 경우는 call의 classifier_name을 globalTimer로, operation_name을 timerTick으로 대응해 call의 행위를 표현하였다.

일반적으로, 생성된 행위는 상태의 분기와 새로운 상태전이의 추가 없이 가장 마지막으로 생성된 상태전이에 추가된다. 하지만 call, signal, if는 행위의 특성상 상태를 분기해야 할 필요성이 있기 때문에 아래에 따로 설명한다.

Fig. 14은 call과 signal의 행위에 따른 상태의 분화를 나타낸 것이다. call과 signal은 행위의 특성상 다른 컴포넌트

```
CreateFunctionBehavior(Node node){
   // recursively traverse 'comp_stmt' under 'node'
   Node ChildNode = GetChildNode(node);
   foreach(comp_stmt in ChildNode){
      CreateFunctionBehavior(comp_stmt);
   }
   NodeType nodeType = node.GetNodeType();
   Switch(nodeType){
   Case assign, return, decl:
      TrPointer.addAction(CreateAction(node));
      break;
   Case call, signal:
      TrPointer.addAction(CreateAction(node));
      if(return value is needed) {
         State waitState = Reg.createState();
         TrPointer.setTarget(waitState);
         // create state transition to catch return value
         TrPointer =
            Reg.createTransition(From: waitState);
         // create trigger event to check whether it caught
            return value
         TrPointer.addTrigger(CreateTrigger(node));
      }
      break;
   Case if:
      HandleIfStmt(node);
      break;
   ...
   }
}
```

Fig. 12. Rule for generating behaviour of state transition

action :	elementary_action | if ;
elementary_ action :	assignment ';' | return ';' | call ';' | signal ';' ;
if:	IF simple_expression THEN action ENDIF | IF simple_expression THEN action ELSE action ENDIF;
assignment:	attribute_name '=' simple_expression ;
call:	classifier_name '!' operation_name '(' paramList ')' | operation_name '(' paramList ')' ;
signal:	signal_name '(' paramList ')' ;
return:	RETURN simple_expression | RETURN paramList
paramList :	/* empty string */ | literal | paramList ',' literal

Fig. 13. Action language representing behaviour of state transition

에 요청을 전송하며, 요청을 받는 컴포넌트가 수신을 했는지의 여부 및 반환 값의 수신을 기다려야 하는 상황이 발생할 수 있다. 따라서 다른 행위와는 다르게 상태를 분할해 수신을 대기하는 상태를 생성하고, 수신을 대기하는 트리거(trigger) 이벤트를 가진 상태전이를 생성해 수신을 대기하는 상황을 처리한다.

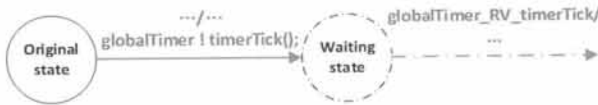


Fig. 14. Branch of condition based on behaviour: call, signal

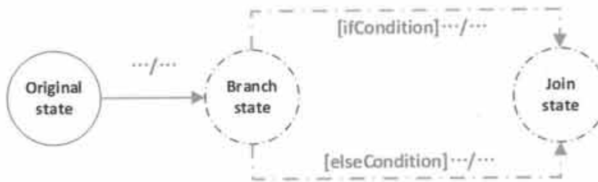


Fig. 15. Branch of condition based on behaviour: if

```
1.   command int Interf1.func1(int b) {
2.       int a = 0;
3.       call Interf1.func2(call Interf1.func3(), b);
4.       a = b;
5.       if(a > 0) {
6.           call Interf2.func4(2);
7.       }
8.       else {
9.           call Interf2.func5(5);
10.      }
11.      return 0;
12.  }
```

