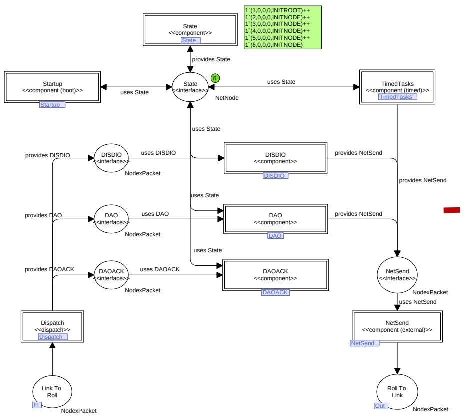
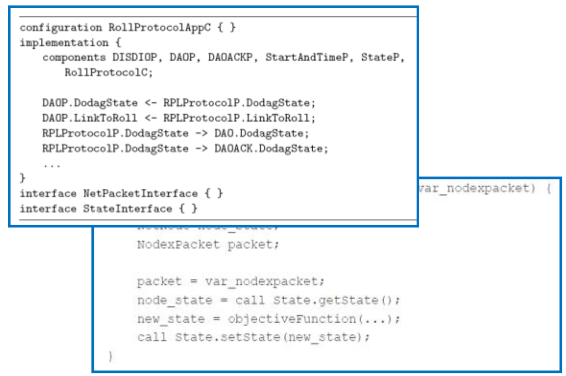
Transforming Coloured Petri Net Models into Code for TinyOS
- A Case Study of the RPL Protocol



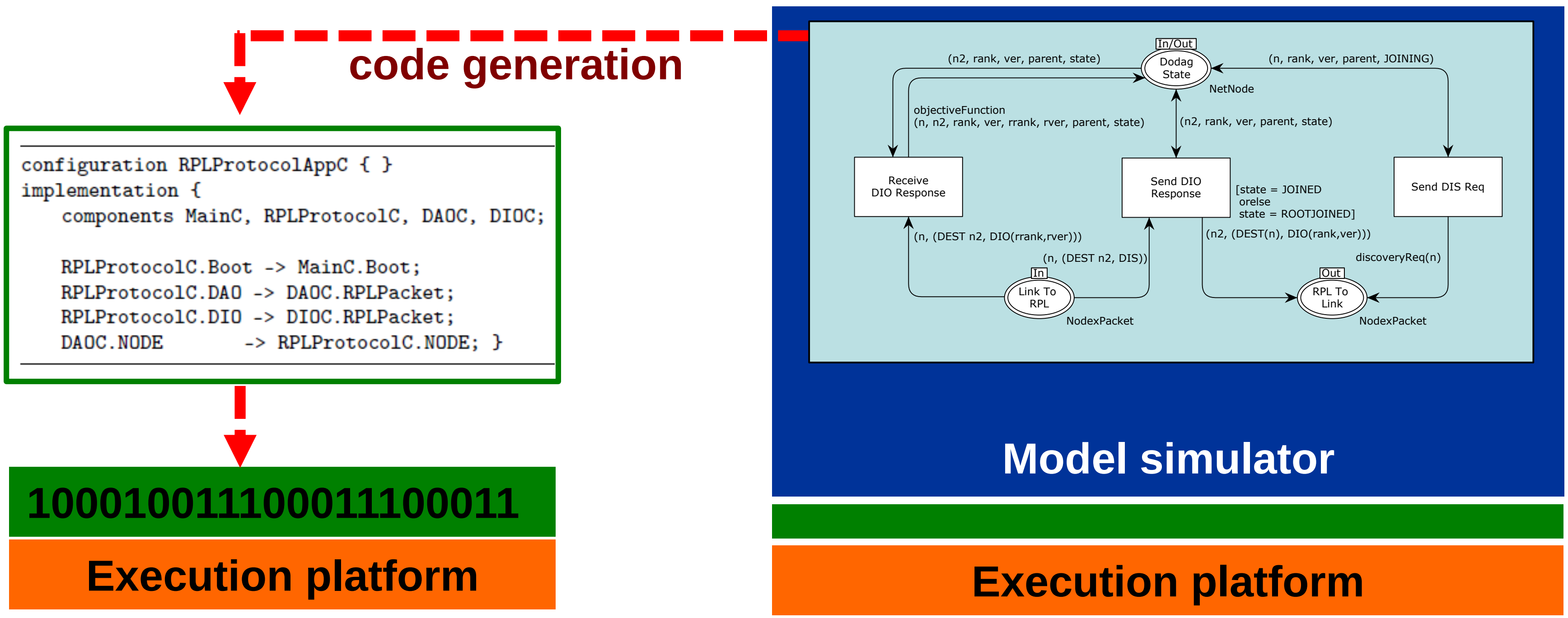


Lars M. Kristensen and Vegard Veiset
Department of Computing
Bergen University College, NORWAY
Email: lmkr@hib.no / vegard.veiset@stud.hib.no

