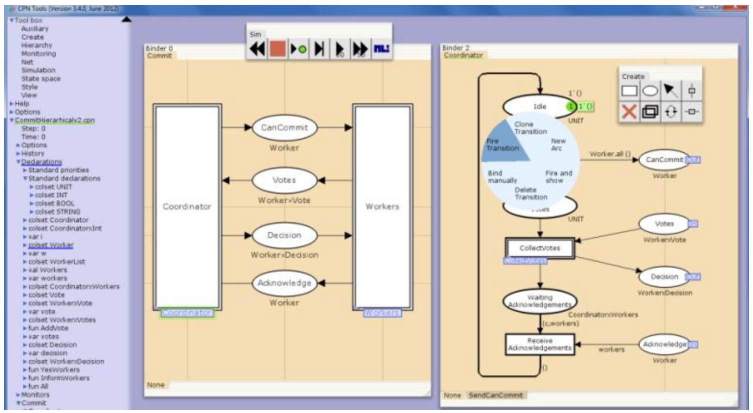
A Pragmatic Approach for Transforming Coloured Petri Net Models into Code

- A Case Study of the IETF WebSocket Protocol*



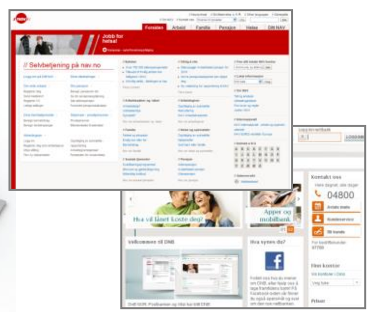
Kent I.F. Simonsen[1,2] and Lars M. Kristensen[1]

[1]Bergen University College [2]Technical University of Denmark

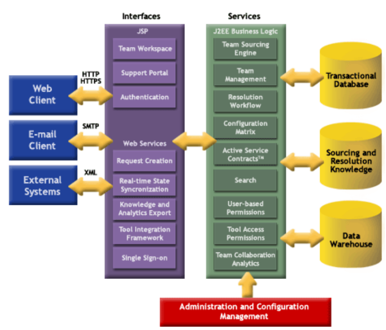
based on:** ***Implementing the WebSocket Protocol based on Formal Modelling and Automated Code Generation. **To appear in at 14th Intl. IFIP Conference on Distributed Applications and Interoperable Systems (DAIS), June 2014.**

Concurrent Systems

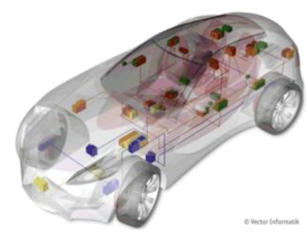
- **The vast majority of IT systems today can be characterised as concurrent software systems:**
  - **Structured as a collection of concurrently executing software components and applications.**
  - **Operation relies inherently on communication, synchronisation, and resource sharing.**

Internet and Web-based applications, protocols



Multi-core platforms and multi-threaded software




Embedded systems and networked control systems

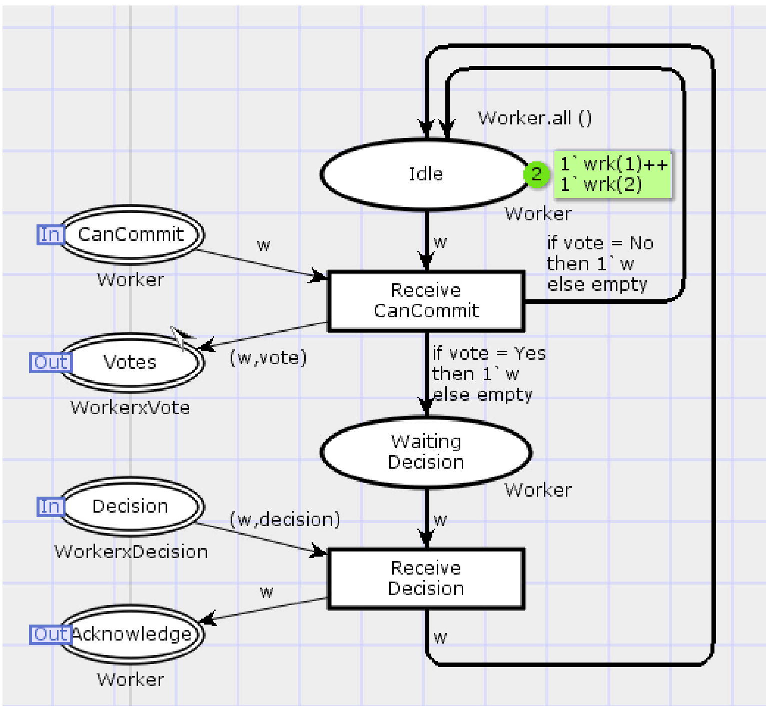
Concurrent Systems

- **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)

