Namur – December 9, 2003

From Craft to Science

Declarative Programming for Business-Critical e-Systems

Michel Vanden Bossche

mvb@missioncriticalit.com

Mission Critical SA, Belgium

1. Introduction

- **An industrial point of view on software construction, from the battlefield**
- **My colleagues**
  - **Requirements** (Colette Roland)
  - **Communication Worlds** (Michel Léonard)
  - **Back-end Services and Embedded Systems** (Gérard Ségarra)
  - **Usability** (Janet Wesson, Karin Becker)
  - **Declarative Approach for Complex Systems** (Michel Theys)
- **My Focus**
  - **Construction**

Introduction…

- **Observations**
  - **Undeniable progresses in some domains**
    - User Interface friendliness
    - Fairly complex technology stack works
    - Google, Deep Jr…
  - **But the software crisis is, more then ever, with us**
    - Concerns about the fragility of our software infrastructure
    - General lack of confidence in software
    - Barely acceptable rate of failures
    - More complexity

Introduction…

- **Fragility**

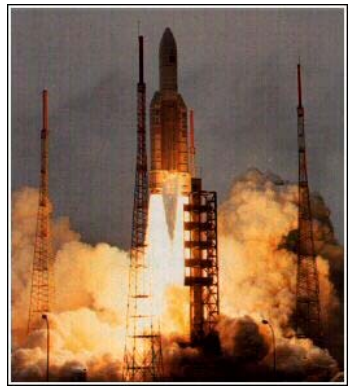
  *A major theme of the PITAC Report was the **"fragility"** of our software infrastructure, where fragility means **"unreliability, lack of security, performance lapses, errors and difficulty in upgrading"**. The PITAC was particularly concerned by this failings, because **software now affects almost every aspect** of personal and professional life in the nation.*

  Final Report, NSF Workshop On a Software Research Program for the 21st century, October 1998

  http://www.cs.umd.edu/projects/SoftEng/tame/nsfw98/FinalRep.pdf

Introduction…

- **Lack of confidence in software**



4. RECOMMENDATIONS

On the basis of its analysis and conclusions, the Board makes the following recommendations.

1. ***Switch off the alignment function of the inertial reference system immediately after lift-off. More generally, no software function should run during flight unless it is needed.***

ARIANE 5 - Flight 501 Failure Report by the Inquiry Board

Introduction…

- **Rate of failures**

  __Only 28%__ of all IT projects successful in 2000 in the USA __were successful__ with regard to budget, functionality and timeliness. __23% were cancelled__ and the remainder succeeded only partially failing on a least one of the three counts

  The Standish Group CHAOS report, in

  “The Hidden Threat to E-Government”, OECD, PUMA Policy brief N°8, March 2001

  http://www.oecd.org/dataoecd/19/12/1901677.pdf

Introduction…

- **Rate of failures…**

SPENDING WATCHDOG

Courts IT system disastrous, say MPs

By Nicholas Timmins, Public Policy Editor

The handling of a project to deliver modern information technology to magistrates' courts, approved by Geoff Hoon when he was at the Lord Chancellor's Department, was yesterday condemned as "disastrous" by parliament's spending watchdog.

The cost of the deal with ICL, now Fujitsu Services, has doubled to almost £400m. It will operate for two years fewer than planned. And it has led to two separate contracts having to be signed with other suppliers to deliver the core software application, which ICL originally promised.

In an attack on what it decribes as "one of the worst PFI deals that we have seen", the public accounts committee makes clear that the project, known as Libra, went wrong from the start.

The Department for Constitutional Affairs yesterday confirmed that Mr Hoon was the minister responsible for the deal in 1998 and 1999 when it was signed.

While the IT infrastructure is now largely in place, the courts are still waiting for the system to start operating five years after the contract was signed. Edward Leigh, committee chairman, said: "The handling of the project by the Lord Chancellor's Department was disastrous at every turn."

All the original bidders other than ICL dropped out, the committee said. Yet the department knew that "a single bid for a major complex project is seldom likely to achieve value for money".

The presence of only one bidder "should have alerted the department to the fact that its project may not have been sufficiently well designed to attract competition".

Warning signs that ICL was already in deep trouble over the benefit swipe card – a project eventually abandoned at enormous cost – were ignored and the department failed to investigate ICL's technical competence to deliver.

The department agreed to the original bid of £146m being raised to £184m once ICL was named the preferred bidder in 1998, and then renegotiated that up to £319m in May 2000 after ICL had discovered it could not use the software originally planned. Even then ICL could not cover its costs, and within 10 months the contract was cut back to cover just the IT hardware. Two other deals have since been signed for the software and its delivery.

ICL is to blame for failing to understand the department's requirements, taking on excessive risk and underpricing its initial bid, says the committee.

But the department is blamed for "failing to take decisive action when ICL did not deliver" and for repeatedly renegotiating rather than terminating when ICL was in breach of contract.

"Departments must be willing to terminate PFI contracts, or take legal action, when contractors fail to deliver," said Mr Leigh. Without that, risk transfer did not really take place and termination "should not automatically be seen as the most difficult and risky option".

Financial Times – November 11, 2003

- **Public Sector IT failures well documented…**
- **...but Private Sector failures abound**

Introduction…

Software crisis in the media

Software: So bad, it can only get better

BY PAUL STRASSMANN
(December 09, 1996)

COMPUTERWORLD

Software easily rates among the most poorly constructed, unreliable and least maintainable technological artifacts ever invented with the exception, perhaps, of Icarus' wings.

The Rising Costs of Software Complexity

Dr. Dobb's Journal April 2001

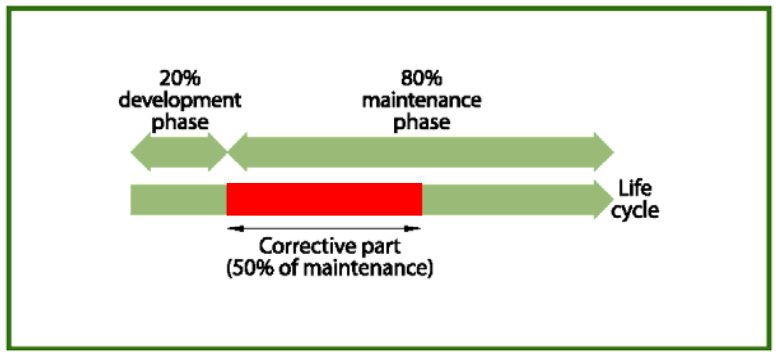
Dr. Dobb's

It's been a problem for 40 years. Now it's a crisis

- ***Software's Permanent Crisis*** (Scientific American, Sep 1994)
- ***How Software doesn't work*** (Byte, Dec 1995)
- ***Six reasons for Software Project Failure*** (IEEE Software, Sep/Oct 2002)
- ***Software Project Failure: The Reasons, The Costs*** (Datamation Jan 3, 2003)
- ***Building a better bug-trap*** (The Economist Jun 19, 2003)

