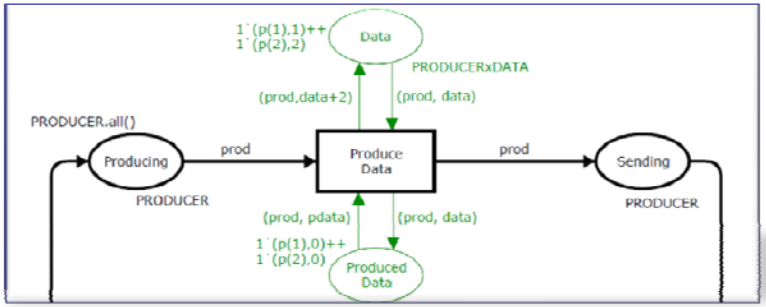
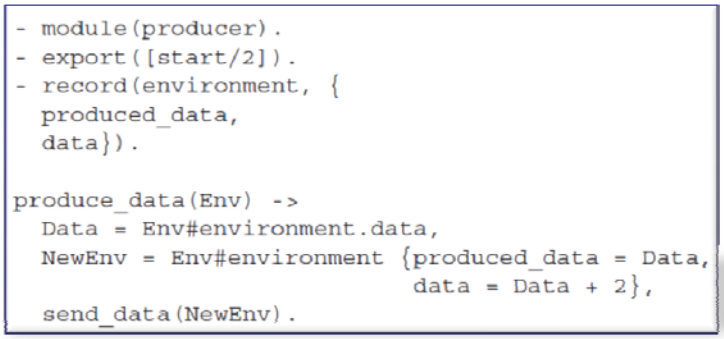
Towards Automatic Code Generation from Process Partitioned Coloured Petri Nets





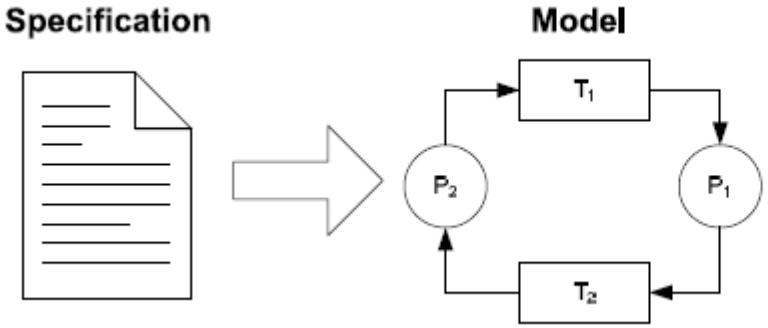
```
- module(producer).
- export([start/2]).
- record(environment, {
  produced_data,
  data}).

produce_data(Env) ->
  Data = Env#environment.data,
  NewEnv = Env#environment {produced_data = Data,
                            data = Data + 2},
  send_data(NewEnv).
```

K.L. Espensen, M.K. Kjeldsen, M. Westergaard
Computer Science Department,
Aarhus University, Denmark

L.M. Kristensen
Department of Computer Engineering,
Bergen University College, Norway

Motivation and Background

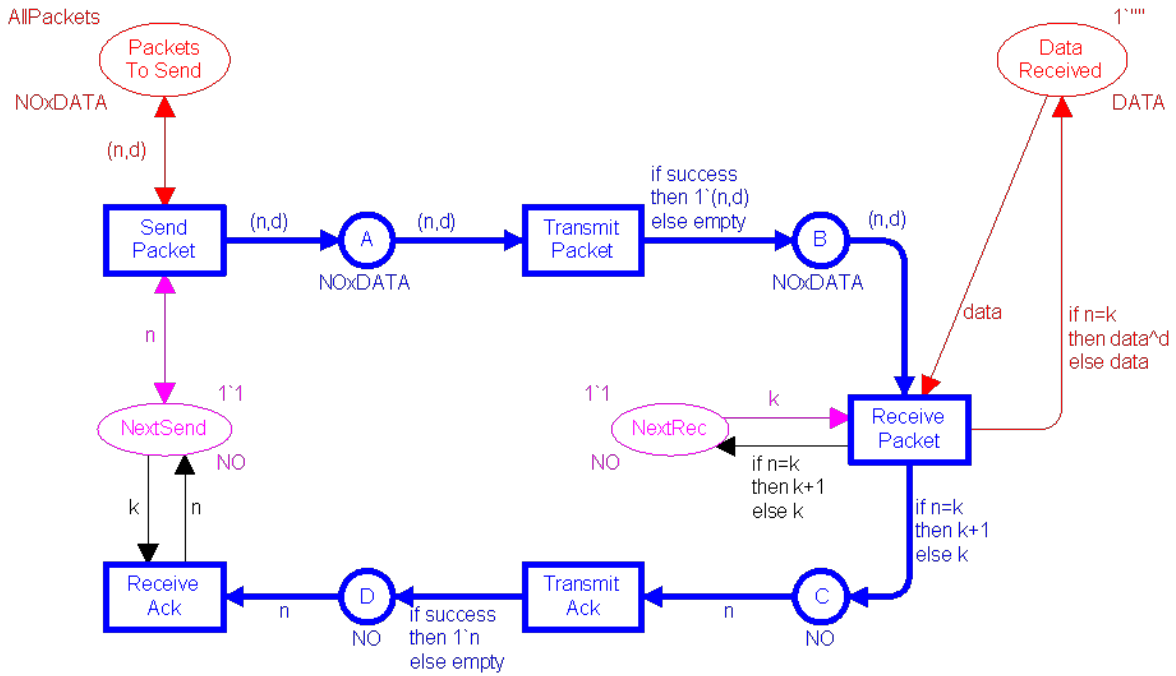
- Modelling, simulation, and analysis yields useful insight into the design and behaviour of a system:



- Several approaches to automatic code generation:
  - Simulation-based: the code constituting the model simulator is embedded directly in the implementation.
  - State space-based: the state space of the model is computed and used in the implementation.
  - Structure-based: structural analysis of the model for translation into programming language constructs.

