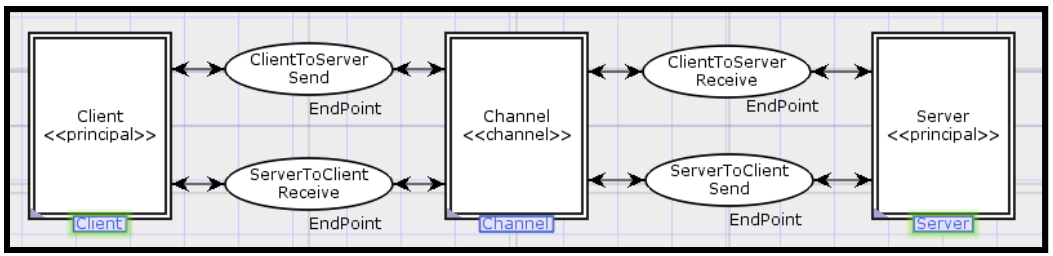
Pragmatics Annotated Coloured Petri Nets for Protocol Software Generation and Verification



Kent I.F. Simonsen[1, 2] **Lars M. Kristensen[1]** **Ekkart Kindler[2]**

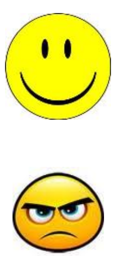
Department of Computing[1]
Bergen University College
NORWAY

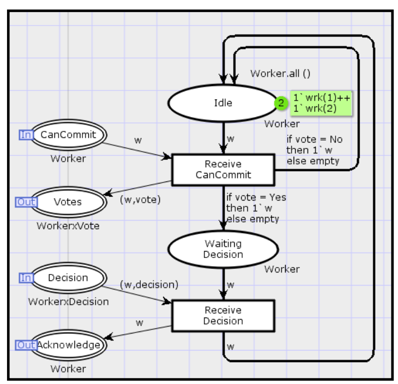
DTU Compute[2]
Technical University of Denmark
DENMARK

HØGSKOLEN I BERGEN
BERGEN UNIVERSITY COLLEGE

Motivation

- **Coloured Petri Nets (CPNs) have been widely used for modelling and verification of protocols:**
  - Application layer protocols: IOTP, SIP, WAP, …
  - Transport layer protocols: TCP, DCCP, SCTP, …
  - Routing layer protocols: DYMO, AODV, ERDP, …
- **Desirable to use the constructed CPN models for automated generation of protocol software:**





automated code generation

Verification **Modelling** **Protocol software (code)**

Background

- **Our earlier work: Pragmatics Annotated CPNs for automated code generation [FMFA'13]:**
  - The PetriCode tool implementation [WS-FMDS'13].
  - Platform independence and code integration [PNSE'14].
  - Readability of the generated code for review [PNSE'14].
  - Scalability and interoperability for industrial-sized protocols [DAIS'14].

- **Main contributions of the present paper:**
  - Formal definition of Pragmatics Annotated CPNs (PA-CPNs).
  - Methodology for protocol software development with PA-CPNs.
  - Efficient verification of PA-CPNs by means of service testers and the sweep-line state space exploration method.

Pragmatic Annotated Coloured Petri Nets

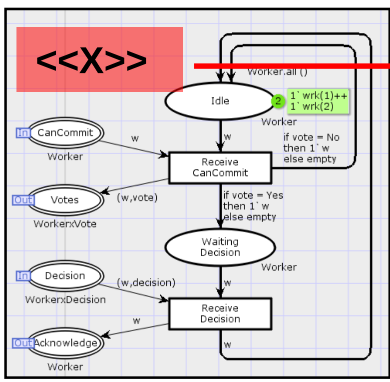
Pragmatic Annotated CPNs

- A restricted form (subclass) of Coloured Petri Nets organised into three levels of modules:
  1. Protocol system level module specifying the principals (protocol entities) and the channels between them.
  2. Principal level modules specifying the life-cycle and services provided by each principal in the system.
  3. Service level modules specifying the detailed behaviour of the services provided by each principal.

- PA-CPN model elements can be annotated with pragmatics used to direct the code generation:
  - Syntactical annotations representing concepts from the domain of communication protocols.
  - Pragmatics are by convention written inside << X >>.