HØGSKOLEN I BERGEN
BERGEN UNIVERSITY COLLEGE

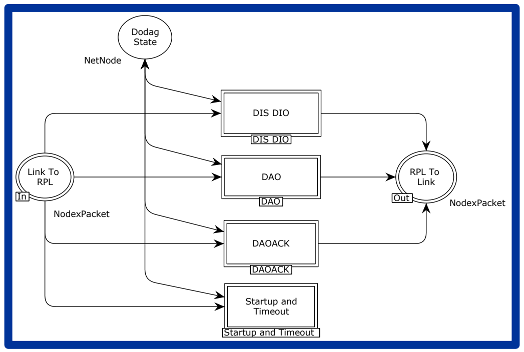
Motivation

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

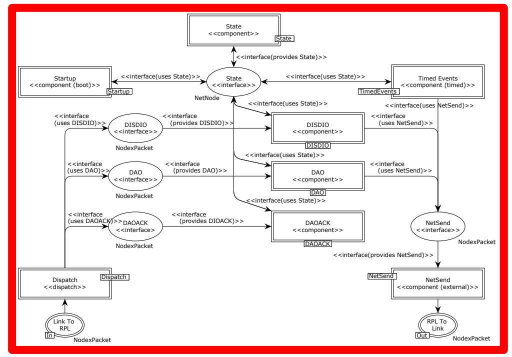


Overview of Approach

- **CPN models are platform independent and at a high level of abstraction:**



manual refinements



code generation

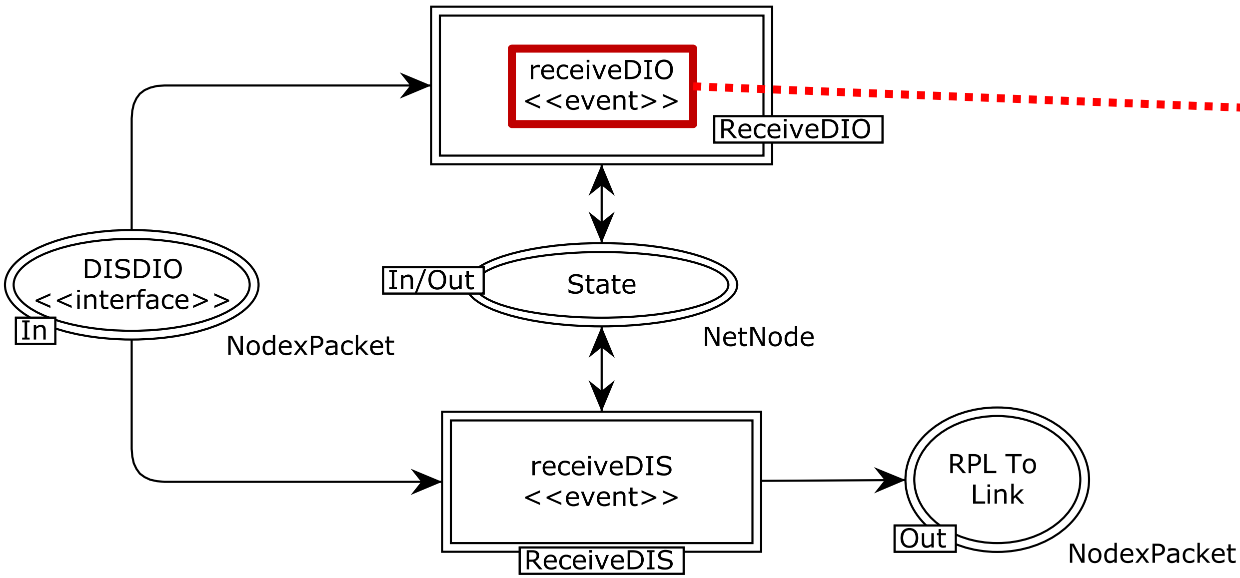
```
configuration RPLProtocolAppC { }
implementation {
   components MainC, RPLProtocolC, DAOC, DIOC;

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

- **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.



code generation template

```
<%import static
org.k1s.petriCode.generation.CodeGenerator.removePrags%>class
${name} {
<%
    if(binding.variables.containsKey('lcvs')){
        for(lcv in lcvs){
        %>def ${removePrags(lcv.name.text)}
${lcv.initialMarking.asString() == '()' ? '= true' : ''}\n<%
        }
    }
    if(binding.variables.containsKey('fields')){
        for(field in fields){
        %>def ${removePrags(field.name.text)}<%
        }
    }
    %>
    %%yield%%
}
```

```
configuration RPLProtocolAppC { }
implementation {
   components MainC, RPLProtocolC, DAOC, DIOC;