2. e-Systems Issues

- **CQFD – Cost, Quality, Function, Delay[1]**
  - Pressure to reduce cost after years of overspending
  - Quality
    - Correctness, usability, robustness, security, performance…
    - Adaptability of paramount importance
  - Time-to-Market
- **Current paradigms do not seem adequate**



IEEE Software, May-June 1998

[1] Jacques Printz, CNAM

e-Systems Issues…

- **Predictability**
  - Predictability of C+Q+F+D is only obtained for **28%** of IT projects (USA, 2000)
  - Example
    - Large insurance company
    - e-Business project (non-life, for brokers and agents)
    - J2EE + commercial CRM framework
    - IT services company (“Big Five”)
    - Initial quotation: **1,500** person-days
    - Operational system: **15,000** person-days, **with problems**

e-Systems Issues…

- **Security**
  - Common Criteria and ISO 15408
  - Analysis and reasoning methods are required to guarantee that a system does not do **“bad things”**
- **Components and Web Services**
  - The use of a component (or service) built by somebody else claiming that it works **requires some way of gaining confidence in it**
  - Analysis and reasoning methods are required to guarantee that a system does the **“right thing”** and not **“bad things”**

3. Approach

- **Analysis**
  - Classical engineering produces reliable physical systems under real-world constraints because they have a consistent framework – their **scientific foundations**
  - In software, nowhere is the gap between **theory and practice** more evident than with **programming languages**
  - The semantics of Cobol, C, C++, Java… is **extremely messy**: in practice, it is very difficult to **reason** about programs in such languages

Approach…

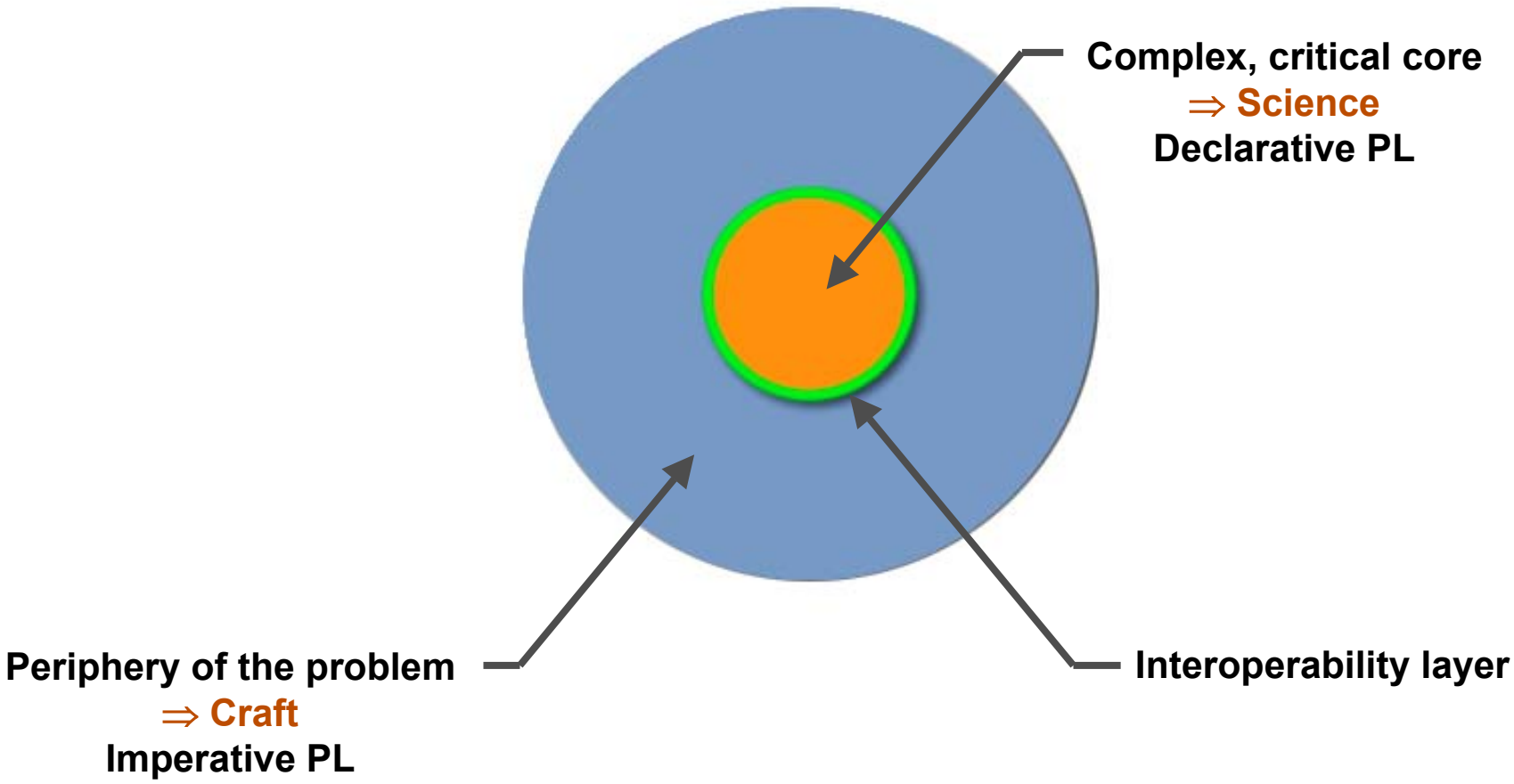
- **Analysis…**
  - Testing is the only solution with these languages.
  - Software systems are **discontinuous**: using extrapolation and interpolation to estimate output behaviour for untested input **does not work**
  - Discontinuity of software **poses problems for testing**: for systems with low reliability requirements, better statistics (time, sample) may work (Windows 95, 400.000 beta sites) but is very expensive
  - Without a **scientific basis** for the software discipline, we cannot build quality systems in a **predictable way**
  - What are the options to move **from Craft to Science**?

Approach…

- **Option #1 – Formal specifications** (Z, B, …)
  - It works, but…
    - Requires **much higher cost** and **takes time** (you « program » **twice**, the **specification** and the **implementation**)
    - Not a constraint for safety critical systems and product-lines
    - Quite an issue for business e-Systems
  - **Impedance mismatch** between specifications and program
- **Option #2 – Declarative languages** (LP/FP)
  - Which PL (programming language: applicability, risk, etc.)?
  - Which methodology?
  - What options in terms of “coexistence” with mainstream paradigm?

Approach…

- General idea



Which Declarative PL?