An Observation (A Claim)

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



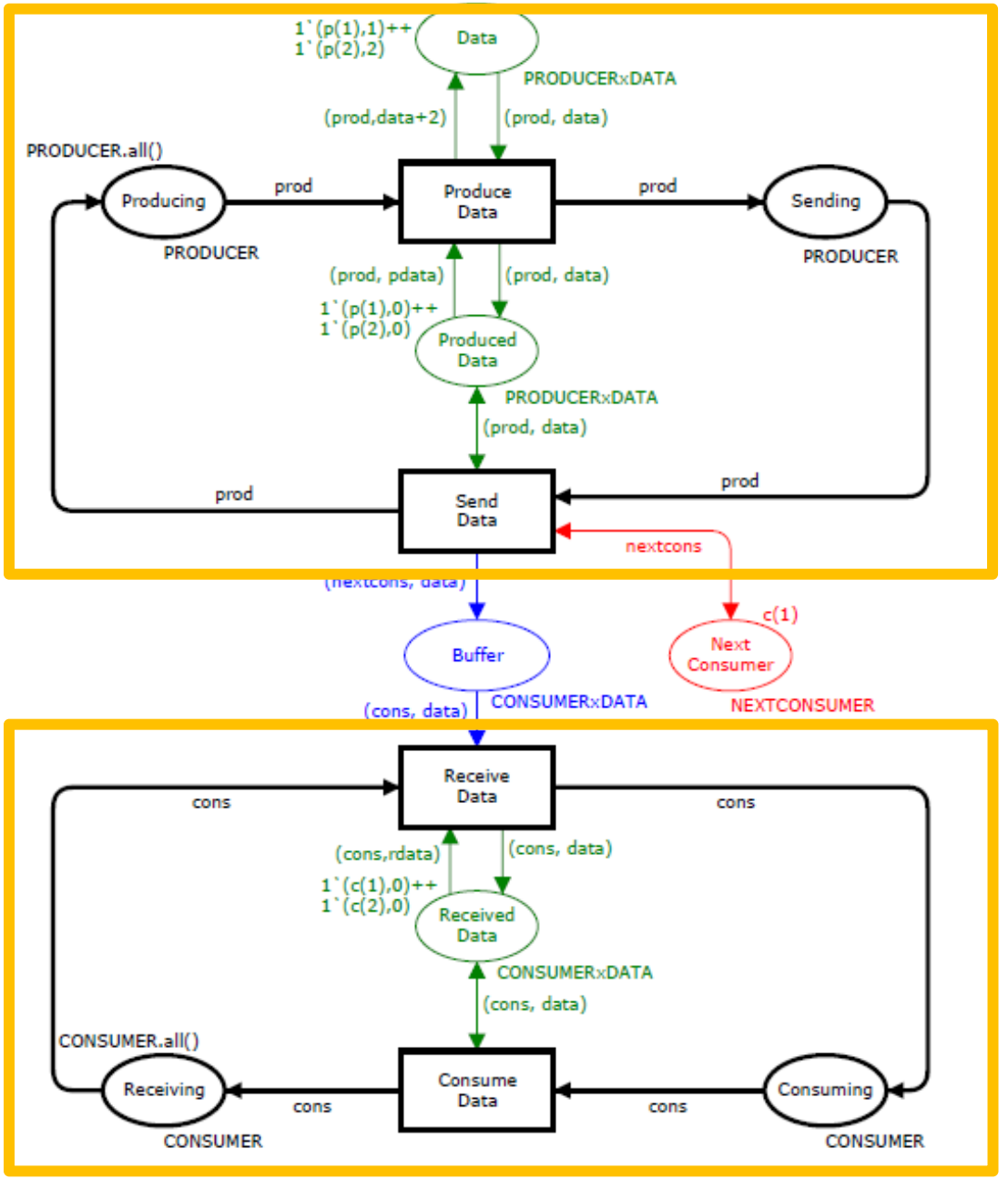
Control flow: if-then-else, case, while,...?

Communication, synchronisation, message passing, shared data,...?

- **Conclusion: Some additional syntactical constraints and annotations are needed.**

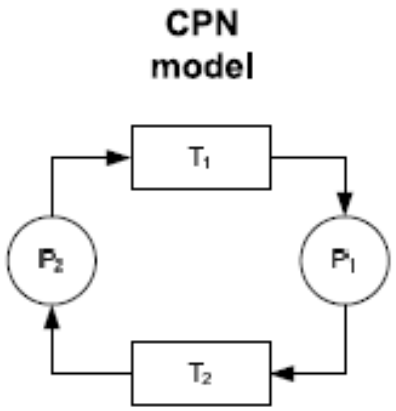
Process Partitioned CPNs

- **Provides an explicit separation of:**
  - Control flow of processes.
  - Message passing.
  - Shared and local data.
- **Key concepts:**
  - Process partitions and process places.
  - Buffer places.
  - Shared and local places.