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

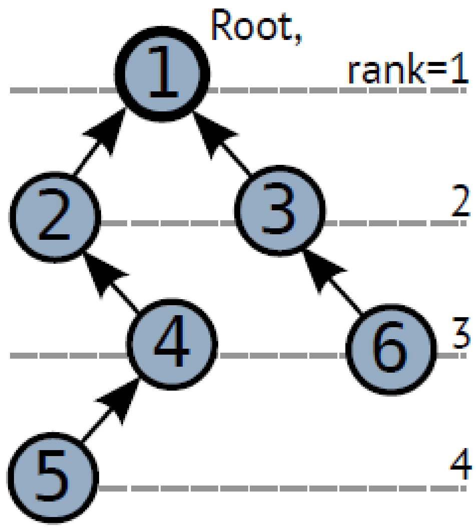
implementation code

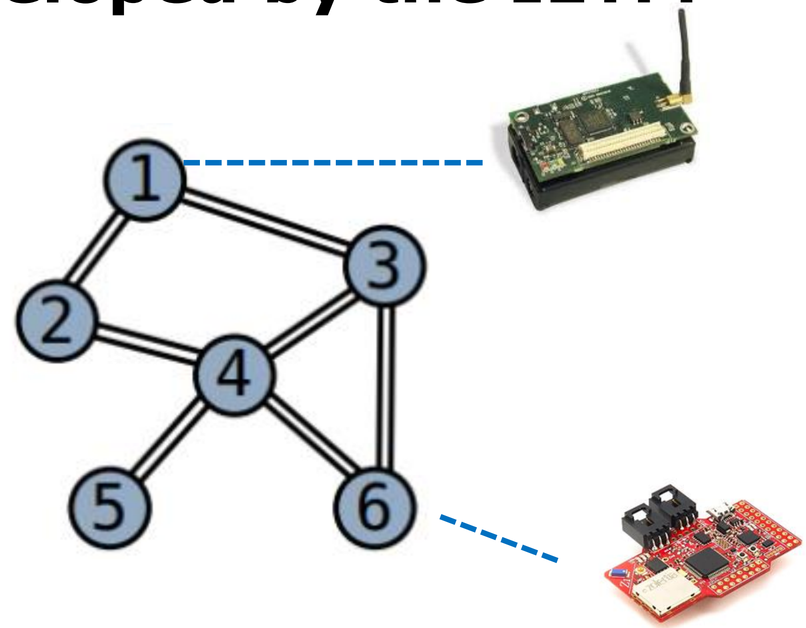
[12] K. Simonsen, L.M. Kristensen, and E. Kindler. Generating Protocol Software from CPN Models Annotated with Pragmatics. In Proc. of SBMF’13, volume 8195 of LNCS, pages 227–242. Springer-Verlag, 2013.