Fig. 16. Example code of func_def

Table 2. Model generation process with example code

L.N	Model generation steps based on algorithm in (Fig. 12)
1	Case func_def: func_def를 만났으므로 StateMachine의 초기상태로부터 나오는 Transition 하나가 생성되었으며, 함수 타입(command), 인터페이스 이름(Interf1), 함수이름(func1)을 통해 'Interf1 _func1'이라는 함수 호출을 나타내는 Trigger가 생성되었다. 그리고 comp_stmt들에 대한 처리(아래 참조)과정을 거치며 가장 마지막으로 생성된 상태전이(from 5)를 초기상태로 돌아오게 하였다.
2	Case assign: 할당문 내용에 해당하는 행위(a=0;)를 가장 최근에 생성된 상태전이(from 1)에 추가한다.
3	Case call: 중첩된 call으로 내부에 있는 함수 호출부터 처리한다.
3-1	인터페이스이름, 함수이름, 매개변수 정보로부터 'Interf1 ! func3();' 이라는 함수 호출행위를 구성하고 가장 최근에 생성된 상태전이(from 1)에 추가한다. 함수 호출의 반환 값이 다음 함수 호출의 매개변수로 사용되므로, 반환 값을 기다리는 상태(*로 표시)가 생성된다. 또한 이 상태로부터 나가는 Transition도 생성된다.
3-2	3-1 의 함수 호출로부터 반환 값을 수신하는 Trigger를 가장 최근에 생성된 상태전이(from 3-1)에 추가한다.
3-3	안쪽 함수 호출과 마찬가지로 인터페이스이름, 함수이름, 매개변수 정보로부터 'Interf1 ! func2(rv, b);'라는 함수호출 행위를 구성하여 가장 최근에 생성된 상태전이(from 3-1)에 추가한다. 여기서 rv는 func3 함수 호출에 대한 반환 값을 나타낸다.
4	Case assign: 할당문 내용에 해당하는 행위(a=b;)를 가장 마지막에 생성된 상태전이 (from 3-1)에 추가한다.
5	Case if: 먼저 분기점이 되는 상태를 생성한다. 다음으로 조건문이 참인 경우에 대해 상태를 생성하고 조건문 내용으로부터 'a>0'라는 제약조건을 생성하여 Transition에 추가한다. 그리고 조건이 참인 경우의 실행구문을 처리한다(6 번째 라인). 다음으로 조건이 거짓인 경우에 대해서도 상태전이를 생성하고 상태전이에 'else'라는 제약조건을 추가한다. 그리고 조건이 거짓인 경우의 실행구문을 처리한다(9 번째 라인). 두 경우에 대한 처리가 모두 끝나면 두 경우의 마지막으로 생성된 상태전이들(from 6, 9)을 합치는 상태를 생성하고, 그 상태로부터 나가는 새로운 상태전이를 생성한다.
6	Case call: 인터페이스이름, 함수이름, 매개변수 정보로부터 'Interf2 ! func4(2);'라는 함수 호출행위를 구성하여 가장 최근에 생성된 상태전이(from 5)에 추가한다.
9	Case call: 인터페이스이름, 함수이름, 매개변수 정보로부터 'Interf2 ! func5(5);'라는 함수 호출 행위를 구성하여 가장 최근에 생성된 상태전이(from 5)에 추가한다.
11	Case return: 반환문 내용에 해당하는 'Interf1 ! RV, 0;'라는 행위를 가장 마지막에 생성된 상태전이(from 5)에 추가한다. 여기서 'RV'는 반환 값을 나타낸다.

다음으로 Fig. 15는 if의 행위에 따른 상태의 분화를 나타낸 것이다. if는 분기조건을 제약조건(guard condition)을 통해 표현하며, 따라서 참/거짓을 표현하기 위해 최소 2개의 상태전이가 필요하다. 또한if 이전까지의 행위가 진행된 이후 분기조건을 검사하기 때문에 if 이전 행위가 수행되는 시점과 if이후의 행위가 수행되는 시점을 분리해야 한다. 따라서 시점을 분리하기 위해 분기 상태를 만든 뒤, 분기 상태에서 출발하고 참/거짓을 표현하는 상태전이를 각각 만들어 if의 행위에 대응한다. 참/거짓을 표현하는 상태전이에서 수행되어야 하는 행위가 모두 작성된 이후에는 분리된 두 상

태전이를 다시 하나로 모으는 상태를 생성하며, 이 상태에서 출발하는 상태전이를 생성해 if이후의 행위를 다시 표현할 수 있도록 한다.