Approach: Overview

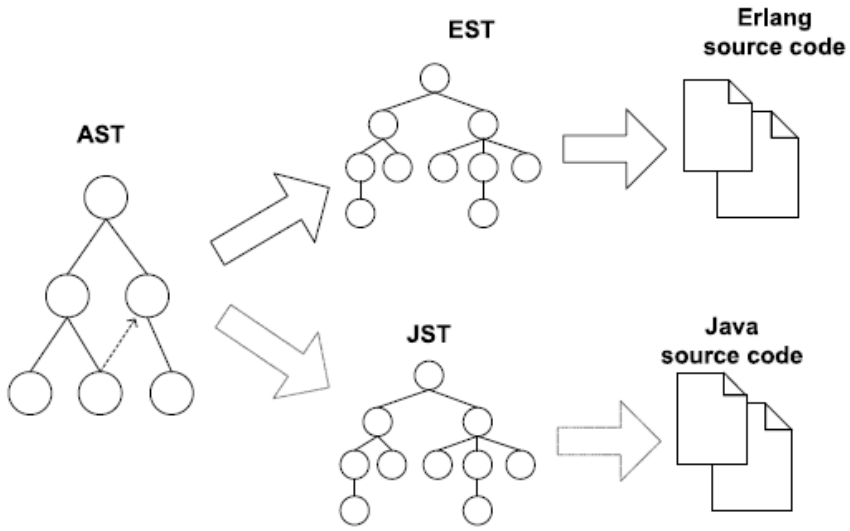
- **The translation of Process-Partitioned CPNs into an implementation is divided into phases:**



- **The abstract syntax tree (AST) is independent of a particular target programming language.**

Target Implementation Language

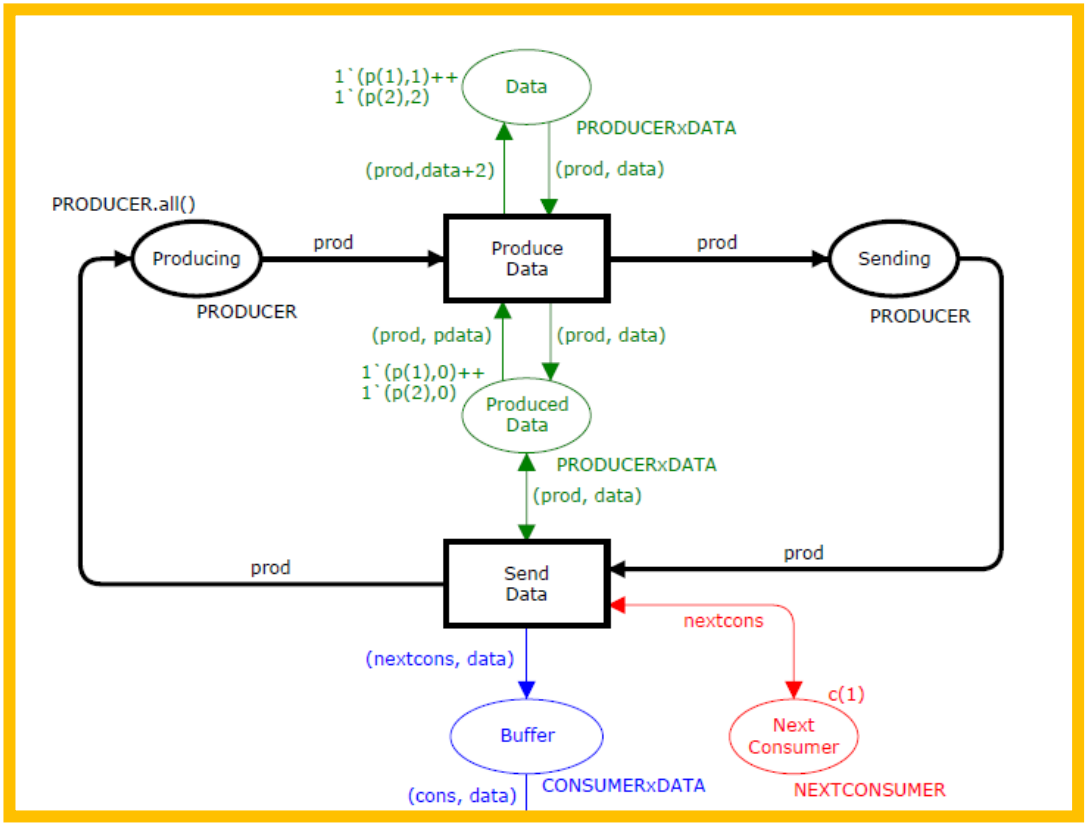
- The two last phases are target programming language specific:



- The Erlang programming language was used as the target programming language.
- The code generation did not consider the sequential CPN ML parts of the CPN model.

Phase 1: CPN Model Decoration

- **Identification of process partitions, process-, local-, buffer- and shared- places.**



Process partitions: Program executed by one or more processes.