Case Study of the RPL Protocol

IEFT RPL Protocol

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

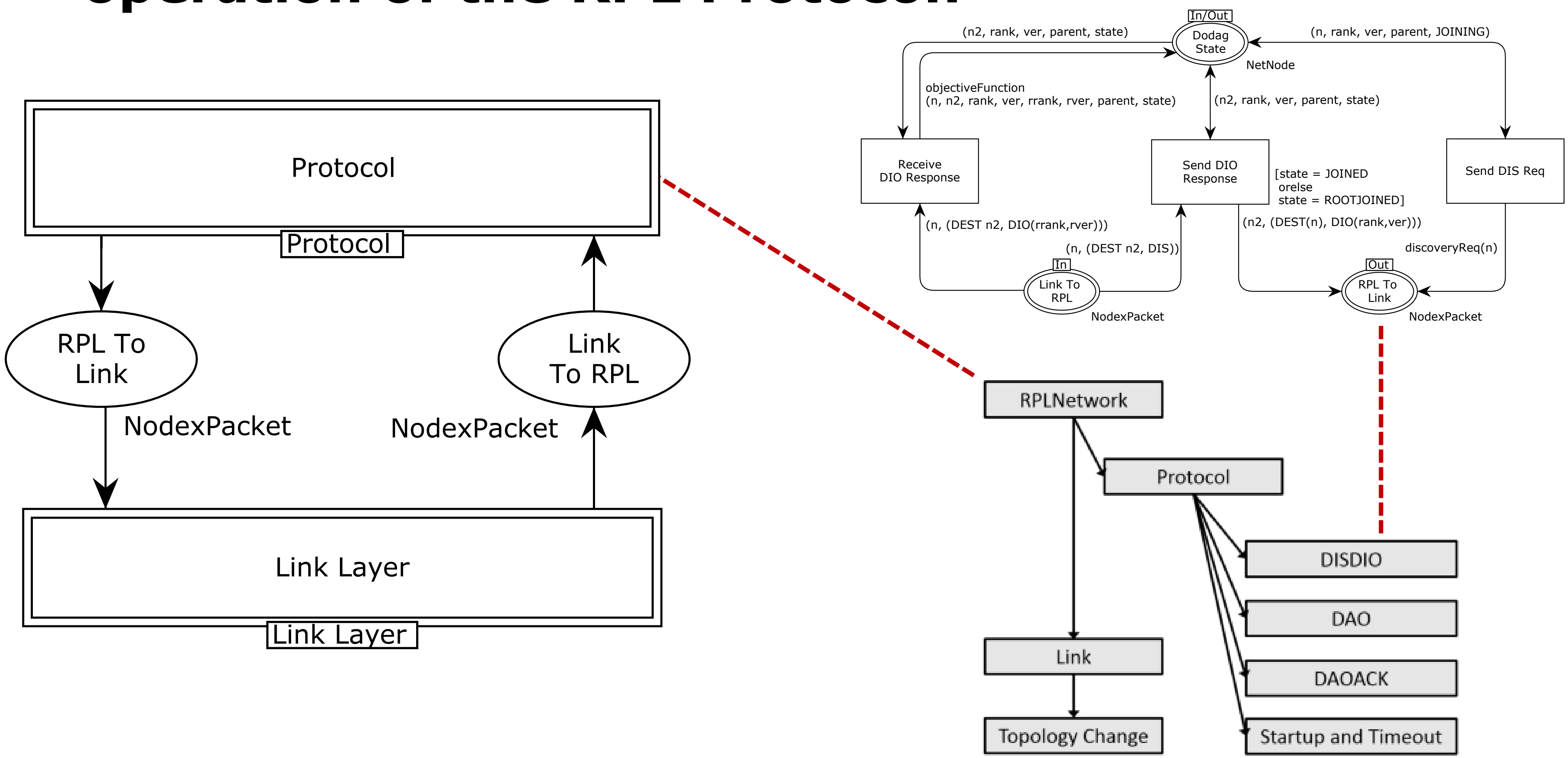


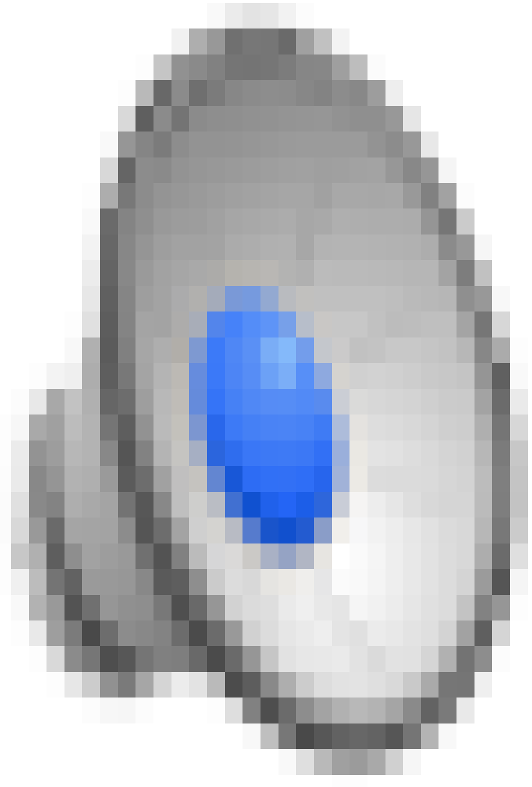


- **Supports a sensor nodes in establishing a DODAG for data collection purposes.**