- **Initially** (1994, complex billing system)
  - **Prolog**
    - \+ Experience
    - \+ Full unification, difference lists, flexibility, …
    - \+ Good performance
    - \+ Rapid prototyping and interactive querying
  - **Issues**
    - – **Not** really **declarative**
    - – **Understanding** of the **program** is very **difficult** for those that have not written it (e.g. control part)
    - – Business-critical problem require **team development**
    - – **Programming-in-the-large** not the design goal of Prolog

Mercury

- **Logic/Functional** PL (University of Melbourne)
- Strong **polymorphic typing**
- **Type classes** (constrained polymorphism)
- Functions and predicates have **modes**
- Procedures have **determinism** (bounds on number of solutions)
- **Backtracking** search available for procedures with multiple solutions
- Fully declarative (= **pure**)
  - IO through unique state parameter
  - Top level of program is deterministic and does IO

Mercury…

Prolog	Mercury[1]

`append([],A,A).`
`append([A|B],C,[A|BB]):-`
`  append(B,C,BB).`
	`:-module list.`
	`:-interface.`
	`:-pred append(list(T),list(T),list(T)).`
	`:-mode append(in,in,out) is det.`
	`:-mode append(out,out,in) is nondet.`
	`:-implementation.`
	`append([],A,A).`
	`append([A|B],C,[A|BB]):-`
	`  append(B,C,BB).`

[1] http://www.cs.mu.oz.au/research/mercury/index.html

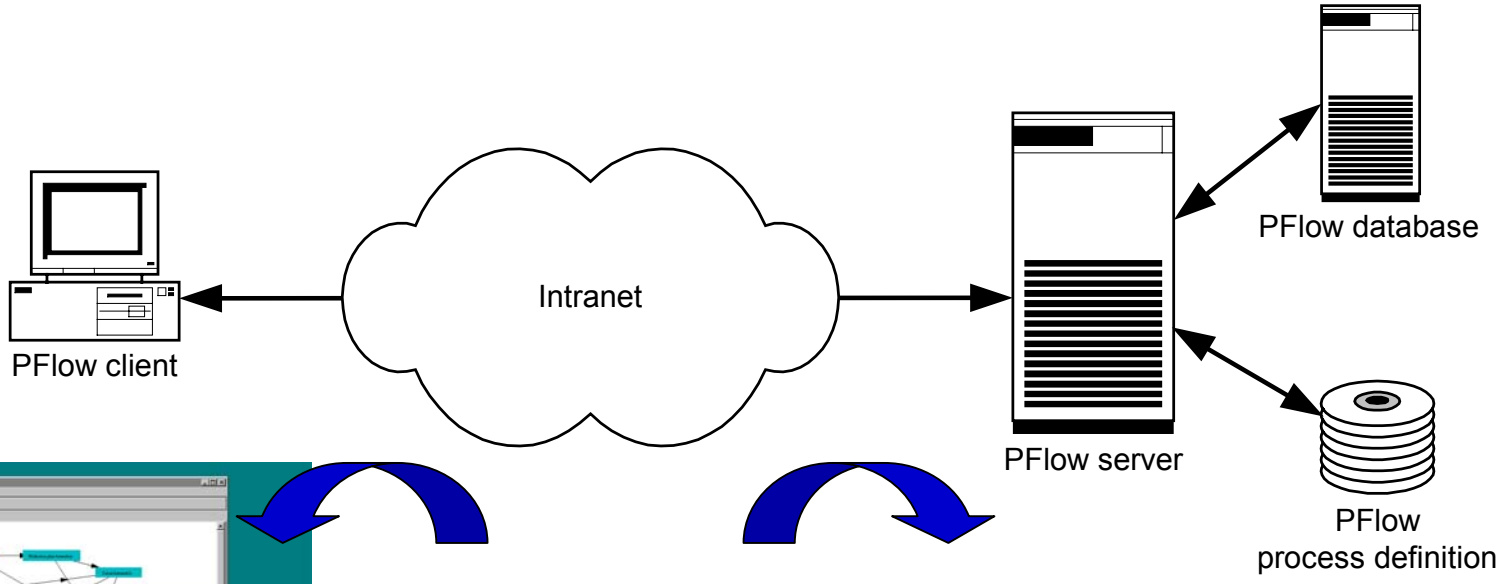
Mercury…

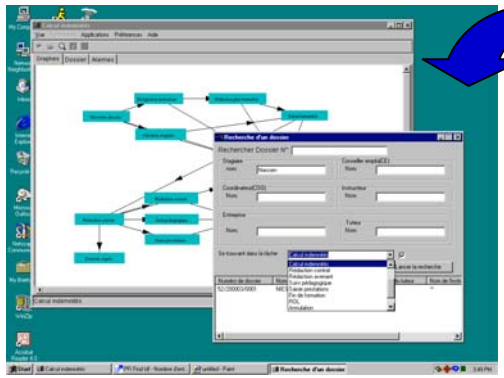
- **Initial observations** (1996 – 1998)
  - Programming in a totally **pure style takes a bit more effort**
  - When the **compiler accepts the program**, it will almost certainly **do what the programmer expected it to**
  - The **extra discipline** required is **good** and **save much time in the future**
  - The compiled code is **fast**
  - Possible to program in a **functional style** (e.g. anything that needs to do IO must be deterministic)

4. Real-Life Application

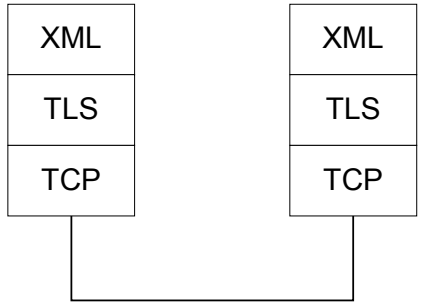
- **PFlow** (1999-2000)
- Business Process modeling: extended **Petri Nets**
- Data modeling with an **Ontology**
- Syntax: **XML** everywhere (data, Petri Net, RPC)
- Architecture
  - Light HTML client developed in Java
  - XML protocol based on WfMC
  - PKI security
  - Petri Net Server develop in **Mercury**
  - Reuse (MS XML parser, Excel as business computing engine)
  - Component integration through COM

Architecture





Java




```
:-pragma memo(dependson/2).
:-pred dependson(sheet_name::in,sheet_name::in) is semidet.
dependson(S1,S2):-
        S1\=S2,ivars(S1,IV1),ovars(S2,OV2),
        set__intersect(IV1,OV2,Intersect),
        not set__empty(Intersect).
dependson(S1,S2):-
        member(Si,allsheets),S1\=Si,S2\=Si,S1\=S2,
        ivars(S1,IV1),ovars(Si,OVi),
        set__intersect(IV1,OVi,Intersect),
        not set__empty(Intersect),
        dependson(Si,S2).
```

