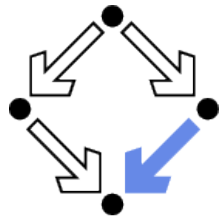
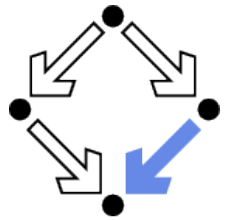
The LogicGuard Project

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.jku.at



LogicGuard



The Efficient Checking of Time-Quantified Logic Formulas with Applications in Computer Security.

- FFG BRIDGE Program

  January 2012 – December 2013

- Partners
  - RISC Institute (JKU Linz/Hagenberg)

    Wolfgang Schreiner, Temur Kutsia
  - RISC Software GmbH (Hagenberg)

    Michael Krieger, Stephan Leitner
  - SecureGUARD GmbH (Linz)

    Helmut Otto, Martin Rummerstorfer.
  - Associated: George Rahonis (Thessaloniki)

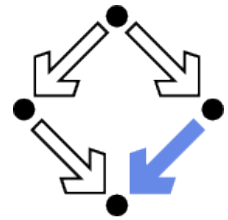
RISC
RESEARCH INSTITUTE FOR SYMBOLIC COMPUTATION

RISC
Software GmbH

SecureGuard

http://www.risc.jku.at/projects/LogicGuard

Topic