RPL CPN Model

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





Platform: TinyOS and nesC

- **Operating system and programming language targeting resource constrained devices.**

```
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.**

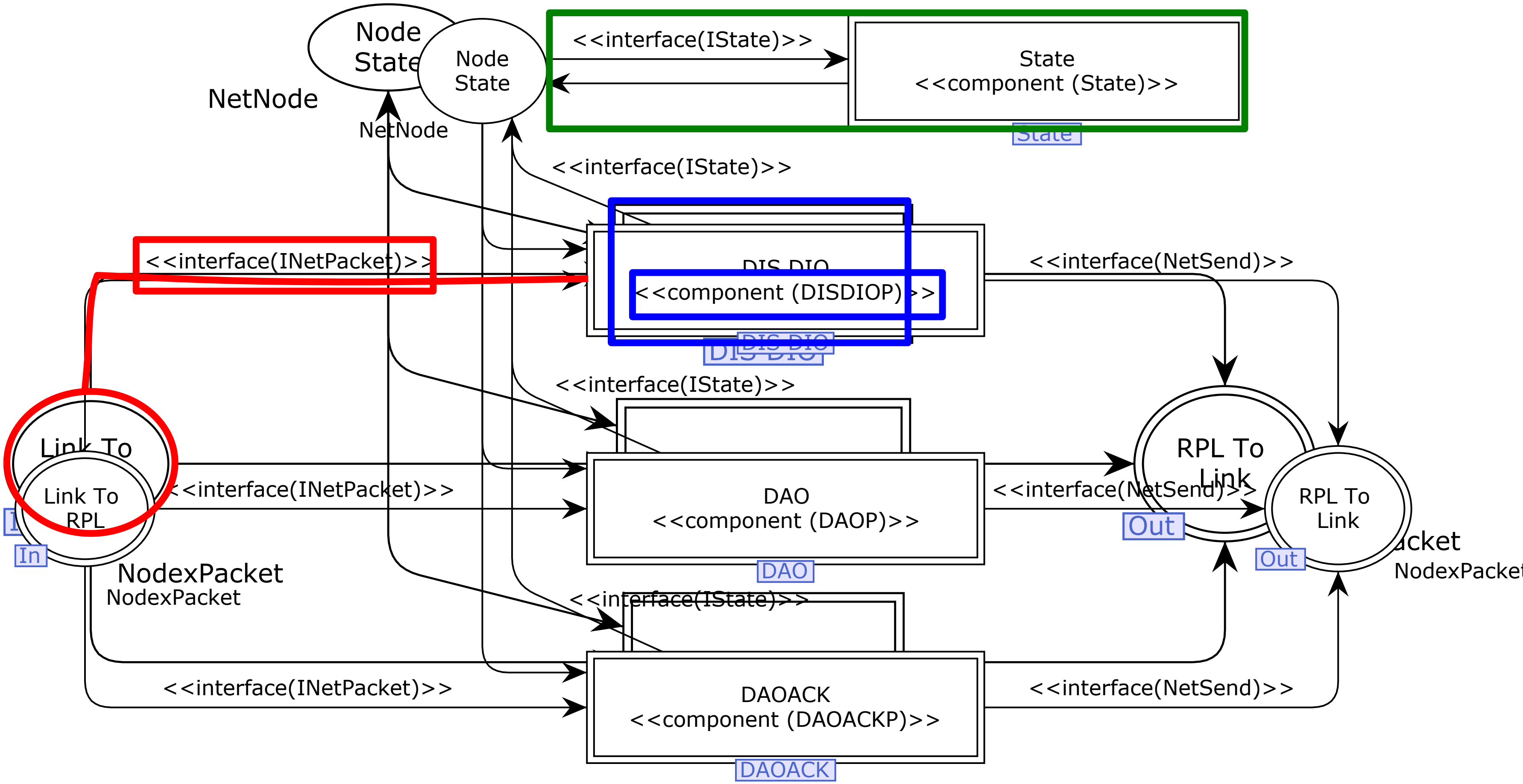
Proposed Refinement Methodology

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

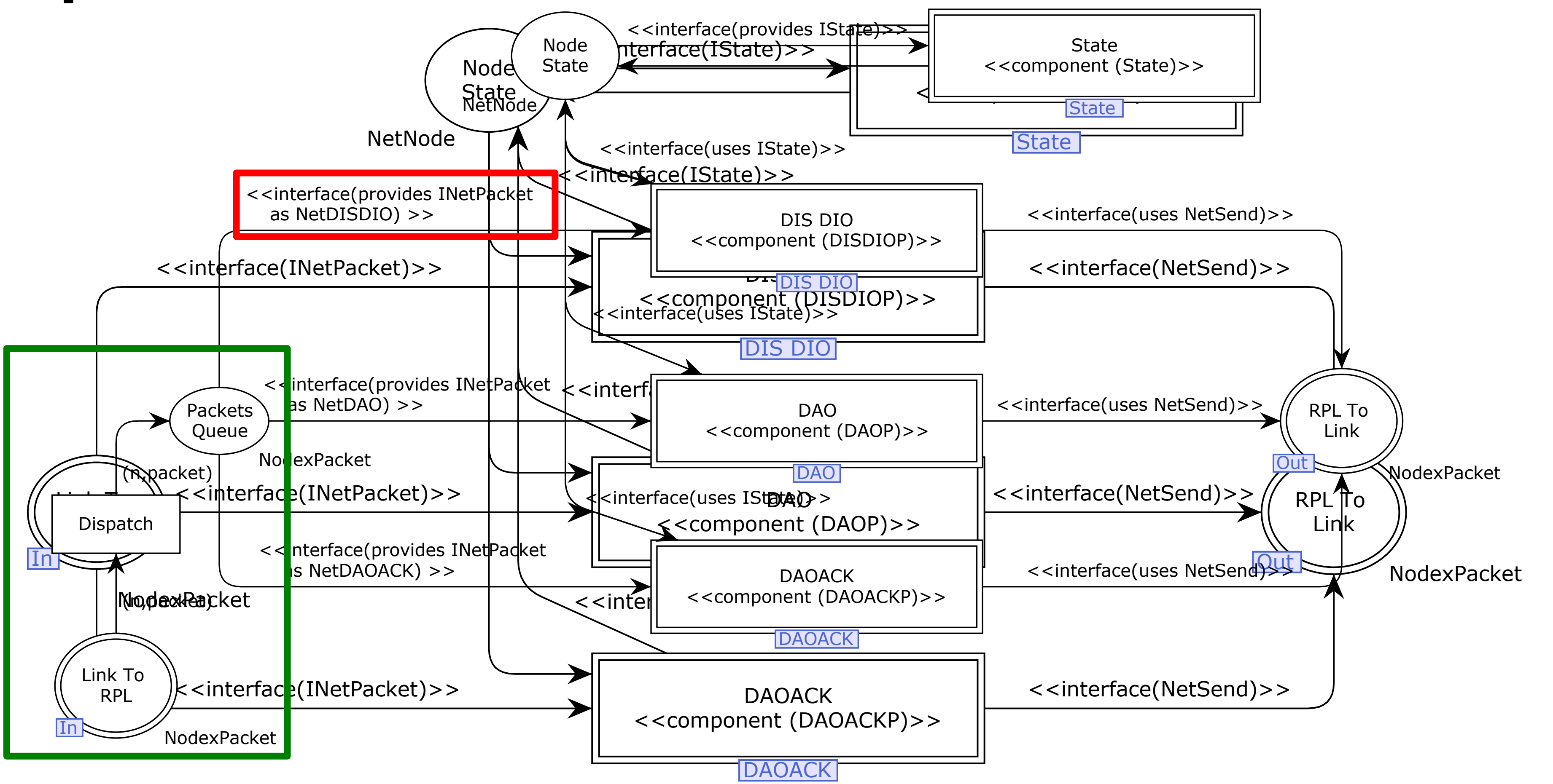
- **Identify <<components>> and <<interfaces>> via substitutions transitions and socket places:**



HØGSKOLEN I BERGEN
BERGEN UNIVERSITY COLLEGE