다음으로 module_impl과 상태기계간 대응 규칙의 구체적인 이해를 돕기 위해 아래의 예제를 통해 규칙이 실제로 어떻게 적용되는지 구체적으로 설명한다. Fig. 16은 nesC의 func_def 구조에 대한 예제 코드이며, 이 코드에 Fig. 11의 규칙을 적용해서 생성된 모델이 Fig. 17이다.

Fig. 11의 내용을 이용해서 Fig. 16의 각 라인 별 모델 생성과정을 설명하면 Table. 2와 같다.

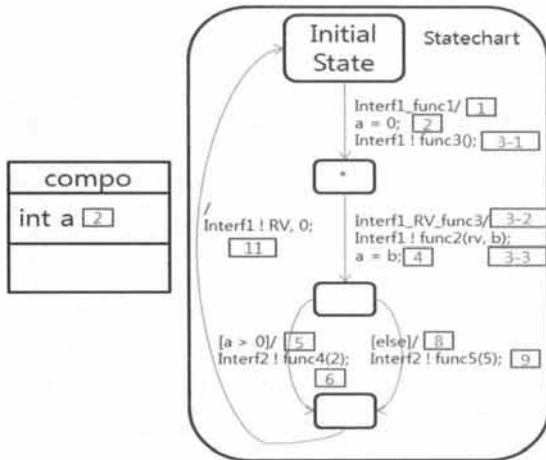


Fig. 17. A model generated from example code

5. 모델 추출기 설계

본 연구에서 제안하는 컴포넌트 모델 추출기는 하나의 nesC 컴포넌트(단위컴포넌트 혹은 조합컴포넌트)를 입력 받아서 그에 해당하는 UML 클래스와 상태차트를 내부적으로 생성하며 최종적으로 XMI 파일 형태로 출력하게 된다(Fig. 18). nesC 파서는 입력된 nesC 파일이 문법에 맞게 작성되었는가를 판단하며, 또한 해당 파일을 컴포넌트 모델로 변환하기 위해 AST를 생성한다. AST Walker는 앞서 생성된 nesC의 AST를 순회하면서 노드 타입에 따라 그에 해당하는 UML 객체를 생성해주며, 마지막으로 XMI 생성기는 생성된 UML 모델을 XMI 파일 형태로 출력한다.

5.1 nesC AST의 설계

AST의 설계는 nesC 구문 구조에서 불필요한 토큰을 제거하는 방식을 사용하였다. 예를 들어 Fig. 19에 나와 있는 module_implementation 구문 구조의 경우 'implementation'이라는 문자열과 중괄호 내부의 translation _unit 심벌로 이루어져 있는데, 여기서 불필요한 'implementation' 문자열과 중괄호를 제거하는 방식으로 AST를 구성하였다('->' 기호 우측). 또한 AST 순회 시 module_implementation 구문임을 인지할 수 있도록 MODULE_IMPL이라는 별도의 토큰을 만들어 루트 노드가 되도록 하였다.

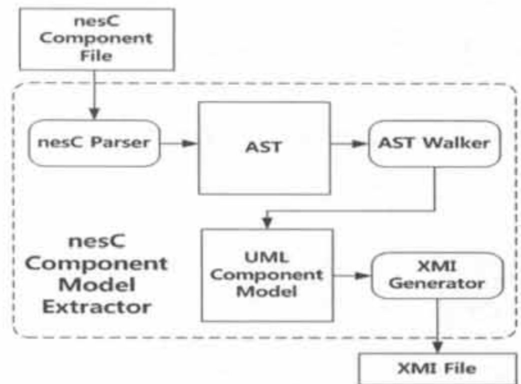


Fig. 18. Structure of component model extractor

```
module_implementation:
        'implementation' '{' translation_unit '}'
  -> (MODULE_IMPL translation_unit);
```

Fig. 19. Example of an AST

5.2 도구의 구현

nesC 파서는 ANTLR [11] 파서 생성기를 이용하여 구현하였다. 또한 nesC의 문법은 ANSI C의 문법을 확장한 것으로[5], 본 도구에서는 ANTLR 용 ANSI C 문법과 참고문헌 [4]에 나와 있는 nesC 확장 문법을 합성하여 전체 문법을 완성하였다. AST Walker는 nesC 파서에 의해 생성된 AST를 순회하며 노드 타입에 따라 해당하는 UML 메타모델 객체를 생성하기 위해 Eclipse UML2 플러그 인을 사용, 제공되는 기본적인 API들을 활용하여 구현하였다. XMI 생성기는 Eclipse UML2에서 제공하는 XMI 파일 저장 기능을 이용하였다.