Pragmatics - Code Generation

- **Extends the CPN modelling language with domain-specific specific elements.**
- **Makes implicit knowledge of the modeller explicit for code generation purposes.**
- **No semantic meaning or impact on the execution of the CPN model (syntactical construct).**



CPN model

```
<%import static
org.k1s.petriCode.generation.CodeGenerator.removePr
ags%>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%%
}
```

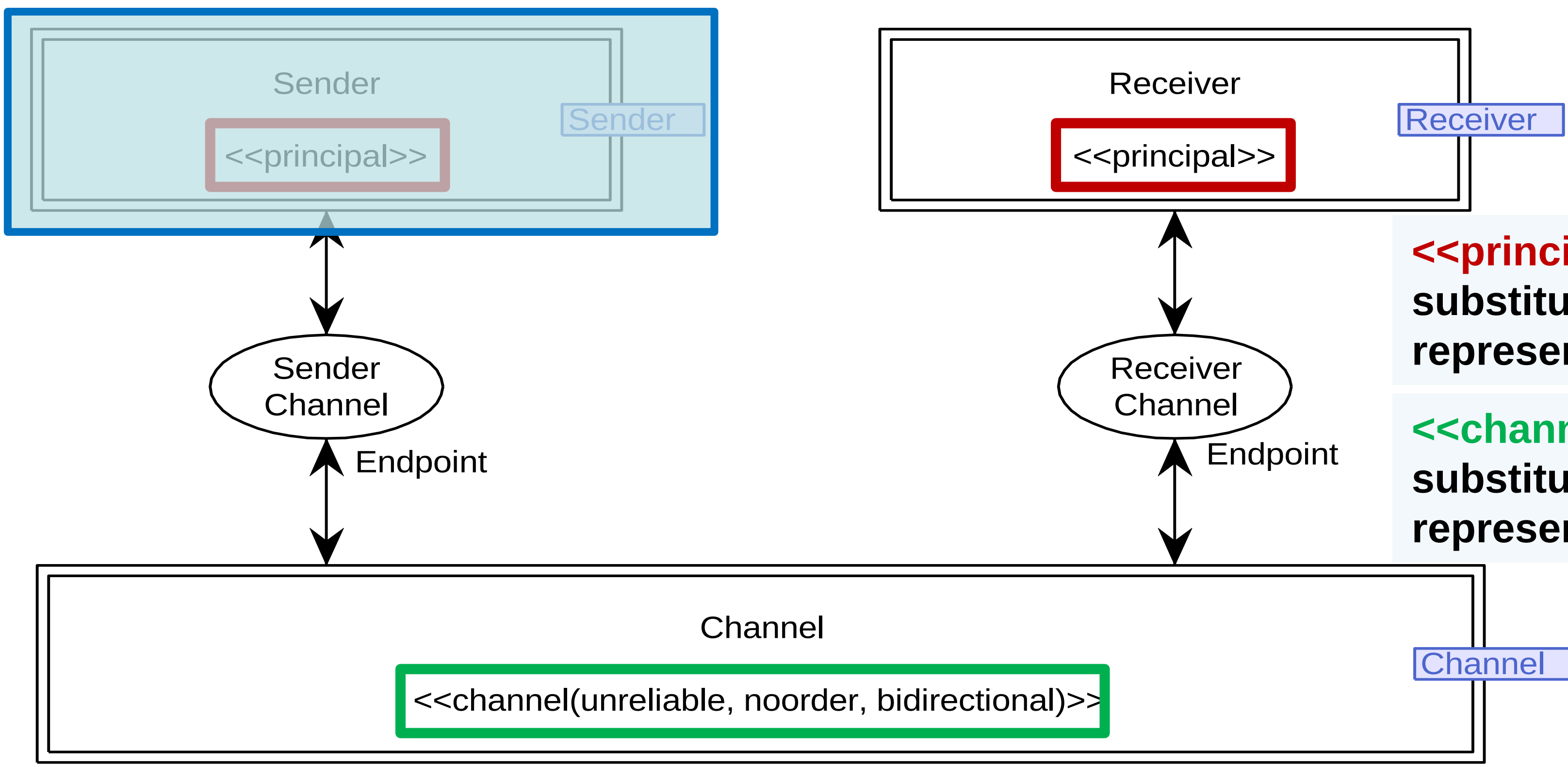
Code generation template

```
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
}
```