Process places: Control flow locations of processes

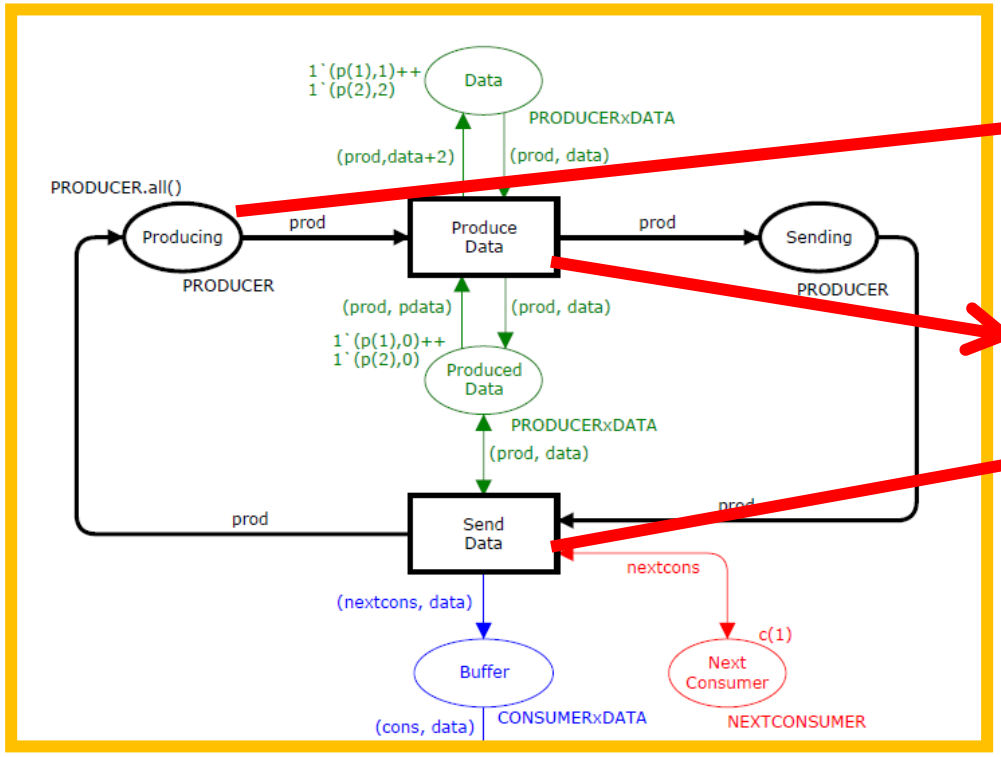
Local places: data local to a process.

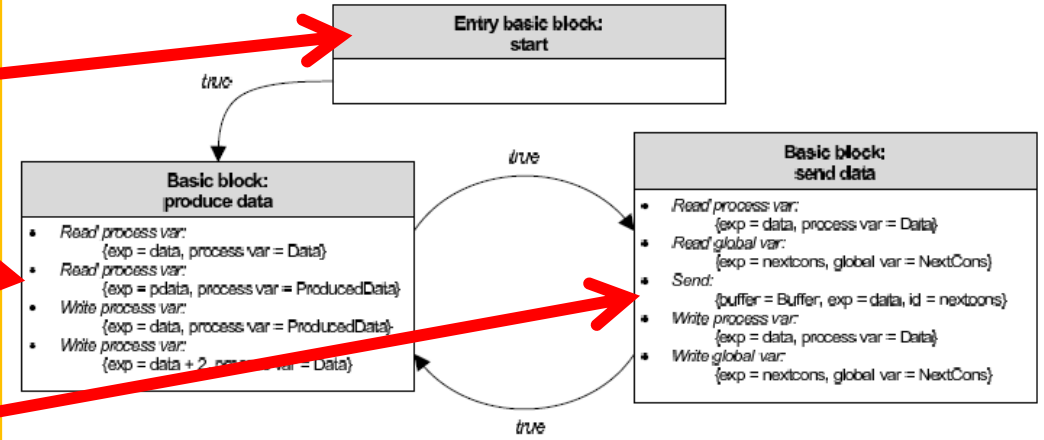
Buffer places: messages passed to a process.

Shared places: data shared between processes.

Phase 2: Translating to CFG

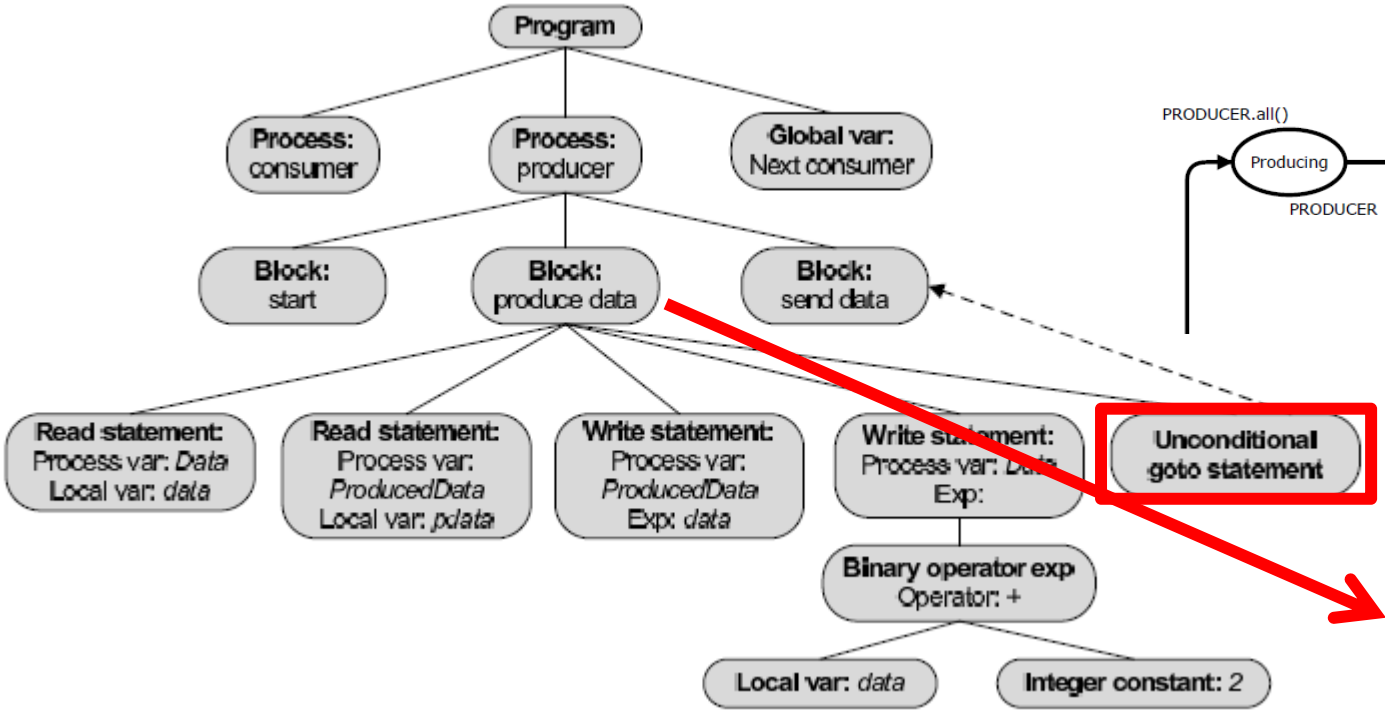
- **Constructs a control-flow graph for each process partition:**