6. 적용결과 및 평가

본 도구를 nesC 단위컴포넌트의 하나인 Msp430AlarmC Fig. 20에 적용하여 컴포넌트 모델을 추출하였다. 추출된 컴포넌트 모델은 XMI 파일 형태로 주어지는데, 이것을 IBM Rhapsody를 이용하여 시각화한 것이 Fig. 21과 Fig. 22이다.

Fig. 21은 해당 단위컴포넌트가 추상컴포넌트 구조모델로 추출된 결과이다. 단위컴포넌트가 사용하는 인터페이스에 대해서 반원 모양의 사용 포트(used port) 3개가 만들어진 것을 볼 수 있으며, 단위컴포넌트가 제공하는 인터페이스에 대해서 둥근 모양의 제공 포트(provided port) 2개가 만들어진 것을 볼 수 있다. Fig. 22는 같은 단위컴포넌트의 구현부분을 추상컴포넌트의 내부행위모델로 표현한 상태도로서 단위컴포넌트가 제공하는 각 커맨드에 대해 커맨드 호출 이벤트가 발생하였을 때 수행되는 행위를 확인할 수 있다. Fig. 22는 같은 단위컴포넌트의 구현부분을 추상컴포넌트의 내부행위모델로 표현한 상태도로서 단위컴포넌트가 제공하는 각 커맨드에 대해 커맨드 호출 이벤트가 발생하였을 때 수행되는 행위를 확인할 수 있다.

```
generic module Msp430AlarmC(typedef frequency_tag)
@safe() {
   provides interface Init;
   provides interface Alarm<int,int> as Alarm;
   uses interface Msp430Timer;
   uses interface Msp430TimerControl;
   uses interface Msp430Compare;
}
implementation {
   command int Init.init()  {
      call Msp430TimerControl.disableEvents();
      call Msp430TimerControl.setControlAsCompare();
      return SUCCESS;
   }
   async command void Alarm.startAt( int t0, int dt ) {
   atomic {
      int now = call Msp430Timer.get();
      int elapsed = now - t0;
      if( elapsed >= dt ) {
         call Msp430Compare.setEventFromNow(2);
      }else {
         int remaining = dt - elapsed;
         if( remaining <= 2 )
            call Msp430Compare.setEventFromNow(2);
         else
            call Msp430Compare.setEvent(now+remaining);
      }
      call Msp430TimerControl.clearPendingInterrupt();
      call Msp430TimerControl.enableEvents();
   }
   (…생략…)
}
```