Code

Protocol System Level Module

- **Specifies the protocol principals and the channels used for communication:**

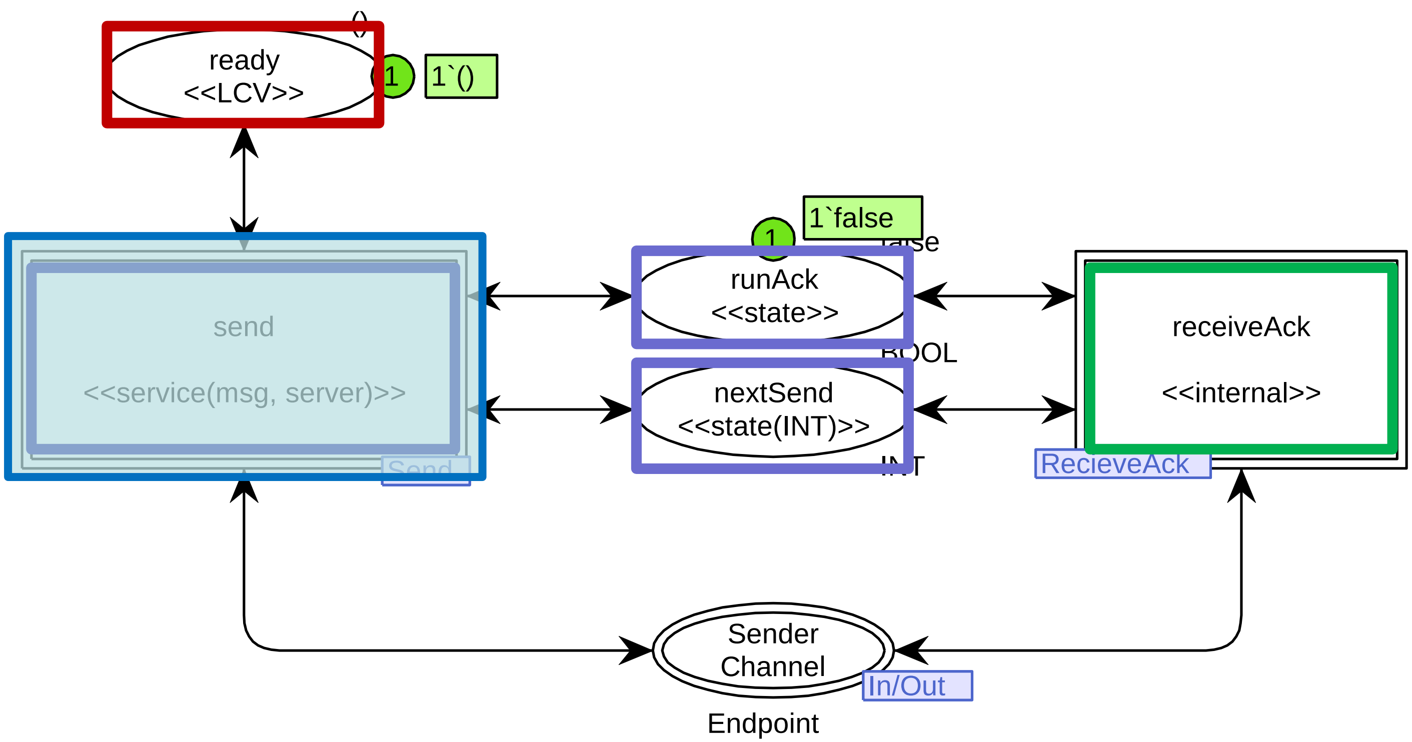


<<principal>> specifies substitution transitions representing principals

<<channel>> specifies substitution transitions representing channels