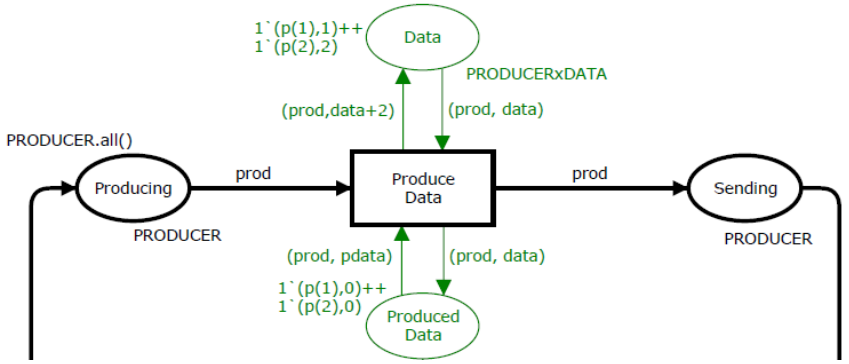




Phase 3: CFG to AST Translation

- **Translates blocks and statements of the CFG into an abstract syntax tree (AST):**



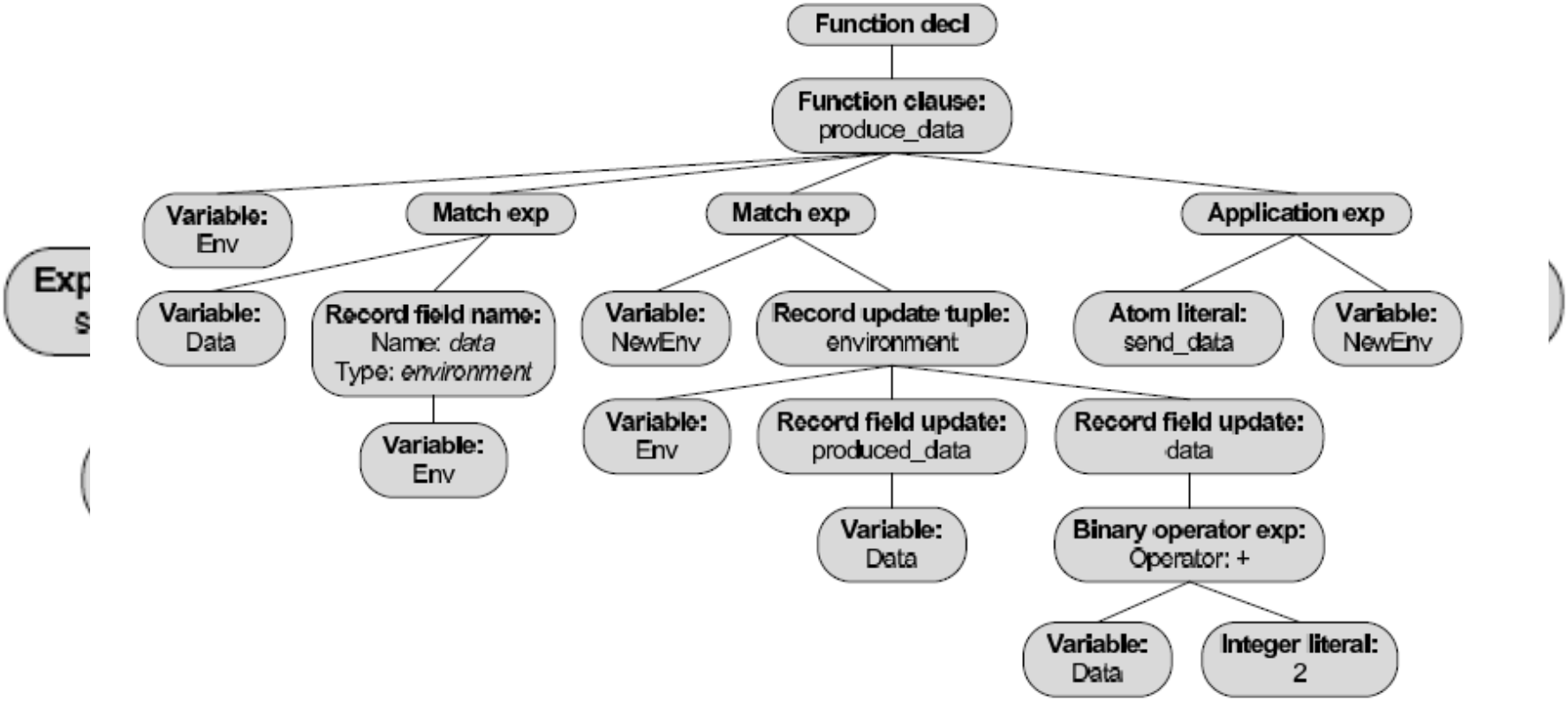


Basic block: produce data

- *Read process var:* {exp = data, process var = Data}
- *Read process var:* {exp = pdata, process var = ProducedData}
- *Write process var:* {exp = data, process var = ProducedData}
- *Write process var:* {exp = data + 2, process var = Data}