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



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

Programming language
- data types
- data manipulation
- compact modelling
- parameterisable models

High-Level Petri Net

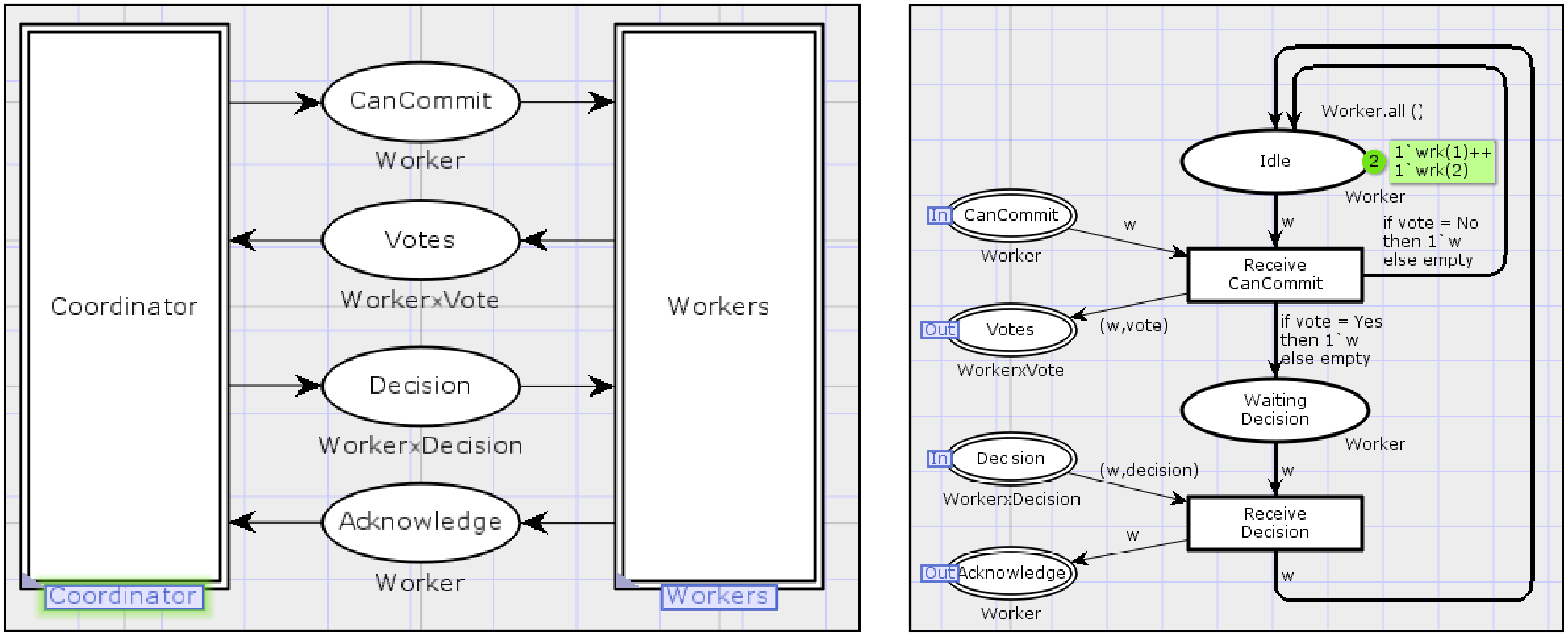
- Supported by CPN Tools [www.cpntools.org]

Application of CPNs

- **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 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:**