Mercury

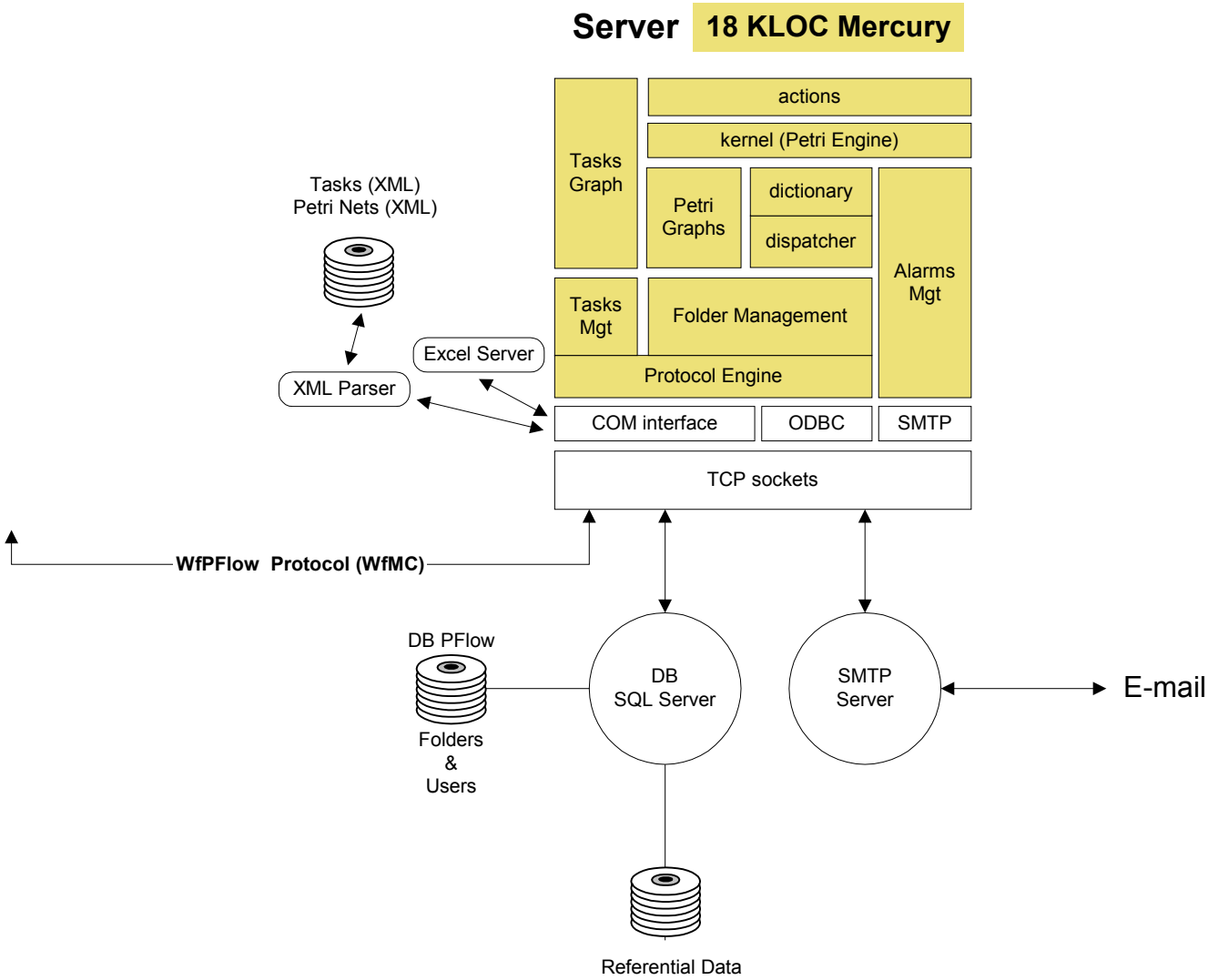
PFlow Client

- **What**
  - Case selection using multi criteria searches
  - Query process variables for active actions in current task
  - Display/set all variables
  - Display/print documents
  - Display alarms status
  - Execute client actions (e.g. external applications)
  - Display the process state through the Petri Net Graph
  - UI completely **driven by the process** (35 different screens)
  - Screens defined in the ontology
- **How**
  - **~ 22 KLOC Java**

PFlow Server

- **Request Processing**
  - Search requests: retrieve sets of cases satisfying criteria
  - Case lookup request: returns current state of a case
  - Case update request: handle data/tokens modifications, returns resulting state
- **Transaction handling**
  - Update requests treated as transactions
- **Alarm handling**
  - Delays tokens for duration of alarms
- **Generate tokens for repetitive alarms**
- **Actions handling**
  - Complex values computation (e.g. travel indemnities)
  - e-mail messages

Server Architecture



Example of Specification

$$
\begin{aligned}
S_1\ dependson\ S_2\ \hat{=}\ & \\
& S_1 \neq S_2 \\
\wedge\ & (\ \ ivars(S_1) \cap ovars(S_2) \neq \emptyset \\
& \vee\ \ \exists S_i\ :\ i \notin \{1,2\} \\
& \qquad \wedge\ ivars(S_1) \cap ovars(S_i) \neq \emptyset \\
& \qquad \wedge\ S_i\ dependson\ S_2 \\
&)
\end{aligned}
$$

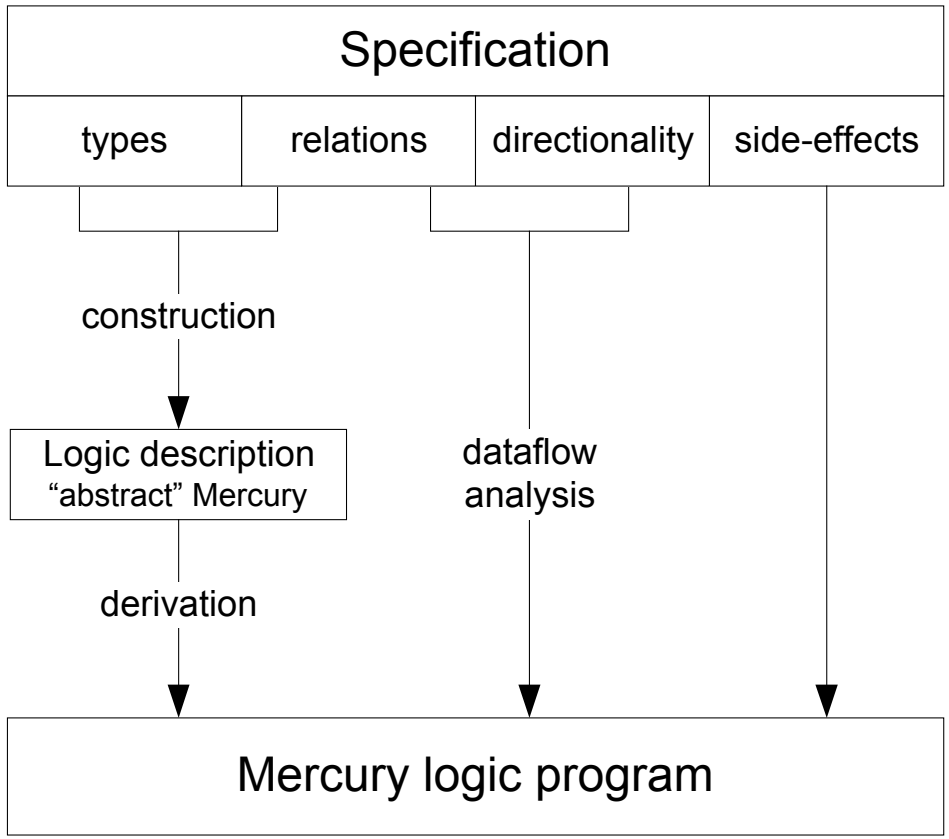
Mercury Code for dependson

```
:-pragma memo(dependson/2).
:-pred dependson(sheet_name::in,sheet_name::in) is semidet.
dependson(S1, S2):-
    S1 \= S2,
    (
        not set__empty(ivars(S1) `intersect` ovars(S2))
    ;
        Si `member` allsheets, S1 \= Si, S2 \= Si,
        not set__empty(ivars(S1) `intersect` ovars(Si)),
        dependson(Si, S2)
    ).
```