Principal Level Modules

- **Models the services, internal state, and the life-cycle of each principal:**



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

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

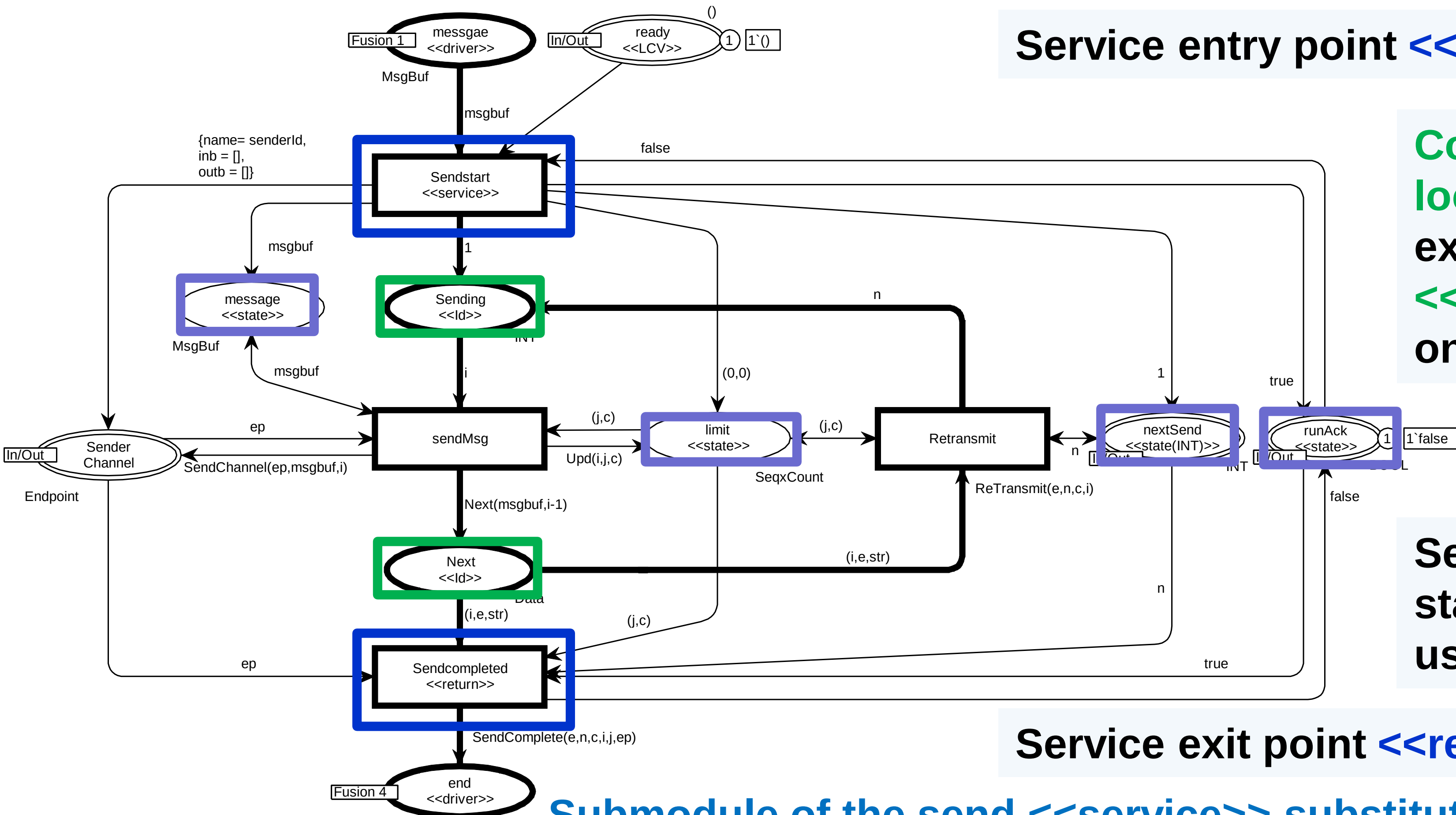
<<LCV>> specifies life-cycle for external services.

