A Model-driven development and verification framework for embedded software

Software Safety Engineering Laboratory
School of Electrical Engineering and Computer Science
Kyungpook National University

Yunja Choi
Motivation

- Software engineering perspective
  - Increasing needs for a structured (or systematic) development methodology for embedded software
  - Increasing need for efficient and effective verification technique

- Verification perspective
  - Programming analysis is limited to certain code-specific properties
    - e.g., array-bound checking, dangling pointer, assertion checking, etc.
  - Design and/or requirements errors are hard to identify and costly to correct
    - e.g. process deadlock, incorrect behavior due to loss of input messages
Our approach

- Integration of verification techniques into existing development process
  - We chose one of the component-based, model-driven development (MDD) methodologies named MARMOT

- Provide a framework for the V&V-integrated development methodology including
  - Modeling language
  - Design simulation
  - Design verification
  - Code generation

- Provide automation to support the V&V-integrated development framework
  - UML subset + action language for the modeling language
  - Extension of existing UML support tools
  - Integration of model checking techniques
MARMOT Methodology

- Stands for Method for Component-Based Real-Time Object-Oriented Development and Testing
- Branched from KobrA by Atkinson et. al
  - Designed for the development of embedded systems
- High quality system through systematic, structured development
- Based on
  - the principle of “separation of concerns”: specification vs. realization
  - Iterative decomposition and refinements
- Components are the focus of entire development process
  - Tree-structured hierarchy of components
  - Flexibility and reuse of components
MARMOT Component

[Diagram of a UML class diagram showing relationships between components, attributes, operations, relations, structure, behavior, and interfaces.]
Component Refinements

- Refined component
- Refining component

Specifying

Refined component

Statecharts

Realization

Object Diagram (Architecture)

Class Diagram

Operation Schemata

Sequence Diagram

Class Diagram

Specification
Recursive Development

- Identification
- Specification
- Realization

Component Reuse

COTS Component

Kpt A
Kpt B
Kpt C
Kpt D
Things to be checked

- Statecharts
- Class Diagram
- Operation Schemata
- Sequence Diagram
- Object Diagram (Architecture)

- Interaction consistency
- Specification–realization consistency

Correctness
A few huddles to get over

- Models do not exist!
  - Where do we start?
- No universally accepted modeling notations
  - UML?
- Model checking does not scale well
  - Is it usable?
Creating models – reverse engineering

- Start from existing codes and reverse engineer them into abstract component models
  - We start from open source wireless sensor network
- Once reverse engineered, the same model can be reused for future developments
Creating models – reverse engineering

Requirements → analysis → Low-level design → implementation → testing → deployment → Abstract component model → Component model with annotation → Code → Reverse engineering

- Requirements
- Analysis
- Low-level design
- Implementation
- Testing
- Deployment
- PIM
- PSM
- Code
- Abstract component model
- Component model with annotation
- Code
- Reverse engineering
Applying MDD and verification

Requirements → analysis

→ Low-level design

→ implementation

→ testing

→ deployment

PSM → code

Component model with annotation

Reverse engineering
Applying MDD and verification

- Requirements analysis
- Low-level design
- Implementation
- Testing
- Deployment

Verification of design change
- Component model with annotation
- Code

Reverse engineering
Ultimate Goal – round trip development

- Requirements
- Analysis
- Low-level design
- Implementation
- Testing
- Deployment

- Abstract component model
- Component model with annotation
- Code
Modeling language & Tools (1)

- UML as a modeling language
  
  **Pros:**
  - Independent from program languages to be used
  - widely accepted in industry
  - A number of CASE tools are available and widely used in industry
    - With simulation, code generation, and reverse engineering capability
  
  **Cons:**
  - Unclear semantics: dynamic semantics is left to the CASE tools
  - Ambiguity: allow informal expressions
  - Existing CASE tools does not support the notion of abstract component
Modeling language & Tools (2)

- Solution
  - Define UML extensions and formal semantics
    - action language
    - Syntax for describing abstract component – stereotype and annotation
  - Utilize existing CASE tools as much as possible
    - We use Telelogic Rhapsody
    - But, our V&V approach and other extensions are to be tool-independent
Verification methods (1)