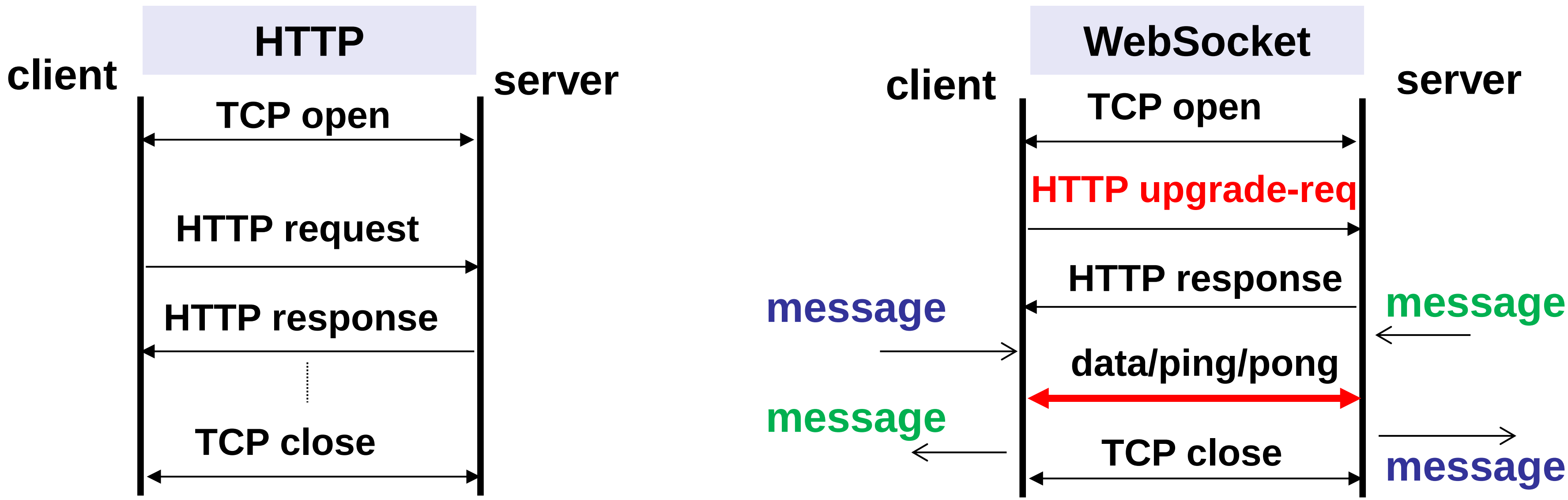
- **Conclusion:** **some additional syntactical constraints and/or annotations are required.**

Main 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 invoke and rely on underlying libraries.
3. **Model checking and property verification:**
   - Code generation capability should not introduce complexity problems for the verification of the CPN models.
4. **Readability of the generated code:**
   - Enable code review of the automatically generated code.
   - Enable performance enhancements (if required).
5. **Scalability:**
   - Applicable to industrial-sized communication protocols.

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.**

Overview of Approach

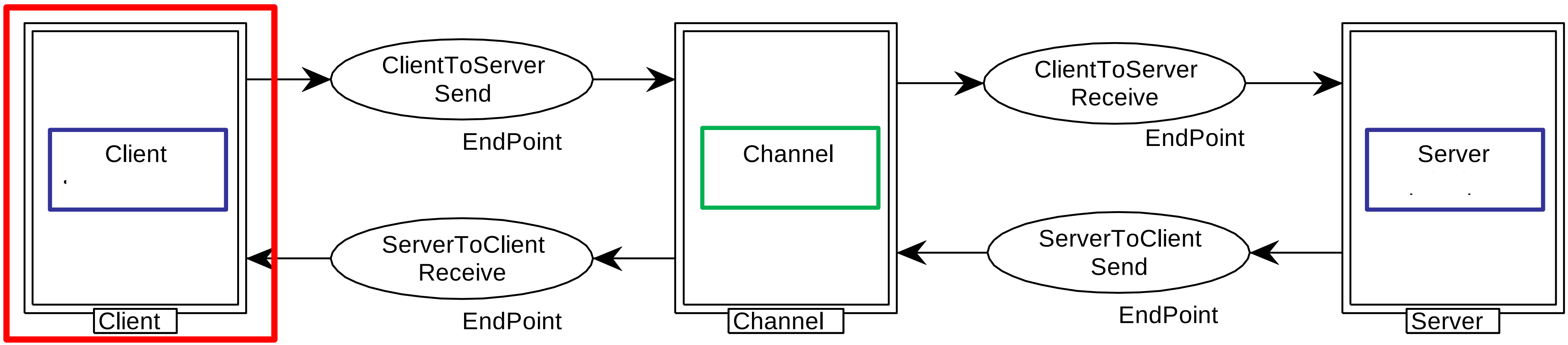
- Modelling structure requiring the CPN model to be organised into three levels:
  1. Protocol system level specifying the protocol principals and the communication channels between them.
  2. Principal level reflecting the life-cycle and services provided by each principal in the protocol system.
  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.