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

Submodule of the Sender <<principal>> substitution transition

Service Level Modules

- **Control-flow oriented modelling of the detailed behaviour of each service:**



Service entry point <<service>>

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

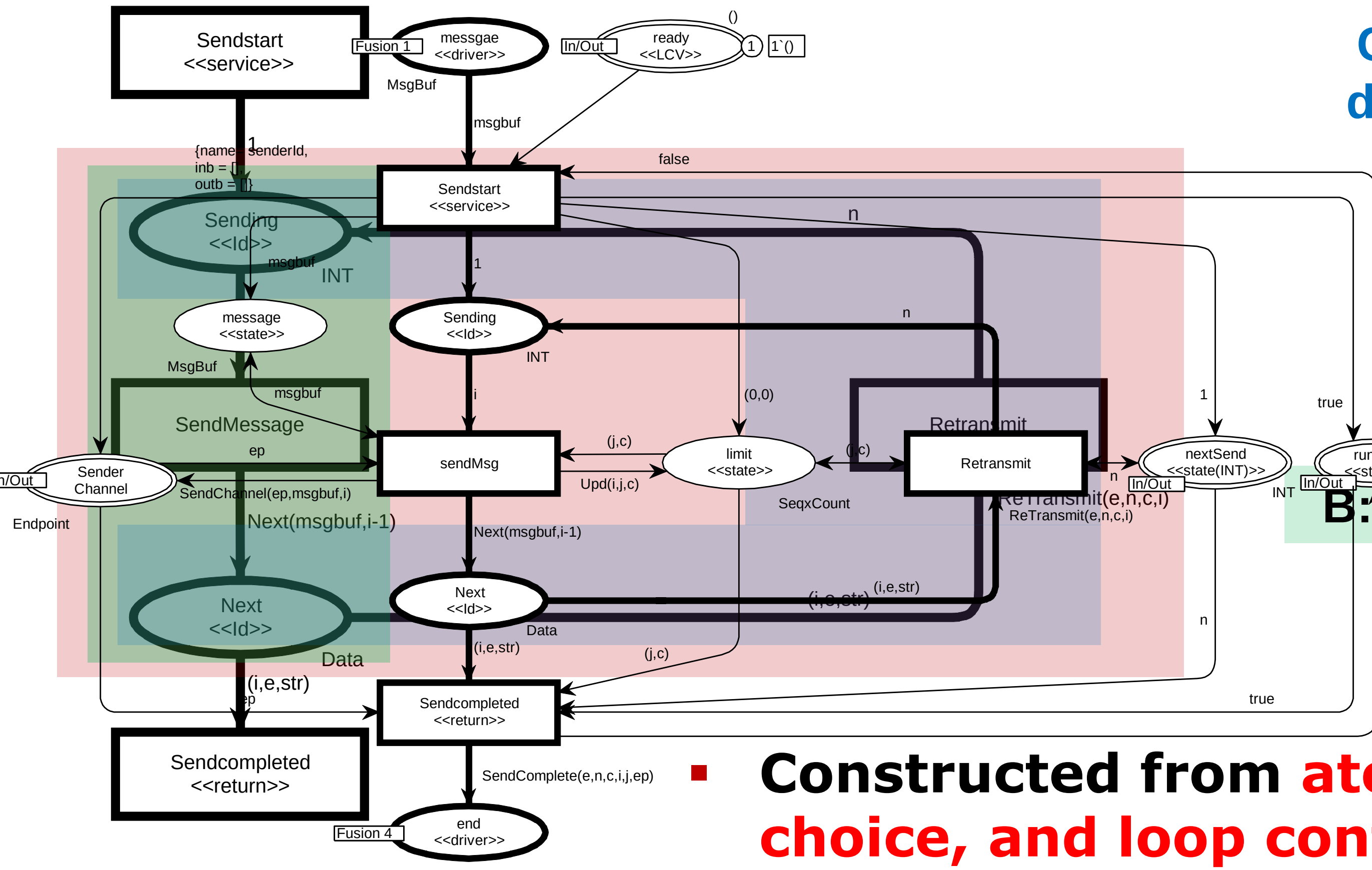
Service-local state is specified using <<state>>

Service exit point <<return>>

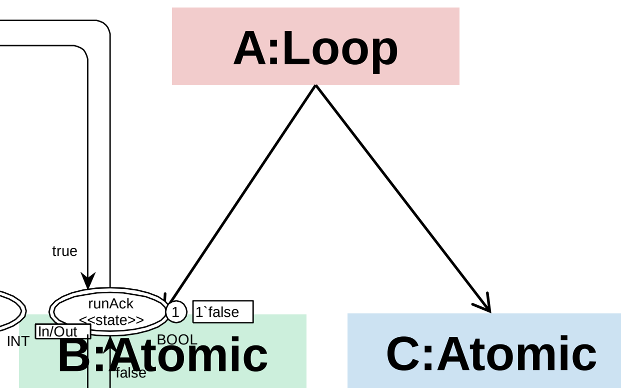
Submodule of the send <<service>> substitution transition

Block Tree Decomposition

- **The underlying control-flow net of a service level module must be block tree decomposable:**



Control-flow block decomposition tree



- **Constructed from atomic, sequence, choice, and loop control-flow constructs.**