- Model simulation for behavior checking
  - Use the simulation tool of existing CASE tools as much as possible
  - Provide extension to the existing simulation tool to support
    - different dynamic semantics
    - Simulation of abstraction components
Verification methods (2)

- Use model checking as a back-end verifier
  - Based on the exhaustive search of system state-space
  - Can check process deadlock and other concurrency-related properties
  - Fully automated
  - Provide counter-examples

- Need a translation to the input language of model checker
  - SPIN, SMV, CADP

- Need to support efficient feed-back
  - Replay of the counter-examples through simulation
Verification methods (2) – framework

- UML model (tool & version dependent)
- XML export
- Model in XML format
- JDOM
- Generate abstract syntax tree
- UML abstract syntax tree (tool & version independent)
- UML 2.0. metamodel library
- SMV metamodel library
- UML-to-SMV translation
- UML-to-PROMELA translation
- PROMELA metamodel library
- consistency model
- SMV consistency model
- PROMELA consistency model

original model
xml DTD/Schema
mapping
Verification methods (3)

- Successive verification through abstraction
  - Verification of the entire system at once is not feasible
    - 311+ nesC files for basic features of TinyOS
  - Mixture of top-down and bottom-up approaches
    - Environmental constraints: top-down extraction
    - Behavior abstraction: bottom-up abstraction
Model verification : Consistency

External stimuli

System processes
Model verification: Consistency

A compositonal process forming an environment

Process P_i from refining component i

Process P_1 from refining component 1

Process P_2 from refining component 2

actions generated by the environment

cchange of states

+refined

+refining

attribute

structure

component

behavior

component model at (i+1)_th level

component model at (i-1)_th level

component model at i_th level

component model at (i+1)_th level

component model at (i-1)_th level

component model at i_th level

external

internal

external

internal

external

internal

external

internal

external

internal
module PlatformP{
  provides interface Init;
  uses interface Init as Msp430ClockInit;
  uses interface Init as MoteInit;
  uses interface Init as LedsInit;
}
implementation {
  command error_t Init.init() {
    call Msp430ClockInit.init();
    call MoteInit.init();
    call LedsInit.init();
    return SUCCESS;
  }
  default command error_t LedsInit.init() { return SUCCESS; }
}

• extraction of statecharts from code
• statechart composition and reduction
Verification methods (4) -- scalability

- Successive verification approach may limit the number of components to be verified at the same time – good!
- Still, scalability issue is potentially the most serious problem
- We will investigate on techniques such as
  - Property-based abstraction
  - Compositional verification
Code generation

- Use intermediate language

Diagram:
- Requirements
  - Analysis
  - Low-level design
    - Implementation
    - Testing
    - Deployment
  - PIM
  - PSM
  - Intermediate code
  - Code
Related work

- **OMEGA project**
- **SYNTHESES**
  - ICSE 2007
- **Adaptor**
  - TSE 2008
  - http://www.ibisc.univ-evry.fr/~poizat
Research Plan (for next 4 years)

- **기초 프레임워크 개발 단계**
  - **1차년**
    - 제어소프트웨어의 컴포넌트 기반 정형적 참조모델 개발
    - 부분적 정형기법의 적용과 동적 코드 검증의 결합을 위한 기초 프레임워크 연구
  - **2차년**
    - 비정형적 소프트웨어 모델의 정형화 기법연구
    - 사례연구를 통한 정형화 기법의 적용성 평가
    - 동적분석의 일반화와 코드-모델간 피드백 시스템 개발
  - **3차년**
    - 컴포넌트 기반 부분 추출기법을 통한 조합적 검증 프레임워크 개발
    - 동적분석을 이용한 정적분석의 허위경보 색출과 피드백 시스템 개발
  - **4차년**
    - 실제 제어 소프트웨어 개발과정에의 적용과 사용성 평가
    - 피드백 시스템의 평가와 보완
Work in progress

- Reverse engineering tinyOS
  - Define modeling notations for abstract components
  - Model extraction from code
- Model simulation using Rhapsody
  - Identify the limitation of Rhapsody simulator
  - Design extensions of Rhapsody simulator
- Participants
  - 1 professor, 2 graduate students, 1 undergraduate student