PFlow Server – Code Statistics

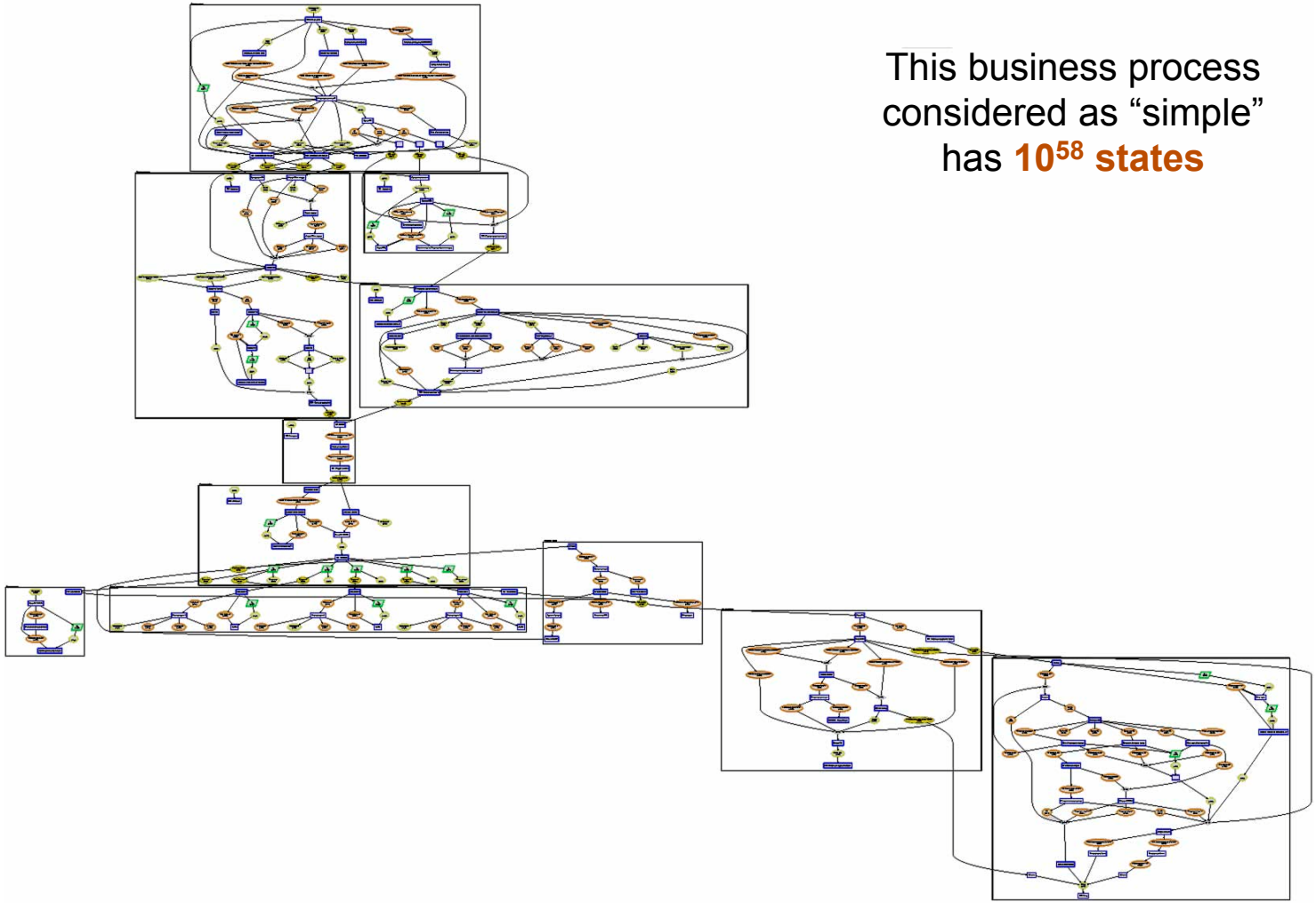
```
Number of types:                    346
Number of insts:                      2
Number of predicates:              1435
Number of functions:                241

Number of predmodes:                479
Number of funcmodes:                 29
Number of separate modes:          1177
Number of implicit function modes:    ?
Total number of modes:        >=   1685
                              =<   1926
        - det:                     1630 ( 96.74%)
        - semidet:                  245 ( 14.54%)
        - nondet:                    30 (  1.78%)
        - multi:                      6 (  0.36%)
        - cc_nondet:                  0 (  0.00%)
        - cc_multi:                   2 (  0.12%)
        - erroneous:                  4 (  0.24%)
        - failure:                    0 (  0.00%)
Average modes per predicate: >=  1.154
                             =<  1.174

Blank lines:                       3261 ( 10.35%)
Comment lines:                     4095 ( 12.99%)
Total whitespace/comment lines:    7356 ( 23.34%)

Function declaration lines:         242 (  0.77%)
Predicate declaration lines:       1678 (  5.32%)
Mode declaration lines:            1194 (  3.79%)
Type declaration lines:            1318 (  4.18%)
Inst declaration lines:              10 (  0.03%)
Other declaration lines:           2304 (  7.31%)
Total declaration lines:           6746 ( 21.41%)

Code lines:                       17412 ( 55.25%)

Total number of lines:            31514 (100.00%)
```

Specification & Programming



Yves Deville (1998), an evolution from *Logic Programming – Systematic Software Development*, Addison-Wesley (1990)