Fig. 20. Implementation of module component - Msp430AlarmC

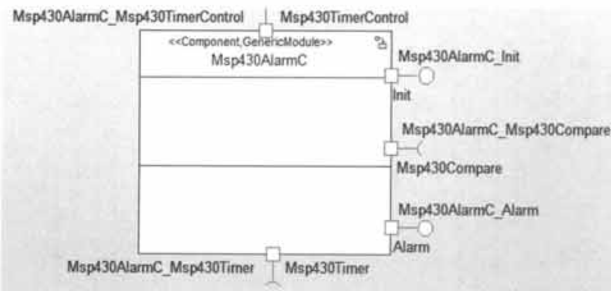


Fig. 21. Automatically extracted structure model - Msp430AlarmC

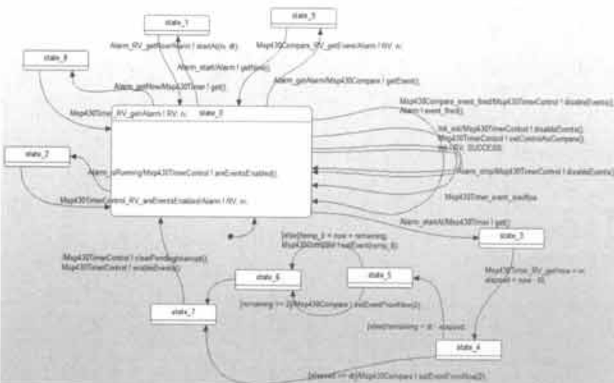
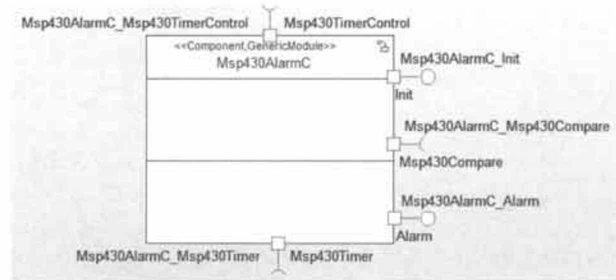
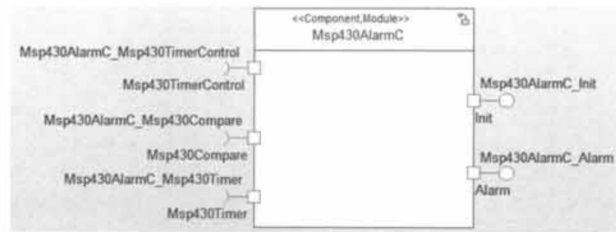


