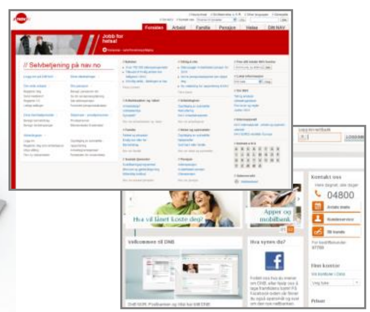
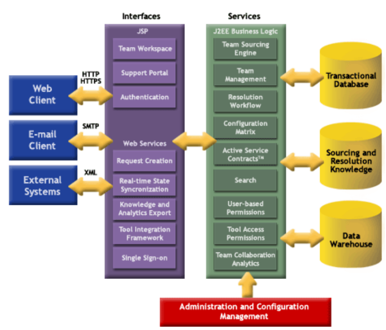
A Pragmatic Approach to Model-driven Code Generation from Coloured Petri Nets Simulation Models

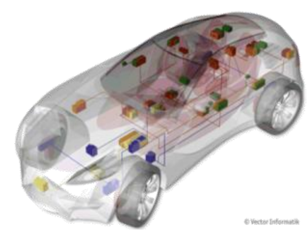
Lars M. Kristensen
Department of Computing
Bergen University College, NORWAY
Email: lmkr@hib.no / WWW: www.hib.no/ansatte/lmkr

HØGSKOLEN I BERGEN
BERGEN UNIVERSITY COLLEGE

Concurrent Systems

- The vast majority of IT systems today can be characterised as concurrent software systems:
  - Structured as a collection of concurrently executing software components and applications (parallelism).
  - Operation relies on communication, synchronisation, and resource sharing.




Internet and Web-based applications, protocols

Multi-core platforms and multi-threaded software

Embedded systems and networked control systems

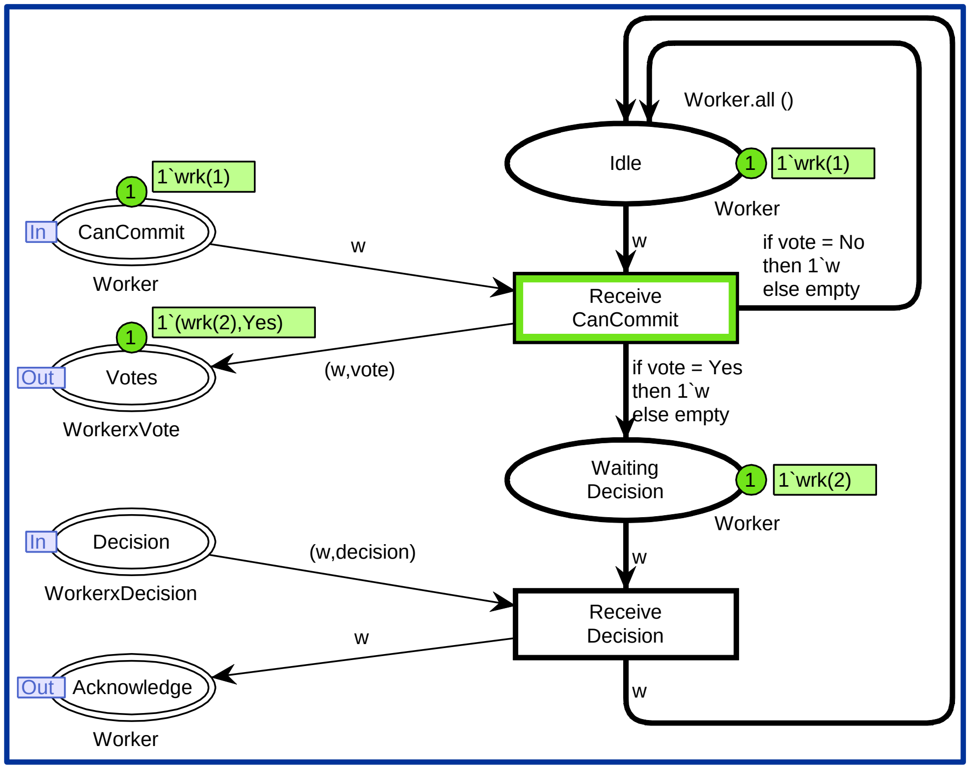
Concurrent Systems

- **Most software development projects are concerned with concurrent software systems.**
- **The engineering of concurrent systems is challenging due to their complex behaviour:**
  - Concurrently executing and independently scheduled software components.
  - Non-deterministic and asynchronous behaviour (e.g., timeouts, message loss, external events, …).
  - Almost impossible for software developers to have a complete understanding of the system behaviour.
  - Reproducing errors is often difficult.

- **Techniques to support the engineering of reliable concurrent systems are important.**

Coloured Petri Nets (CPNs)

- General-purpose graphical modelling language for the engineering of concurrent systems.
- Combines Petri Nets and a programming language:



Petri Nets: [C.A. Petri'62]

- graphical notation
- concurrency
- communication
- synchronisation
- resource sharing

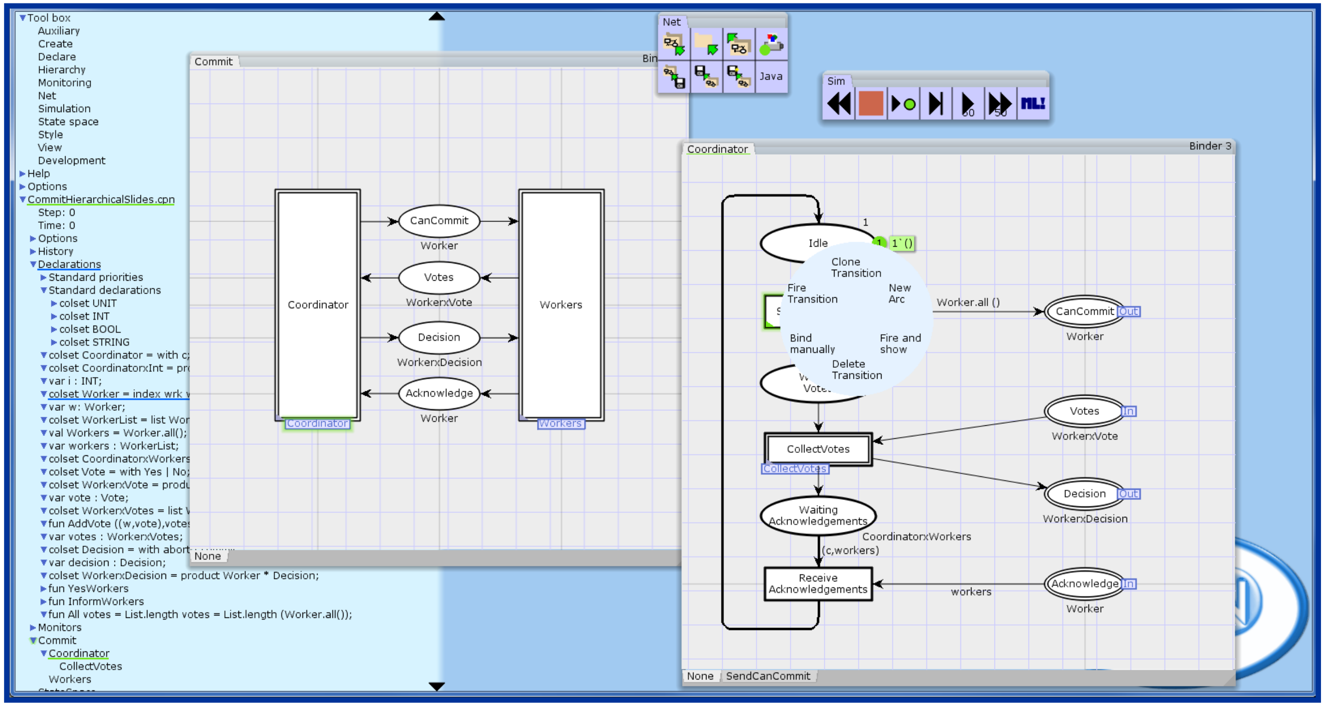
CPN ML (Standard ML):

- data manipulation
- compact modelling
- parameterisable models

High-Level Petri Net

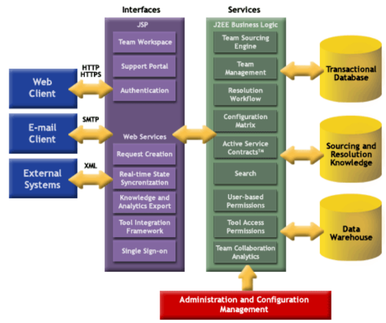
CPN Tools [www.cpntools.org]

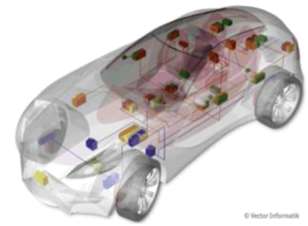
- **Practical use of CPNs is supported by CPN Tools:**



- **Editing and syntax check.**
- **Interactive- and automatic simulation.**
- **Application domain visualisation.**
- **Verification based on state space exploration.**
- **Simulation-based performance analysis.**

Application Areas





- **Communication protocols and data networks.**
- **Distributed algorithms and software systems.**
- **Embedded systems and control software.**
- **Business processes and workflow modelling.**
- **Manufacturing systems.**
- **... [http://cs.au.dk/cpnets/industrial-use/]**

1 2

Examples of CPN Tools users

North America

- Boeing
- Hewlett-Packard
- Samsung Information Systems
- National Semiconductor Corp.
- Fujitsu Computer Products
- Honeywell Inc.
- MITRE Corp.,
- Scalable Server Division
- E.I. DuPont de Nemours Inc.
- Federal Reserve System
- Bell Canada
- Nortel Technologies, Canada

Asia

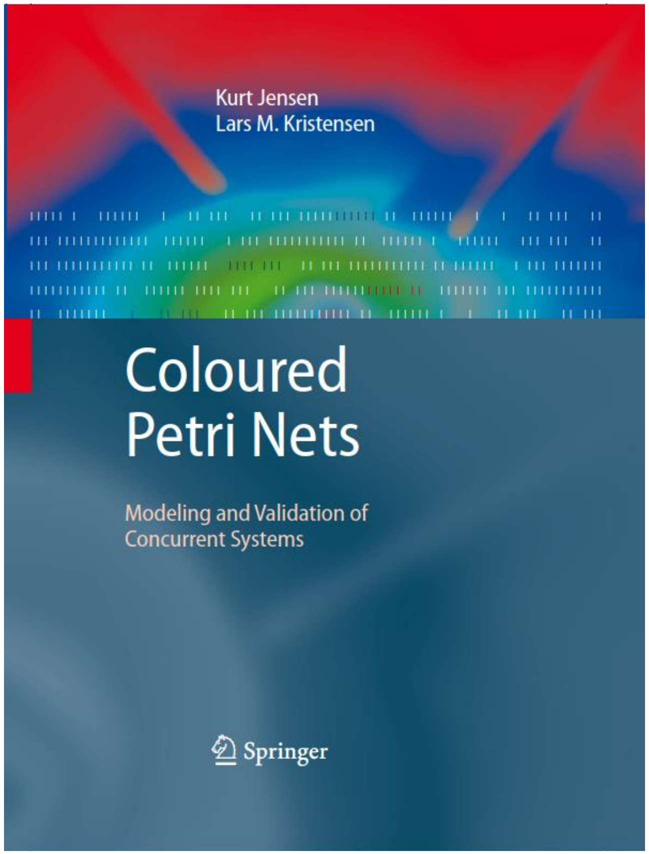
- Mitsubishi Electric Corp., Japan
- Toshiba Corp., Japan
- SHARP Corp., Japan
- Nippon Steel Corp., Japan
- Hongkong Telecom Interactive Multimedia System

Europe

- Alcatel Austria
- Siemens Austria
- Bang & Olufsen, Denmark
- Nokia, Finland
- Alcatel Business Systems, France
- Peugeot-Citroën, France
- Dornier Satellitensysteme, Germany
- SAP AG, Germany
- Volkswagen AG, Germany
- Alcatel Telecom, Netherlands
- Rank Xerox, Netherlands
- Sydkraft Konsult, Sweden
- Central Bank of Russia
- Siemens Switzerland
- Goldman Sachs, UK

1

Most Recent CPN Book



www.hib.no/ansatte/lmkr/cpnbook/

- K. Jensen and L.M. Kristensen. Coloured Petri Nets: Modelling and Validation of Concurrent Systems, Springer, 2009.
- K. Jensen, L.M. Kristensen, L. Wells: Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. Intl. Journal on Software Tools for Technology Transfer, Vol. 9, pp. 213-254, Springer, 2007.