Complexity is managed



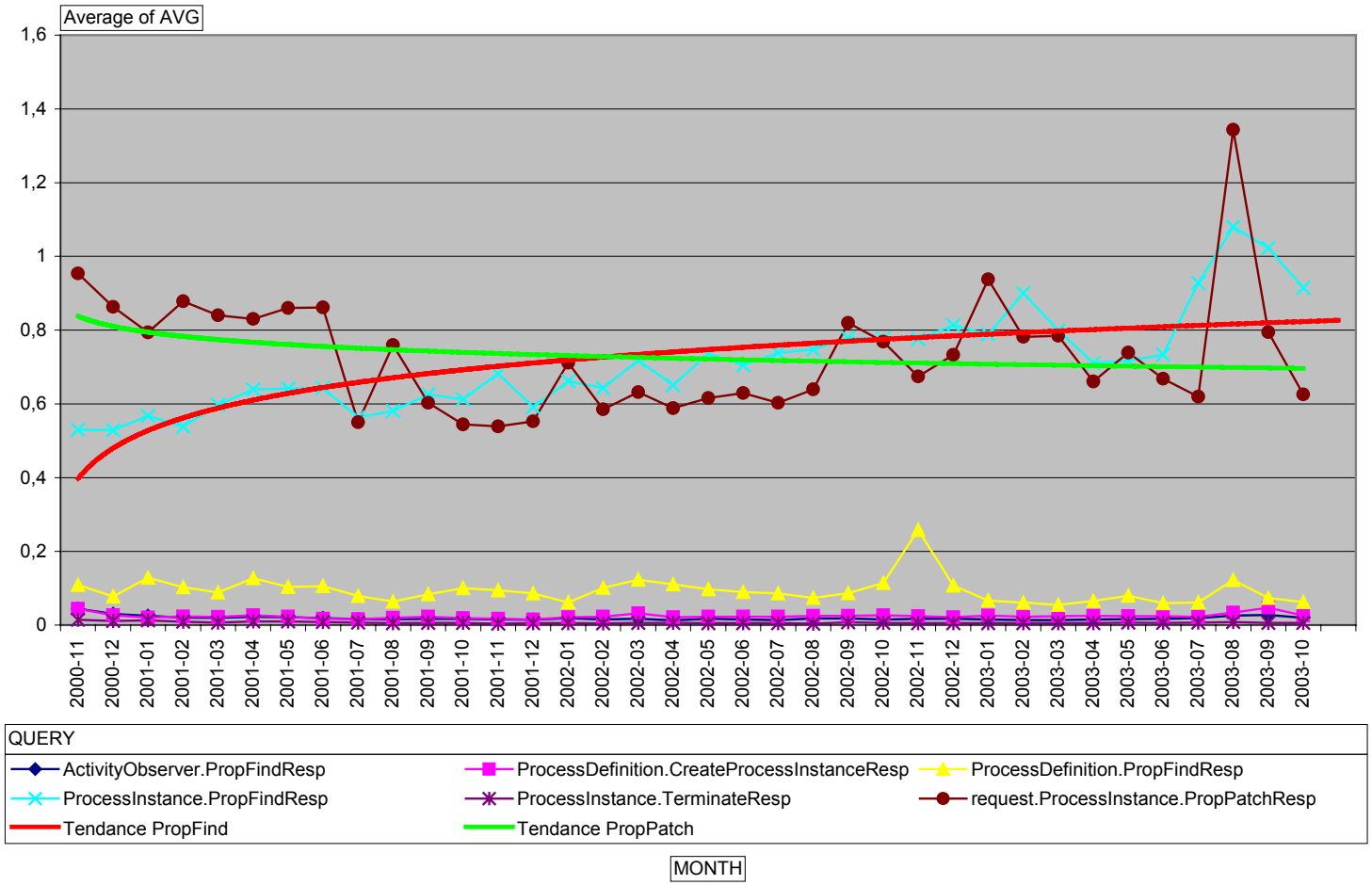
This business process considered as “simple” has **10^{58} states**

Appraisal of Mercury

- Programming in a totally pure style **takes a bit more effort** in the very early phase (specifying and programming)
- When the compiler accepts the program, **the program is correct** wrt to the intentions of the programmer
- The extra discipline required is good and **save much time in the future** (adaptive maintenance is very important)
- The compiled code is **fast**
- Possible to program in a **functional style** (important for real-world programs)
- Extremely robust server (**“0 defect”**)
- Client built in **Java is still not bug free** (memory leaks in spite of Java, compatibility issues with IE 5.0, 5.5, 6.0)