WebSocket: Protocol System

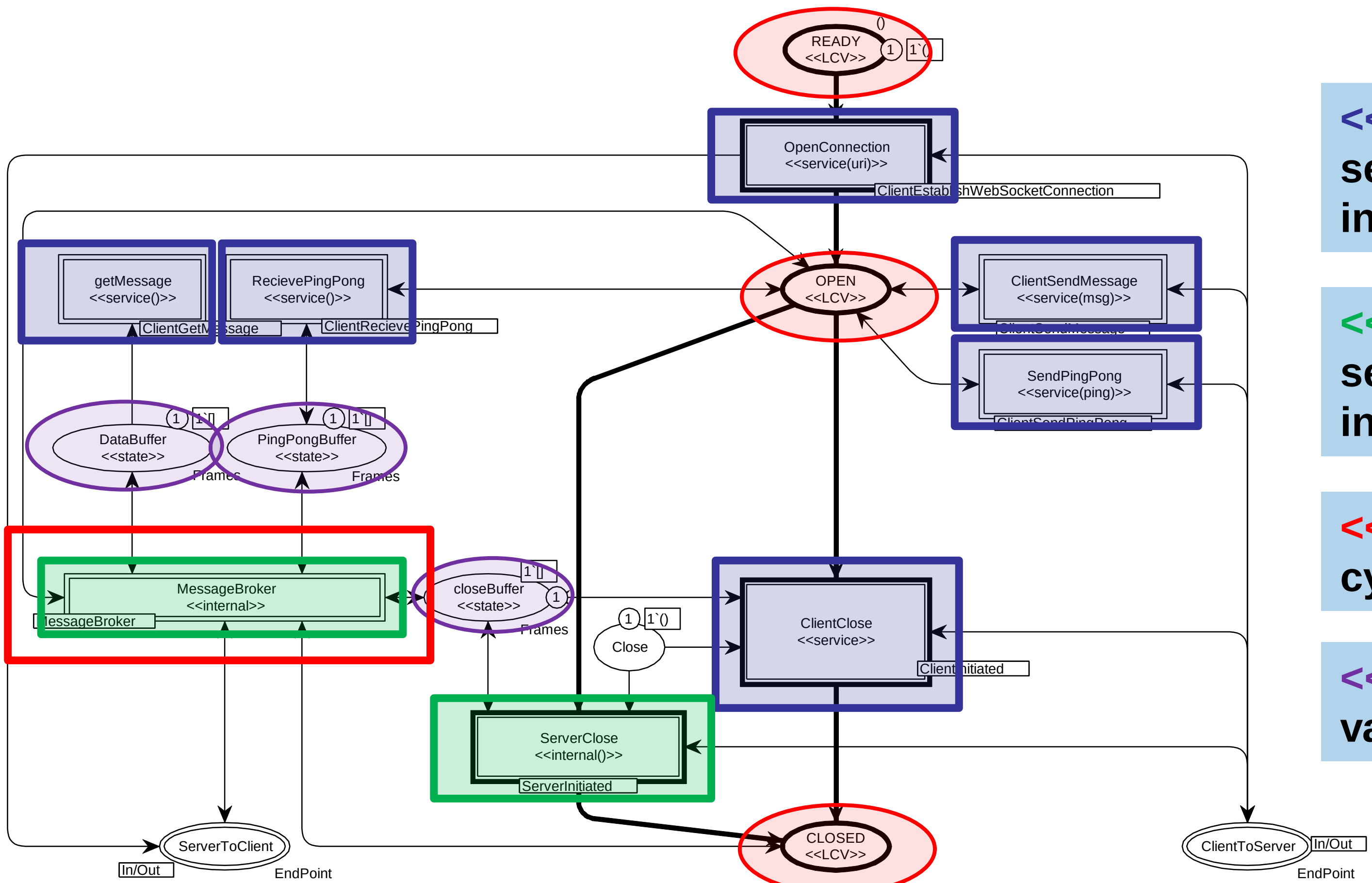
- The complete CPN model consists of 19 modules, 136 places, and 84 transitions:



- 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):**



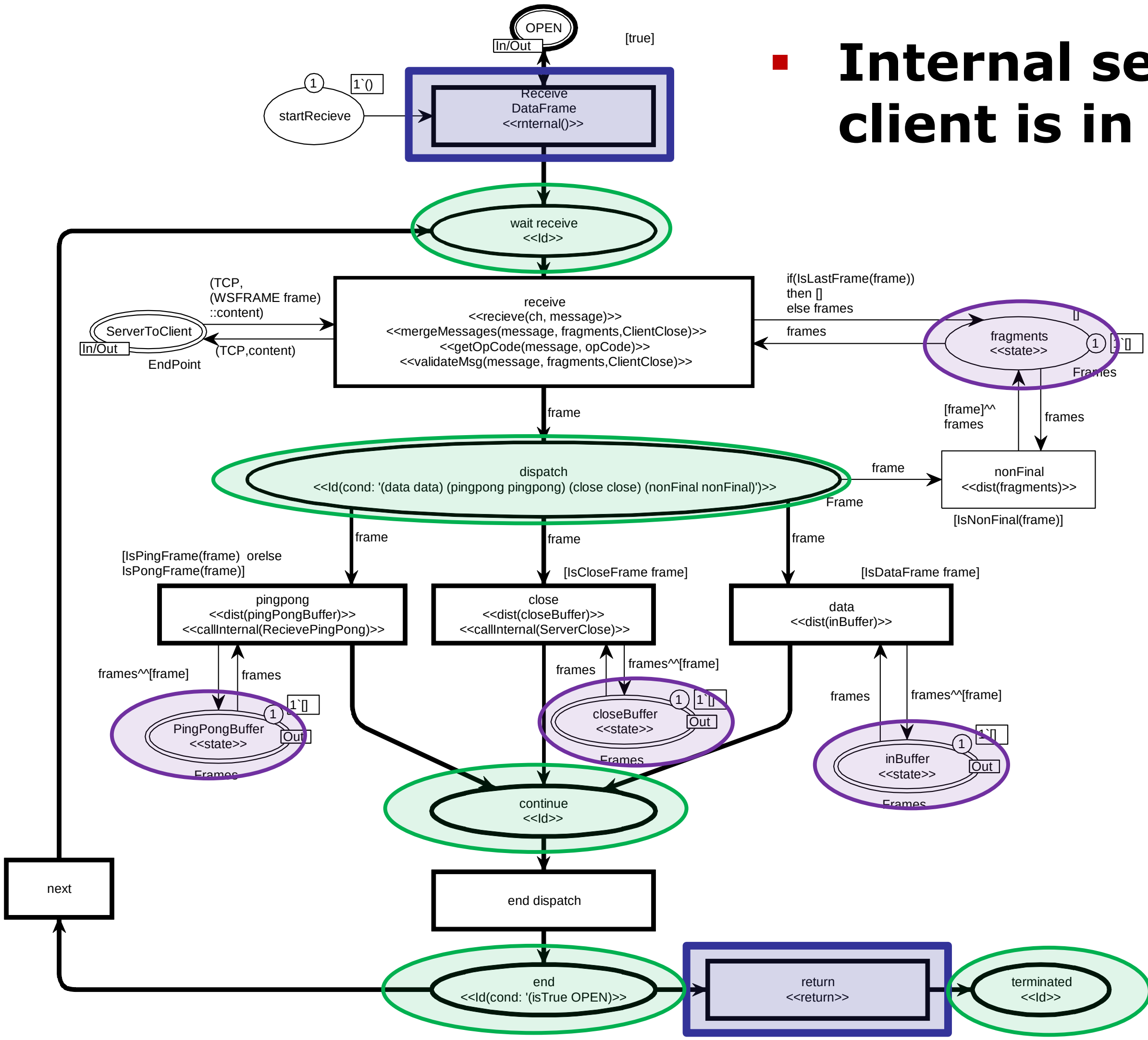
<<service>> specifies services that can be invoked externally.

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

<<LCV>> specifies life-cycle 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:**

P0 From the initial state it is possible to reach states in which the WebSocket connection has been opened.

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

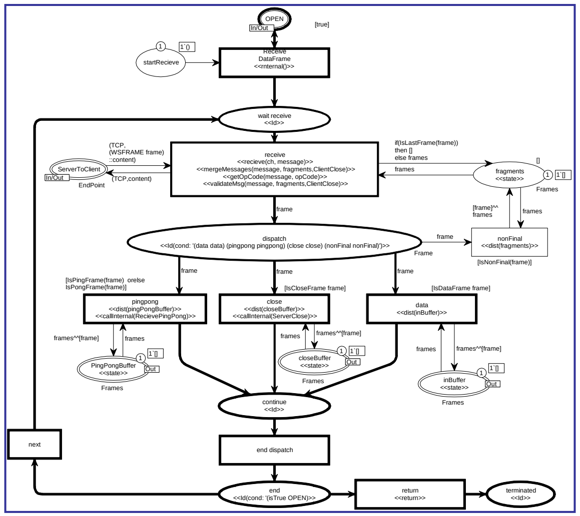
P2 From any reachable state, it is always possible to reach a state in which the WebSocket connection has been properly closed.