Outline of this Talk

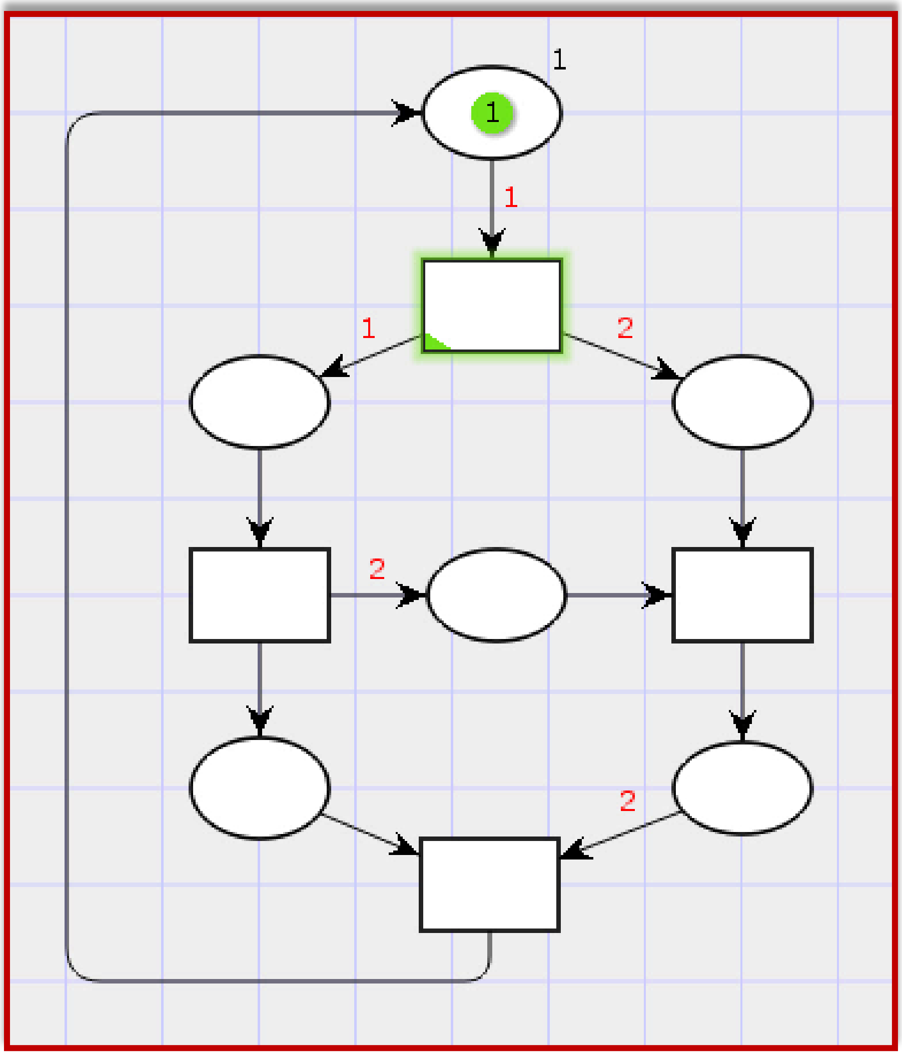
- **Part I: Coloured Petri Nets and CPN Tools**
  - **Example:** Two-phase commit transaction protocol
  - Basic concepts of Coloured Petri Nets (CPNs)
  - Short demonstration(s) of CPN Tools

- **Part II: Automated Code Generation**
  - **Case study:** The IETF WebSocket Protocol
  - Pragmatic-annotated CPN models
  - Template-based code generation for protocol software

Part I:
The Coloured Petri Nets Modelling Language and CPN Tools

Based on:
Kurt Jensen (Aarhus University, Denmark) and Lars M. Kristensen:
Coloured Petri Nets - A Graphical Modelling Language for Formal Modelling and Validation of Concurrent Systems. **Submitted to Communications of the ACM, February 2014.**

Quick Recap: Petri Net Concepts



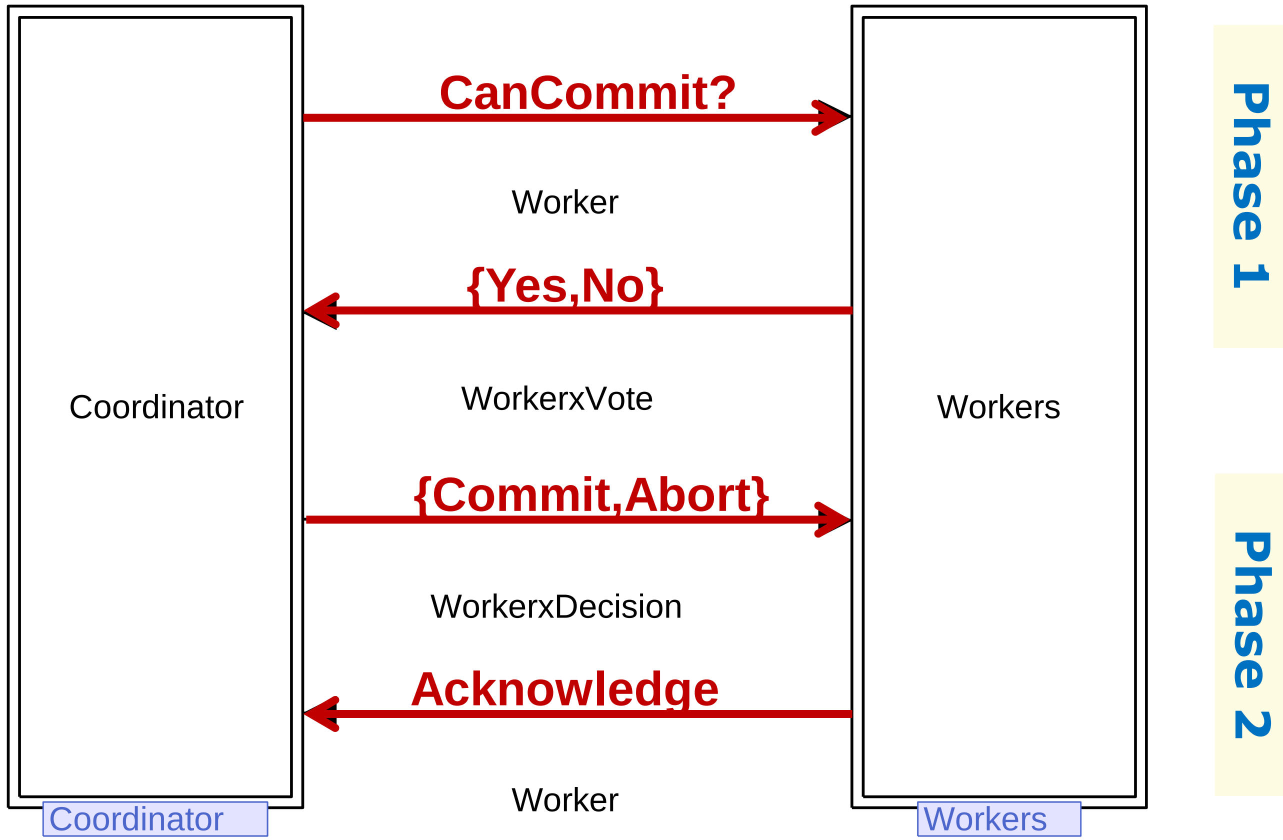
State modelling:
- **Places** (ellipses) that may hold **tokens**.
- **Marking (state)**: distribution of **tokens** on the places.
- **Initial marking**: initial state.

Event (action) modelling:
- **Transitions** (rectangles)
- **Directed arcs:** connecting places and transitions.
- **Arc weights:** specifying tokens to be added/removed.

Execution (token game):
- **Current marking**
- **Transition enabling**
- **Transition ocurrence**

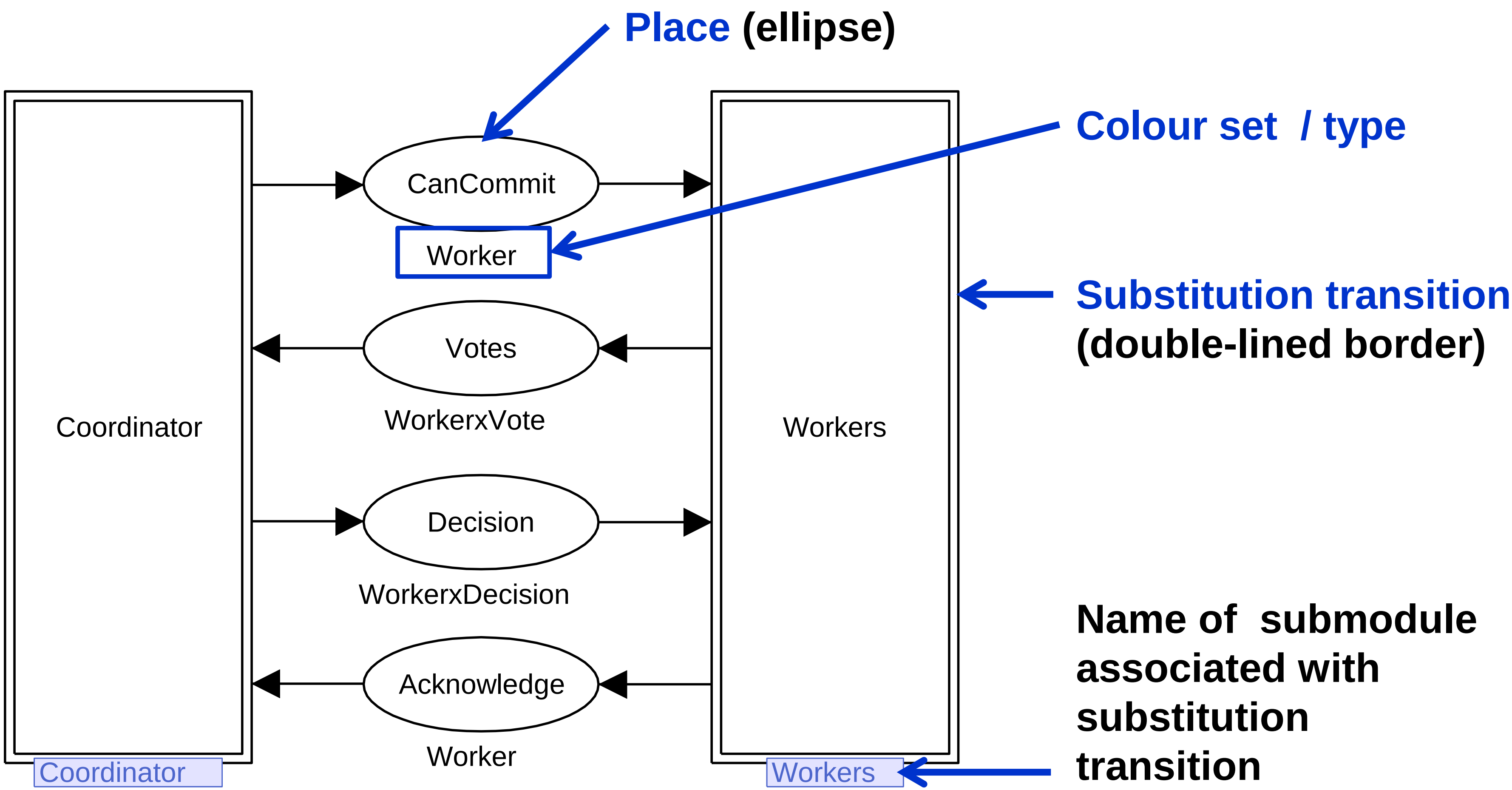
Example: Two-phase Commit Transaction Protocol

- A concurrent system consisting of a coordinator process and a number of worker processes:



CPN Model: Top-level Module

- **The CPN model is comprised of four modules hierarchically organised in three levels.**



Colour Sets and Tokens

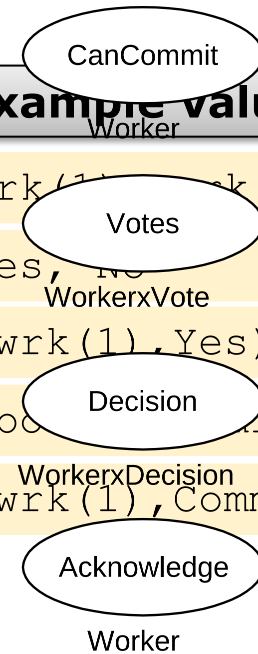
- **The colour set (or type) of a place determines the kinds of tokens that may reside on a place:**

Colour set definitions

```
val W = 2;
colset Worker         = index wrk with 1..W;
colset Vote           = with Yes | No;
colset WorkerxVote    = product Worker * Vote;

colset Decision           = with Abort | Commit;
colset WorkerxDecision = product Worker * Decision;
```

Example values

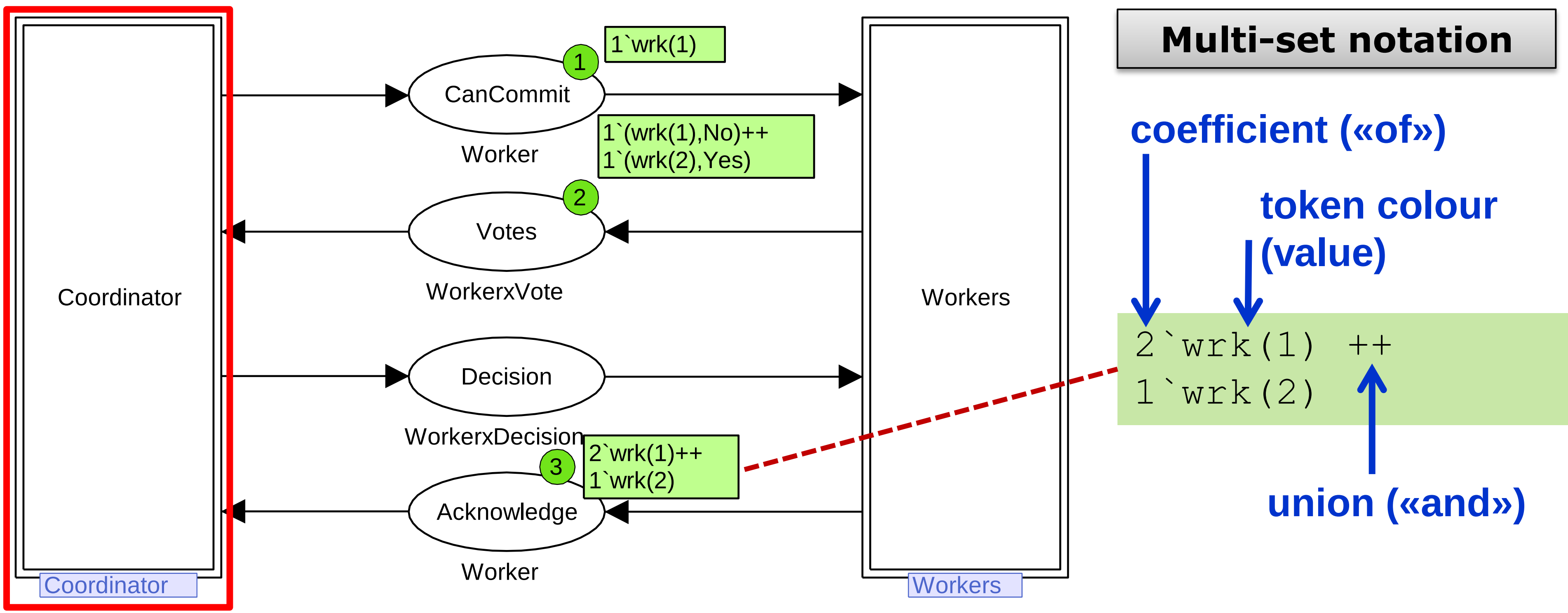
wrk(1), wrk(2)
Yes, No
(wrk(1),Yes)
Abort, Commit
(wrk(1),Commit)