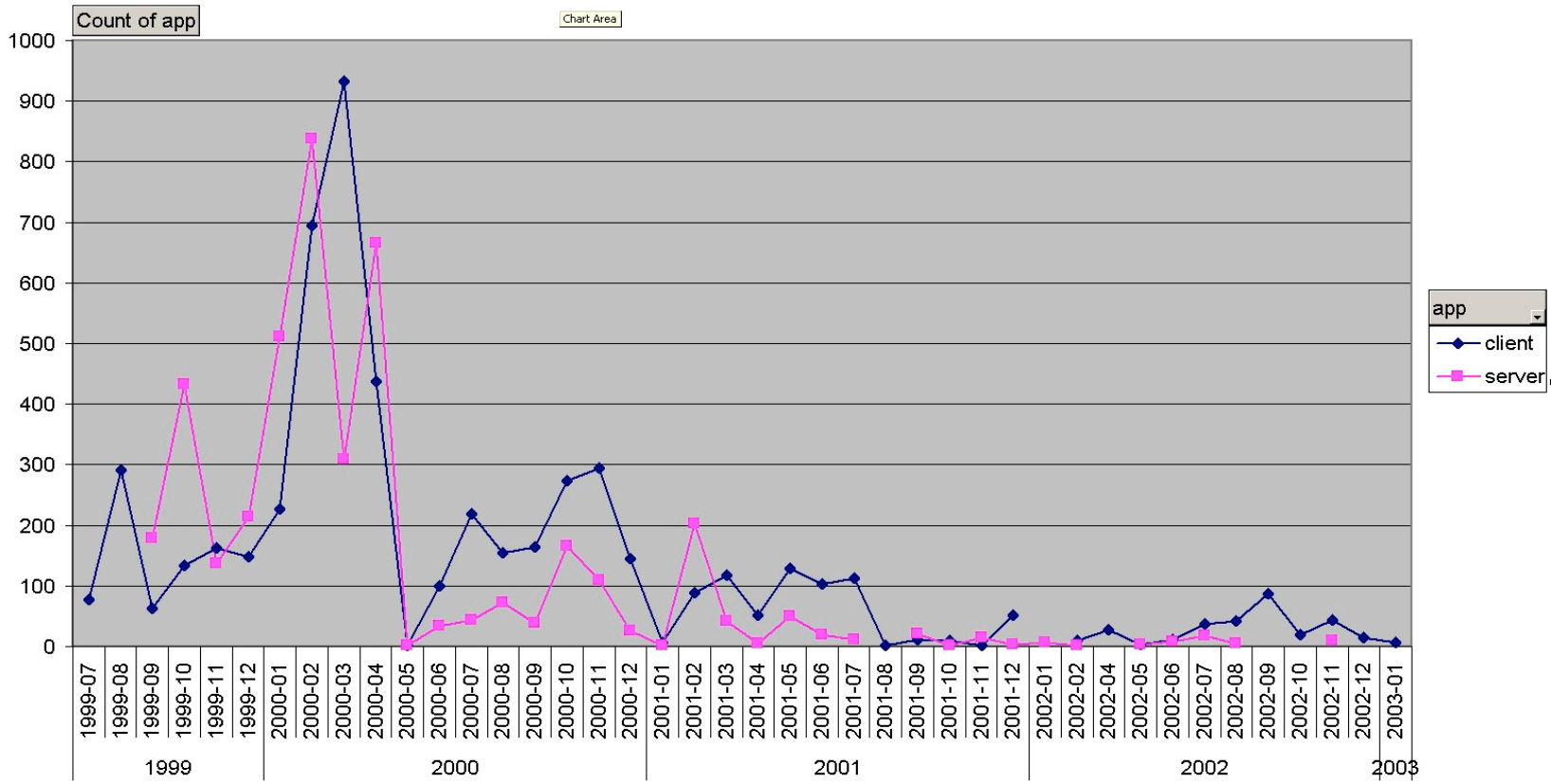
Appraisal of Mercury…

Performance



Appraisal of Mercury…

- **Maintenance**
  - Still have problems with Java client
  - Updates to the server very easy



Appraisal of Mercury…

- **Maintainability…**
  - **One technical issue**
    - Business Computing with Excel
    - 1st release of server uses COM between Mercury and Excel
    - Response time **~1 sec, too slow**
    - Marshalling problem
  - **Solution**
    - Keep same worksheet definitions
    - Rebuild subset of Excel in Mercury
    - One person, 3 weeks
    - Response time **33 msec**

Appraisal of Mercury…

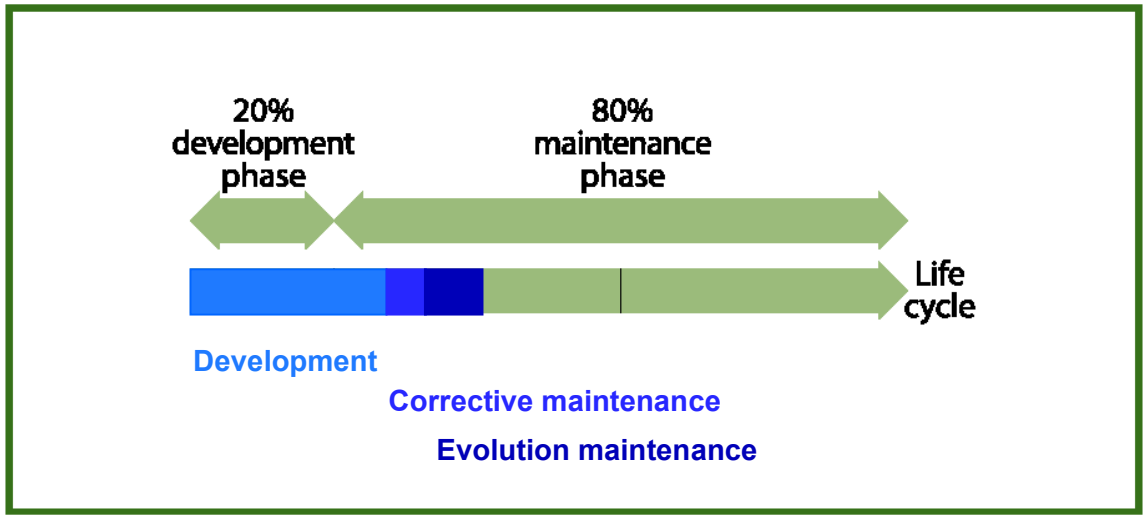
- **Current conclusions**
  - **Very encouraging, but not yet a very large application**
  - **Team of 5 Software Engineers/Computer Scientists**

Design, Project Management	12 p.m
Client (Java)	16 p.m
Server (Mercury)	18 p.m
Other (XML, print, email, …)	10 p.m
Total	**56 p.m**

  - **Total effort = 56 p.m** (Basic COCOMO for 40 KLOC = 115 p.m)
  - **LP for the "complex" part, 32 % of the total effort**

Appraisal of Mercury…

- Current conclusions…



Quality, Cost and Time-to-Market

5. Moving to Web Services

Coloured Petri Nets

- Lessons learned with our extensions to Petri Net
- CPN – tokens with type – is an improvement
- Goals of implementation
  - Arbitrary Mercury functions for arc expressions and types for tokens
- Existential type ensures type safe CPN
- Serialization of CPN state: a place must be a member of a **typeclass**
  - Type checking ensures that CPN always serializable
- Non-determism in CPN:
  - Transition fires if compatible token at each input place to the transition
  - Committed choice non-determinism: prunes the choice point stack once a solution is found
  - Benefit: automatic search without paying the price of non-determinism once no longer needed

Moving to Web Services…

- **Web Services**
  - **Mercury.NET**
    - No change to the language, compiles to IL
    - True interoperability with CLR requires some extensions

      Tyson Dowd, Fergus Henderson, and Peter Ross
      Compiling Mercury to the .NET Common Language Runtime
      http://www.cs.mu.oz.au/research/mercury/information/papers.html
  - **Benchmarks**
    - Platform overhead (e.g. GC support) is the same for all languages
    - Best compilation technology wins
    - Purity allows better optimizations
    - Mercury wins over C# for some benchmarks

Moving to Web Services…

- **Development**
  - Took about a **week** to get the **types** right
  - Took about **½ a day** to code the engine
  - So the question seems to be how do we **support** the programmers **in getting the types right**
- **.NET**
  - **Coexistence Declarative – Imperative** straightforward
  - No marshalling
  - Some C# still required
  - **Mono**: Open Source version of .NET