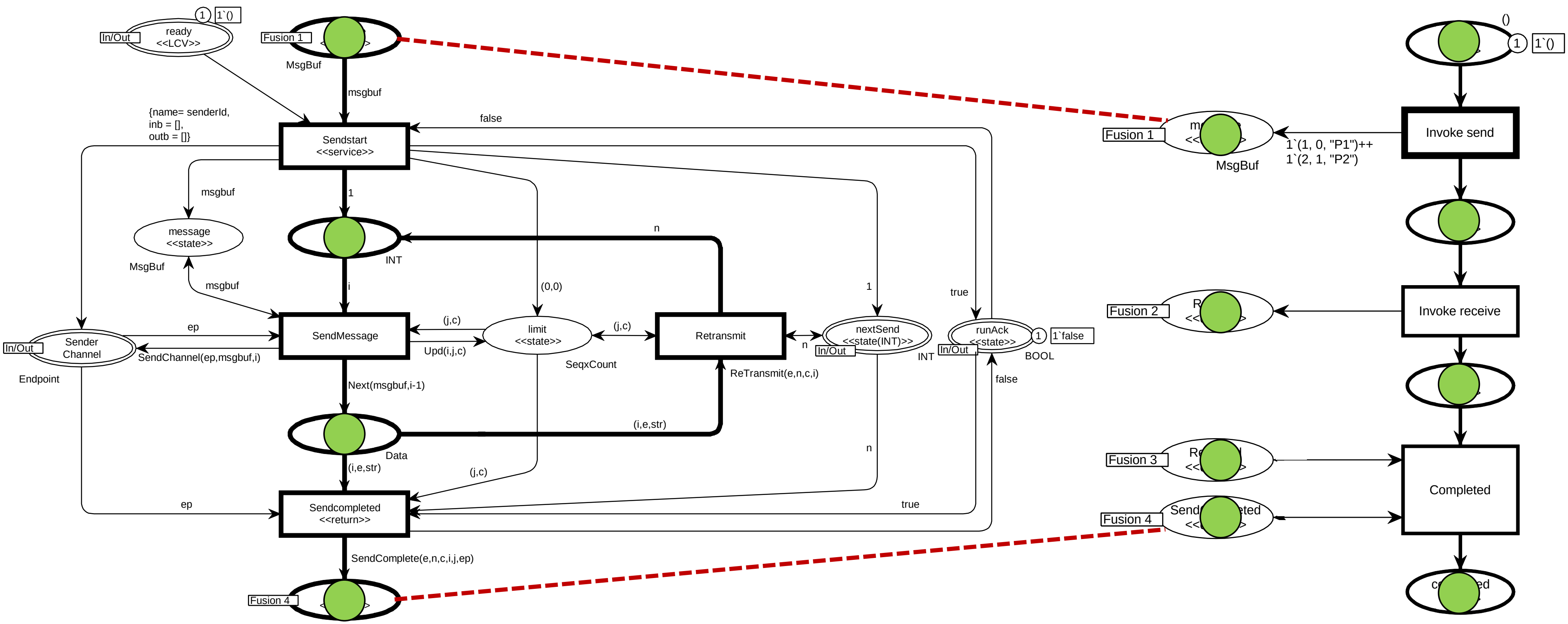
HØGSKOLEN I BERGEN
BERGEN UNIVERSITY COLLEGE

Service Testers and Verification

Service Tester Modules

- **The execution of PA-CPNs can be controlled by connecting service testers to the service modules:**

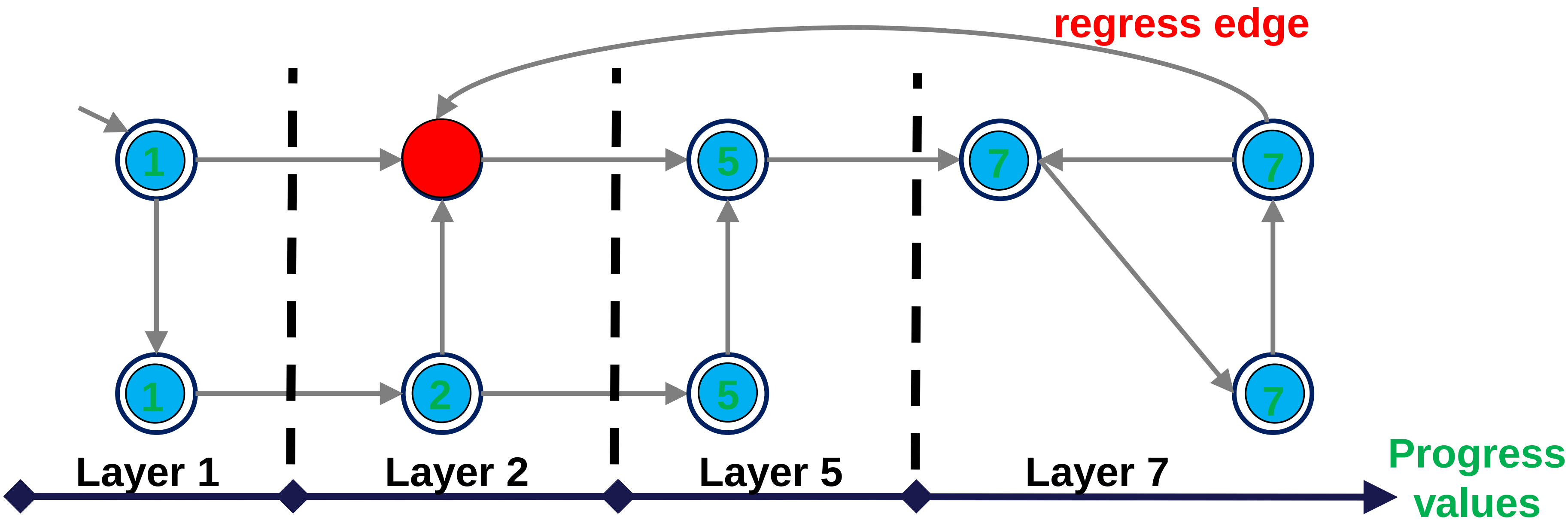
Service level module: send <<service>> Service tester module