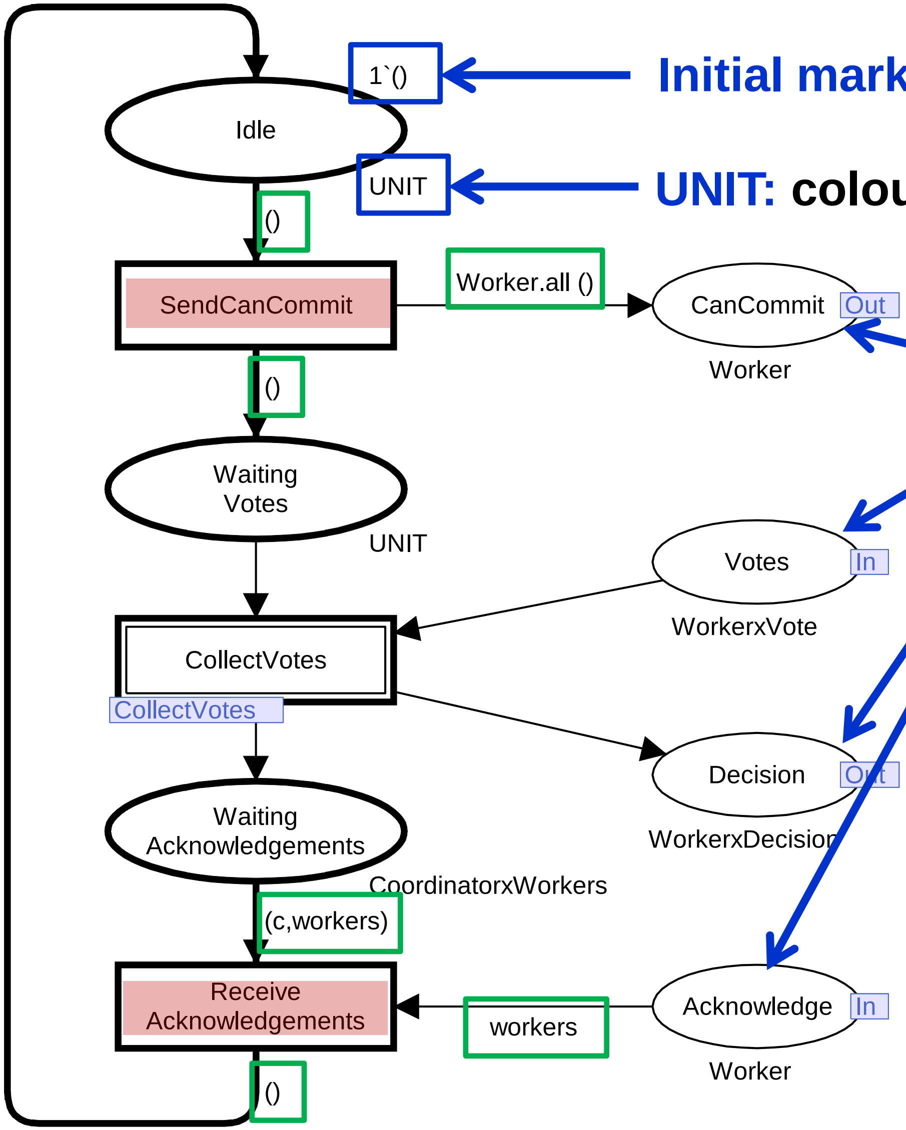
- **Colour sets are defined using the Standard ML based programming language CPN ML.**

Markings and Multi-sets

- A **marking** (state) is a distribution of **tokens** on the places of the model.
- Each place may hold a **multi-set of tokens** over the colour set of the place:



Coordinator Module



Initial marking: determines tokens in the initial state

UNIT: colour set with a single value () «unit»

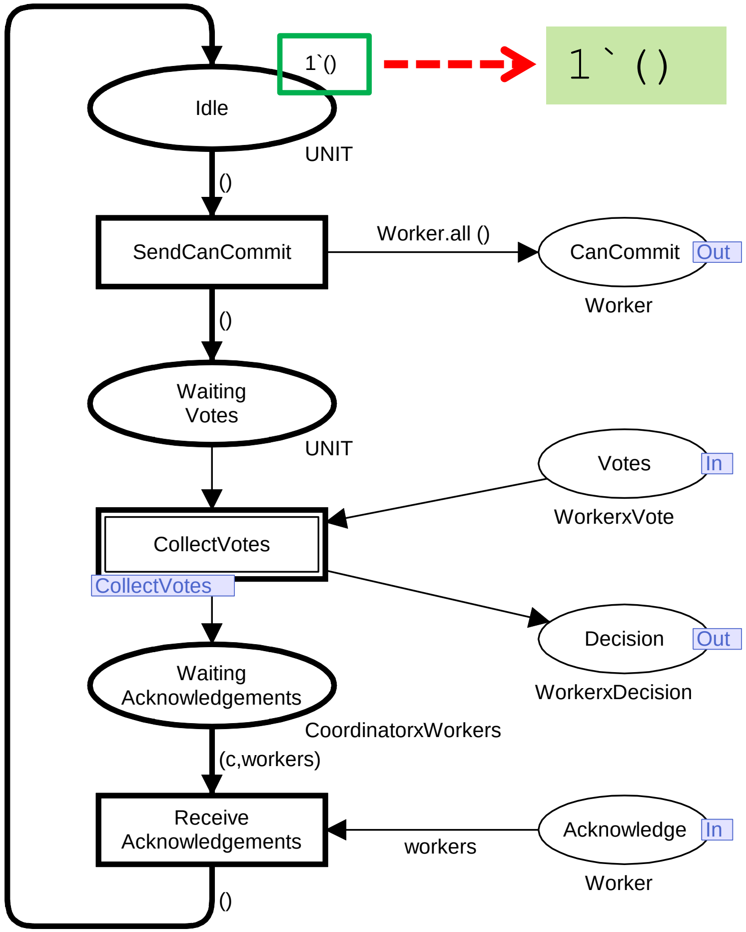
Port places: used for exchanging tokens with the upper-level module.

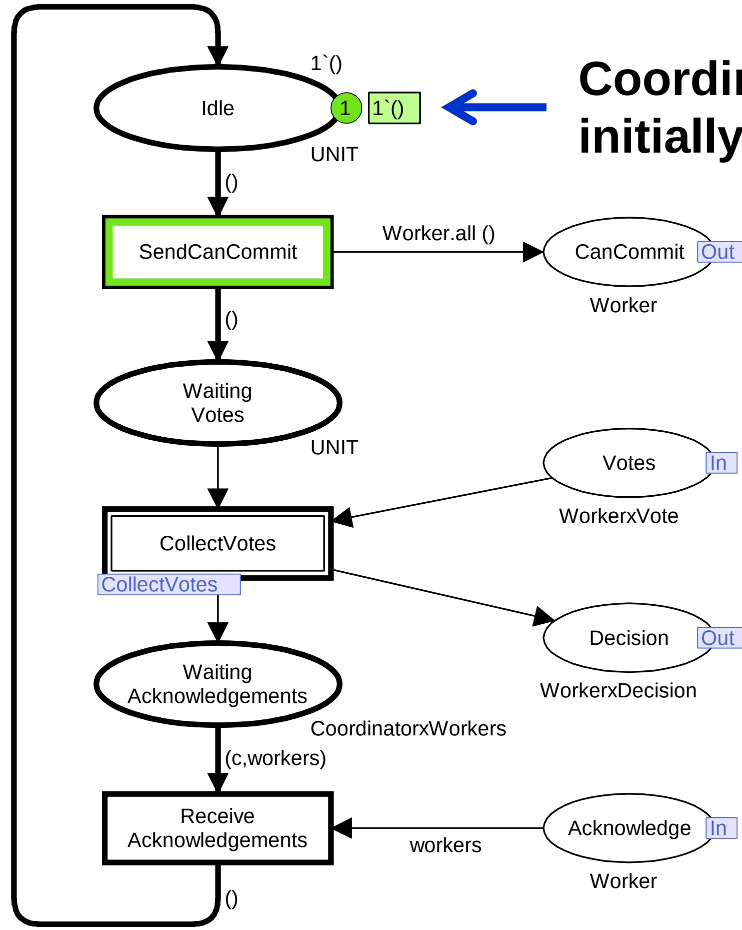
SendCanCommit and **ReceiveAcknowledgement** are ordinary transitions.

Arc expressions: determine the tokens added and removed by occurrences of enabled transitions.

Initial Marking

- The **initial marking** (state) is obtained by evaluating the **initial marking expressions**:

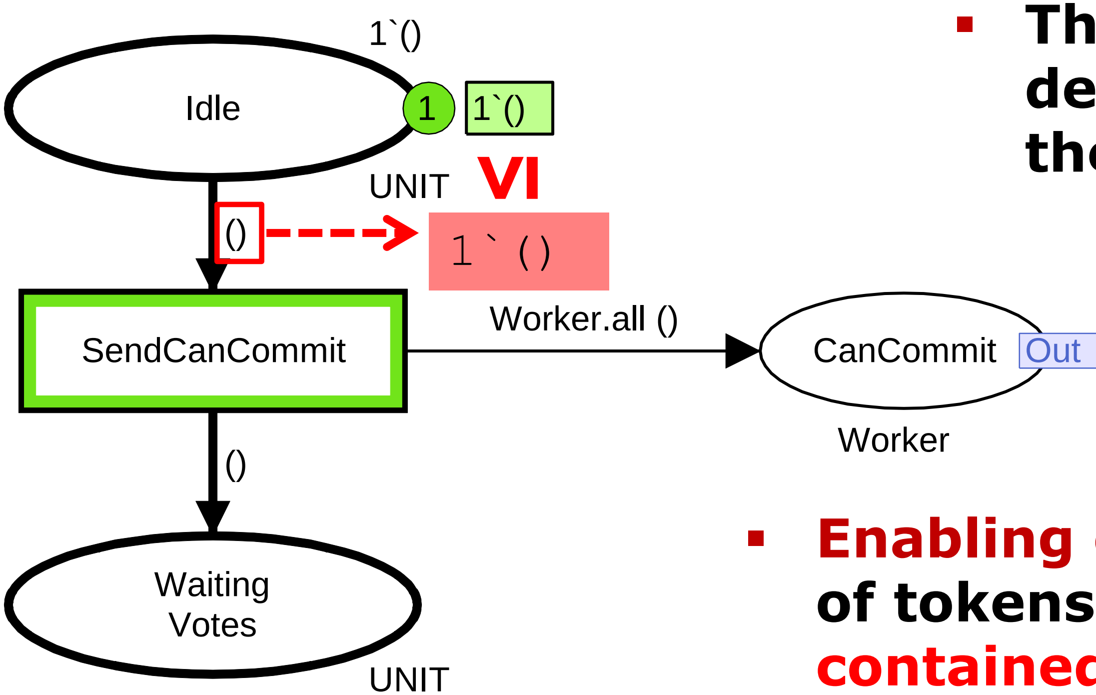




Coordinator is initially Idle

Transition Enabling

- A transition is enabled is there are sufficient tokens on each input place:

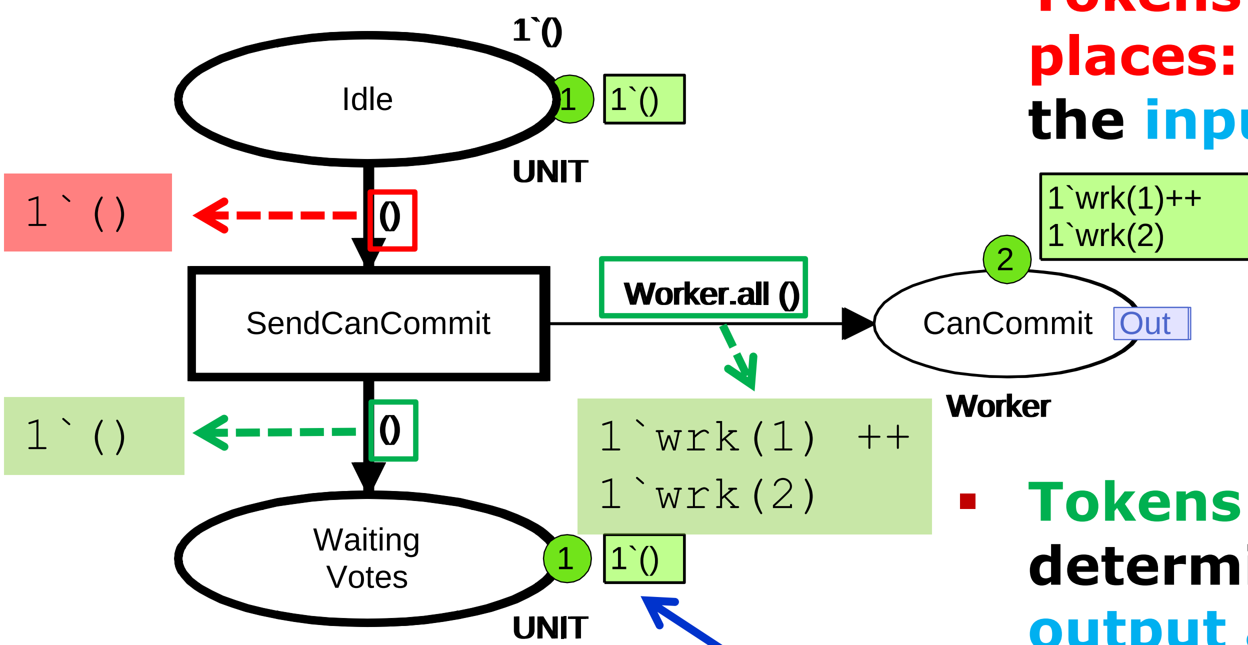


- The required tokens are determined by evaluating the input arc expressions.

- Enabling condition: the multi-set of tokens obtained must be contained in the multi-set of tokens present on the corresponding input place.

Transition Occurrence

- An enabled transition may occur changing the current marking (state) of its connected places:



- **Tokens removed from input places:** determined by evaluating the input arc expressions.

A message to each worker asking whether they CanCommit.

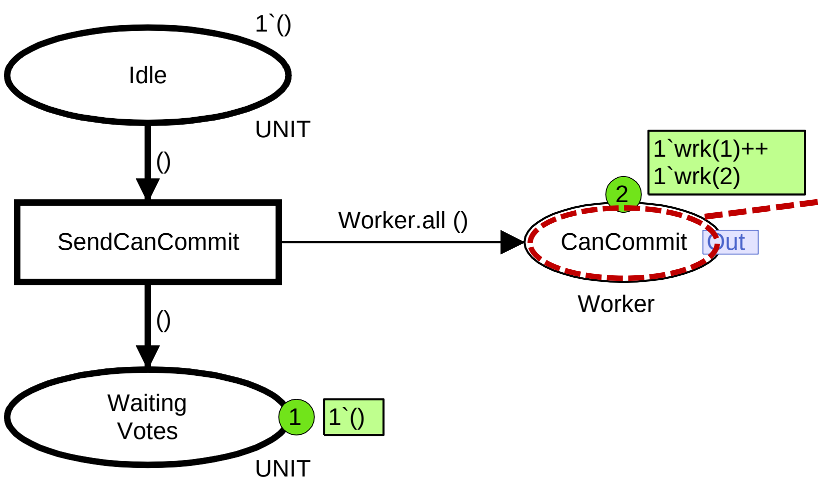
- **Tokens added to output places**: determined by evaluating the output arc expressions.