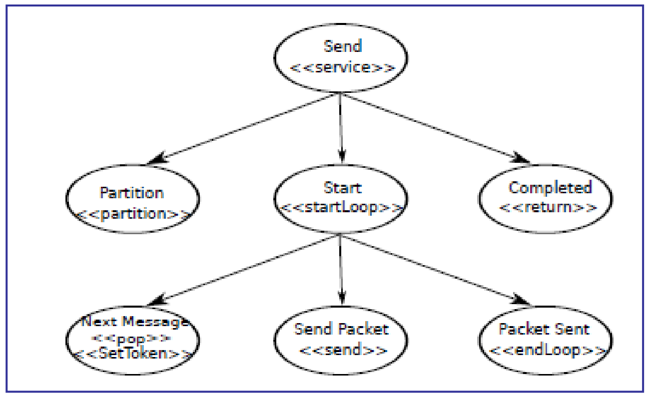
ClientM	ServerM	#Nodes	#Arcs	Time (secs)	#Terminal states
+	-	2,747	9,544	1	2
-	+	2,867	9,956	2	2
+	+	39,189	177,238	246	4

Automated Code Generation

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

Step 1	Step 2	Step 3
Computing Derived Pragmatics	Abstract Template Tree (ATT) Construction	Pragmatics binding and emitting code