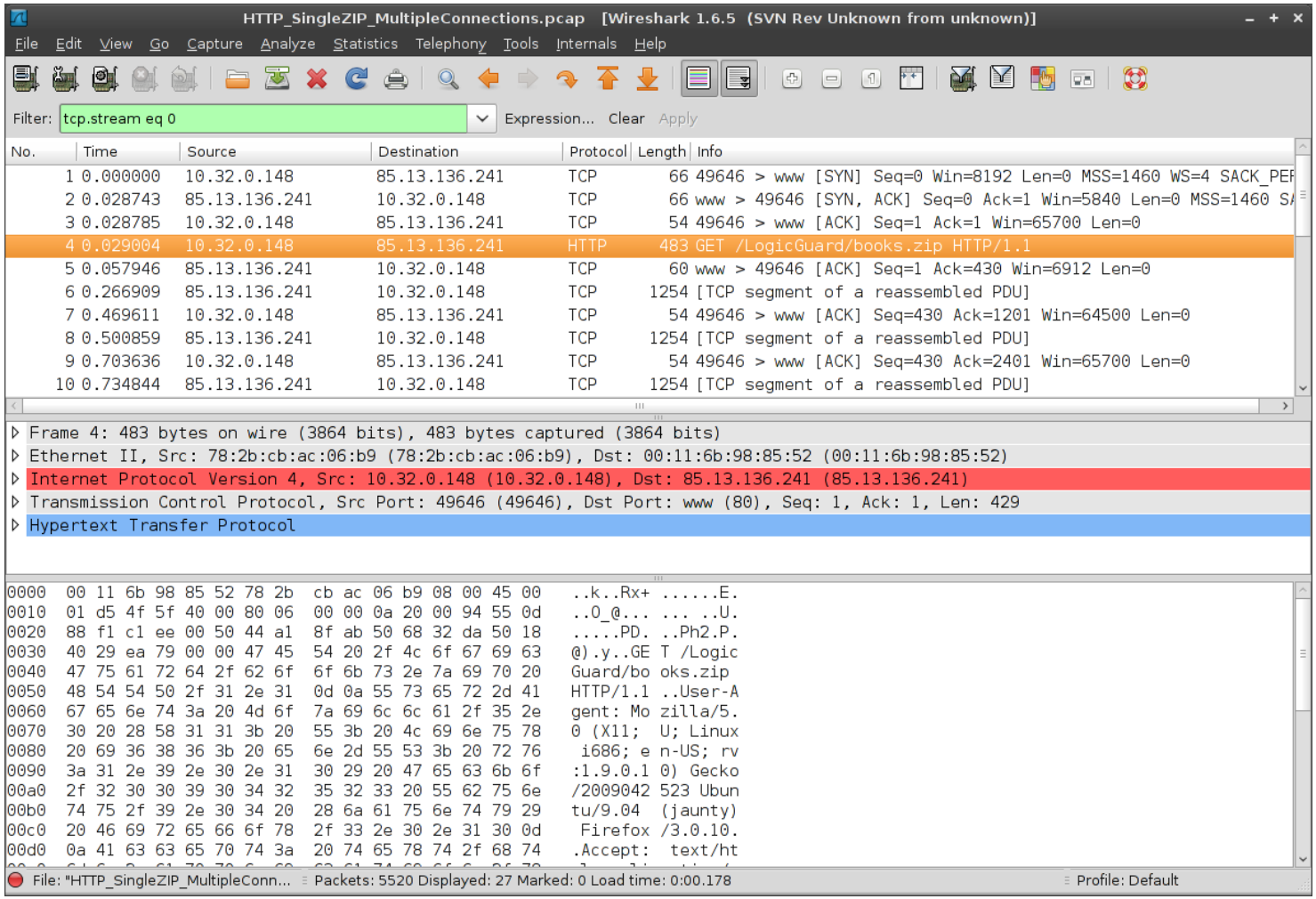


A special application of "runtime verification".

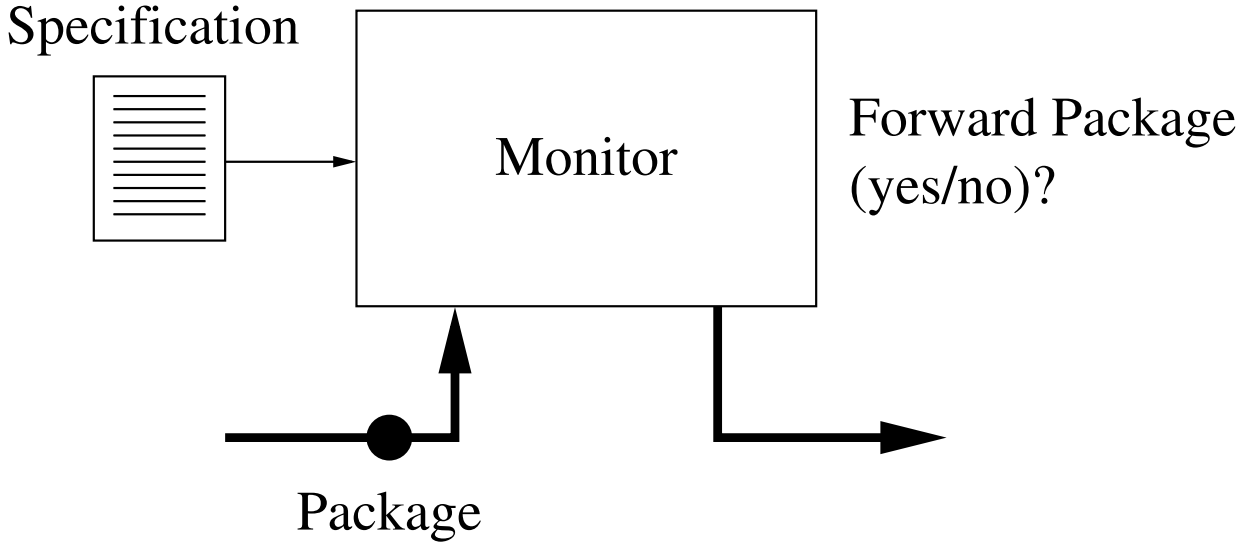
- Monitor network traffic for security breaches.
  - Traffic is an infinite stream of TCP/IP packets.
- Specify safety property in a high-level declarative form.
  - A predicate logic formula interpreted over infinite streams with quantification over stream positions.
- Automatically translate specification into a monitor.
  - A program that surveils the traffic for violations of the property.
- Advantage: no manual low-level coding of monitors required.
  - Tedious and error-prone, difficult to maintain.
- Problem: time and space complexity of the monitor.
  - Must operate with limited time and memory resources.

Use predicate logic as the specification formalism for a runtime monitor.