Coordinator is now Waiting for Votes

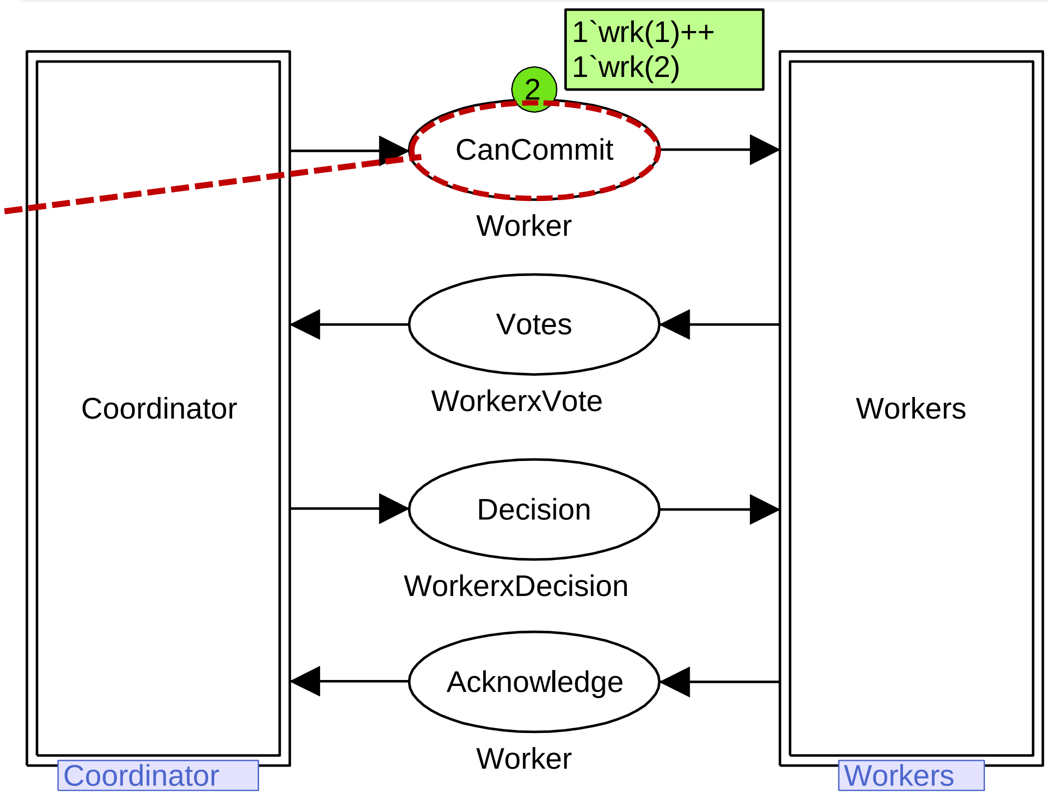
Port and Socket Places

- **Tokens added (removed) on a port place are added (removed) on the associated socket place:**

Coordinator module

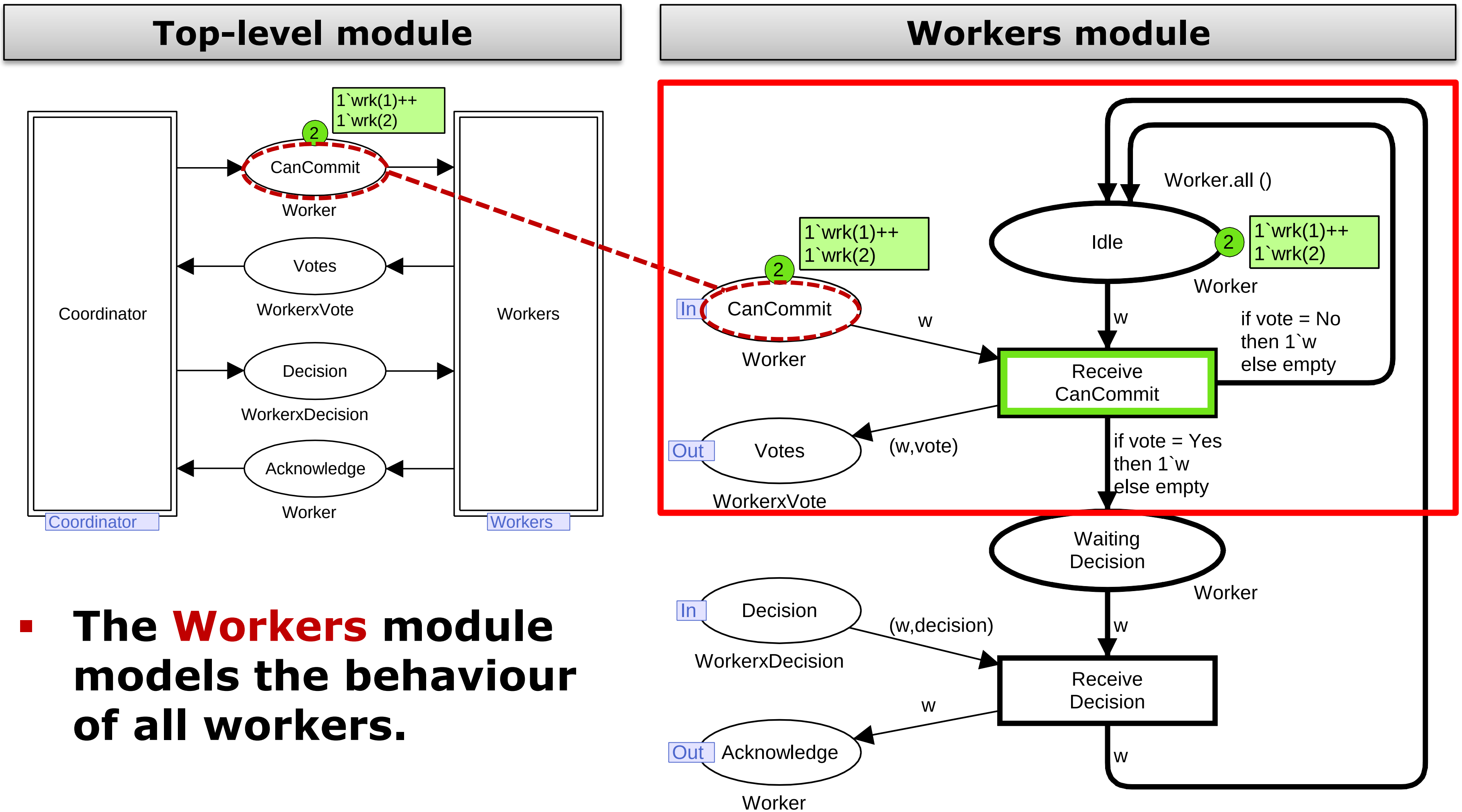


Top-level module



- **Associated port and socket places constitute a compound place.**

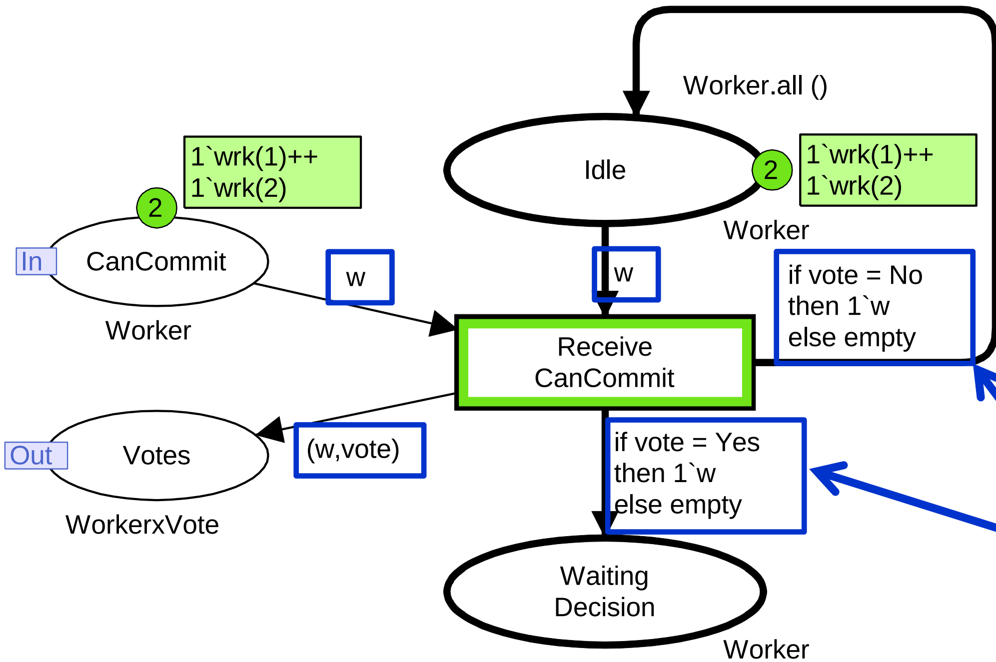
Workers Module



- The **Workers** module models the behaviour of all workers.

Transition Variables

- The arc expressions on the arcs of a transition may contain free variables:



Variable declarations

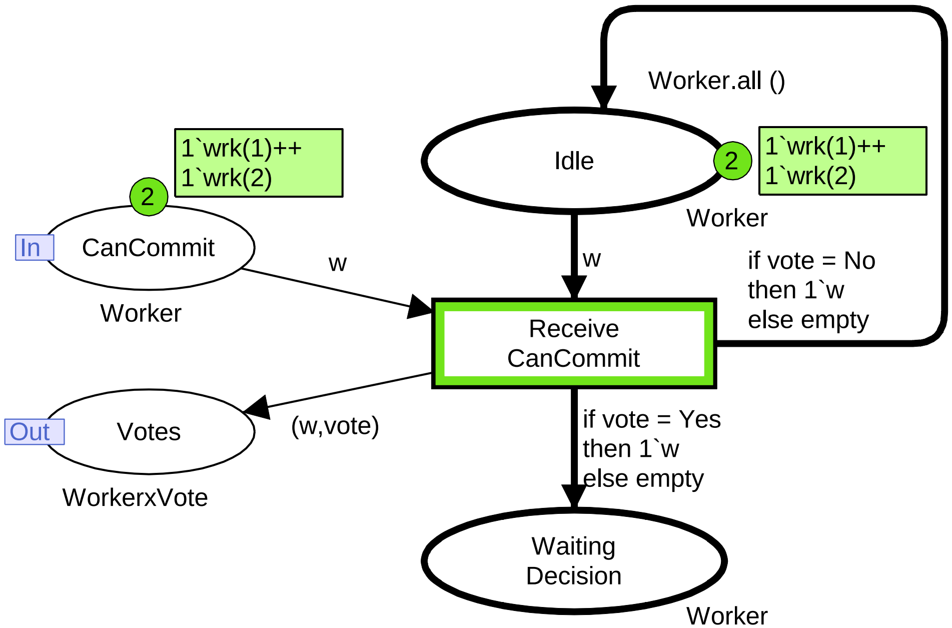
```
val W = 2;
colset Worker = index wrk with 1..W;
var w : Worker;

colset Vote = with Yes | No;
var vote : Vote;
```

if-then-else expressions with free variables vote and w.

- Transition ReceiveCanCommit has two free variables: vote and w.

Transition Bindings

- **Variables must be bound to values for a transition to be enabled and occur:**



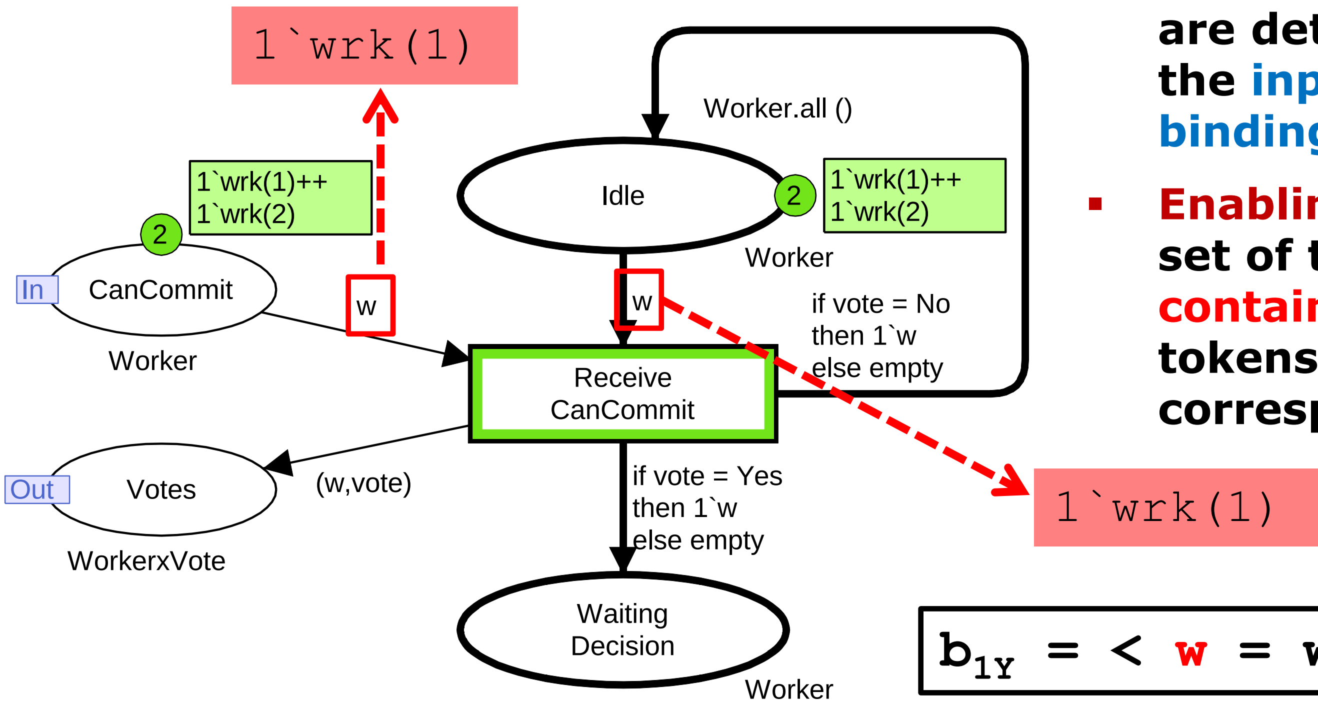

```
val W = 2;
colset Worker = index wrk with 1..W;
var w : Worker;

colset Vote = with Yes | No;
var vote : Vote;
```