The Sweep-Line Method

[Jensen, Kristensen, and Mailund (TCS'12)]

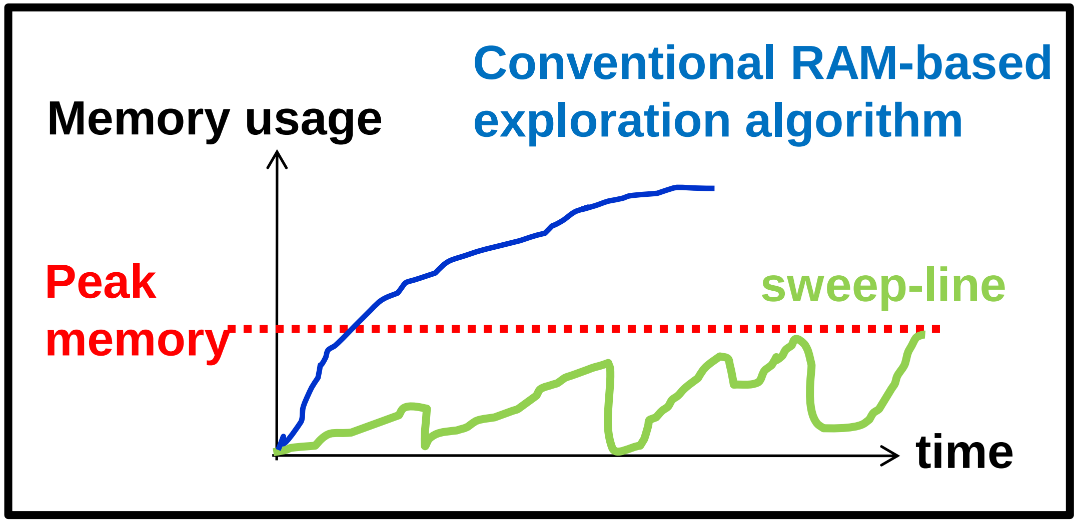
- A method to reduce peak memory usage in explicit state space exploration by exploiting a notion of progress:



- Conduct a least-progress-first layer-by-layer exploration.
- Delete states when a complete layer has been processed.
- Detect regress edges on-the-fly and mark as persistent.