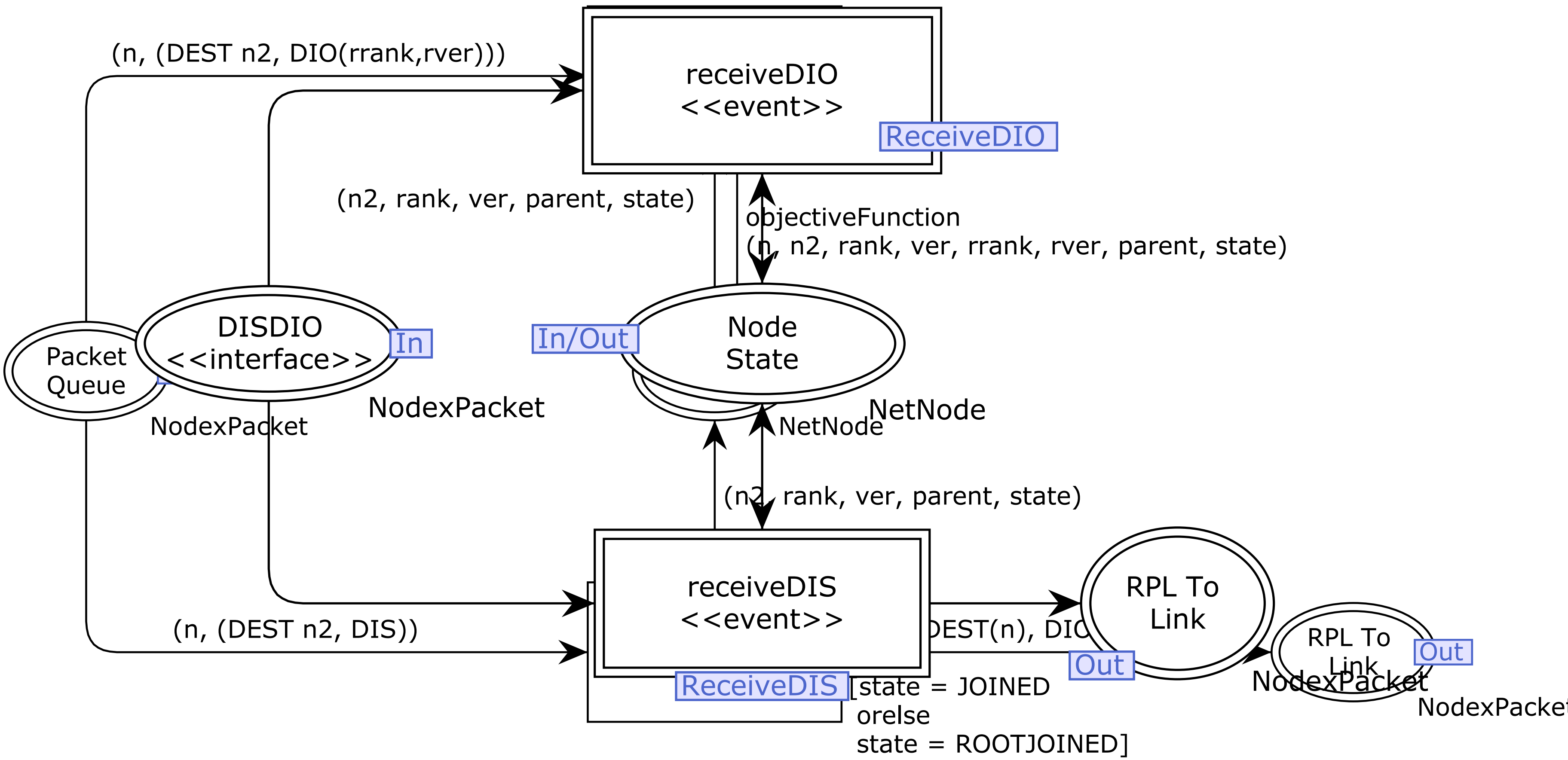
Step 2: Interface Naming and Use

- **Resolve naming conflicts and specify use and provision of interfaces:**



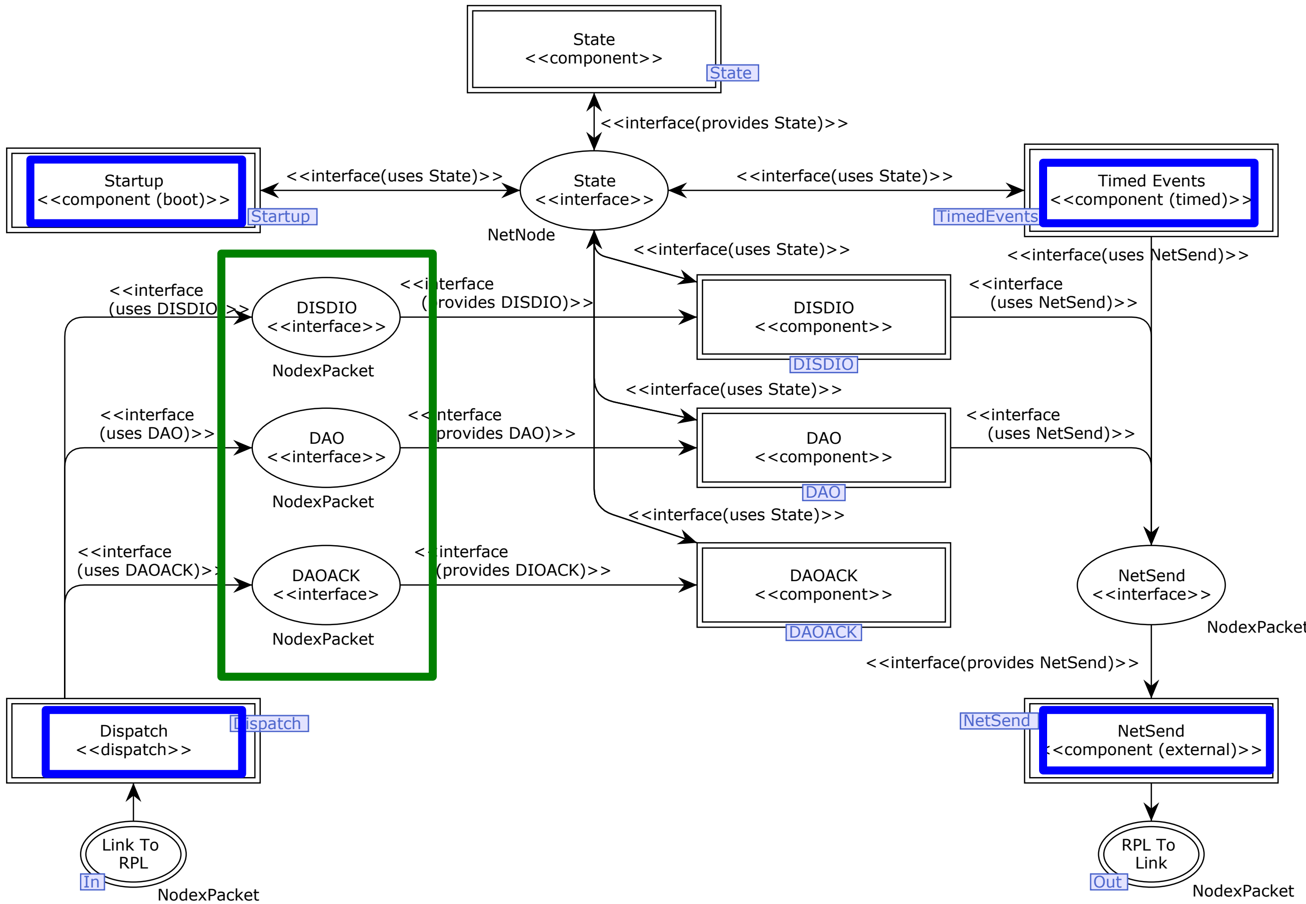
Step 3: Interface Signatures

- **Refine component <<interfaces>> to specify <<commands>> and <<events>>:**



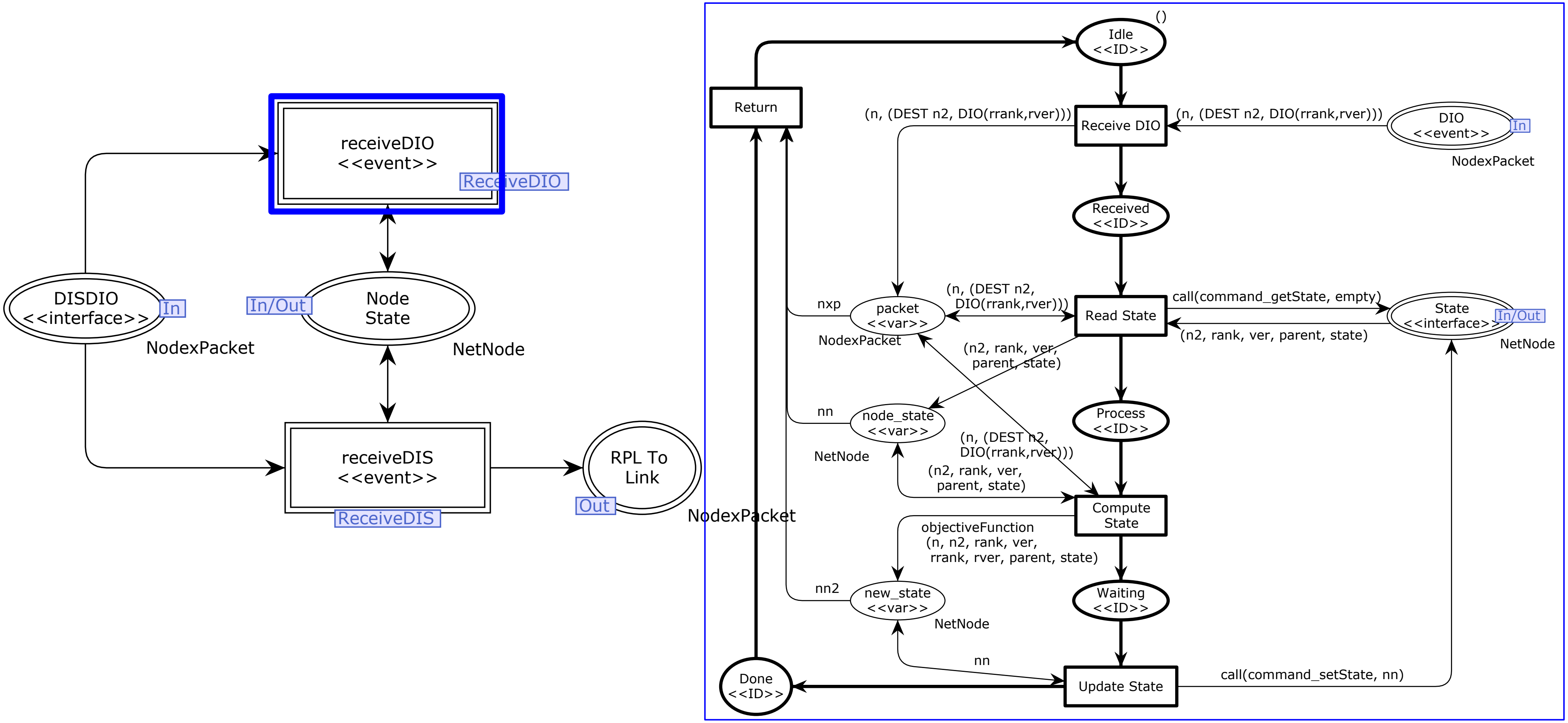
Step 4: Component Classification

- **Classifies components as boot-, timed-, dispatch-, external-, and regular components:**



Step 5: Internal Behaviour

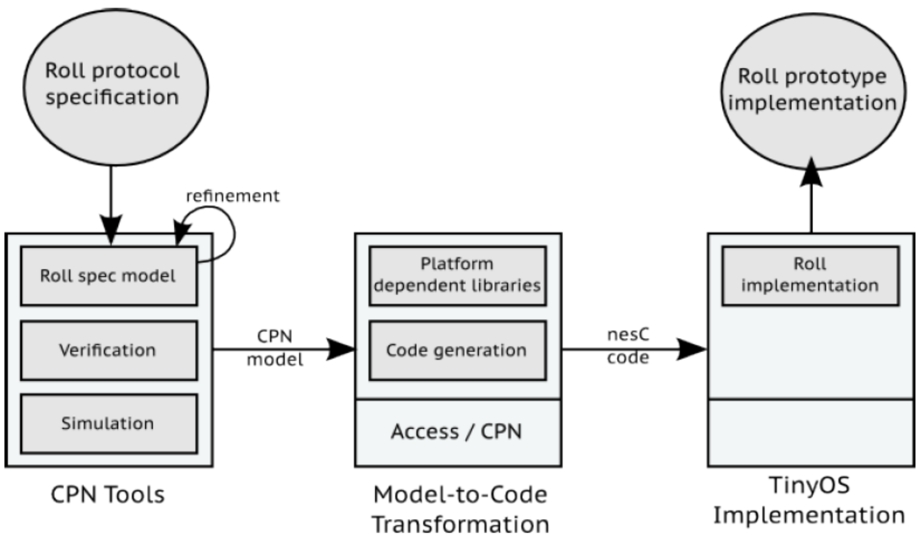
- **Makes explicit control flow and data access in the command and event implementations:**