Possible bindings
b_{1Y} = < w = wrk(1), vote = Yes >
b_{1N} = < w = wrk(1), vote = No >
b_{2Y} = < w = wrk(2), vote = Yes >
b_{2Y} = < w = wrk(2), vote = No >

- **The bindings correspond to possible enabling and occurrence modes of the transition.**

Enabling: Transition Bindings

- **A transition binding is enabled if there are sufficient tokens on each input place:**

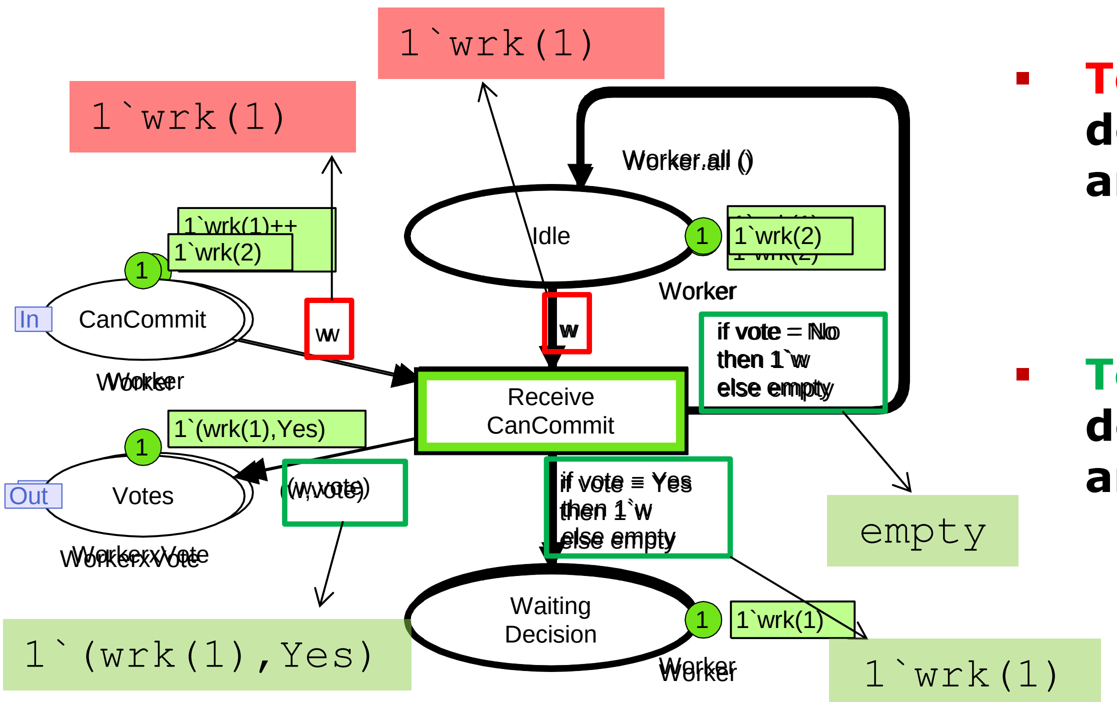


- **Tokens required on input places are determined by evaluating the input arc expressions in the binding under consideration.**
- **Enabling condition: the multi-set of tokens obtained must be contained in the multi-set of tokens present on the corresponding input place.**

$$b_{1Y} = < w = wrk(1), vote = Yes >$$

Occurrence: Transition Bindings

- **An enabled transition binding may occur changing the current marking (state):**



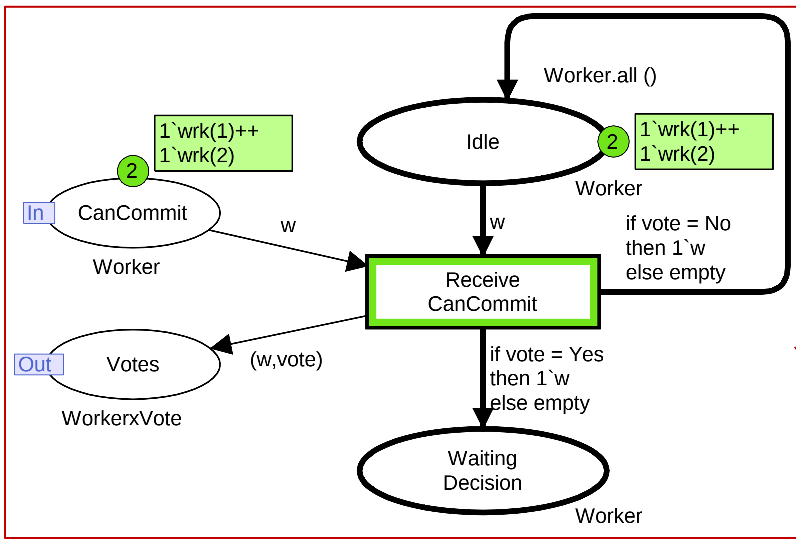
- **Tokens removed from input places:** **determined by evaluating the input arc expression in the binding.**
- **Tokens added to output places:** **determined by evaluating the output arc expressions in the binding.**

$$b_{1Y} = \langle\ w = wrk(1), vote = Yes\ \rangle$$

HØGSKOLEN I BERGEN
BERGEN UNIVERSITY COLLEGE

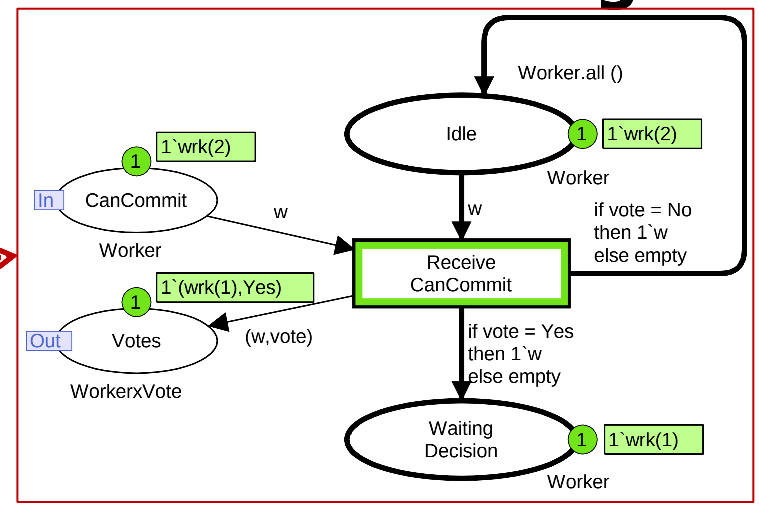
Occurrence: Transition Bindings

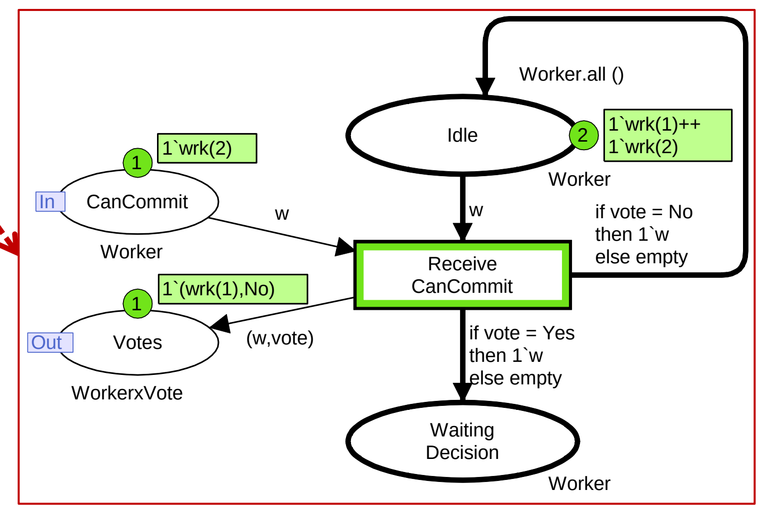
- **A transition may have several enabled bindings:**



b_{1Y}

b_{1N}

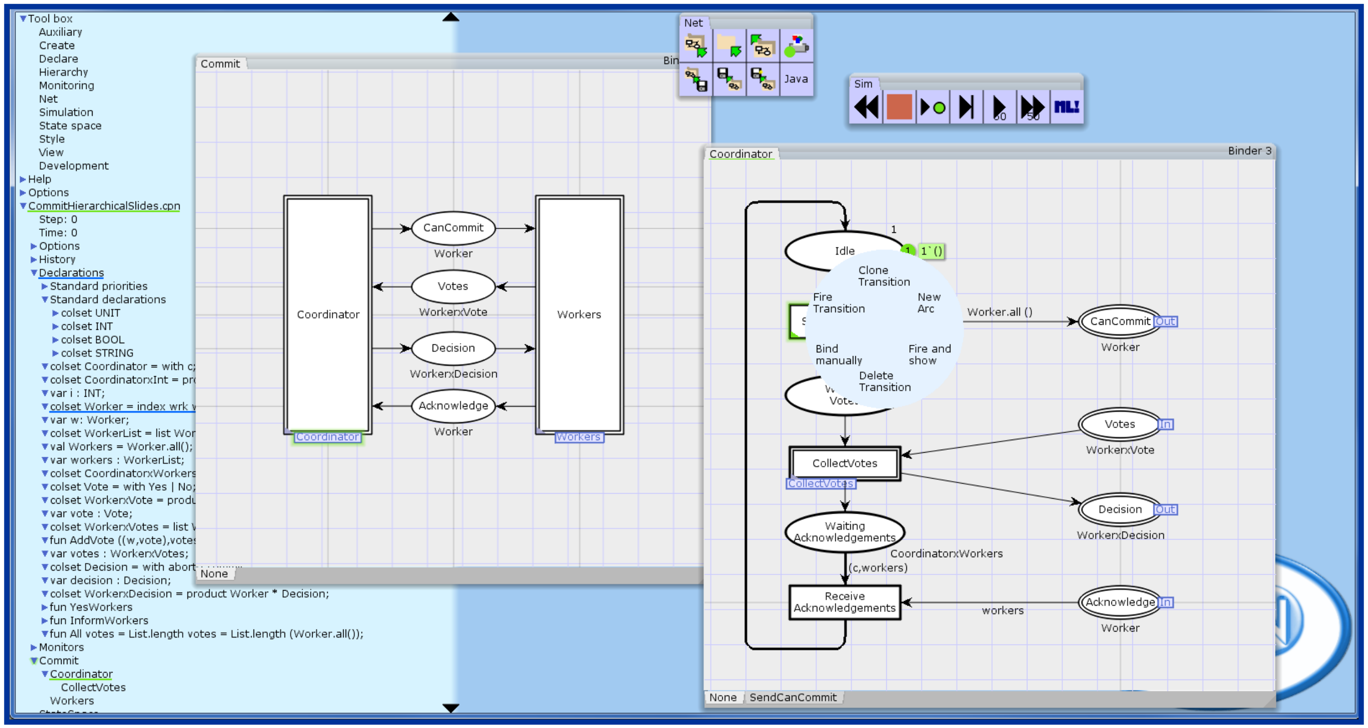




Bindings
b_{1Y} = < **w** = wrk(1),**vote** = Yes >
b_{1N} = < **w** = wrk(1),**vote** = No >
b_{2Y} = < **w** = wrk(2),**vote** = Yes >
b_{2Y} = < **w** = wrk(2),**vote** = No >

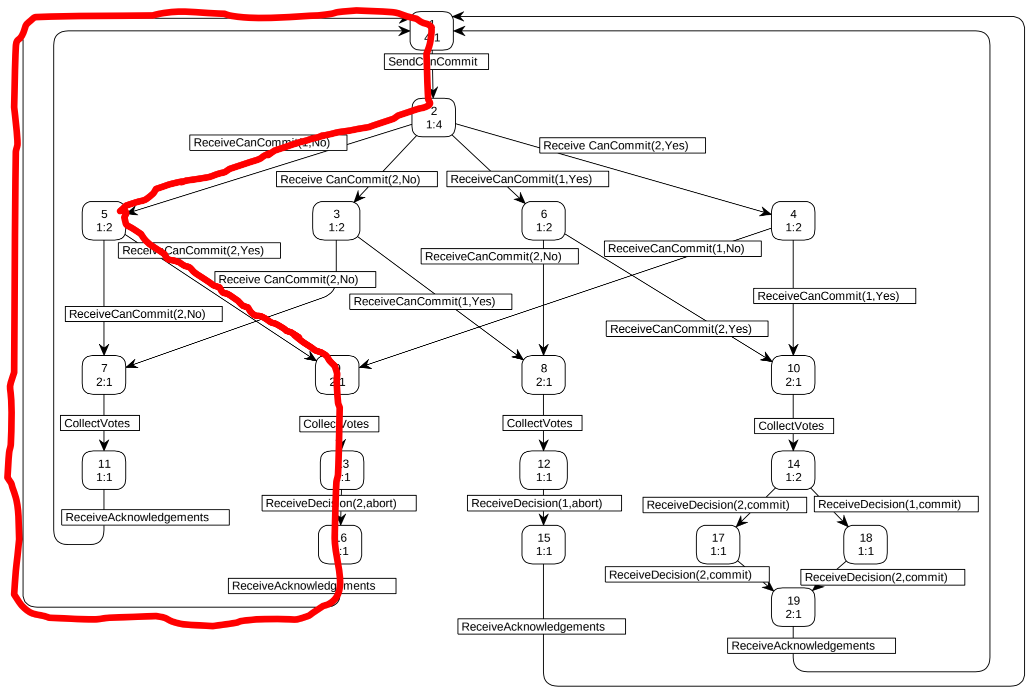
CPN Tools: Demo

- Simulation
- (Editing)



Verification and Model Checking

- **Formal verification of CPN models can be conducted using explicit state space exploration:**