Network Traffic

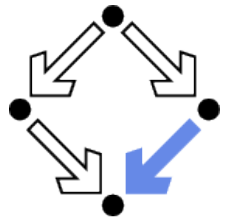


A Monitor



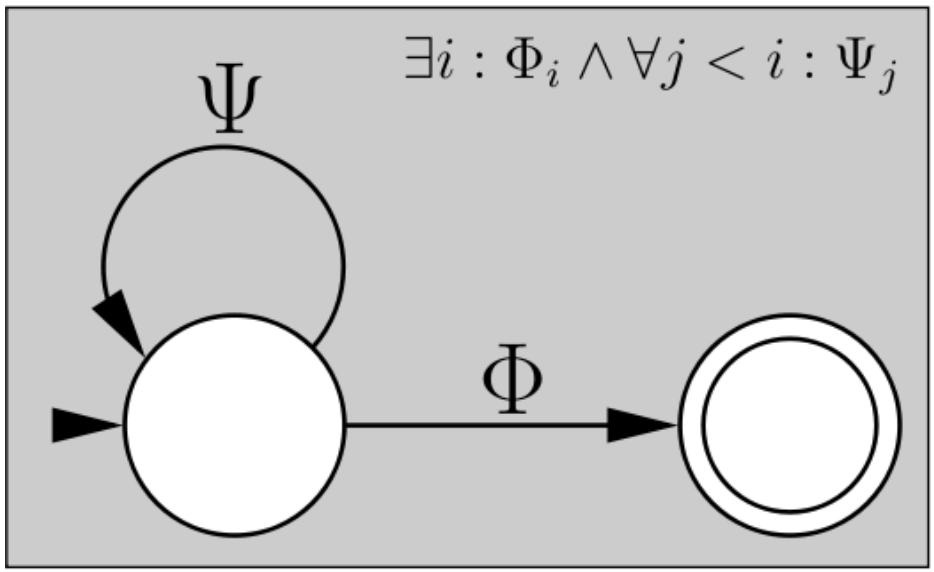
Block (or just report) every package that triggers a violation of the specified safety property.

Core Idea



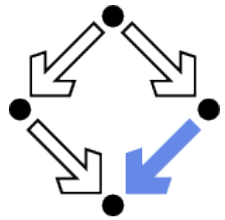
$$\Psi_0 \rightarrow \Psi_1 \rightarrow \Psi_2 \rightarrow \Psi_3 \rightarrow \Phi_4 \rightarrow \ldots$$

$$\exists i : \Phi_i \wedge \forall j < i : \Psi_j$$



Monadic second order logic (MSO): the size of the automaton is non-elimentary in the size of the formula.

Current Work

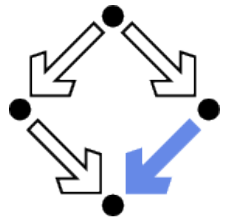


- Abstract Language
  - Syntax F and semantics $[\![F]\!]$

    $$[\![F]\!] : (P^{\omega} \times \ldots) \rightarrow Bool$$
  - Translation $T[\![F]\!]$

    $$T[\![F]\!] : Step[Bool]$$
    $$Step[T] = (P \times \ldots) \rightarrow Answer[T]$$
    $$Answer[T] = T + Step[T]$$
- Concrete Language and System
  - Parsing and type checking.
  - Translation.
  - Runtime system.
- Application Scenarios
  - Modeling of "interesting" properties.
  - Validation of language design.

Application Scenario



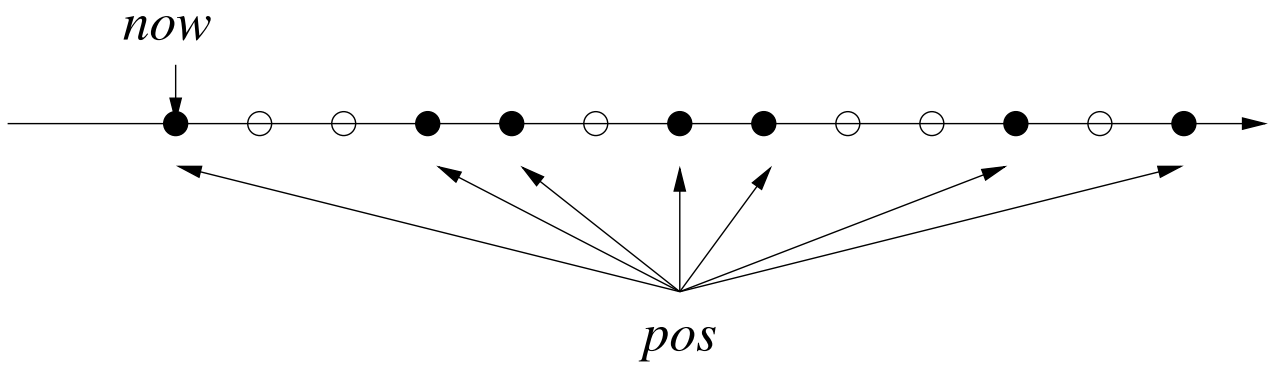
Assume that stream `parts` consists of downloaded parts of files.

```
predicate files <=>
  forall var now
    let part = parts@now
    with startFile(now, part)
    let file = getFile(now, part) :
      NOVIRUS(file)
```

