Transforming Coloured Petri Net Models into Code for TinyOS
- A Case Study of the RPL Protocol

Lars M. Kristensen and Vegard Veiset
Department of Computing
Bergen University College, NORWAY
Email: lmrk@hib.no / vegard.veiset@stud.hib.no
Motivation

- Coloured Petri Nets (CPNs) have been widely used for modelling of concurrent systems:
  - specification, validation, and verification
  - what about executable software?

```java
configuration RPLProtocolAppC {} 
implementation {
  components MainC, RPLProtocolC, DAOC, DIOC;
  RPLProtocolC.Boot -> MainC.Boot;
  RPLProtocolC.DAO -> DAOC.RPLPacket;
  RPLProtocolC.DIO -> DIOC.RPLPacket;
  DAOC.NODE -> RPLProtocolC.NODE; }
```

100010011100011100011
Overview of Approach

- **CPN models are platform independent** and at a high level of abstraction:
  - Each manual refinement step consists of:
    - Increasing the level of details to the CPN model.
    - Adding **pragmatic annotations** to the CPN model.
  - The result is a **platform-specific CPN model** for automated code generation.
Pragmatic <<annotations>>

- **Syntactical annotations [12] on model elements:**
  - Adds platform dependent and domain-specific elements.
  - Can be bound to code generation templates.

```
configuration RPLProtocolAppC { }
implementation {
  components MainC, RPLProtocolC, DAOC, DIOC;
  RPLProtocolC.Boot -> MainC.Boot;
  RPLProtocolC.DAO -> DAOC.RPLPacket;
  RPLProtocolC.DIO -> DIOC.RPLPacket;
  DAOC.NODE -> RPLProtocolC.NODE;
}
```

Case Study of the RPL Protocol
IEFT RPL Protocol

- IoT routing protocol for **distributed sensor networks** currently being developed by the IETF:

  - Supports a sensor nodes in establishing a **DODAG** for data collection purposes.
RPL CPN Model

- A platform independent model specifying the operation of the RPL Protocol:

```
RPL To Link

NodexPacket

Link To RPL

NodexPacket

Link Layer

Link Layer

Protocol

RPL Network

Protocol

DISDIO

DAO

DAOACK

Topology Change

Startup and Timeout
```

odicFucntion

(n2, rank, ver, parent, state)

(n, (DEST n2, DIO(rank,ver)))

Send DIS Req

(n, (DEST n2, DIS))

discoveryReq(n)

(n, rank, ver, parent, JOINING)

Receive DIO Response

Send DIS Req

(state = JOINED
eroelse state = ROOTJOIN)
Platform: TinyOS and nesC

- Operating system and programming language targeting resource constrained devices.

```plaintext
configuration RPLProtocolAppC { }

implementation {

components MainC, RPLProtocolC,
     DAOC, DIOC;

RPLProtocolC.Boot -> MainC.Boot;
RPLProtocolC.DAO -> DAOC.RPLPacket;
RPLProtocolC.DIO -> DIOC.RPLPacket;
DAOC.NODE -> RPLProtocolC.NODE;
}
```

- Applications are structured into components providing and using interfaces.
- Split-phase programming model based on commands, and calls, events and signals.
- Component are wired into a configuration constituting an application.
Proposed Refinement Methodology
Refinement Methodology

- A five step methodology for refining models to an abstraction level suited for code generation:
  1. **Component architecture** identifying components and interfaces, and determining an application configuration.
  2. **Interface naming, provision, and use** allowing reference to the same interface provided by multiple components.
  3. **Component and interface signatures** identifying commands and events and associated types.
  4. **Component classification** into boot-, dispatch-, external-, timed-, and regular components.
  5. **Internal component behaviour** providing control-flow oriented modelling of command and event implementations.
Step 1: Component Architecture

- Identify <<components>> and <<interfaces>> via substitutions transitions and socket places:
**Step 2: Interface Naming and Use**

- Resolve naming conflicts and specify use and provision of interfaces:
Step 3: Interface Signatures

- Refine component «interfaces» to specify «commands» and «events»:

```
Node State

In/Out
NetNode In/Out
Packet Queue

NodexPacket

RPL To Link Out
NodexPacket

ReceiveDIS
[state = JOINED orelse state = ROOTJOINED]

ReceiveDIO

objectiveFunction
(n, n2, rank, ver, rrank, rver, parent, state)

(n2, rank, ver, parent, state)

(n, (DEST n2, DIS))

(n, (DEST n2, DIO(rrank,rver)))
```

```
Node

State

In/Out
NetNode

DISDIO «interface»

In

NodexPacket

RPL To Link Out
NodexPacket

receiveDIO «event»

ReceiveDIO

receiveDIS «event»

ReceiveDIS

In

NodexPacket

RPL To Link Out
NodexPacket
```
Step 4: Component Classification

- Classifies components as boot-, timed-, dispatch-, external-, and regular components:
Step 5: Internal Behaviour

- Makes explicit control flow and data access in the command and event implementations:
Automated Code Generation
Code Generation

- A template-based code generator implemented based on the Access/CPN Framework [15]:

1. Mapping CPN ML datatypes into corresponding nesC datatypes.
2. Interfaces based on places annotated with <<interface>>.
3. Components based on substitution transition with <<component>>.
4. Configuration and wiring based on <<component>> substitution transitions and <<interface>> arcs.
5. Command and event behaviour based on <<var>> and <<id>> places and structural pattern matching.

- Top-down traversal of the CPN model invoking templates according to encountered pragmatics.

Code Validation

- Deployment in a virtualised sensor networks using the TOSSIM emulator:

- Instrumentation and inspection of event-logs:

```
DEBUG (0): 0:0:0.0000000300 RPL | Application booted.
DEBUG (0): 0:0:0.0000000300 RPL | State change: 0 -> 2.
```
Conclusions and Future Work

- **A semi-automatic approach to code generation for the TinyOS Platform:**
  - A five step methodology refining the model to a level of detail suitable for generating nesC code for the target platform.
  - Pragmatics used to relate CPN model construct and elements to the target platform via code generation templates.

- **The approach has been initially validated on the IETF RPL routing protocol for sensor networks.**

- **Future work:**
  - Formalisation of meta-models and transformation steps for the refinement methodology.
  - Model checking techniques for verification of refined models.
  - Model-based testing for validating the generated code.