Fig. 22. Automatically extracted behavior model - Msp430AlarmC



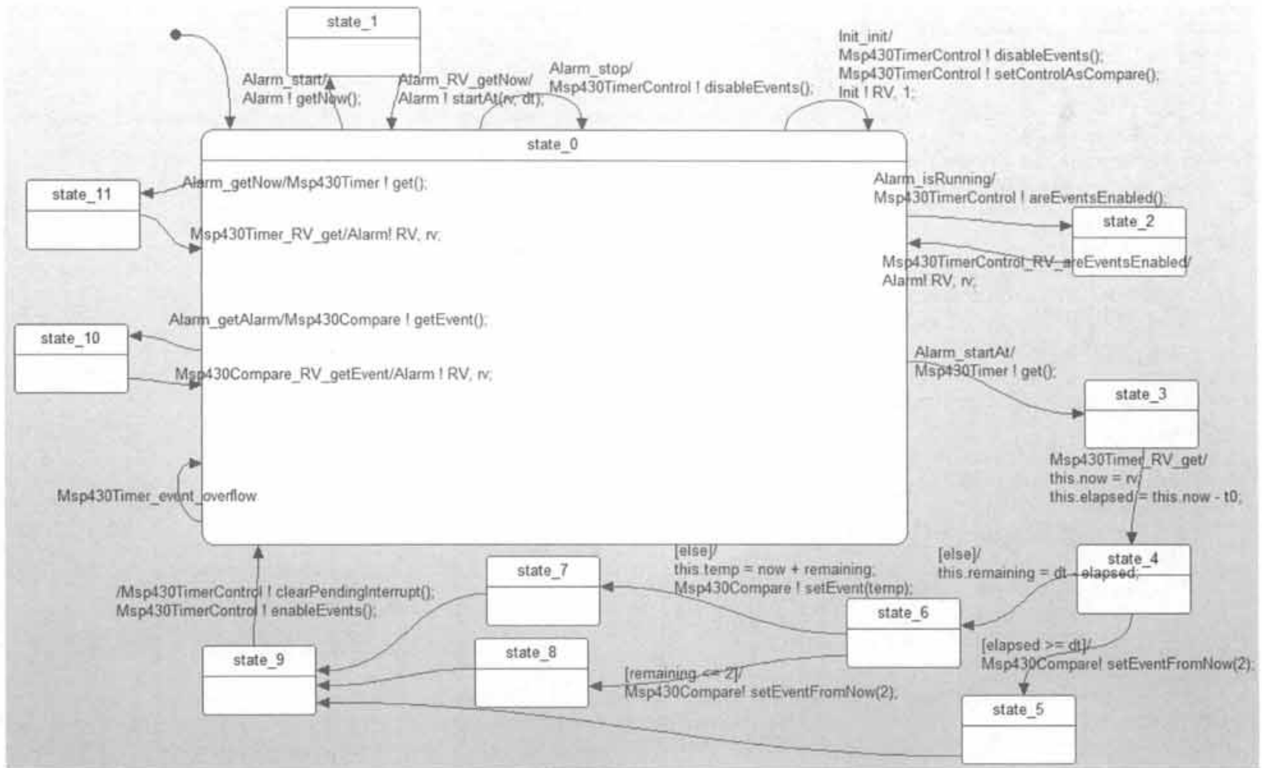
(a) Automatically extracted model



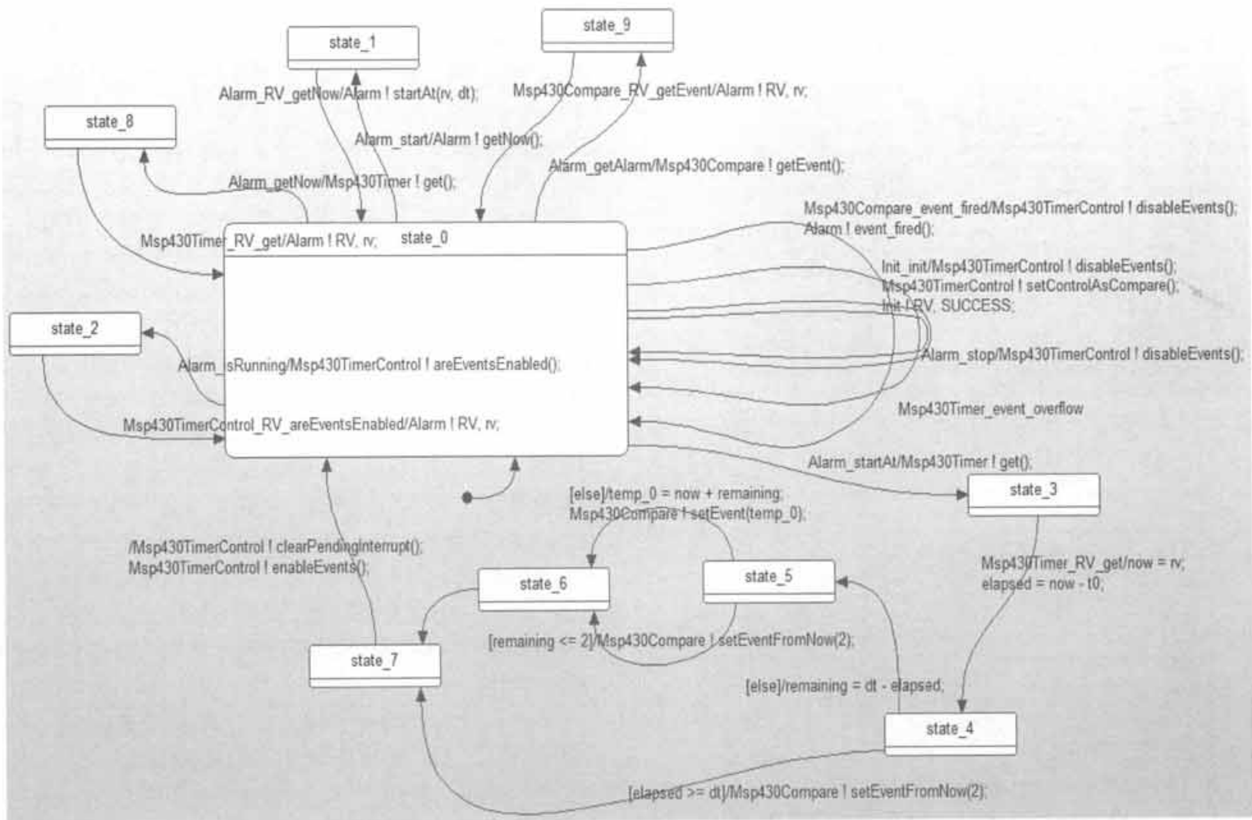
(b) Manually generated model

Fig. 23. Comparison of structure model - Msp430AlarmC

다음으로, 동일한 TinyOS 프로그램에 대해 모델링 도구를 통해 수작업으로 생성한 컴포넌트 모델과, 컴포넌트 모델 추출기를 통해 자동으로 추출한 모델을 비교함으로써 컴포넌트 모델 추출기의 타당성을 평가한다. 평가를 위해 사용한 TinyOS 프로그램은 Msp430AlarmC이며, 이 프로그램에 대한 수작업 모델은 참고문헌[8]의 연구 결과에서 차용하였다. 먼저 Fig. 23의 구조 모델에서는 두 컴포넌트 모두 Msp430AlarmC라는 동일한 컴포넌트 이름을 가지며, Component, Module 이라는 동일한 스테레오타입을 가짐을 알 수 있다. 또한 같은 이름과 개수의 사용/제공 포트를 가짐을 알 수 있다. 따라서 두 구조 모델은 동일한 의미를 가진다. 다음으로 Fig. 24의 행위 모델을 비교해보면, 두 모델 모두 하나의 초기 상태를 가지며, Msp430AlarmC 컴포넌트의 각 오퍼레이션에 대한 전이를 가지고 있음을 알 수 있다. 그림을 살펴보면 Alarm_startAt 오퍼레이션과 Msp430 Compare_event_fired 오퍼레이션에 대한 전이에서 두 모델 간에 차이가 발생하며, 나머지 오퍼레이션에 대해서는 동일함을 알 수 있다. Alarm_startAt 오퍼레이션의 경우, 수작업 모델에서는 트리거, 제약조건, 행위가 없는 비어 있는 전이(ε move)가 총 3개(state_9로 들어오는 전이들) 발생한 반면, 자동추출의 경우 비어 있는 전이가 총 1개(state_6과 state_7 사이)만 발생한 것을 알 수 있다. 이 차이는 두 모델이 if문에 대해서 상태를 생성하는 방식이 달랐기 때문에 생긴 차이다. 즉, 수작업 모델의 경우 if문 조건이 참인 경우와 거짓인 경우에 대해서 무조건 상태를 만들었지만, 자동추출 모델의 경우 중간 상태가 반드시 필요하지 않은 이상 새로운 상태를 생성하지 않았기 때문에, 비어있는 전이들을 축약하게 되면 결국 두 모델의 상태차트가 같아지므로 결국 동일한 의미의 상태차트임을 알 수 있다. 따라서 두 행위 모델은 동일한 의미를 가진다고 할 수 있다. 다음으로 Msp4