$$
\begin{array}{l}
\mathit{files} \Leftrightarrow \\
\quad \forall \mathit{now} : \\
\qquad \textbf{let } \mathit{part} = \mathit{parts@now} : \\
\qquad \mathit{startFile}(\mathit{now}, \mathit{part}) \Rightarrow \\
\qquad\quad \textbf{let } \mathit{file} = \mathit{getFile}(\mathit{now}, \mathit{part}): \\
\qquad\quad \mathit{NOVIRUS}(\mathit{file})
\end{array}
$$

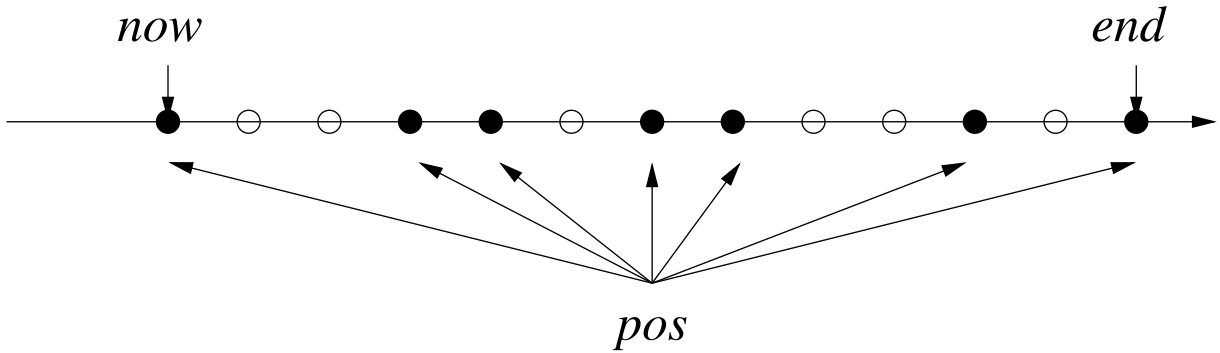
The combined files must not contain a virus.

Application Scenario (Contd)




```
function getFile(now, part) =
  let set = combine[EMPTYSET, ADDPART, FILETIMEOUT]
              var pos with now <= pos
              let part0 = parts@pos
              with SAMEFILE(part, part0)
              resettimer
              with COMPLETEPART(part0)
              until COMPLETESET(this) :
              part0 :
  FILE(set)
```

Application Scenario (Contd)

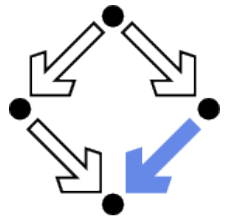


$getFile(now, part) =$
  let $set = C(now, part)$:
  $FILE(set)$

$C(now, part) =$
  let $end = \min\ p \geq now$ such that $COMPLETESET(C(now, part, p))$:
  $C(now, part, end)$

$C(now, part, p) = combine[EMPTYSET, ADDPART]\ \{part0 \mid$
    $now \leq pos \leq p \wedge part0 = parts@pos \wedge$
    $SAMEFILE(part, part0) \wedge COMPLETEPART(part0)\}$

Stream Transformations



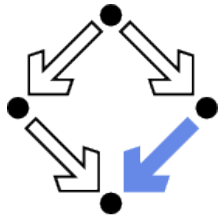
Actually, the input stream consists of TCP/IP packets, not file parts.