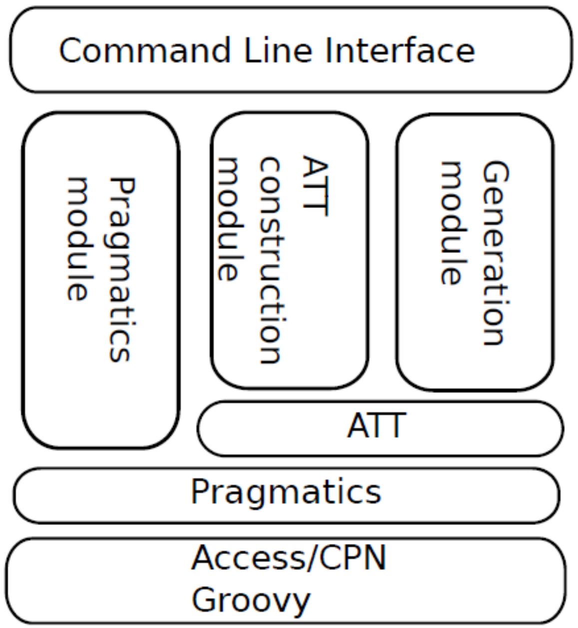





```
def getMessage(){
  /*vars: [__TOKEN__:, message:]*/
  def __TOKEN__
  def message
  //getMessage
  if(inBuffer != null && inBuffer.size() > 0){
    message = inBuffer.remove(0)
    byte[] bArr = new byte[message.payLoad.size()]
    for(int i = 0; i < bArr.length; i++){
      bArr[i] = message.payLoad.get(i)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

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.

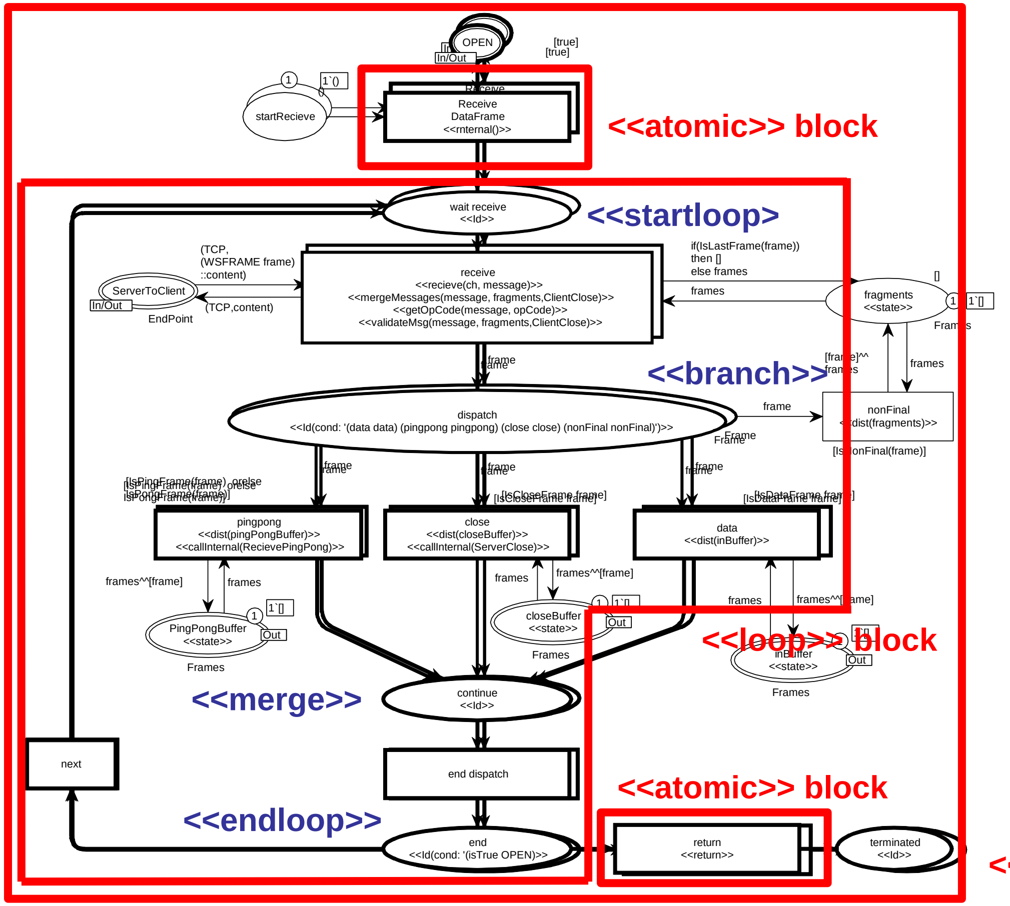
ATT construction module: performs block decomposition and constructs the ATT.

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.**

Step 1: Derived Pragmatics

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



<<sequence>> control block

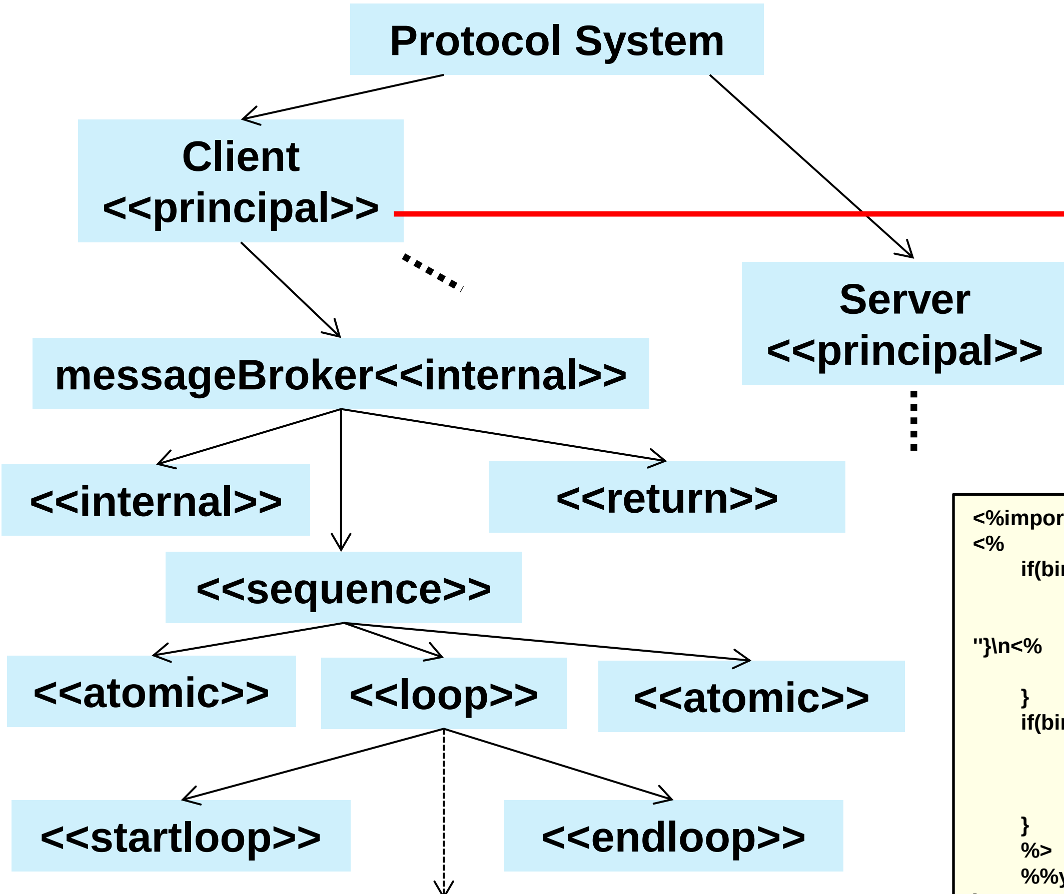
A DSL is used for specifying **pragmatic descriptors**.