The Sweep-Line Method

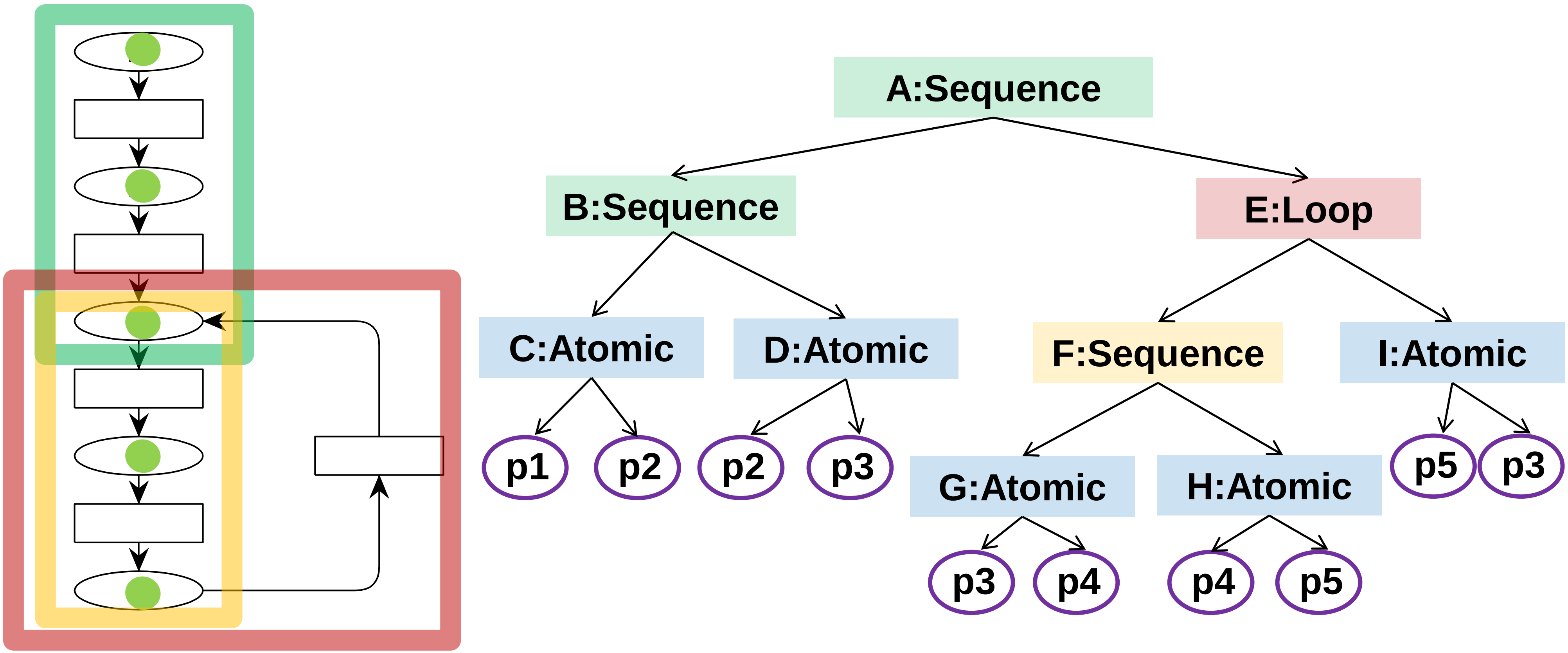
- Guarantees full coverage of the set of reachable states but may re-explore states:



- PA-CPNs structure induces sources of progress:
  - Control-flow from entry to exit point in service level modules.
  - Life-cycle of principals via intended order of service invocation.
  - Control-flow from start of test to end in service tester modules.

Progress Measures for PA-CPNs

- **Synthesised based on block tree decompositions:**



Simple (monotonic) progress measure: $(M(p_1), M(p_2), M(p_3) + P(p_4) + M(p_5))$

Complex (non-monotonic) progress measure: $(M(p_1), M(p_2), M(p_3), P(p_4), M(p_5))$

Experimental Evaluation

- **Initial results on the framing protocol:**

Config		Simple PM				Complex PM			
	Reachable	Explored	Peak	Ratio	Time	Explored	Peak	Ratio	Time
1:noloss	156	156	77	49.3	<1 s	165	63	40.3	<1 s
1:lossy	186	186	99	53.2	<1 s	196	78	41.9	<1 s
3:noloss	2,222	2,222	2,014	90.6	<1 s	2790	1,582	71.2	<1 s
3:lossy	2,928	2,928	2,700	92.2	<1 s	4037	2,187	75.7	<1 s
7:noloss	117,584	117,584	115,373	98.1	216 s	143,531	86,636	73.6	32 s
7:lossy	160,620	160,620	158,888	98.1	532 s	263,608	124,661	77.6	80 s

- **Complex progress measure gives better reduction but at the expense of exploring more states.**

Conclusions and Future Work

- **Formalisation of PA-CPNs completes the development of the code generation approach.**
- **Demonstrated how PA-CPNs structure can be exploited to make verification more efficient.**
- **Conclusion: PA-CPNs supports both automated code generation and more efficient verification.**
- **Directions for future work:**
  - Comprehensive experimental evaluation of the verification approach based on service testers.
  - Automated synthesis of progress measures that exploit also the life-cycle of services.
  - Using the service tester module to derive test cases for the automatically generated protocol implementation.

Thank you for your attention!

www.petricode.org