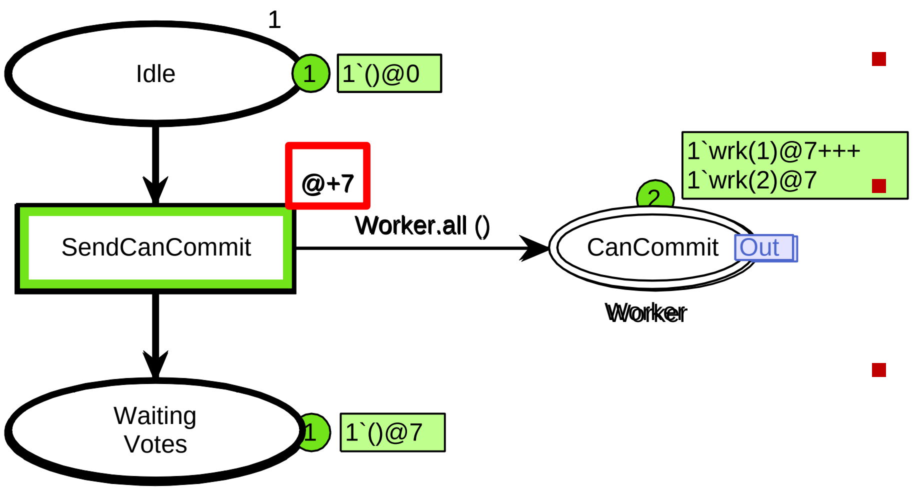


- **A state space represents all possible executions of the CPN model.**
- **Standard behavioural properties can be investigated using the state space report.**
- **Model-specific properties can be verified using queries and temporal logic model checking.**

- **Several advanced techniques available to alleviate the inherent state explosion problem.**

Performance Analysis

- **CPNs include a concept of time that can be used to model the timed taken by activities:**



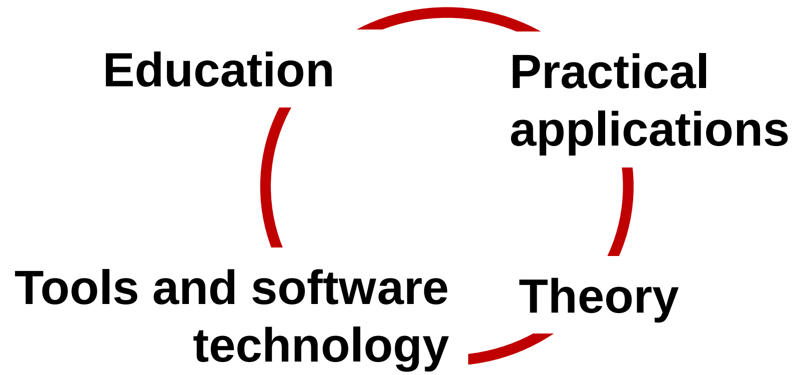
- **A global clock representing the current model time.**

  Tokens carry time stamps describing the earliest possible model time at which they can be removed.
- **Time inscriptions on transitions and arcs are used to give time stamps to the tokens produced on output places.**

- **Random distribution functions can be used in arc expressions (delays, packet loss, ...).**
- **Data collection monitors and batch simulations can be used to compute performance figures.**

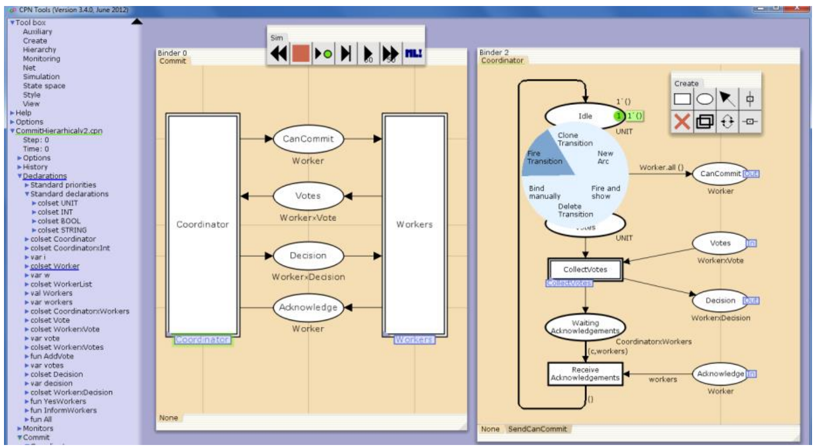
Perspectives on CPNs

- **Modelling language combining Petri Nets with a programming language.**
- **The development has been driven by an application-oriented research agenda**



- **Key characteristics:**
  - **Few but still powerful and expressive modelling constructs.**
  - **Implicit concurrency inherited from Petri nets: everything is concurrent unless explicit synchronised.**
  - **Verification and performance analysis supported by the same modelling language.**

Part II: Automated Code Generation from CPN Simulation Models



Based on:

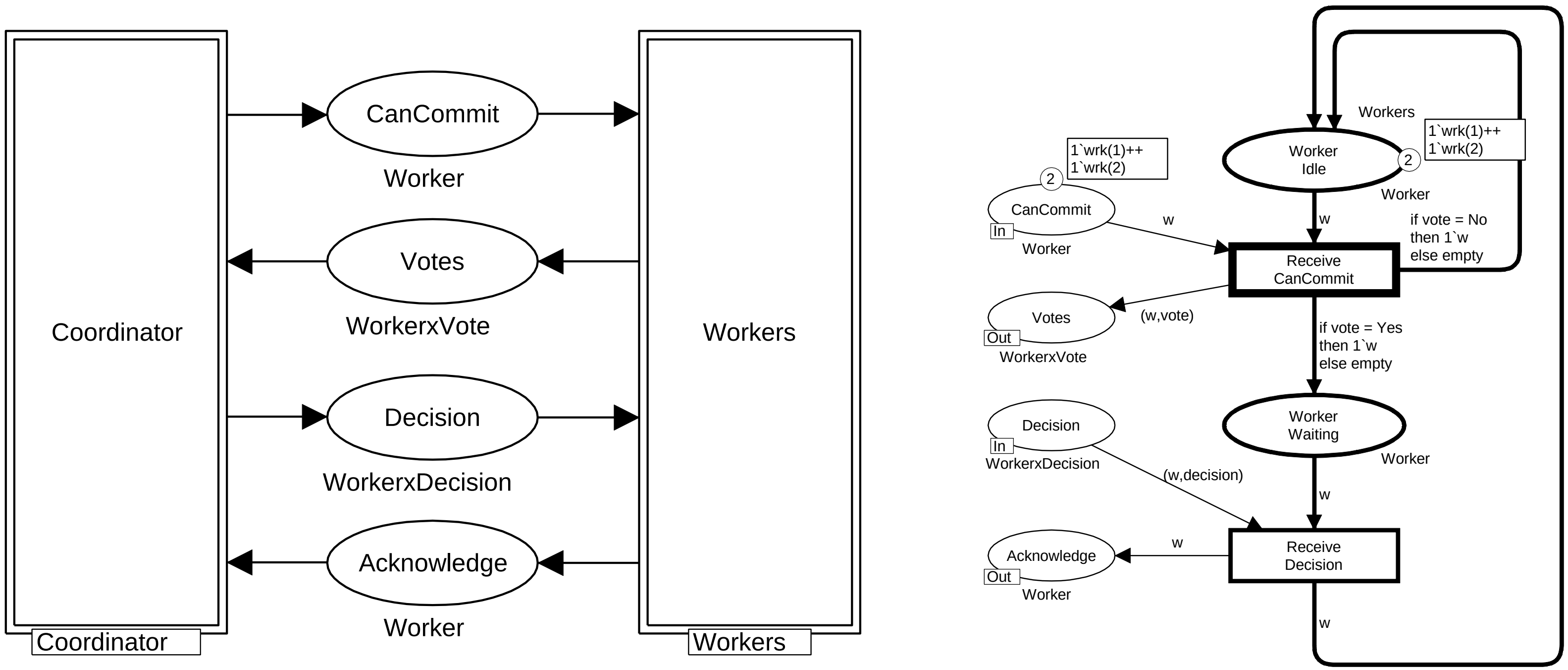
Kent I.F. Simonsen and Lars M. Kristensen:
Implementing the WebSocket Protocol based on Formal Modelling and Automated Code Generation. **To appear in Proc. of 14th Intl. Conference on Distributed Applications and Interoperable Systems, Springer, 2014.**

Motivation and Background

- **CPNs have been widely used for modelling and validation of communication protocols*:**
  - Application Layer Protocols: IOTP, SIP, WAP, …
  - Transport Layer Protocols: TCP, DCCP, SCTP, …
  - Routing Layer Protocols: DYMO, AODV, ERDP, …
- **It would be desirable to use CPN models more directly for implementation of protocol software.**
- **Limited work on automatic code generation.**
- **This part of the talk:**
  - **A newly developed approach to structure-based code generation from CPN models.**
  - **Application to the IETF WebSocket Protocol.**

Automated Code Generation

- **It is difficult (in general) to recognize programming language constructs in CPNs:**



- **Conclusion:** **some additional syntactical constraints and/or annotations are required.**

Requirements

1. **Platform independence:**
   - Enable code generation for multiple languages / platforms.
2. **Integratebility of the generated code:**
   - Upwards integration: the generated code must expose an explicit interface for service invocation.
   - Downwards integration: ability for the generated code to call and rely on underlying libraries.
3. **Model checking and property verification:**
   - Code generation capability should not introduce complexity problems for the verification of the model.
4. **Readability of the generated code:**
   - Enable code review of the automatically generated code.
   - Enable performance enhancements (if required).

Overview of Approach

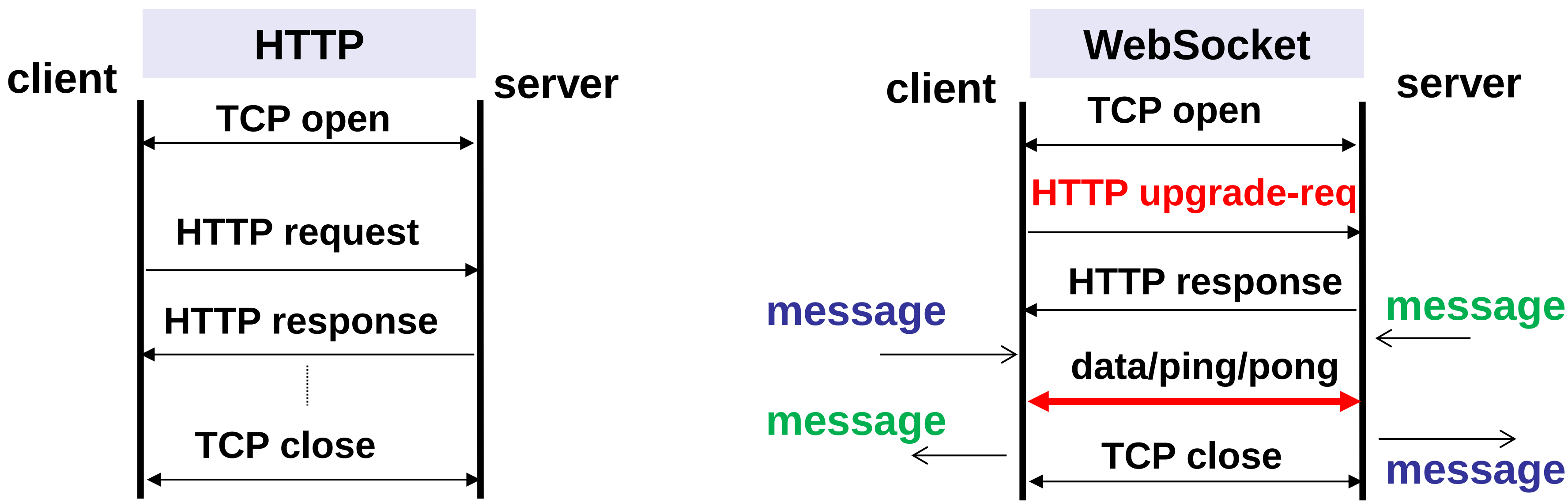
- **Modelling structure requiring the CPN model to be organised into three levels:**
  1. **Protocol system level specifying the protocol principals and the communication channels between them.**
  2. **Principal level reflecting the life-cycle and services provided by each principal in the protocol system.**
  3. **Service level specifying the behaviour of the services provided by each principal.**
- **Annotate the CPN model elements with code generation pragmatics to direct code generation.**
- **A template-based model-to-text transformation for generating the protocol software.**

Code Generation Pragmatics

- **Syntactical annotations (name and attributes) that can be associated with CPN model elements:**
  - Structural pragmatics designating principals and services.
  - Control-flow pragmatics identifying control-flow elements and control-flow constructs.
  - Operation pragmatics identifying data manipulation.
- **Template binding descriptors associating the pragmatics and code generation templates:**
  - Bridges the gap between the platform independent CPN simulation model and the target platform considered.
  - Code can be generated for different platforms (Groovy, Clojure, Java, Python) by changing the template binding descriptors.

The IETF WebSocket Protocol

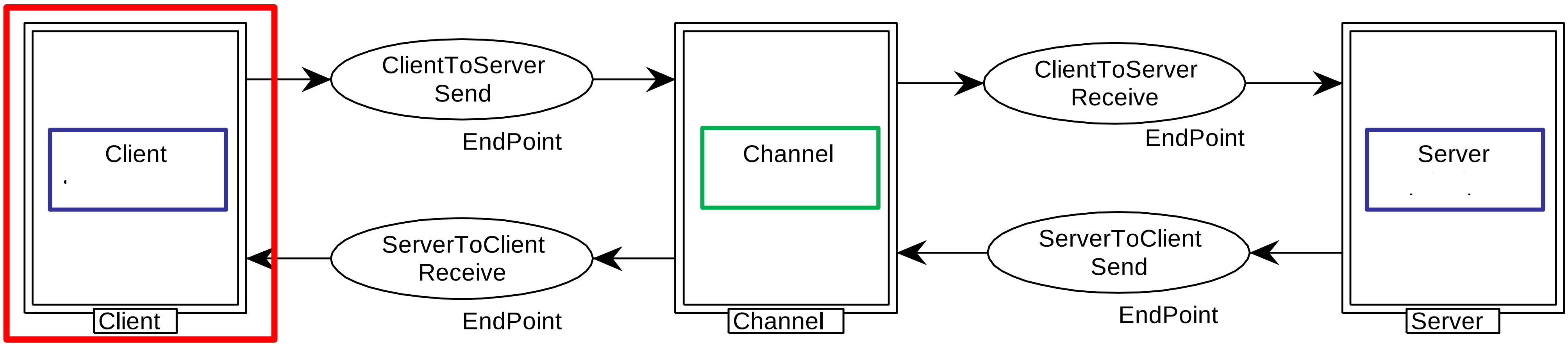
- **Provides a bi-directional and message-oriented service on top of the HTTP protocol:**



- **Three main phases: connection establishment, data transfer, and connection close.**