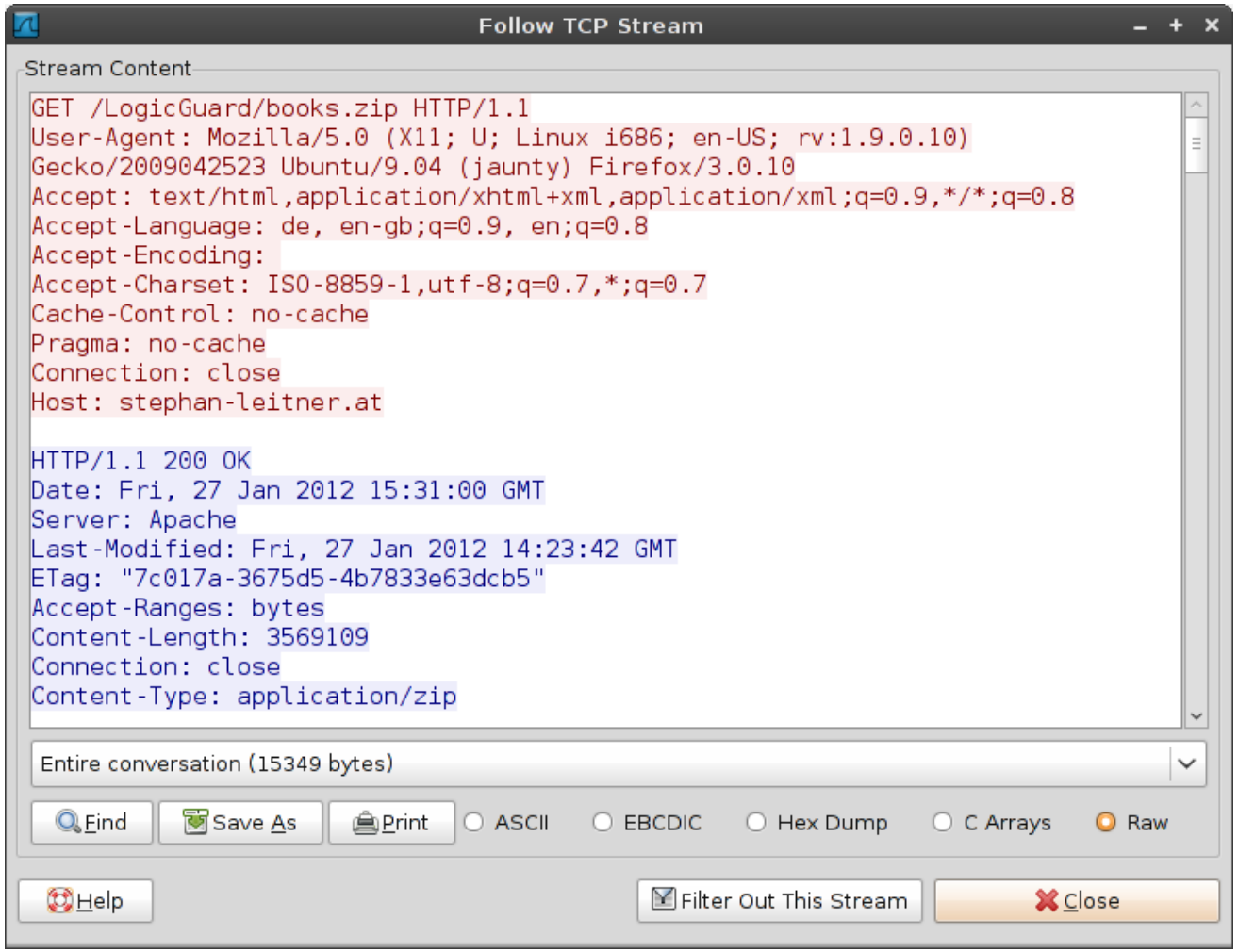
```
tcpip -> connections ---> http ---+--> downloads -> parts -> files
              |    |               ^
              |    +--------+      |
              |             |      |
              v             v      |
         ftprequests ----> ftp ---+
```

let *connections* = . . . *tcpip* . . . :
let *http* = . . . *connections* . . . :
let *ftprequests* = . . . *connections* . . . :
let *ftp* = . . . *ftprequests* . . . *connections* . . . :
let *downloads* = . . . *http* . . . *ftp* . . . :
let *parts* = . . . *downloads* . . . :
files

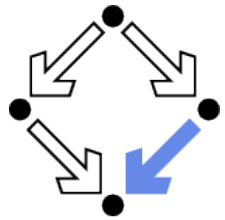
The stream has to be transformed to appropriate layers of abstraction.