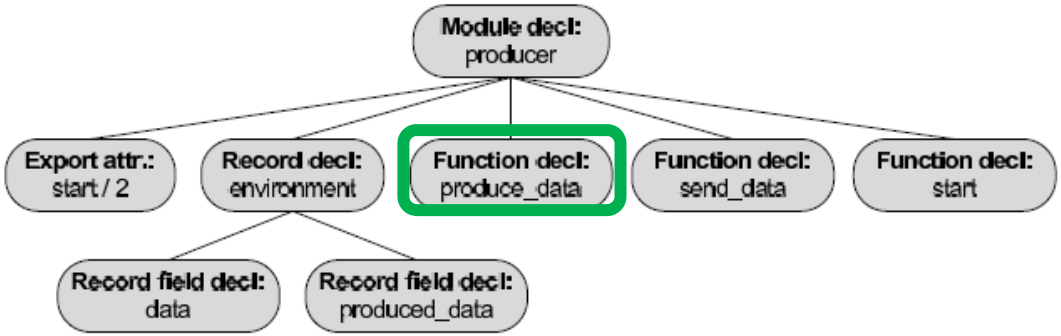
Phase 4: AST to EST Translation

- **Generates an Erlang syntax tree (EST) from the abstract syntax tree (AST):**

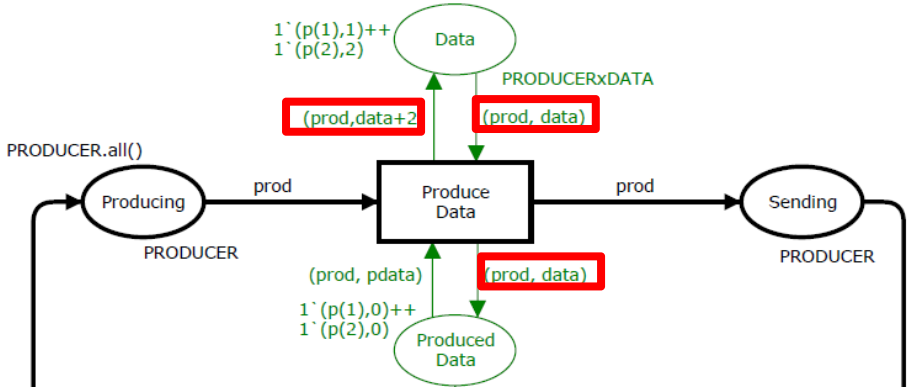


Phase 5: EST to Erlang Code

- **Traverse the EST and writes out a textual representation of the Erlang program:**