(a) Manually generated model



(b) Automatically extracted model

Fig. 24. Comparison of two behavior models - Msp430AlarmC

30Compare_event_fired 오퍼레이션의 경우, 수작업 모델에서는 이에 대한 전이가 존재하지 않고 자동 추출된 모델에만 존재하고 있다. 이것은 수작업으로 모델링 하는 과정에서 Msp430Compare_event_fired 오퍼레이션에 대한 전이를 빠뜨린 것으로 추측된다.

따라서 수작업 모델의 생성과정에서 빠뜨린 Msp430Compare_event_fired 오퍼레이션에 대한 전이를 제외하면, 수작업으로 생성한 Msp430AlarmC 모델과 자동으로 추출한 Msp430AlarmC 모델이 구조와 행위의 관점에서 각각 동일한 의미를 가짐을 알 수 있다.

7. 결론 및 향후 과제

본 연구에서는 모델 중심·컴포넌트 기반 개발 방식인 MARMOT의 지원 도구로서 nesC 컴포넌트 모델 추출기를 개발하였다. 도구를 개발하기 위해 nesC 프로그래밍 언어의 구문에 대한 AST를 설계하였고, AST 노드 타입과 UML 메타모델 요소 사이의 대응관계를 정의하였다. 또한 지원 도구를 실제 nesC 코드에 적용해본 결과 정상적으로 컴포넌트 모델 생성이 되었으며, IBM Rhapsody 도구를 통해 시각적인 확인을 할 수 있었다. 또한 수작업으로 생성한 모델과 자동으로 추출된 모델의 비교를 통해 자동 추출된 모델이 올바른 의미를 가짐을 보였다. 개발된 자동 추출기는 코드로부터 기반 컴포넌트 모델을 추출하는 과정을 자동화함으로써 현재 개발 마무리단계에 있는 상향식 추상컴포넌트 합성도구[15] 와 함께 MARMOT프레임워크의 지원을 위한 기초도구로 활용될 것이다. 향후 과제는 여러 TinyOS 코드에 대한 추가 실험을 통해 자동추출 가능한 범위를 늘리는 것이며, 세부 과제로는 외부파일에 정의되어 있는 매크로 표현 및 사용자 정의 타입을 자동으로 검색하는 기능 추가, Generic 컴포넌트/인터페이스 등을 지원하고자 한다.