WebSocket: Protocol System

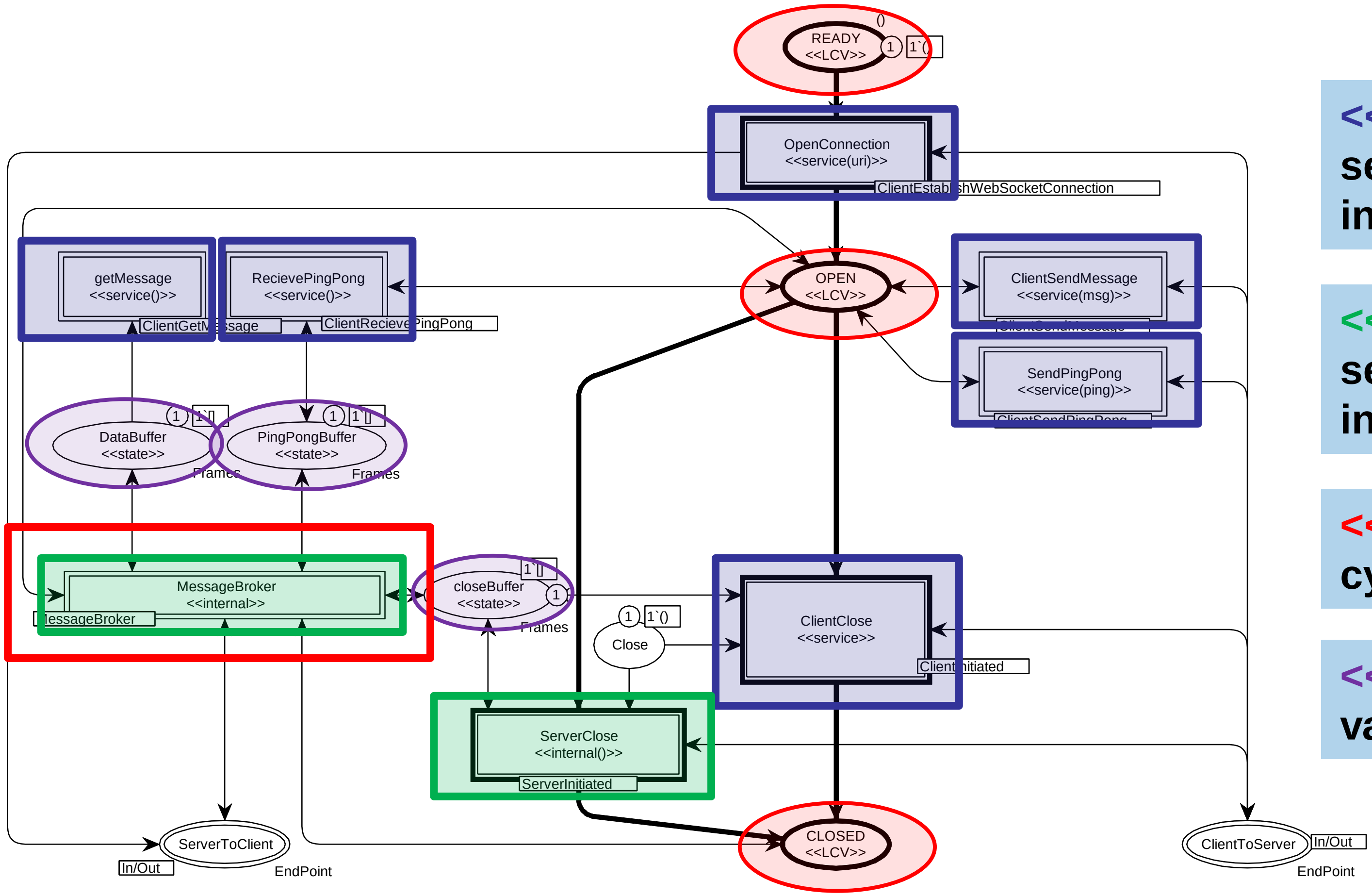
- **The complete CPN model consists of 19 modules, 136 places, and 84 transitions:**



- **The <<principal>> pragmatic is used on substitution transitions to designate principals.**
- **The <<channel>> pragmatic is used to designate channels connecting the principals.**

Client: Principal Level

- **Makes explicit the services provided and their allowed order of invocation (API life-cycle):**



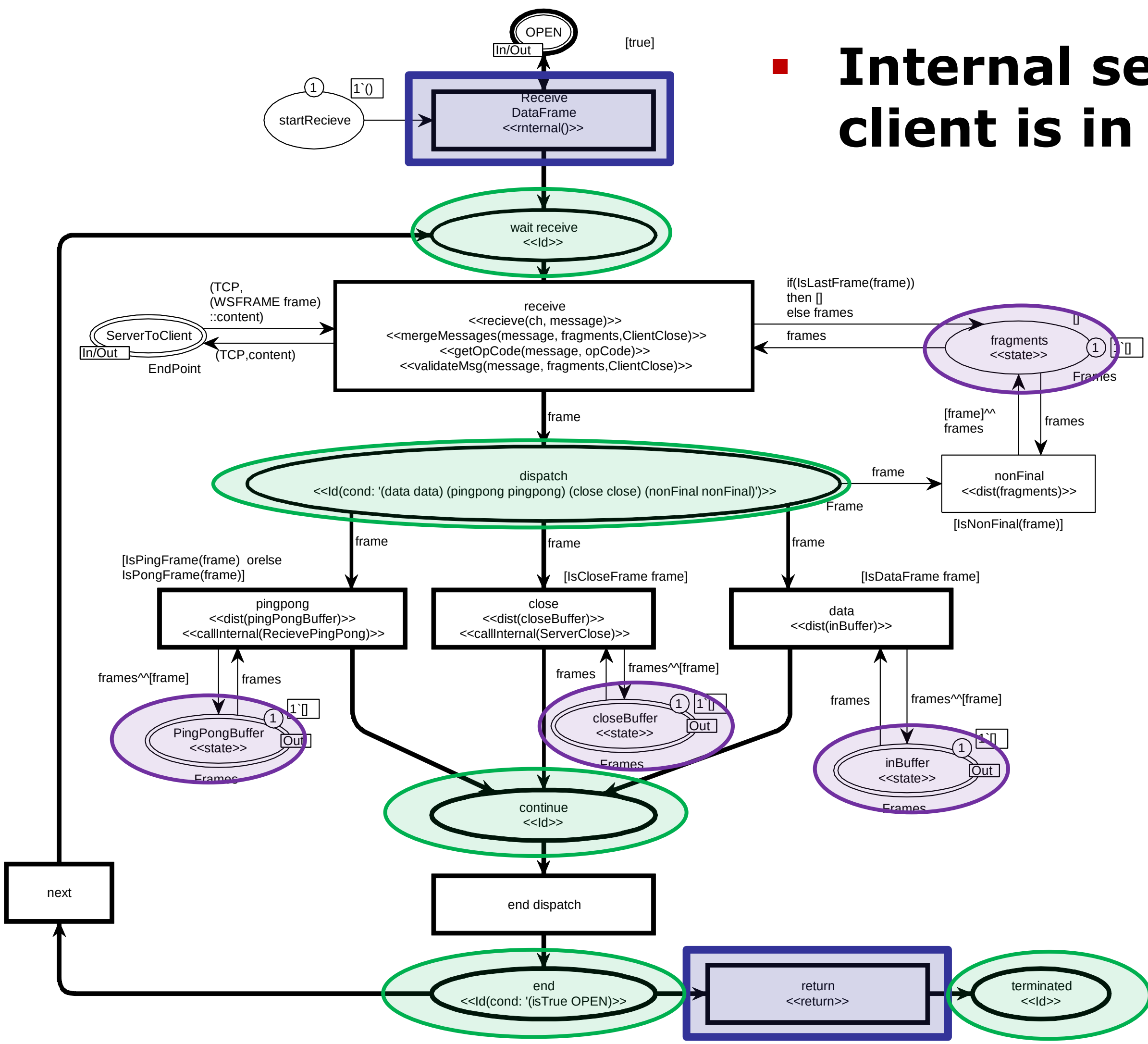
<<service>> specifies services that can be invoked externally.

<<internal>> specifies services that are invoked internally in the principal.

<<LCV>> specifies life-cycle variables for services.

<<state>> specifies state variables of the principal.

Client: MessageBroker Service



- **Internal service started when the client is in the OPEN state.**

Service entry point <<internal>>

Service-local state is specified using <<state>>

Control-flow locations is made explicit using <<ID>> pragmatic on places.

Service exit point <<return>>

WebSocket Verification

- **State space exploration prior to code generation used to model check basic connection properties:**

P1 All terminal states correspond to states in which the WebSocket connection has been properly closed.

```
fun isProperClosed : state -> bool
```

```
List.all isProperClosed (ListTerminalStates ())
```

P2 From any reachable state, it is always possible to reach a state in which the WebSocket connection has been properly closed.

```
HomeSpace (PredAllNodes isProperClosed)
```

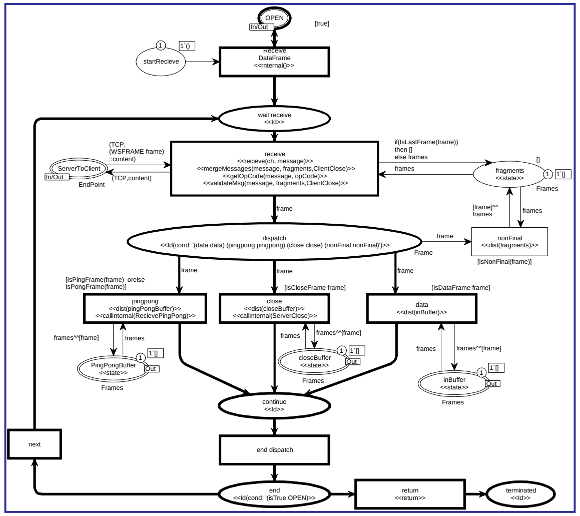
ClientM	ServerM	#Nodes	#Arcs	Time (secs)	#Terminal states
+	-	2,747	9,544	1	2
-	+	2,867	9,956	2	2
+	+	39,189	177,238	246	4

HØGSKOLEN I BERGEN
BERGEN UNIVERSITY COLLEGE

Automated Code Generation

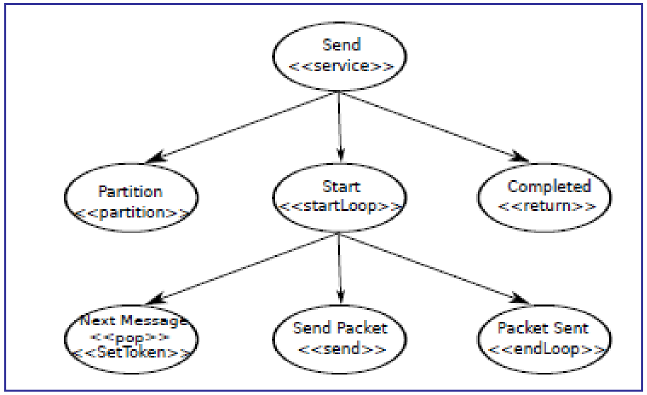
- **Template-based code generation consisting of three main steps:**

Step 1



Computing Derived Pragmatics

Step 2



Abstract Template Tree (ATT) Construction

Step 3

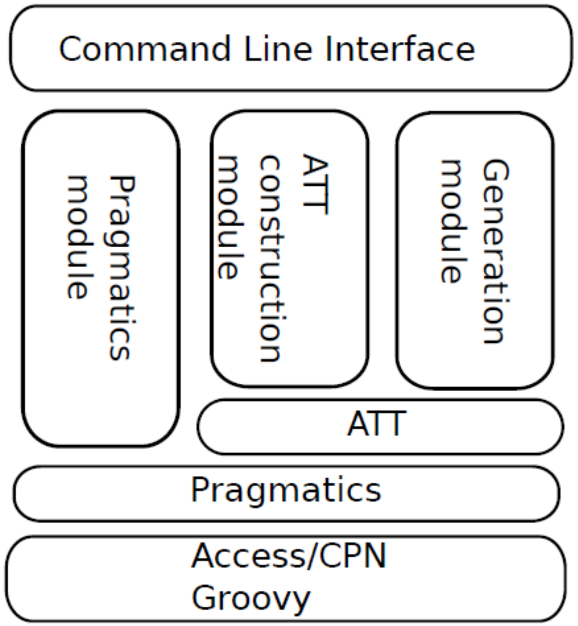
```
def getMessage(){
  /*vars: [__TOKEN__:, message:]*/
  def __TOKEN__
  def message
  //getMessage
  if(inBuffer != null && inBuffer.size() > 0){
    message = inBuffer.remove(0)
    byte[] bArr = new byte[message.payLoad.size()]
    for(int i = 0; i < bArr.length; i++){
      bArr[i] = message.payLoad.get(i)
    }
    if(message.opCode == 1){
      message = new String(bArr)
     }else if(message.opCode == 2) {
      message = bArr
     }
  }else{
    message = null
  }
  return message
}
```

Pragmatics binding and emitting code

HØGSKOLEN I BERGEN
BERGEN UNIVERSITY COLLEGE

PetriCode [www.petricode.org]

- **Command-line tool reading pragmatic-annotated CPN models created with CPN Tools:**



Pragmatic module: parses CPN models and computes derived pragmatics.

ATT construction module: performs block decomposition and constructs the ATT.