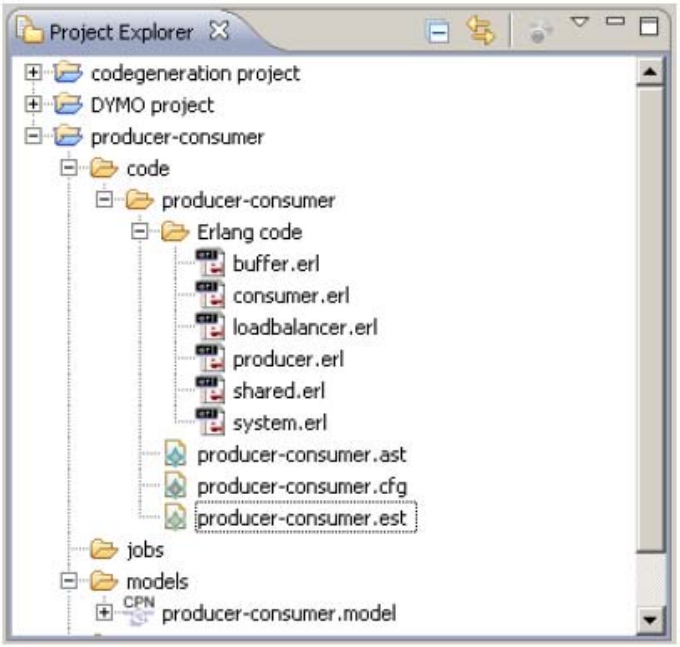


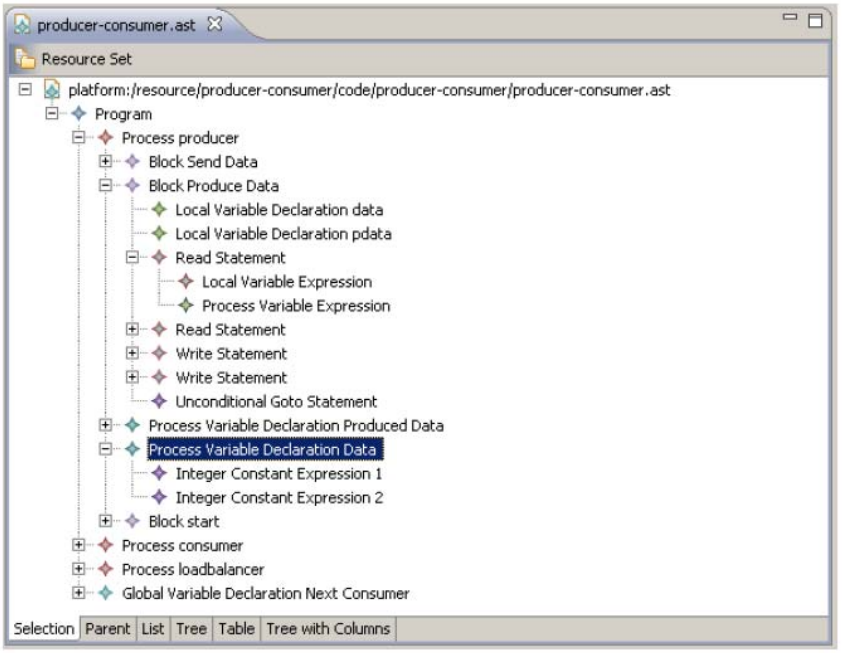

```
- module(producer).
- export([start/2]).
- record(environment, {
  produced_data,
  data}).

produce_data(Env) ->
  Data = Env#environment.data,
  NewEnv = Env#environment {produced_data = Data,
                            data = Data + 2},
  send_data(NewEnv).
```



Computer Tool Support

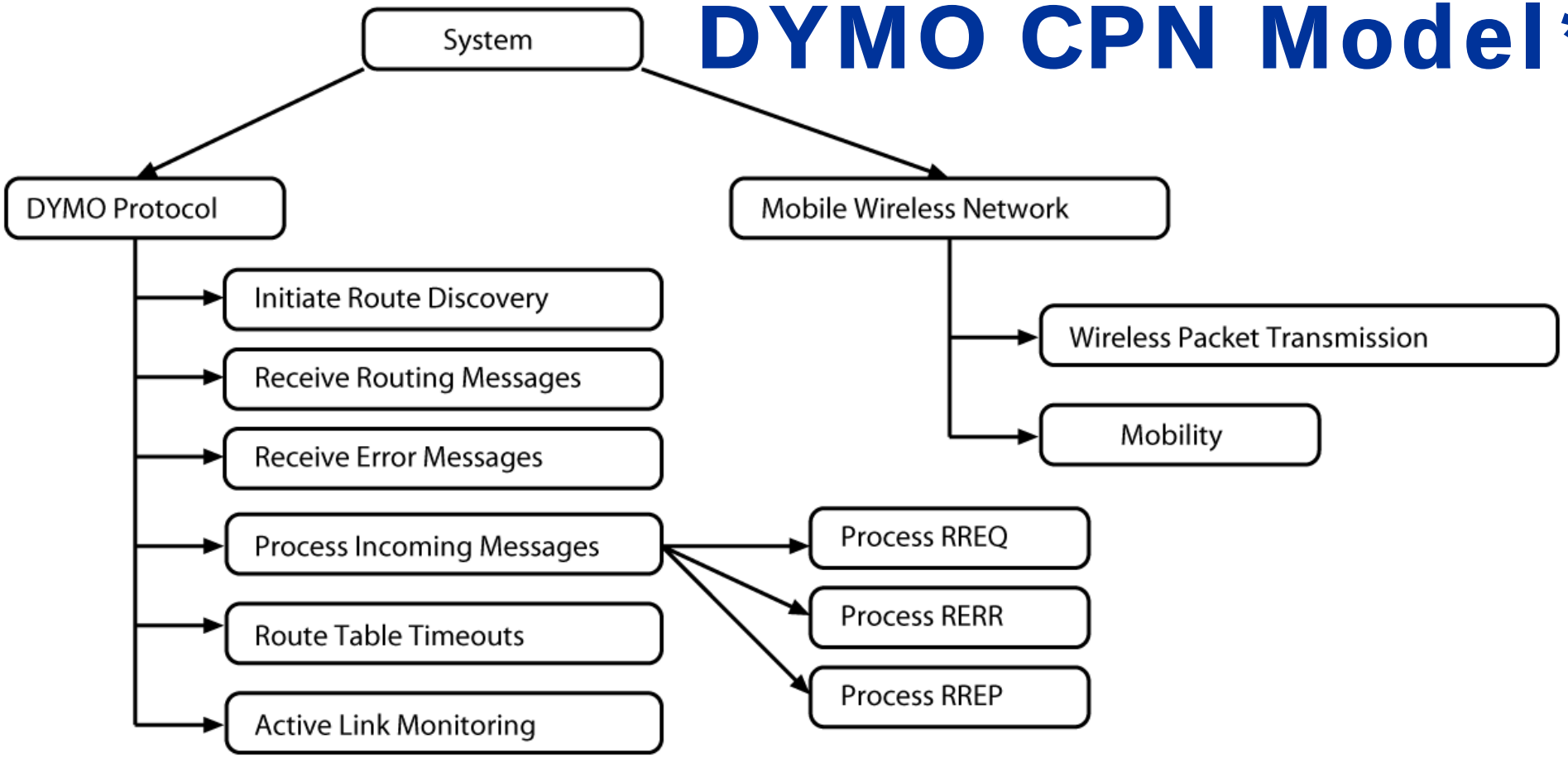
- A prototype implementation realised as an Eclipse plug-in to the ASAP Verification Platform.
- The existing EMF CPN model of ASAP and the CPN importer was used to load CPN models:





A Larger Example:
The DYMO MANET Routing Protocol

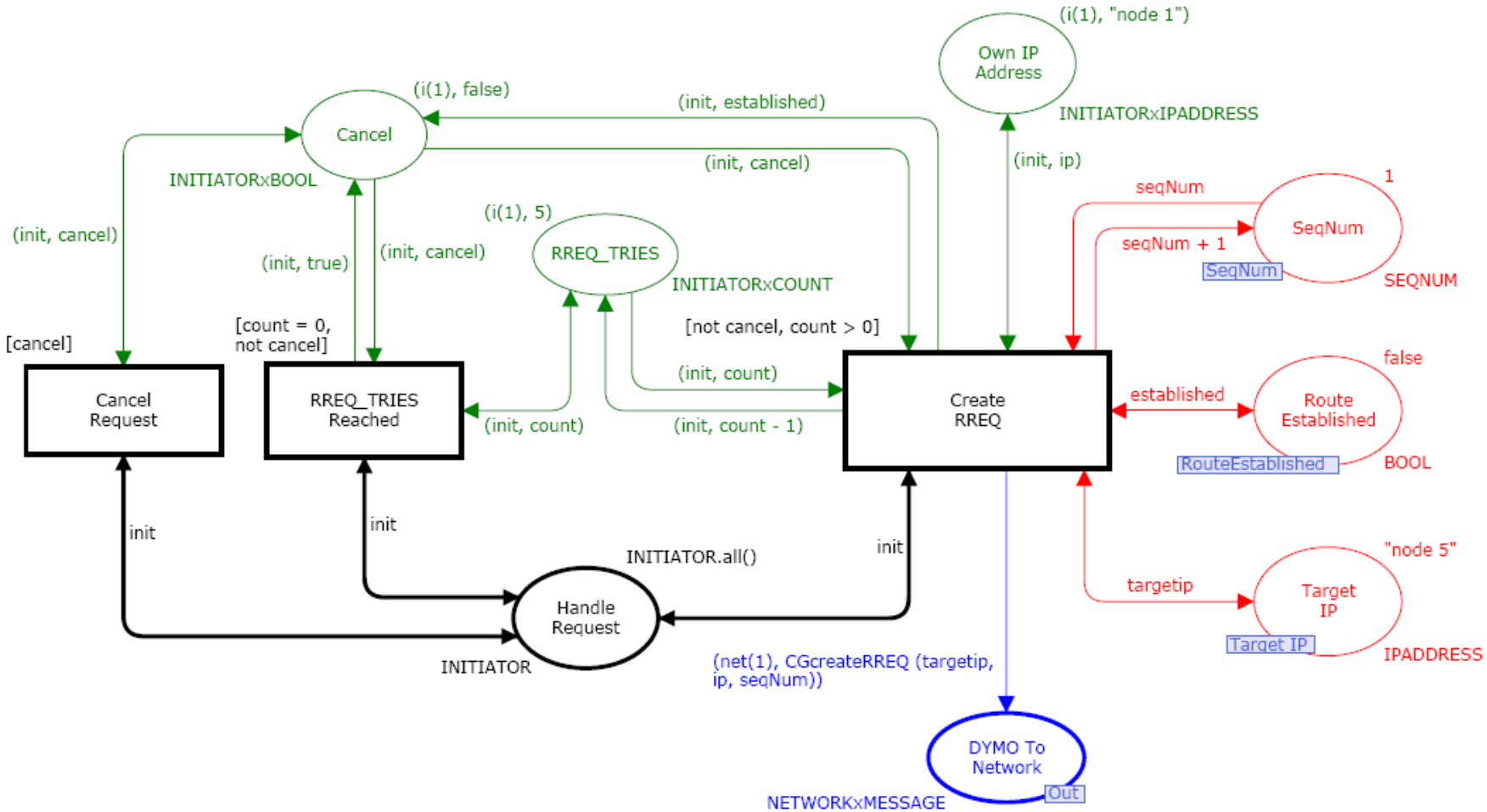
DYMO CPN Model*



- **Adjusted to satisfy the syntactical constraints of Process-Partitioned CPNs in approximately 20 person-hours.**

***K. L. Espensen, M. K. Kjeldsen, and L.M. Kristensen. Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In Proc. of ICATPN'2008. Vol. 5062 of LNCS, pp. 152-170. Springer, 2008.**

Example: Initiate Module



Code Generation and Validation

- **Statistics from code generation:**

Module name	L.O.C.	Functions to implement
system.erl	20	0
buffer.erl	36	0
shared.erl	16	0
initiator.erl	116	1
receiver.erl	116	7
processer.erl	111	4
establishchecker.erl	126	0
network.erl	22	0
Total	563	12

  - **Approximately 12 person-hours for manual implementation of the sequential functions.**

- **The generated code was validated by setting up a distributed Erlang system:**
  - Executes a set of Erlang nodes (run-time systems) on a single PC.
  - Nodes communicate by means of an underlying network simulator.
  - Each Erlang node independently executes the generated DYMO protocol entity implementation.

Conclusions and Future Work

- **A proof-of-concept for structure-based code generation has been established:**
  - Erlang programming language used as target language.
  - Applied on an large example: the DYMO Routing Protocol.
- **Main ideas for code generation:**
  - Introduce a CPN model class with explicit notions of control flow, messaging passing, and shared and local data.
  - Translation divided into five phases deferring the choice of the target programming language.
- **Some directions for future work:**
  - Extending the Process-Partitioned CPN model class.
  - Improve control structure recognition and maybe use GCC GENERIC as the target language.