참 고 문 헌

[1] B. Selic, "The pragmatics of model-driven development," in IEEE SOFTWARE, Vol.20, pp.19-25, 2003.

[2] Y. Choi and H. Jang, "Reverse Engineering Abstract Components for Model-Based Development and Verification of Embedded Software", in Proc. IEEE 12th Int High-Assurance Systems Engineering (HASE) Symp, pp. 122-131, 2010.

[3] Y. Choi and C. Bunse, "Design verification in model-based μ-controller development using an abstract component", Software and System Modeling, pp.91-115, 2008.

[4] D. Gay, "nesC 1.3 Language Reference Manual", 2009.

[5] D. Gay, "Software design patterns for TinyOS," Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, Vol.40, pp.40-49, 2005.

[6] A. MacDonald, D. Russell and B. Atchison, "Model-driven Development within a Legacy System: An industry experience report," in Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05), pp.14-22, 2005.

[7] M. Snoeck and G. Dedene, "Experiences with Object Oriented Model-Driven Development," in Software Technology and Engineering Practice, 1997. Proceedings, Eighth IEEE International Workshop on [incorporating Computer Aided Software Engineering], pp.143-153, 1997.

[8] H.Jang, M.Park, Y.Choi, "A Model Translator for Checking Behavioral Consistency of Abstract Components", The KIPS transactions. Part D. Vol.18D, No.6, pp.443-450, 2011.

[9] E. Korshunova, "CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code," in Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06), pp.297-298, 2006.

[10] C. Abbaneo, F. Flammini, A. Lazzaro and P. Marmo, "UML Based Reverse Engineering for the Verification of Railway Control Logics," in Proceedings of the International Conference on Dependability of Computer Systems (DEPCOS-RELCOMEX'06), pp.3-10, 2006.

[11] T. Parr, The Definitive ANTLR Reference, Pragmatic Bookshelf, 2007.

[12] N. Kothari, T. Millstein and R. Govindan, "Deriving State Machines from TinyOS Programs using Symbolic Execution," in 2008 International Conference on Information Processing in Sensor Networks, pp.271-282, 2008.

[13] Chouambe, L., Klatt, B. and Krogmann, K., "Reverse Engineering Software-Models of Component-Based Systems," in 12th European Conference on Software Maintenance and Reengineering. CSMR, pp.93-102, 2008.

[14] E. Cho, C. Kim, "A Design Technique of Meta-Model for Reengineering from Legacy to CBD", Journal of Korea Multimedia Society, Vol.8, No.3, pp.398-412, 2005.

[15] M.Park and Y.Choi, "A tool for scalable verification of multi-component distributed systems", International conference on Computer, Networks, Systems, and industrial Applications, pp.541-545, 2012.

윤 상 권

e-mail : skyun@hancom.com

2010년 경북대학교 수학과(학사)

2012년 경북대학교 전자전기컴퓨터학부 (석사)

2012년~현 재 한글과컴퓨터

관심분야 : 컴포넌트 기반 개발 방법, 컴파일러

박 민 규

e-mail : pqrk8805@hanmail.net

2011년 경북대학교 IT 대학 컴퓨터학부 (학사)

2013년 경북대학교 전자전기컴퓨터학부 (석사)

2013년~현 재 경북대학교 IT 대학 컴퓨터학부 박사과정

관심분야 : 컴포넌트 기반 개발방법, 컴포넌트 모델 최적화

최 윤 자

e-mail : yuchoi76@knu.ac.kr

2003년 (미)미네소타대학 전산과(박사)

2003년~2006년 (독)프라운호퍼연구소 연구원

현 재 경북대학교 IT 대학 컴퓨터학부 부교수

관심분야 : 소프트웨어 안전성 분석, 정형검증, 모델기반 개발방법론