Statistics for CPN Service

```
[238]>awk -f /cygdrive/x/ddw/source_stats.awk *.m
Number of types:                      48
Number of insts:                       4
Number of predicates:                 40
Number of functions:                  55

Number of predmodes:                  39
Number of funcmodes:                   3
Number of separate modes:              1
Number of implicit function modes:     ?
Total number of modes:          >=    43
                                =<    98
        - det:                        27 ( 62.79%)
        - semidet:                     5 ( 11.63%)
        - nondet:                      2 (  4.65%)
        - multi:                       0 (  0.00%)
        - cc_nondet:                   3 (  6.98%)
        - cc_multi:                    8 ( 18.60%)
        - erroneous:                   0 (  0.00%)
        - failure:                     0 (  0.00%)
Average modes per predicate: >=  1.000
                             =<  1.075

Blank lines:                        372 ( 25.43%)
Comment lines:                      167 ( 11.41%)
Total whitespace/comment lines:     539 ( 36.84%)

Function declaration lines:          61 (  4.17%)
Predicate declaration lines:         70 (  4.78%)
Mode declaration lines:               1 (  0.07%)
Type declaration lines:             134 (  9.16%)
Inst declaration lines:               8 (  0.55%)
Other declaration lines:            115 (  7.86%)
Total declaration lines:            389 ( 26.59%)

Code lines:                         535 ( 36.57%)

Total number of lines:             1463 (100.00%)
```

Less than 100 functions/predicates

924 real lines of code

6. The Future

Compile-time Garbage Collection

- Purity eases the possibility to use **Abstract Interpretation**
- Theoretical work done at KUL
- MC contributes to the implementation
- Around 50% structure reuse on ICFP'00 code

Nancy Mazur, Peter Ross, Gerda Janssens and Maurice Bruynooghe

Practical aspects for a working compile time garbage collection system for Mercury

http://www.cs.mu.oz.au/research/mercury/information/papers.html

The Future…

- **Web Technologies**
  - Web and Internet support technologies derived from a logic/declarative formalism (XML, XSL, XQL…)
  - **Ontologies**, RDF…enhance resource description to the point where inference become possible (*Semantic Web*)
  - OWL extends RDF to provide some of the basic language primitives found in frame based systems (AI?)
  - OWL leverages the power of **Declaration Logic**, with a clean separation between intentional (T-Box) and extensional knowledge (A-Box)

7. Conclusions

- **Practical advantages of declarative programming**
  - **Semantics**
    - Without simple and clear semantics, programming will inevitably be a time and error-prone task
    - Key to many techniques: program analysis, optimization, synthesis, verification and systematic program construction
    - Possibility to effectively reason about the correctness or otherwise of programs
  - **Productivity**
    - With imperative languages, the logic and control are mixed: programmers have no choice but to be concerned about a lot of low level details
    - Having to deal only (or mostly) with the logic component simplifies many things

Conclusions…

- **Methodologies with Mercury** (with FUNDP)
  - **Very-High Level programming & Reverse engineering**
    - Write the solution to the problem (correctness)
    - Document the code with the rationale behind the construction
  - **Derivation from formal specification**
    - Two versions of the applications are required
    - Burden can be lowered by means of systematic derivation techniques
  - **Reuse**
    - Component-based
    - Specifications and compositionality
  - **Declarative reasoning**
    - Enforce a complete understanding of the problem and its solution
    - Program testing can be almost eliminate
  - **Transformational approach**
    - Starting from a “simple (and inefficient) but obviously correct program”
    - Finishing with a “possibly intricate but efficient program”, whose correctness is no longer evident but is ensured by correctness preserving transformations

Conclusions…

- **Observations wrt to methologies[1]**
  - Developers preferring either an a priori or an a posteriori approach can collaborate in the same development activity.
  - The methodology based on declarative reasoning is well-suited to Mercury, especially when the full expressive power of the language can be used.
  - The transformation approach greatly benefits from the multi-paradigm nature of Mercury; it is possible to identify subsets of relational, functional and imperative paradigms taking into account different kinds of operational constraints.
  - Although Mercury is a multidirectional PL, procedures are often restricted in practice to a single usage (correct specifications requires preconditions on their input parameters), but not always…

[1] Baudouin Le Charlier, Yves Deville

Conclusions…

- **Issues with Declarative Programming**
  - **Skepticism**

    I do not see much future in the next few years for some approaches that were heralded as promising: functional programming, logic programming, and expert systems (in their application to software).

    B. Meyer, “Where is software headed”, IEEE Computer, August 1995

  - **LP/FP not for the masses**

    … aimed at people who have mathematical sophistication, who are able to think in more abstract ways, ***that lots of other folks, and I include myself, have trouble with***”

    Brian Kernighan

Conclusions…

- **Issues with Declarative Programming…**
  - **Lack of morale**

    Pessimism of the intellect
  - **Education and the impact of the environment**
    - The weight of Java
    - The new generation of designers and programmers **have been changed by the environment**

Conclusions…

- **But**
  - Software crisis: **predictability** (quality, cost, time) requires **science**
  - Science of software based on discrete mathematics, i.e. **logic**
  - Practical **relevance** of using next generation logic/functional programming is very important
  - Pure declarative language like Mercury **helps a lot** (error catching, evolution of code, optimizations, transformations)

Conclusions…

- **But…**
  - Very good for **modeling** and **specification**
  - **Coexistence** (imperative–declarative) very important in the real-life; emerging platforms (.NET, Mono) simplify that coexistence
  - CBD and Web Services: **trusted**, well formed components are key
  - Opportunities: **ontologies, semantic web**, etc.

Conclusions…

- **Edsger W. Dijkstra**

  Why is software so expensive?

  Because it is tried with cheap labour

  Why is it tried that way?

  Because its intrinsic difficulties are widely and grossly underestimated

Conclusions…

- **Three Personal Concerns**
  - The software profession is, for a large majority, at risk of **losing credibility**
  - This evolution, combined with the market downturn, could have a **negative impact** on **software education**
  - Software jobs will move **abroad**, while they are required to build a competitive **knowledge economy** here
- **Moving programming from Craft to Science**
  - An opportunity to reverse these trends

Acronyms

CBD	Component-Based Development
CLR	Common Language Runtime
COCOMO	Cost Construction MOdel
COM	Component Object Model
CQFD	Cost, Quality, Functions, Delay
CPN	Coloured Petri Net
CRM	Customer Relationship Management
FP	Functional Programming
GC	Garbage Collector
IE	Internet Explorer
IL	Intermediate Language
IO	Input Output
IT	Information Technologies
J2EE	Java 2 Enterprise Edition
KLOC	Kilo Lines Of Codes
LP	Logic Programming
MC	Mission Critical
ODBC	Open DataBase Connectivity

Acronyms

OWL	Web Ontology Language
PITAC	President's Information Technology Advisory Committee
PKI	Public Key Infrastructure
PL	Programming Languages
p.m	person-month
RDF	Resource Description Framework
RPC	Remote Procedure Call
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
TLS	Transport layer Security
WfMC	Workflow Management Coalition
XML	eXtensible Markup Language
XQL	XML Query Language
XSL	eXtensible Stylesheet Language