```
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:**



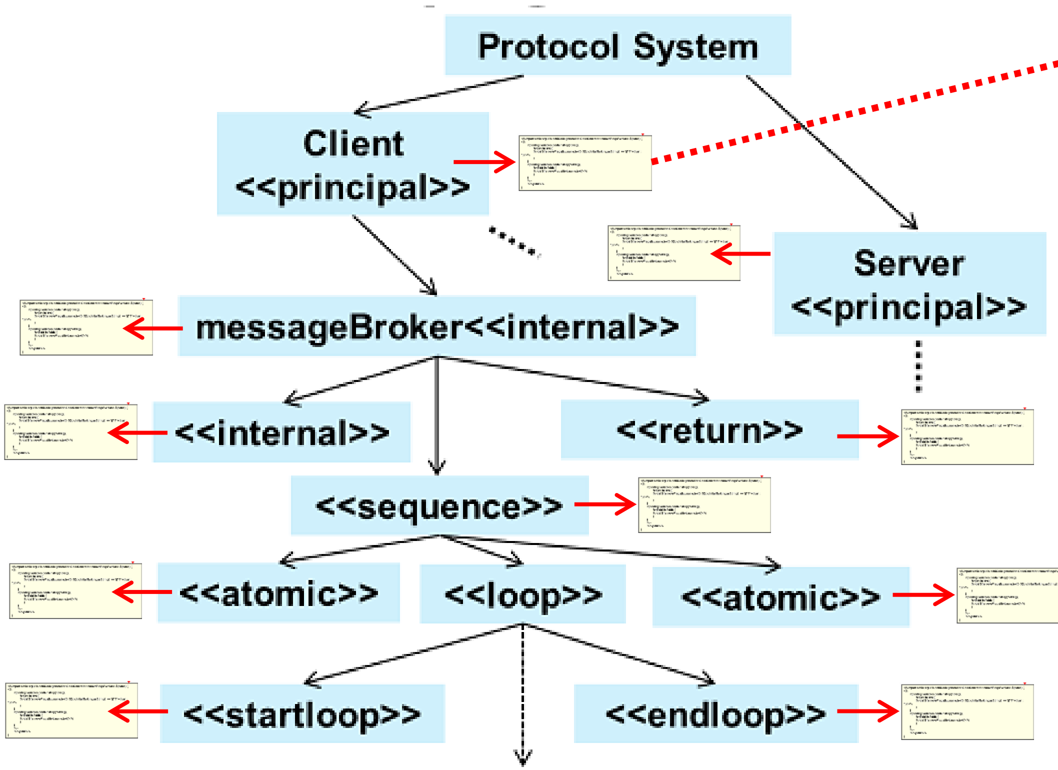
A DSL for template bindings and linkage to the target platform.

```
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:**



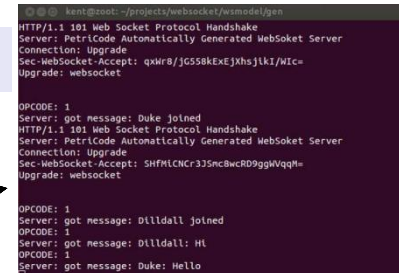

```
def getMessage(){
  /*vars: [__TOKEN__:, message:]*/
  def __TOKEN__
  def message
  //getMessage
  if(inBuffer != null && inBuffer.size() > 0){
    message = inBuffer.remove(0)
    byte[] bArr = new byte[message.payLoad.size()]
    for(int i = 0; i < bArr.length; i++){
      bArr[i] = message.payLoad.get(i)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

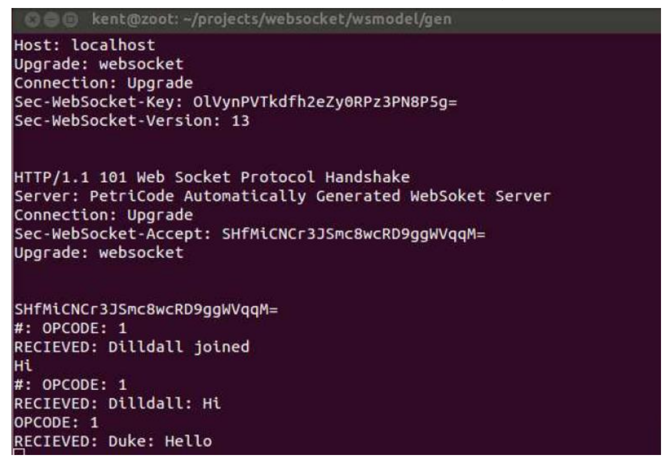
Chat Application

- **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/]

Autobahn

- **Test-suite used by several industrial WebSocket implementation projects** (Google Chrome, Apache Tomcat,..).
- **Errors encountered with the generated code:**
  - **One protocol 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	48/54
10. Auto-Fragmentation	1/1	1/1

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

Conclusions

- **An approach enabling CPN models to be used for code generation of protocol software:**
  - Pragmatic annotations and enforcing modelling structure.
  - Binding of pragmatics to code generation templates.
- **Implemented in the PetriCode tool to allow for practical applications and initial evaluation.**
- **Scalability of the approach has been evaluated via application to the IETF WebSocket Protocol:**
  - State space-based verification was feasible for verifying 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!