Automated Code Generation

Code Generation

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



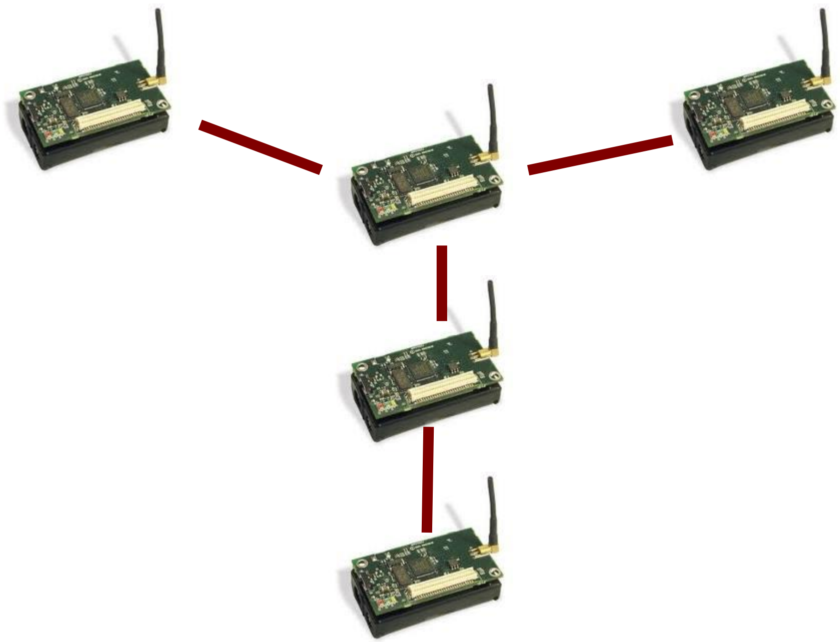
1. **Mapping CPN ML datatypes into corresponding nesC datatypes.**
2. **Interfaces based on places annotated with <<interface>>**
3. **Components based on substitution transition with <<component>>**
4. **Configuration and wiring based on <<component>> substitution transitions and <<interface>> arcs**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

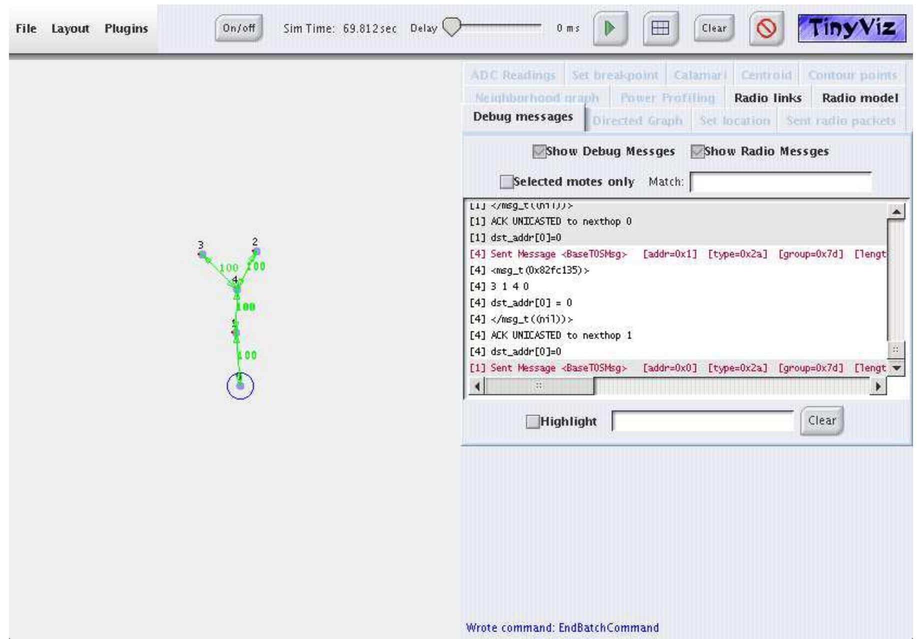
- **Top-down traversal of the CPN model invoking templates according to encountered pragmatics.**

[15] M. Westergaard. Access/CPN 2.0: A High-Level Interface to CPN Models. In Proc. of ICATPN´11, pp. 328-337, Vol. 6709 of LNCS, 2011.

Code Validation

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





- **Instrumentation and inspection of event-logs:**

```
DEBUG (0): 0:0:0.000000300 RPL | Application booted.
DEBUG (0): 0:0:0.000000300 RPL | State change: 0 -> 2.
```

HØGSKOLEN I BERGEN
BERGEN UNIVERSITY COLLEGE

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.