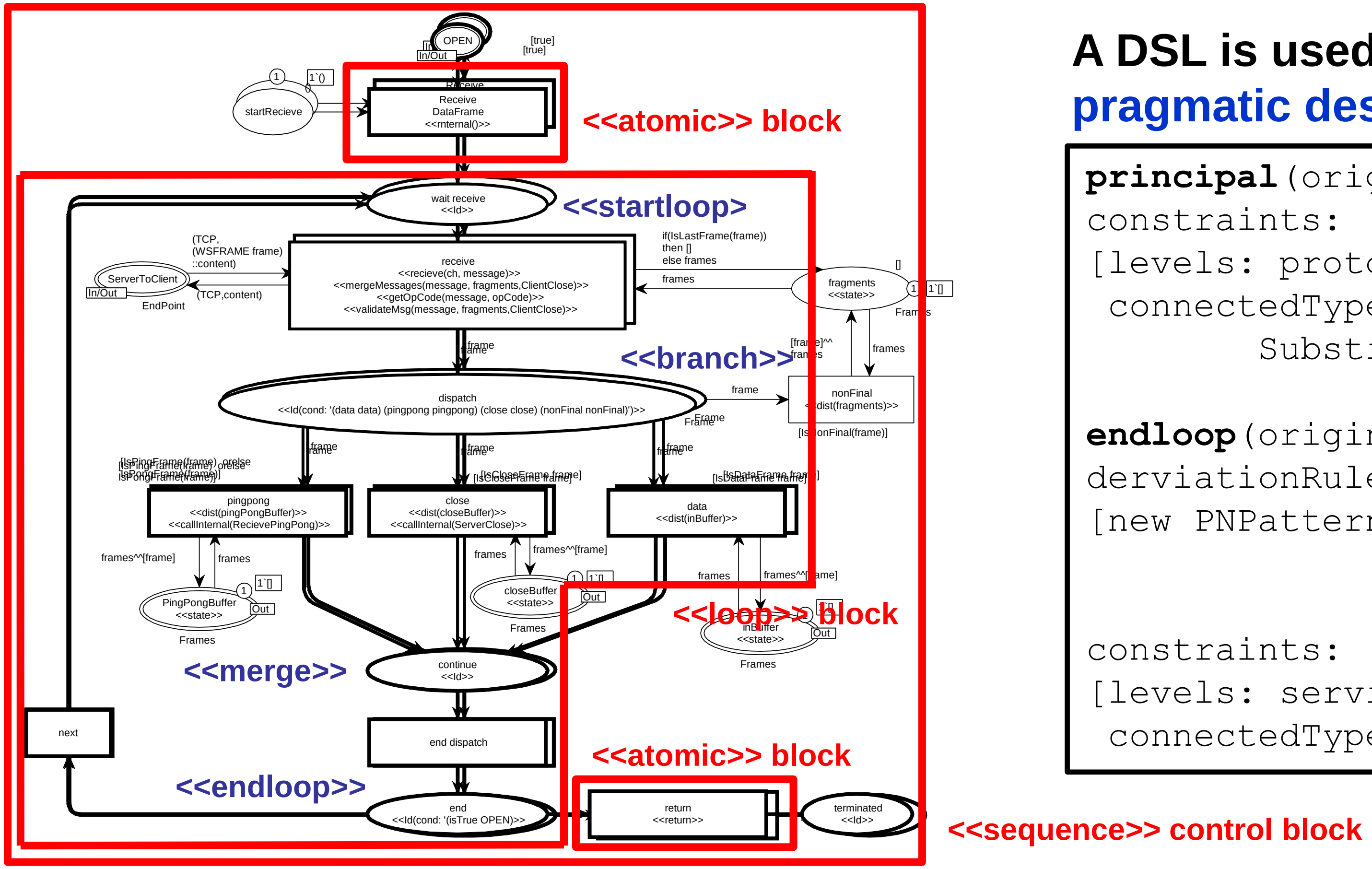
Code generation module: binds templates to pragmatics and generates source code via ATT traversal.

- **Implemented in Groovy and uses the Groovy template engine for code generation.**

1

Step 1: Derived Pragmatics

- **Derived pragmatics computed for control-flow constructs and for data (state) manipulation.**

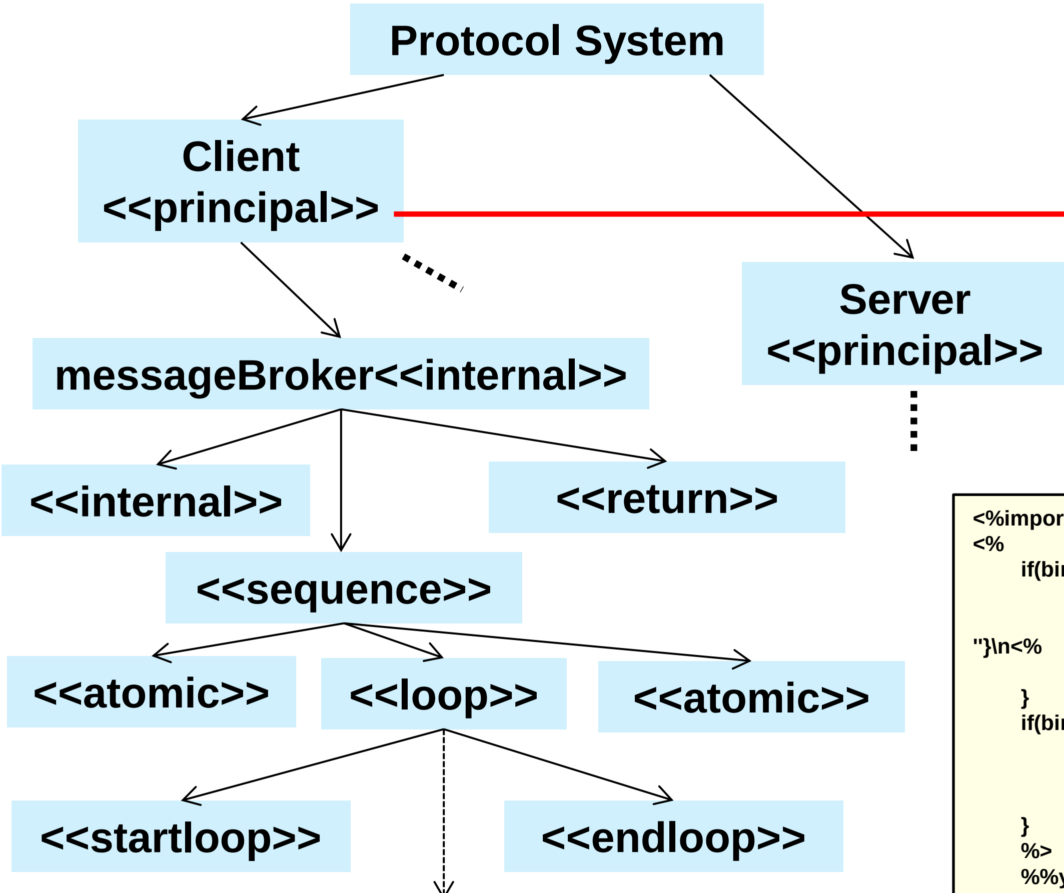


A DSL is used for specifying pragmatic descriptors.

```
principal(origin: explicit,
constraints:
[levels: protocol,
 connectedTypes:
        SubstitutionTransition])

endloop(origin: derived,
derviationRules:
[new PNPattern(pragmatics: [Id],
               minOutEdges: 2,
               backLinks: 1)],
constraints:
[levels: service,
 connectedTypes:Place])
```

Step 2: Abstract Template Tree

- **An intermediate syntax tree representation of the pragmatic-annotated CPN model:**



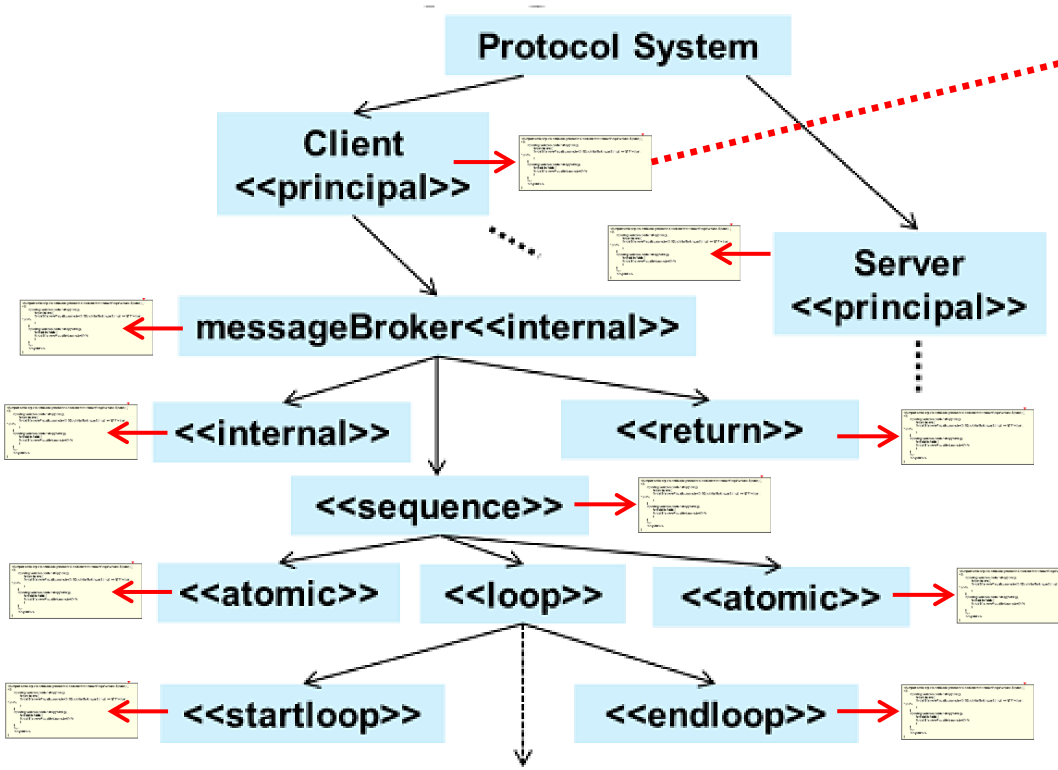
A DSL for template bindings and linkage to the target platform.

```
classTemplate(
pragmatic: 'principal',
template: './groovy/mainClass.tmpl',
isContainer: true)
endloop(
pragmatic: 'endloop',
template: './groovy/endLoop.tmpl')
```

```
<%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%%
}
```

Step 3: Emitting Code

- **Traversal of the ATT, invocation of code generation templates, and code stitching:**



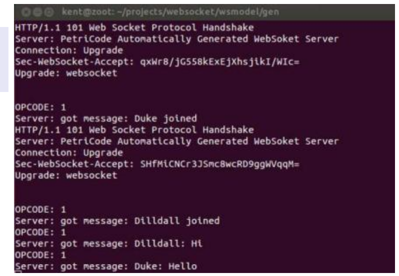

```
def getMessage(){
  /*vars: [__TOKEN__:, message:]*/
  def __TOKEN__
  def message
  //getMessage
  if(inBuffer != null && inBuffer.size() > 0){
    message = inBuffer.remove(0)
    byte[] bArr = new byte[message.payLoad.size()]
    for(int i = 0; i < bArr.length; i++){
      bArr[i] = message.payLoad.get(i)
    }
    if(message.opCode == 1){
      message = new String(bArr)
    }else if(message.opCode == 2) {
      message = bArr
    }
  }else{
    message = null
  }
  return message
}
```

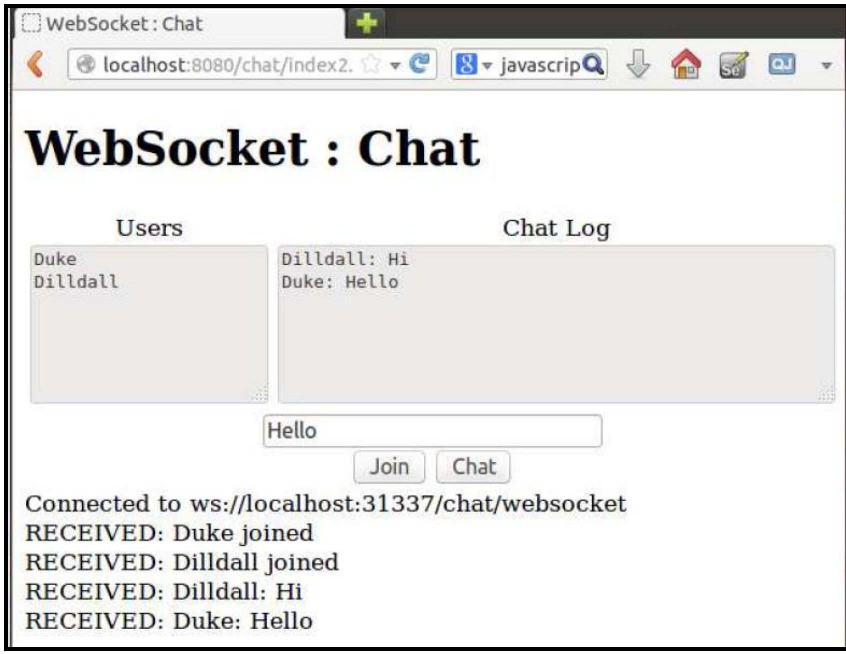
```
def SendPingPong(ping){ ... }
def ClientClose(){ ... }
def getMessage(){ ... }
}
```

Chat Application*

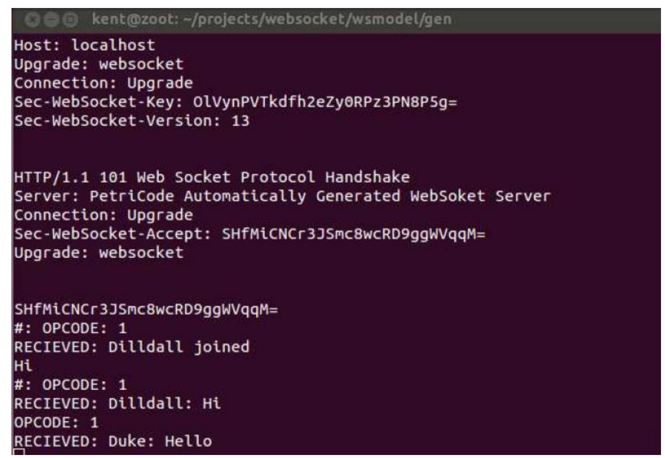
- **WebSocket tutorial example provided with the Java EE 7 GlassFish Application Server:**

Chat Server [CPN WebSocket model]





Web-based Chat Client [WebSocket Browser]



Chat Client [CPN WebSocket model]

Autobahn Testsuite [autobahn.ws/testsuite/]

- **Test-suite used by several industrial WebSocket implementation projects** (Google Chrome, Apache Tomcat,..).
- **Errors encountered with the generated code:**
  - **One global logical error related to the handling of fragmented messages (CPN model change).**
  - **Several local errors in the code-generation templates were encountered (template change).**

Tests	Server Passed	Client Passed
1. Framing (text and binary messages)	16/16	16/16
2. Pings/Pongs	11/11	11/11
3. Reserved bits	7/7	7/7
4. Opcodes	10/10	10/10
5. Fragmentation	20/20	20/20
6. UTF-8 handling	137/141	137/141
7. Close handling	38/38	38/38
9. Limits/Performance	54/54	54/54
10. Auto-Fragmentation	1/1	1/1

Autobahn

http://t.k1s.org/wsreport/

Conclusions

- **An approach allowing CPN simulation and verification models to be used for code generation:**
  - Pragmatic annotations and enforcing modelling structure.
  - Binding pragmatics to code generation templates.
- **Implemented in the PetriCode tool to allow for practical applications and initial evaluation.**
- **The approach has been evaluated via application to the IETF WebSocket Protocol:**
  - State space verification was feasible for verifying some basic connection properties prior to code generation.
  - The implementation was tested for interoperability against a comprehensive benchmark test-suite with promising results.
  - A proof-of-concept on the scalability and feasibility of the approach for the implementation of real protocols.

Thank you for your attention!