The Connection Layer





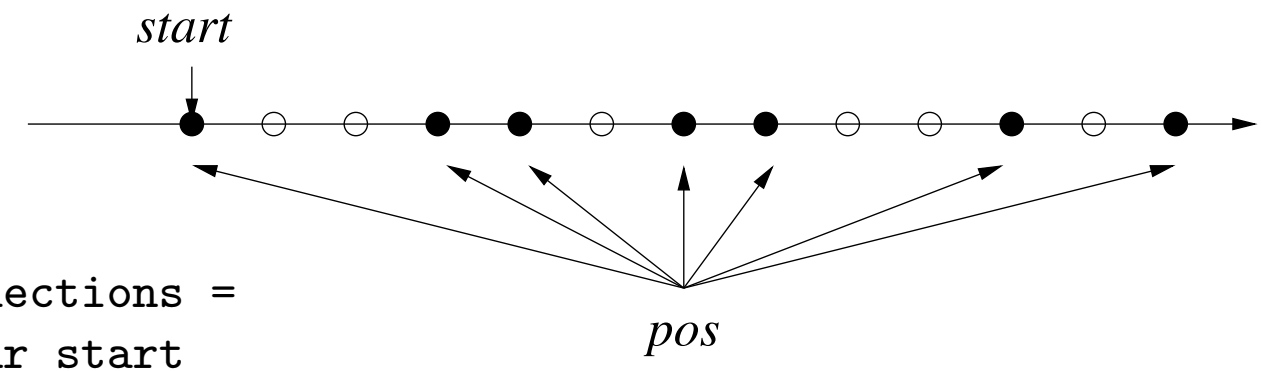
Abstraction Layers



- Packages are merged to *connections*.
  - A connection is a sequence of bytes flowing between two hosts.
  - Content is formed according to some protocol (FTP, HTTP, ...).
- From connections *downloads* are extracted.
  - A download is a range of bytes (*XXX–YYY*) from a file part.
  - Different downloads may use different protocols.
  - Some protocols (FTP) involve multiple connections for a download.
- Downloads are combined to *file parts*.
  - A file part (*file.zip.001*) is part of a file located on a host.
  - Different parts of the same file may be on different hosts.
- File parts are combined to *files*.
  - The content of a file (*file.zip*) can be monitored for a virus.

All these layers are described in predicate logic.

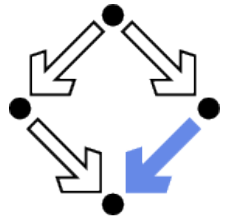
Application Scenario (Contd)




```
stream connections =
  stream var start
    let packet0 = tcpip@start
    with startConnection(packet0) :
    partial combine[EMPTYCONNECTION, ADDPACKET, CONNECTIONTIMEOUT]
      var pos with start <= pos
      let packet = tcpip@pos
      with sameConnection(packet0, packet)
      resettimer
      with ~SKIP(packet)
      until endConnection(packet0, packet) :
      packet
```

Combine all packets from the start of a TCP/IP connection till its end; the result is a stream of (partial) connections.

Status



- Start with a simple core language.
  - Abstract syntax, denotational semantics, translation.
  - Not yet adequate to cover desired scenarios.

    Basis: 4-valued logic (true, false, $\perp$, ?).
- Modeling application scenarios in a revised and extended language.
  - Iterative process until language seems adequate.
  - Syntax, semantics, translation still to be defined.
  - Major issues: semantics of stream transformations and timeouts.
- Prototype implementation.
  - Runtime system (C#): read stream from network or file.
  - Parser/type checker (C#): process specification and construct AST.
  - Monitor (F#): translate AST to monitor.
- Theoretical analysis.
  - Time/space complexity of monitor depending on specification.

Still at an early/exploratory stage.